Shabal, a Submission to NIST's Cryptographic Hash Algorithm Competition

Initiated by the Saphir project

http://www.shabal.com

Editors:
Anne Canteaut, Benoît Chevallier-Mames, Aline Gouget, Pascal Paillier, Thomas Pornin

Contributors:
Emmanuel Bresson, Christophe Clavier, Thomas Fuhr, Thomas Icart, Jean-François Misarsky, María Naya-Plasencia, Jean-René Reinhard, Céline Thuillet, Marion Videau

October 28, 2008

Shabal

Shabal is a cryptographic hash function submitted by the France funded research project SAPHIR to NIST's international competition on hash functions. More specifically, the research partners of SAPHIR (with the notable exception of LIENS) initiated the conception of Shabal and were later joined by partners of the soon-to-be research projectSAPHIR2 who actively contributed to the final design of Shabal. SAPHIR2 is a 4-year research project funded by the French research agency (ANR) and will continue the works and achievements of the SAPHIR project starting from 2009. Partners of SAPHIR2 come from both industry and academia; in addition to partners of SAPHIR, 4 new partners (EADS SN, INRIA, Sagem Sécurité and UVSQ) are about to join and contribute.

SAPHIR (Security and Analysis of Hash Primitives[1]) is an ANR[2] funded project on hash functions. SAPHIR has started on March 2006 for a duration of three years and brings five partners together: Cryptolog International, DCSSI, France Telecom (leader), Gemalto and LIENS. The goal of SAPHIR is to develop a better understanding of recent attacks on hash functions and their potential impact; to extend their scope; to reconsider the design of secure hash functions. The project also aims at proactively anticipating new research directions in the area of hash functions, and at making subsequent results available to the largest audience.

About submitters

Cryptolog International is a software editor specialized in digital signatures and paperless procedures. Founded in 2001 by researchers in cryptography, it has always maintained strong links with fundamental research, through collaborative research projects and participation to various international conferences (Eurocrypt, Crypto) and standardization bodies (ETSI).
WebSite: http://web.cryptolog.com/

DCSSI

The DCSSI (Central Information Systems Security Division) is the State's focal center for Information Systems Security. It was created by decree on July 31, 2001 and is under the authority of the General Secretary for National Defense. As a part of DCSSI, the Crypto Laboratory takes part in the SAPHIR project.
WebSite: http://www.ssi.gouv.fr/en/dcssi/index.html

EADS Secure Networks is a world leading manufacturer and provider of Professional Mobile Radio (PMR) networks, mainly for public safety and governmental users. EADS SN presently provides more than 130 networks worldwide with more than one million users, most of them using access security and end-to-end security.
WebSite: www.eads.net/pmr

[1] http://www.crypto-hash.fr
[2] ANR: Agence Nationale de la Recherche - The French National Research Agency
http://www.agence-nationale-recherche.fr/Intl

France Telecom is the current leader of SAPHIR. France Telecom has a cryptographic team involved in the conception of major products, in different research projects (RNRT SAPHIR, ANR PACE, NoE Ecrypt, etc.) and in standardization activities (AFNOR, ISO, ETSI, etc.).
WebSite: http://www.francetelecom.com

Gemalto is a world leader in digital security and provides end-to-end digital security solutions, from the development of software applications to the design and production of secure personal devices. Gemalto actively contributes to several standardization groups, especially around mobile communications and open platforms for smart cards.
WebSite: http://www.gemalto.com/

INRIA, the French national institute for research in computer science and control, is dedicated to fundamental and applied research in information and communication science and technology. The research work within the SECRET project-team is mostly devoted to the design and analysis of cryptographic algorithms, especially through the study of the involved discrete structures. Most notably, SECRET is the INRIA research team working on symmetric primitives.
WebSite: http://www-rocq.inria.fr/secret

Sagem Sécurité is a high-technology company within the SAFRAN Group. As a world leader on identification solutions, Sagem Sécurité is specialized in people's rights management and physical and logical access applications based on biometrics, as well as secure terminals and smart cards. Integrated systems and equipment by Sagem Sécurité are used worldwide to ensure transport safety, data and personal security, and high-level governmental security. Through the SAFRAN Group, Sagem Sécurité operates worldwide.
WebSite: http://www.sagem-securite.com

Contents

List of Figures

List of Tables

Part 2.B.1

A Complete Written Specification of the Algorithm

Chapter 1

A Short Introduction to Hash Functions

Contents

1.1 Modes for Iterative Hash Functions

Being able to hash a message on the fly without prior knowledge of the whole message or even of its length requires the use of iterative constructions. A well-known example resides in the Merkle-Damgård (MD) construction where a message suitably padded with an injective padding scheme is cut up into blocks which are sequentially processed together with a chaining value through a (finite-length input) compression function. This construction suffers from many attacks even though it has been proven collision resistant provided that the underlying compression function is collision resistant [18, 33]. Almost all known examples of iterated hash functions currently in use are derived from this original Merkle-Damgård principle.

While designing a hash function, one has to get close to a model of what can be an ideal behavior for the algorithm. It has been widely acknowledged that the random oracle model [4] while catching this ideal behavior is also an unreachable goal [12]. However, there are ways to somehow quantify the distance between a given construction and a random oracle. In a black-box setting, a hash function has to be *indistinguishable* from a random oracle. Since in most cases the algorithm is known — especially in the case of an iterative hash function where the underlying iterated function is publicly available — and cannot really be considered as a black box, one has to rely on a more appropriate notion, namely *indifferentiability*. This notion has been introduced in [31] and applied to iterative constructions of hash functions such as [13]. Briefly speaking, this notion takes into account the composite nature of the hash function by considering a *mode*, that is, the way the internal function is employed in the construction. Indifferentiability means that there exists an algorithm (referred to as a *simulator*) which simulates consistently with a random oracle the behavior of the inner function (which the attacker can access too, since it is non black-box), in such a way that the two resulting constructions are indistinguishable.

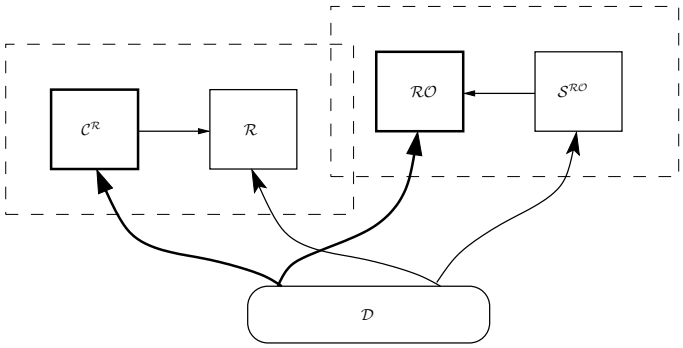

Figure 1.1: Indifferentiability setup. The internal function $\mathcal{R}$ is considered perfect. The mode $\mathcal{C}^{\mathcal{R}}$ has access to $\mathcal{R}$. The simulator $\mathcal{S}^{\mathcal{RO}}$ has oracle access to the random oracle $\mathcal{RO}$. The distinguisher interacts either with $(\mathcal{C}^{\mathcal{R}}, \mathcal{R})$ or $(\mathcal{RO}, \mathcal{S}^{\mathcal{RO}})$ and has to tell them apart.

Still, proofs of indifferentiability assume that the inner functions are perfect which is certainly not the case for a real hash function. A complementary approach to prove the soundness of a construction has been based on the formalization of several properties that a hash function should verify in order to be secure [38]. The idea is to rely on a finite-length input compression function verifying some properties and to specify a domain extension transform to build a hash function which is property preserving (for at least some of them) [3, 1]. An example of this property preservation is the well-known MD-strengthening which ensures collision resistance of the MD construction assuming the collision resistance of the compression function. In this context, one can see indifferentiability as *pseudorandom oracle preservation* [3].

1.2 A General Description of a Sequential Iterative Hash Function

Generally speaking, most of the (sequential) iterative hash constructions have the following structure — we do not describe parallel constructions. We denote by S the internal state of the hash function. For an input message M and a hash value H that can be written as blocks $H_1, \ldots, H_t$, the following informal process is applied:

- *Initialization*:
  - apply appropriate block formatting (including special encoding and/or padding) to the input message and get k blocks with equal size: $M_1, \ldots, M_k$,
  - give an initial value to the internal state and get S_0,
- *Block processing* or *message rounds*: for i from 1 to k, insert the block M_i in the state S_{i-1}, get $S_i = \mathcal{R}(M_i, S_{i-1})$, where $\mathcal{R}$ is called the *compression function*,
- *Discontinuity*: apply a *final transformation* after the last message round: $S_{k+1} = \mathcal{F}(S_k)$
- *Producing the hash value*: sequentially apply for j from 1 to t:
  - extract one block of hash value H_j from the state S_{k+j}: get $H_j = \mathrm{ext}(S_{k+j})$,
  - update the internal state with a *transition function*: get $S_{k+j+1} = \mathcal{T}(S_{k+j})$.

Depending on the construction, some steps can be canceled or slightly twisted, for example in order to use the same underlying function for $\mathcal{R}$, $\mathcal{F}$ and $\mathcal{T}$. Indeed, we have to keep in mind that two quite contradictory goals are aimed to in the design of a hash function: security and

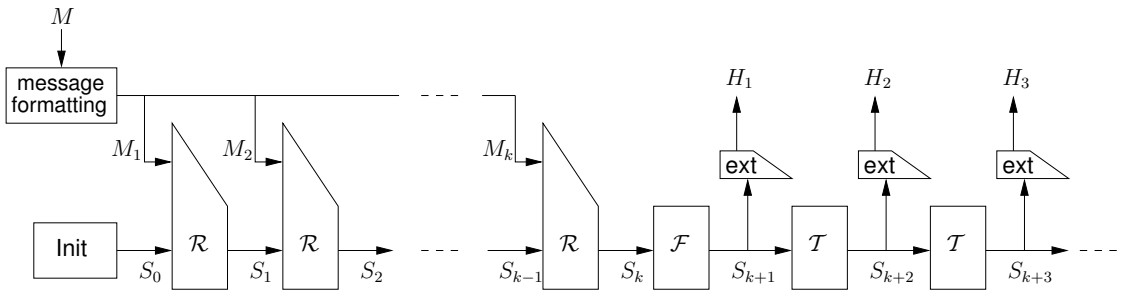

Figure 1.2: A general iterative hash function construction.

performance. Security would require domain separation, independent and perfect functions while performance would require the reuse of existing components and imperfect functions. This explains why the vast majority of existing algorithms only use one underlying function for the definition of the compression function and/or final/transition function, if any. This situation explains the need for assessing the security of the *mode i.e.,* the formal description of how a small function is used to define the overall algorithm and to clarify the security relation between the hash function and the underlying function.

1.3 Some Existing Iterative Modes

1.3.1 Plain Merkle-Damgård

Known as the *plain Merkle-Damgård construction*, this totally insecure mode (which is never used in practice) is simply cited here as it provides the basis for all iterative constructions through the use of a *compression function* $\mathcal{R}$. The compression function has a fixed input bitsize greater than its (fixed) output bitsize. The hash function is obtained as the value of the last internal state.

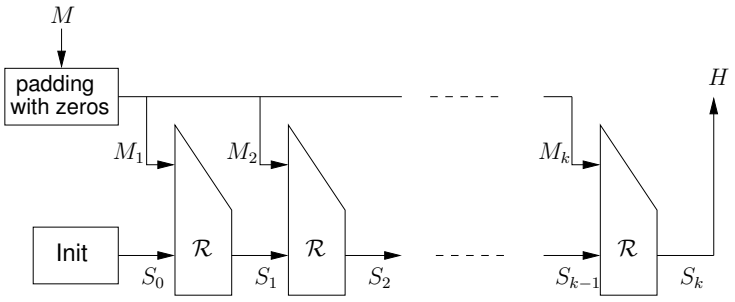

Figure 1.3: Plain Merkle-Damgård construction.

1.3.2 MD With Special Message Formatting

Strengthened MD.

The well-known Merkle-Damgård construction, as it is referred to, is also known as *strengthened Merkle-Damgård*. It is followed by a large majority of algorithms which are still in use (the MD and SHA families of functions follow this paradigm). The only difference with its plain version lies in the padding function which also appends the length of the message. In fact the padding function is required to be injective. When using such a padding scheme, the hash function is collision resistant as soon as the compression function is collision resistant.

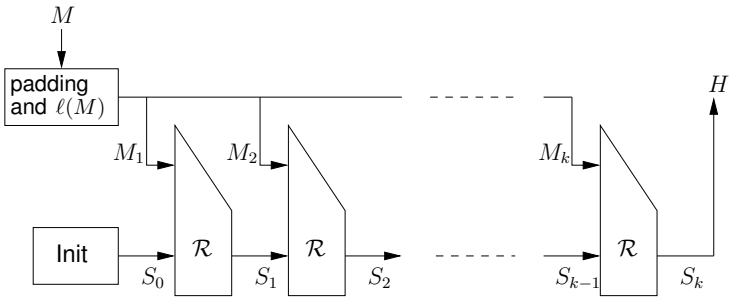

Figure 1.4: Merkle-Damgård construction with MD-strengthening.

Prefix-Free MD.

The prefix-free construction aims at providing a mode which is indifferentiable from a random oracle. This is obtained by modifying the message before hashing it. More precisely, a prefix-free code has to be applied on the incoming message. It is then processed through a plain Merkle-Damgård construction.

This scheme has been proposed in [13] where the authors suggested two prefix-free encoding as examples. In the first one each message block is concatenated with the length of the message and its index while in the second one a 0 bit is prefixed to each message block except the last one which is prefixed with a bit set to 1. Unfortunately, those two solutions suffer from a major drawback, both require the loss of a part of the bandwidth and the first one implies that the length is known before processing the message.

MD with a Counter.

To avoid attacks that rely on finding fixed points in the compression function *i.e.,* values (x, y) such that $\mathcal{R}(x, y) = y$ (for example [26]), a simple idea is to make the input of the compression function depend on the index of the block that has to be processed. A simple way to get this result is to use a counter as an input to the compression function, concatenated with the message block. By doing so, the use of fixed points is only possible at the very moment when the right index appears. A natural drawback of this solution is either the decrease of the size of message blocks if used to patch existing compression functions or a larger memory occupancy if considered during the design of a new compression function. However when this last point is not crucial, the simplicity of the solution and the security gain makes it very straightforward to use.

1.3.3 MD with Larger Internal State

Chop-MD.

In this mode a plain Merkle-Damgård construction is performed and a fraction of the output (the last internal state) is removed. This mode has been proven indifferentiable in [13]. Such an idea, but without the purpose of indifferentiability in mind, is already in use in SHA-384 and SHA-224 respectively obtained by dropping some output bits from SHA-512 and SHA-256.

1.3.4 MD with Discontinuity

NMAC.

The NMAC construction applies an independent hash function to the output of the plain MD construction. It has been proven indifferentiable in [13].

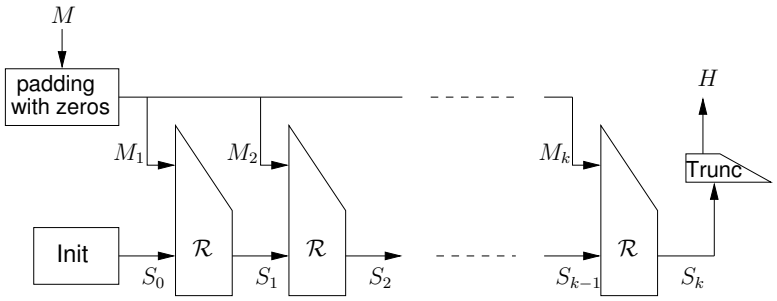

Figure 1.5: Chop Merkle-Damgård construction.

HMAC.

In order to spare the use of another hash function in the NMAC construction, the HMAC construction, proven indifferentiable in [13] prepends a block of 0 bits to the message before processing it through the plain MD. Then it expands or truncates the hash output to fit the size of one block and feed one last time the same MD construction.

Wide Pipe Hash.

This mode proposed in [30] with an instantiation named *double-pipe hash* is equivalent to NMAC. The double-pipe instance shows how to use only one function to get two different compression functions and an internal state twice the size of the original one. The mode is very interesting to mention as it has been introduced for a very practical reason *i.e.,* in order to be "failure-friendly". It means that given a compression function that is known not to be perfect — this is the case for all real world compression functions — the mode aims to compensate for its imperfection.

EMD.

Very similar to the HMAC construction, the EMD construction in [3] aims at providing an indifferentiable construction which is also collision resistance preserving. The collision resistance is provided by the MD-strengthening and the block formatting treats differently the last block whose length must be $\ell_m - \ell_h$ (length of the other blocks minus the size of the hash value). Then the discontinuity is provided by an application of the compression function with a different IV and with input block equal to the last message block concatenated with the result of the chaining value.

1.3.5 Sponge Functions

Recently, *sponge functions* have been introduced in [6] as a new model to capture the behavior of a real-world iterative hash function. The design strategy lies also in making the internal state size grow (similarly to some of the former strategies). This hidden part acts like a reservoir meant to make it difficult to generate and detect internal collisions. What makes the sponge construction specific is that it is intended to mimic the behavior of a random oracle, including the generation of virtually infinite outputs. Thus, the authors also propose to consider it as a model for MAC and stream ciphers. For this purpose, they need to use a transition function for the state on which the security of the scheme strongly relies. The same building block being used for the compression function, a way to insert the message is to XOR it with a part of the internal state.

The sponge construction can be described as follows. The internal state is split into two parts $S = (S^A, S^C)$, with $|S^A| = |M_i|$.

- *Initialization*: apply an appropriate padding to the input message and get k blocks: $M_1, \ldots, M_k$. Give an initial value to the internal state and get $S_0 = (S_0^A, S_0^C)$,

- *Block processing* or *message rounds*: for i from 1 to k, insert the block M_i in the state S_{i-1} and get $S_i = \mathcal{T}(S^A_{i-1} \oplus M_i, S^C_{i-1})$
- *Producing the hash value*: sequentially apply for j from 1 to t:
  - extract one block of hash value H_j from the state S_{k+j-1}: get $H_j = \text{Trunc}(S_{k+j-1}) = S^A_{k+j-1}$,
  - update the internal state with a transformation: get $S_{k+j} = \mathcal{T}(S_{k+j-1})$,

The above model is proven to be indifferentiable in [7]. It appears as a formalization of some algorithms proposals that do not completely fit the scheme principles.

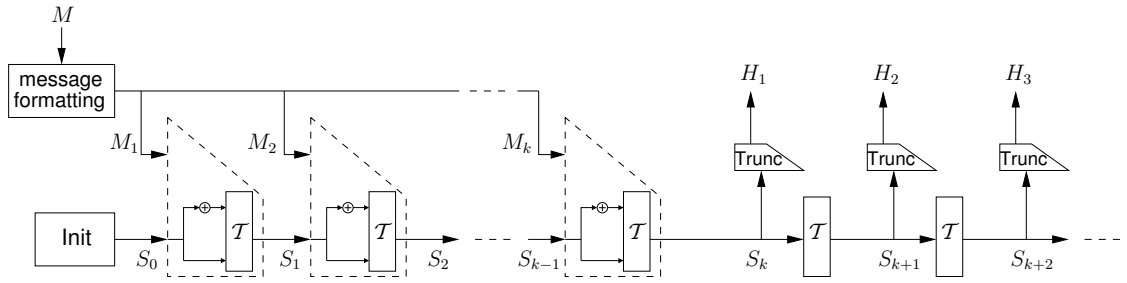

Figure 1.6: The sponge construction.

The "Concatenate-Permute-Truncate" Design.

This design was named in [29] describing the proposal Grindhal and referring to the proposal Snefru [34]. The original idea developed in Snefru is to use an alternative way to design a compression function which would not be based on a traditional adaptation of a block cipher and most notably would spare key derivation.

In this design, the insertion of the message is not made as an XOR but by concatenating the input block to a truncated internal state. There have been early attacks against Snefru [10] improved in [8] as well as for Grindhal [35, 24].

Belt-and-Mill Hash Functions.

Following the ideas developed in [15], the idea behind sponge functions has first been used in the proposal Panama [17]. The algorithm can be used both as a hash function and a stream cipher. Unfortunately, Panama in hash function mode has been severely broken [36, 16]. The idea to formalize a new mode was however on its way, and what was named an *iterative mangling function* has been designed (precisely called a *belt-and-mill hash function*).

Largely inspired by the work done on its predecessor, Radiogatún [5] appears as the result of the formalization of Panama's design and attacks. As a recent proposal, it has not been as thoroughly reviewed as older algorithms. However, some recent analysis have been published which do not break the original security claim of its designers [11, 28, 27].

Strictly speaking, none of the above proposals follows blindly the sponge design, as a discontinuity can be added or the insertion be slightly twisted.

Chapter 2

Complete Description of Shabal

Contents

In this section, we describe our candidate function to the NIST competition, which we facetiously baptized Shabal. The name of our algorithm was chosen as a tribute to Sébastien Chabal, a French rugby player known for his aggressive playing as well as for his beard and long hair which got him the nickname of "Caveman".

This section contains the description of our algorithm. We also explain intuitions behind the reasons that made us shape Shabal the way it; the alternative possibilities and precise explanations for our design choices are dedicated to Chapter 4. Moreover, the description of Shabal may be easier to understand using the patterns given in Chapter 3 (one can also take a look at the detailed execution trace given in Appendix B). Implementation tricks aiming at simplifying or accelerating your Shabal implementation are discussed in Chapter 7. Finally, basic implementation is provided in Appendix A.

2.1 Conventions

2.1.1 Endianess

The input of Shabal is an ordered sequence of bits of arbitrary length. An empty sequence is allowed; Shabal accommodates to bitstreams of any length — however we evaluated its security only for inputs of length smaller than 2^{73} bits. The input length can be any integer value and is not restricted to multiples of 8. Given a sequence of bits, bits are numbered by their index, the first bit having index 0. We use the terms *left* and *right* to describe an ordered sequence of bits: the first bit in the sequence is called the *leftmost* bit, the last bit is the *rightmost* bit.

The input sequence is first *padded*: extra bits are added in a way which implies (among other properties) that the length of the padded sequence is not equal to 0 and is a multiple of 32. The padded sequence is then split into groups of eight bits. We will make use of the term *byte* to denote such groups of bits[1]: the first byte consists of the eight first (leftmost) bits in the padded sequence; the next eight bits are grouped into the second byte, and so on. Since the length of the padded sequence is a multiple of 32, this process yields an integral number of bytes and that number is itself a multiple of 4. Each byte has a *value* which is an integer between 0 and 255 (inclusive). The byte value is derived from the sequence of eight bits by using representation in base 2, the leftmost bit being most significant: if the eight bits of an octet, from left to right, are denoted b_0, b_1,... b_7 then the value of this byte is equal to $\sum_{i=0}^{7} 2^{7-i} b_i$.

As an illustration, the padding procedure begins by appending a bit set to 1. Thus, when the input sequence has a length which is a multiple of 8 (*i.e.,* the unpadded/raw input sequence is an integral number of bytes), then this additional bit becomes the addition of a new byte which has its upper (leftmost) bit set to 1: the new byte has value 128.

Many protocols and software platforms define data as streams of bytes and not individual bits. On such architectures, the process of grouping bits together into bytes is assumed to have already taken place using the conventions discussed above. These conventions directly comply with NIST's API for reference implementations within the SHA-3 competition; they are also compliant with widespread conventions such as the BER encoding of structures expressed in ASN.1 notation which are ubiquitous to many standards related to X.509.

When the padded sequence has been converted into a sequence of bytes, these bytes are assembled into groups of four consecutive values: the first (leftmost) four bytes become the first group, the next four bytes become the second group, and so forth. Each group is hereafter called a *32-bit word* or more simply a *word*. Since the length of the padded input is a multiple of 32, this process yields an integral number of words. Each word has a *value* which is derived from the four bytes with the so-called *little-endian* convention: the first (leftmost) byte is least significant. Thus if the four bytes taken from left to right have values c_0, c_1, c_2 and c_3 — all lying in the range $[0, 255]$) then the value of the word is $c_0 + 2^8 c_1 + 2^{16} c_2 + 2^{24} c_3$.

The operations of Shabal are expressed in terms of words. The output of Shabal is a sequence of words which is transformed into bits using the same conventions in reverse order: words become bytes with the little-endian convention and each byte represents a sequence of eight bits, the most significant one being the leftmost bit. Note that the final output bit sequence is truncated to a configurable output length[2].

It shall be noted that these conventions for the order of bits within a byte and bytes within a word are identical to those used by the well-known hash function MD5. They are sometimes referred to as *mixed-endian*: big-endian at the bit level, and little-endian at the byte level.

2.1.2 Notation

In this section, we introduce notation that are extensively used in the remainder of this document. Let x, y be n-bit words ($n = 32$ for non-weakened versions of Shabal). We denote by $x \oplus y$ the bitwise *exclusive or* (or XOR) of x and y. By $x \wedge y$ we denote the bitwise logical *and* of x and y. We will also denote by $\overline{x}$ the complement of x *i.e.,* $x \oplus \mathbf{1}$ — the notation $\mathbf{1}$ (bold 'one') stands for `0xFFFFFFFF` for a 32-bit word. Finally $x \lll j$ denotes the rotation of x by j bits to the left and $x \ll j$ denotes the shift of x by j bits to the left. Rotation differs from shift in that bits disappearing on the left side come back on the right side in the former while they are simply erased in the latter (so $x \ll j$ means that j zero-bits enter from the right). It is expected that j be lower than the bitsize of a word (*i.e.,* 32 for non-weakened version of Shabal). If this is not the case, j is reduced modulo the word bitsize before the rotation is carried out.

All logical operations used in this document are bitwise *i.e.,* are applied separately on each and every bit in words. We will also use *wordwise* operations *i.e.,* operations on words such as addition

[1] The equivalent term *octet* is also often encountered in technical documents.

[2] The intended output length also modifies internal processing.

and subtraction modulo 2^{32}. We will denote additions modulo 2^{32} by $\boxplus$ or $+$, whose meaning will be clear from the context. In other words, if X and Y are arrays of 32-bit words, $X + Y$ means that the result is an array of words containing words of X and Y added together with no carry propagating from one word to the next. The same convention applies for subtraction.

2.2 Description of the Mode of Operation

The construction on which Shabal is based makes use of a keyed permutation $\mathcal{P}$ and is proven to be indifferentiable from a random oracle. Shabal is entirely defined by this generic construction together with some particular specification of $\mathcal{P}$ which we define in Section 2.3.

Let ℓ_h be the output length of Shabal. For notational simplicity, we will assume that only multiples of 32 are allowed (and most noticeably 192, 224, 256, 384 and 512). Throughout the rest of this document, Shabal with a message digest of ℓ_h bits is referred to as Shabal-ℓ_h as long as $\ell_h \in \{192, 224, 256, 384, 512\}$.[3]

2.2.1 Description

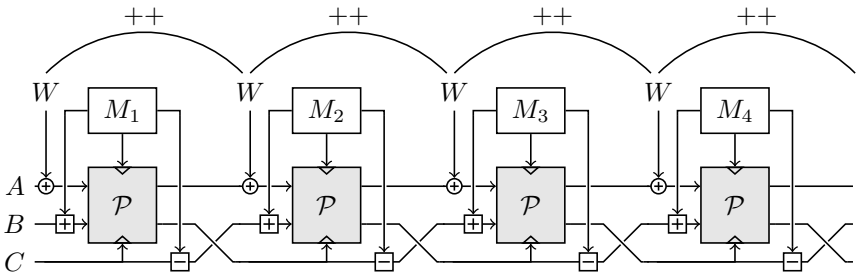

Figure 2.1: The mode of operation: Message rounds

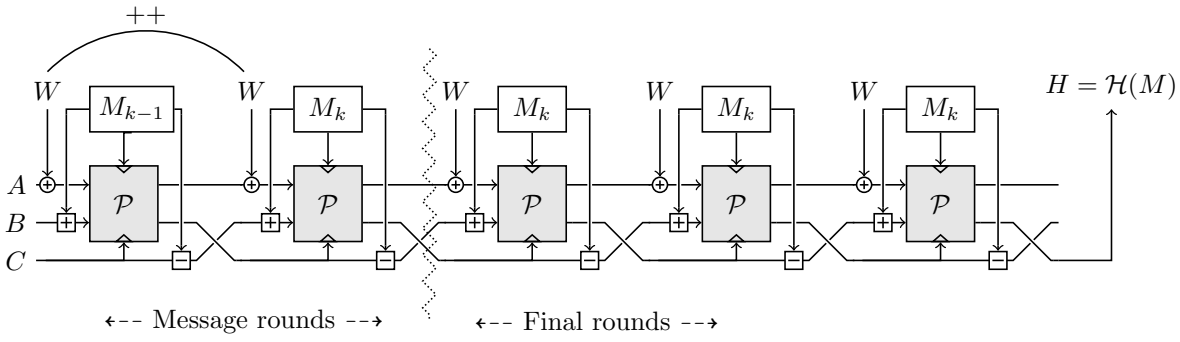

Figure 2.2: Final rounds: View 1

Our hash construction uses an internal buffer divided into three different parts $(A, B, C) \in \{0,1\}^{\ell_a} \times \{0,1\}^{\ell_m} \times \{0,1\}^{\ell_m}$ which at initialization are set to initial values (A_0, B_0, C_0). An auxiliary buffer $W \in \{0,1\}^{64}$ is used as a counter to number message blocks. Due to its particular role, W is not considered as a part of the internal buffer. Shabal hashes ℓ_m-bit message blocks iteratively. The construction uses a keyed permutation $\mathcal{P}$ where $\mathcal{P} : \{0,1\}^{\ell_m} \times \{0,1\}^{\ell_a} \times \{0,1\}^{\ell_m} \times$

[3]We explicitly consider the output size of 192 bits – which is not a request from NIST – since one may find it to be of particular interest for ECDSA-192.

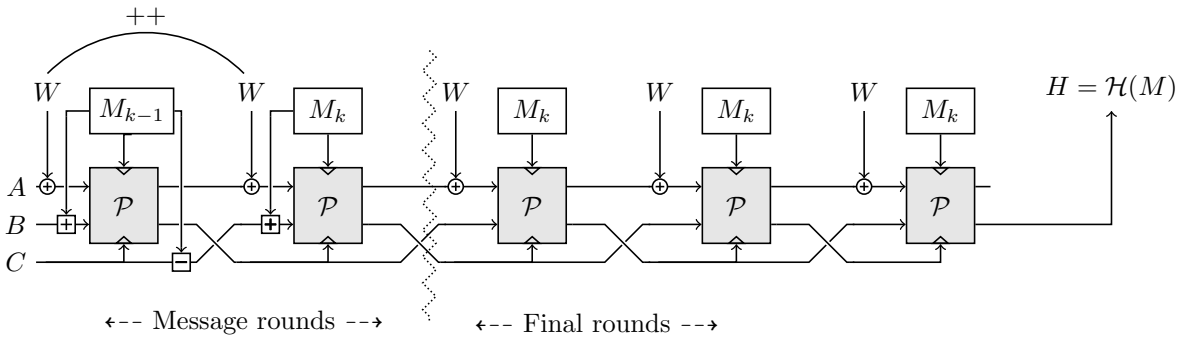

Figure 2.3: Final rounds: View 2

$\{0,1\}^{\ell_m} \to \{0,1\}^{\ell_a} \times \{0,1\}^{\ell_m}$. By definition, for any key $(M, C) \in \{0,1\}^{\ell_m} \times \{0,1\}^{\ell_m}$, the function $\mathcal{P}_{M,C} : (A, B) \to \mathcal{P}_{M,C}(A, B) = \mathcal{P}(M, A, B, C)$ is a permutation.

Description of the Mode of Operation

Initialization: $(A, B, C, W) \leftarrow (A_0, B_0, C_0, 1)$.

Padding: Post-pad the message with a bit set to 1 followed by as many 0 bits as required to yield a padded message with an exact number of ℓ_m-bit blocks.

Message rounds: For w ranging from 1 to k (w being equal to $w = 2^{32} \cdot W[1] + W[0]$), do:

- **add:** the message is introduced.

$$B \leftarrow B + M_w,$$

  where $B \leftarrow B + M_w$ means that B and M_w are added wordwise (again, there is no carry from one word to the next).

- **counter:** XOR the counter in $A[0]$ and $A[1]$.

$$A[0] \leftarrow A[0] \oplus W[0], \quad A[1] \leftarrow A[1] \oplus W[1].$$

- **permute:** apply the keyed permutation.

$$(A, B) \leftarrow \mathcal{P}_{M_w, C}(A, B).$$

- **sub:** the message is subtracted.

$$C \leftarrow C - M_w,$$

  where $C \leftarrow C - M_w$ means that C and M are subtracted wordwise.

- **swap:** B and C are exchanged.

$$(B, C) \leftarrow (C, B).$$

Final rounds: At the end of message rounds, perform a series of final rounds: the message round is applied 3 times with the lastly inserted message block M_k, the counter w being left unchanged and fixed to k.

Output: Output words $C[16 - \ell_h/32]$ to $C[15]$. The contents of A and B are ignored.

A graphical view of the hash construction is displayed on Figure 2.1. At this stage, note that simple optimizations are possible in the final rounds (see the differences between Figures 2.2 and 2.3): in particular, the last *sub* operation is removed, the last *swap*, and the *sub* and *add* between applications of $\mathcal{P}$ in final rounds. The first picture provides a view on atomic rounds made of sequences of *add, counter, permute, sub, exchange* operations while the second picture shows a more efficient but somewhat more code-consuming presentation on the final rounds of Shabal.

The effect of a message round on the internal state is denoted $(A, B, C, w+1) = \mathcal{R}(M_w, A, B, C, w)$ or $(S, w+1) = \mathcal{R}(M_w, S, w)$ for short. The effect of final rounds is referred to as $\mathcal{F}$ (with the notation $(S, w) = \mathcal{F}(M_k, S, w)$); we remind that the only difference between $\mathcal{R}$ and $\mathcal{F}$ is that the counter is not incremented in $\mathcal{F}$ as opposed to $\mathcal{R}$.

2.2.2 A High-Level View

We give below a more synthetic view of Shabal.

Initialization: $(A, B, C) \leftarrow (A_0, B_0, C_0)$

Message Rounds: $M = M_1, \ldots, M_k$

For w from 1 to k **do**

1. $B \leftarrow B + M_w$
2. $A \leftarrow A \oplus w$
3. $(A, B) \leftarrow \mathcal{P}_{M_w, C}(A, B)$
4. $C \leftarrow C - M_w$
5. $(B, C) \leftarrow (C, B)$

End do

Final rounds:

For i from 0 to 2 **do**

1. $B \leftarrow B + M_k$
2. $A \leftarrow A \oplus k$
3. $(A, B) \leftarrow \mathcal{P}_{M_k, C}(A, B)$
4. $C \leftarrow C - M_k$
5. $(B, C) \leftarrow (C, B)$

End do

Output: $H = \mathrm{msb}_{\ell_h}(C)$

2.2.3 Security Results

Chapter 5 focuses on security properties of the mode of operation and provides proofs that Shabal is *(a)* indifferentiable from a random oracle, *(b)* collision resistant, *(c)* preimage resistant and *(d)* second preimage resistant, assuming that the inner keyed permutation $\mathcal{P}$ behaves as a random keyed permutation. All bounds are shown to be optimal in Chapter 11 where we exhibit generic attacks that meet these security bounds. We refer the reader to these sections for more details.

2.3 Specifying the Hash Function Shabal

In Section 2.2, we have described the mode of operation on which our proposition Shabal is based. In this section, we describe a number of implementation details which characterize Shabal. Although other implementation choices of the mode could be defined as well to yield other hash functions, we stress that the design choices we make in what follows are integral parts of Shabal and that any other setting cannot be considered as being Shabal.

2.3.1 Overview

Shabal only defines message blocks of $\ell_m = 512$ bits. For two tunable security parameters $p \geq 2$ and $r \geq 2$, we define the internal state buffer as a (A, B, C) which is a $(1024 + 32r)$-bit buffer viewed as arrays of 32-bit words. More precisely, B and C are 16-word arrays while A is an r-word

buffer. We thus have $\ell_a = 32r$. The counter W, which is not considered as a part of the internal buffer, is viewed as a 2-word buffer. Shabal is then defined as follows.

Description of Shabal (prefix approach)

Initialization: $(A, B, C) \leftarrow 0$, $w \leftarrow -1$.

Prefixing: The message is prefixed with 32 words set to fixed values ranging from ℓ_h (written as a 32-bit word) to $\ell_h + 31$ where $\ell_h \in \{192, 224, 256, 384, 512\}$ is the output length.

Padding: Post-pad the input message with a bit set to 1 followed by as many 0 bits as required so that the padded message can be split into 512-bit blocks.

Message rounds: For w ranging from -1 to k (w being equal to $w = 2^{32} \cdot W[1] + W[0]$), do:

- **add:** the current message block is inserted.

$$B \leftarrow B + M_w.$$

- **counter:** XOR the counter in $A[0]$ and $A[1]$.

$$A[0] \leftarrow A[0] \oplus W[0], \quad A[1] \leftarrow A[1] \oplus W[1].$$

- **permute:** apply the keyed permutation described in Section 2.3.2.

$$(A, B) \leftarrow \mathcal{P}_{M_w, C}(A, B).$$

- **sub:** the message block is subtracted.

$$C \leftarrow C - M_w.$$

- **swap:** B and C are swapped.

$$(B, C) \leftarrow (C, B).$$

Final rounds: When all message blocks are treated, perform 3 final rounds. A final round performs a message round with the last message block M_k, the counter w being fixed to the total number k of message blocks inserted in the message insertion phase.

Output: Finally output the words $C[16 - \ell_h/32]$ to $C[15]$ in that order. The contents of A and B are ignored.

The initialization value of w is chosen to be -1 so that once the 2-block prefix message is treated, the index of the first input message block is $w = 1$. Throughout the document, the prefix is denoted by (M_{-1}, M_0). In particular, it holds that $M_{-1}[0] = \ell_h$, $M_{-1}[15] = \ell_h + 15$, $M_0[0] = \ell_h + 16$ and $M_0[15] = \ell_h + 31$.

It is worth noticing that, as an alternative to the above, one may ignore the prefixing of the message and precompute the contents $(A, B, C) = \mathsf{IV}_{\ell_h}$ of the internal state resulting from hashing the two blocks (M_{-1}, M_0). The simplified algorithm is described below.

Description of Shabal (IV approach)

Initialization: $(A, B, C) \leftarrow \mathsf{IV}_{\ell_h}$, $w \leftarrow 1$.

Padding: Post-pad the input message with a bit set to 1 followed by as many 0 bits as required so that the padded message can be split into 512-bit blocks.

Message rounds: For w ranging from 1 to k (w being equal to $w = 2^{32} \cdot W[1] + W[0]$), do:

- **add:** the current message block is inserted.

$$B \leftarrow B + M_w.$$

- **counter:** XOR the counter in $A[0]$ and $A[1]$.

$$A[0] \leftarrow A[0] \oplus W[0], \quad A[1] \leftarrow A[1] \oplus W[1].$$

- **permute:** apply the keyed permutation described in Section 2.3.2.

$$(A, B) \leftarrow \mathcal{P}_{M_w, C}(A, B).$$

- **sub:** the message is subtracted.

$$C \leftarrow C - M_w.$$

- **swap:** B and C are swapped.

$$(B, C) \leftarrow (C, B).$$

Final rounds: When all message blocks are treated, perform 3 final rounds. A final round performs a message round with the last message block M_k, the counter w being fixed to the total number k of message blocks inserted in the message insertion phase.

Output: Finally output the words $C[16 - \ell_h/32]$ to $C[15]$ in that order. The contents of A and B are ignored.

In Section 3.1, the initialization vectors IV_{ℓ_h} are provided for all supported values of ℓ_h. Let us stress once again that these two ways of defining Shabal are strictly equivalent. Depending on several parameters (see Section 4.5) among which performance tradeoffs, it is left as an implementation choice to follow one or the other approach.

2.3.2 The Keyed Permutation

We now move on to the description of the inner keyed permutation of Shabal. We make use of an NLFSR-based construction (see also Figure 2.4), whose design rationale are provided in Section 4.2.1.

Keyed Permutation $\mathcal{P}$ used in Shabal

Input: M, A, B, C
Output: A, B

For i from 0 to 15, do:

- $B[i] \leftarrow B[i] \lll 17$

Next i

For j from 0 to $p-1$, do:

- For i from 0 to 15, do:
  - Compute

$$\begin{aligned} A[i+16j \bmod r] \quad \leftarrow \quad & \mathcal{U}\big(A[i+16j \bmod r] \oplus \mathcal{V}(A[i-1+16j \bmod r] \lll 15) \\ & \qquad \oplus C[8-i \bmod 16]\big) \\ & \oplus B[i+o_1 \bmod 16] \\ & \oplus (B[i+o_2 \bmod 16] \wedge \overline{B[i+o_3 \bmod 16]}) \\ & \oplus M[i] \end{aligned}$$

    where $(o_1, o_2, o_3) = (13, 9, 6)$ are offset values discussed later in Section 4.3.
  - $B[i] \leftarrow (B[i] \lll 1) \oplus \overline{A[i+16j \bmod r]}$
- Next i

Next j

For j from 0 to 35, do:

- $A[j \bmod r] \leftarrow A[j \bmod r] + C[j+3 \bmod 16]$

Next j

In the above description, $\mathcal{U} : x \mapsto 3 \times x \bmod 2^{32}$ and $\mathcal{V} : x \mapsto 5 \times x \bmod 2^{32}$ are used as nonlinear functions (see Section 4.2.3). Offset values $(o_1, o_2, o_3) = (13, 9, 6)$ are carefully chosen as explained in Section 4.3. Parameters (p, r) may have several acceptable values $p \geq 2$ and $r \geq 2$; however Shabal defines specific values for (p, r) as discussed in Section 2.5.

The final loop of $\mathcal{P}$ (*i.e.*, where $A[j \bmod r] \leftarrow A[j \bmod r] + C[3+j \bmod 16]$) is not fully generic towards the parameter r as explained in Section 4.2.6. Changing a value for r that differs from the one given in Section 2.5 implies applying modifications to this last loop.

2.4 Tunable Security Parameters

Shabal features two security parameters:

Parameter p: the number of loops performed within one application of the keyed permutation; larger values of p provide better security guarantees.

Parameter r: the remanence of A. The minimal value for r is 2 due to the insertion of the 64-bit counter W in $A[0]$ and $A[1]$. r corresponds to a security margin as extensively discussed in Chapter 5.

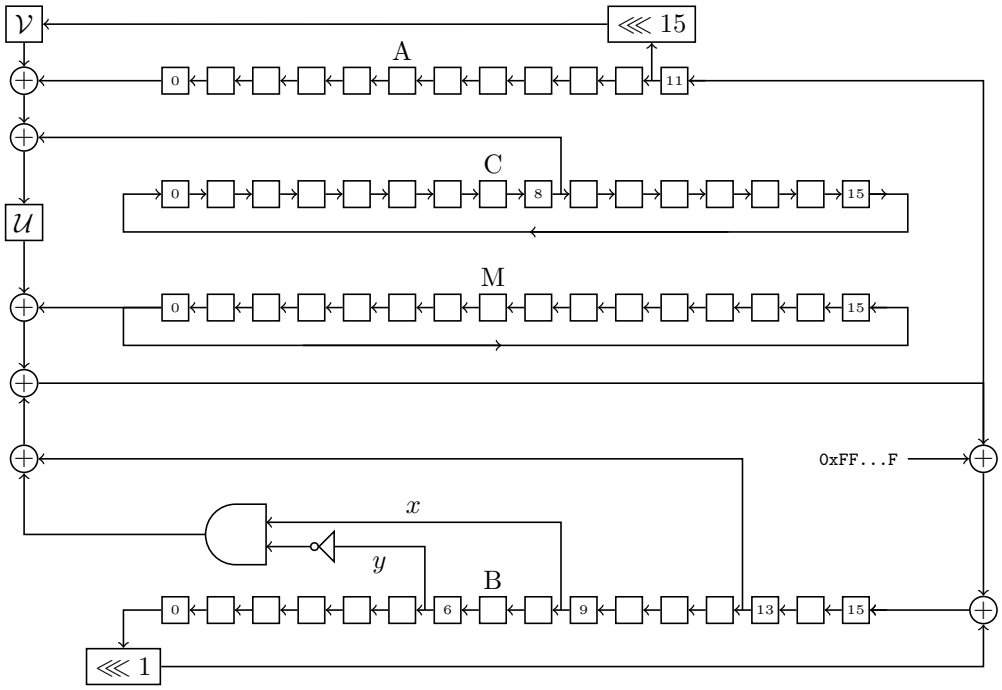

Figure 2.4: Main structure of the keyed permutation used in Shabal.

In our security analysis, we only consider the case where

$$16 \cdot p \equiv 0 \mod r$$

since otherwise certain words of A are more intensively used than others.

We note however that parameters p and r have a different impact on the security of the hash function. Parameter r *increases* the *capacity* (in the sense of [7]) of the mode of operation of Shabal detailed in Section 2.2. Increasing r is therefore a direct way to add a (provable security minded) security margin. However, we also note that too large a value for r is not compatible with a correct level of diffusion and real-world security (furthermore, r is structurally upper-bounded by $16p$). On the contrary, parameter p does not increase the size of the internal state but has the effect to *strengthen* the keyed permutation. Larger enough values of p make the permutation behave in a less controllable way. In a sense, increasing p makes the permutation closer to an idealized permutation. This is true up to a certain threshold above which taking larger values for p will not increase security anymore.

2.5 Parameter Choices in Shabal

The submitted algorithm Shabal strictly uses $(p, r) = (3, 12)$. Other choices of parameters must *not* be considered as Shabal, even though their study may reveal interesting from a research perspective. In Shabal, it always holds that $16p = 0 \bmod r$ so that all the words of A are used equally often.

Chapter 3

Some Test Patterns

Contents

We give in this chapter, for all output bitsizes $\ell_h \in \{192, 224, 256, 384, 512\}$, different test patterns which everyone's implementation must comply with. These data include the initialization vector IV_{ℓ_h} to use when writing Shabal in the IV manner, as well as the final content of the state and the hash result when hashing two example messages. The first example message (message A) is an all-zero full block, which may equivalently be denoted as ${0_1}^{512}$ (bit list), ${0_8}^{64}$ (byte list) or ${0_{32}}^{16}$ (word list). The second example message (message B) is a 102-byte string defined as:

"`abcdefghijklmnopqrstuvwxyz-0123456789-ABCDEFGHIJKLMNOPQRSTUVWXYZ-0123456789-abcdefghijklmnopqrstuvwxyz`"

Note that message B is longer than one block but does not exactly fit on two blocks.

With the aim to facilitate the writing and debugging of Shabal, we also provide the complete lists of all successive intermediate states when hashing message A and message B with all five functions Shabal-ℓ_h (see in Appendix B).

3.1 The Different Initialization Vectors

3.1.1 Initialization Vector for Shabal-192

```
A : FD749ED4 B798E530 33904B6F 46BDA85E 076934B4 454B4058 77F74527 FB4CF465
    62931DA9 E778C8DB 22B3998E AC15CFB9

B : 58BCBAC4 EC47A08E AEE933B2 DFCBC824 A7944804 BF65BDB0 5A9D4502 59979AF7
    C5CEA54E 4B6B8150 16E71909 7D632319 930573A0 F34C63D1 CAF914B4 FDD6612C

C : 61550878 89EF2B75 A1660C46 7EF3855B 7297B58C 1BC67793 7FB1C723 B66FC640
    1A48B71C F0976D17 088CE80A A454EDF3 1C096BF4 AC76224B 5215781C CD5D2669
```

3.1.2 Initialization Vector for Shabal-224

```
A : A5201467 A9B8D94A D4CED997 68379D7B A7FC73BA F1A2546B 606782BF E0BCFD0F
    2F25374E 069A149F 5E2DFF25 FAECF061

B : EC9905D8 F21850CF C0A746C8 21DAD498 35156EEB 088C97F2 26303E40 8A2D4FB5
    FEEE44B6 8A1E9573 7B81111A CBC139F0 A3513861 1D2C362E 918C580E B58E1B9C

C : E4B573A1 4C1A0880 1E907C51 04807EFD 3AD8CDE5 16B21302 02512C53 2204CB18
    99405F2D E5B648A1 70AB1D43 A10C25C2 16F1AC05 38BBEB56 9B01DC60 B1096D83
```

3.1.3 Initialization Vector for Shabal-256

```
A : 52F84552 E54B7999 2D8EE3EC B9645191 E0078B86 BB7C44C9 D2B5C1CA B0D2EB8C
    14CE5A45 22AF50DC EFFDBC6B EB21B74A

B : B555C6EE 3E710596 A72A652F 9301515F DA28C1FA 696FD868 9CB6BF72 0AFE4002
    A6E03615 5138C1D4 BE216306 B38B8890 3EA8B96B 3299ACE4 30924DD4 55CB34A5

C : B405F031 C4233EBA B3733979 C0DD9D55 C51C28AE A327B8E1 56C56167 ED614433
    88B59D60 60E2CEBA 758B4B8B 83E82A7F BC968828 E6E00BF7 BA839E55 9B491C60
```

3.1.4 Initialization Vector for Shabal-384

```
A : C8FCA331 E55C504E 003EBF26 BB6B8D83 7B0448C1 41B82789 0A7C9601 8D659CFF
    B6E2673E CA54C77B 1460FD7E 3FCB8F2D

B : 527291FC 2A16455F 78E627E5 944F169F 1CA6F016 A854EA25 8DB98ABE F2C62641
    30117DCB CF5C4309 93711A25 F9F671B8 B01D2116 333F4B89 B285D165 86829B36

C : F764B11A 76172146 CEF6934D C6D28399 FE095F61 5E6018B4 5048ECF5 51353261
    6E6E36DC 63130DAD A9C69BD6 1E90EA0C 7C35073B 28D95E6D AA340E0D CB3DEE70
```

3.1.5 Initialization Vector for Shabal-512

```
A : 20728DFD 46C0BD53 E782B699 55304632 71B4EF90 0EA9E82C DBB930F1 FAD06B8B
    BE0CAE40 8BD14410 76D2ADAC 28ACAB7F

B : C1099CB7 07B385F3 E7442C26 CC8AD640 EB6F56C7 1EA81AA9 73B9D314 1DE85D08
    48910A5A 893B22DB C5A0DF44 BBC4324E 72D2F240 75941D99 6D8BDE82 A1A7502B

C : D9BF68D1 58BAD750 56028CB2 8134F359 B5D469D8 941A8CC2 418B2A6E 04052780
    7F07D787 5194358F 3C60D665 BE97D79A 950C3434 AED9A06D 2537DC8D 7CDB5969
```

3.2 Final States and Outputs when Hashing Message A

The value of message A, expressed as a list of bytes, is the following:

```
M1 : 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
     00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
     00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
     00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

Note that for each output length the hash value of message A is given twice: first as a word list directly extracted from the end of the state buffer C, then as a byte list expressed in accordance with its *little-endian* representation.

3.2.1 Final State and Output for Shabal-192

```
A : A38C0C63 17C2CAE8 3248572C 1C89CAD5 176ED597 B242B8AD 73298C22 7ADF1817
    00D909DA 61AD8518 90266914 9DC1F617

B : 260A3D42 E9E62340 385A3EBF 2978F492 A1DE4E1A AEDBB855 49DB44CD D0B179F3
    7D7FAAE0 87798FA8 9F7F5E35 4A9F52FE A0F35652 65A6D26E 320A1851 EFF9A7CD

C : F91EEE5E 99DCC78C 82F72599 8CACD775 09544255 ED275CF3 0166F95E 2C375AFA
    49AAFBE0 4D9C01C6 CB6E700F CE4DCF97 D2BBBF00 0C5364FB B40C8732 0D733948

H : CB6E700F CE4DCF97 D2BBBF00 0C5364FB B40C8732 0D733948

H : 0F 70 6E CB 97 CF 4D CE 00 BF BB D2 FB 64 53 0C
    32 87 0C B4 48 39 73 0D
```

3.2.2 Final State and Output for Shabal-224

```
A : 08FC66FC CE392D14 42C29F35 5649D86A DCD65214 9A423F72 99D2F688 5073E130
    1E1F9B61 A28A416E 9C9572AB C3A9B2BC

B : 33D58779 1AD91014 D67F57EC FDADD8EC 75AB31C5 C50ECE4D B7C2CD52 3B8500C5
    FCCD0F17 F7EED488 FF906EED DB95D041 965CBF04 B0E05609 1B26DB06 843B9DE7

C : BE6AC9DC 86AFB8BB 300C6C1B 237F0C8C 6DEF5EEF 599CA070 540040F7 EEA985E4
    4A5B8375 14A6DD99 E8D207F9 F7187681 326F6930 8BCAAE00 25F4855F 3120BA43

H : 14A6DD99 E8D207F9 F7187681 326F6930 8BCAAE00 25F4855F 3120BA43

H : 99 DD A6 14 F9 07 D2 E8 81 76 18 F7 30 69 6F 32
    00 AE CA 8B 5F 85 F4 25 43 BA 20 31
```

3.2.3 Final State and Output for Shabal-256

```
A : 3DBA182A D0D6787E DAD8F4C9 CC065328 A36A08C7 902C794E 43E5A220 E2F378F1
    1E35B4C3 EF6B834E 8E442A11 6922E895

B : 66DF96F1 E4D309BD 6377D5E2 48F253E3 F8E9B974 0773C11C 8B2B886F 9D7AC002
    343612B3 80C441F4 F676FCAD 3D6A453A 6CBEC284 2B3DE748 57EE16DB 80DBFF08

C : 57E837B3 3B2C6ACA E0358DC2 2BD758E9 30F7A2ED DF3516C7 253CB0E0 1A1A98FC
    C0088FDA 9ABA672A 79D0BD56 07AE488E 095E2114 06855B3B 1877A349 A2543F99

H : C0088FDA 9ABA672A 79D0BD56 07AE488E 095E2114 06855B3B 1877A349 A2543F99

H : DA 8F 08 C0 2A 67 BA 9A 56 BD D0 79 8E 48 AE 07
    14 21 5E 09 3B 5B 85 06 49 A3 77 18 99 3F 54 A2
```

3.2.4 Final State and Output for Shabal-384

```
A : 37661E10 1BEDBBD5 B022D077 CB1781BD 23DCFA84 AF4946EC 9C681ADD 8C48B88C
    6BC4D0CB 1F4A95CD 0F2C5CD4 D1BC38C6

B : CA3AFDBC 298219E1 5E0BF2B8 8F151F6D 648A1FC9 C1F29FEE CD9F4978 F81C514A
    44EB858B D33B06CF 929CBC10 60672CBB 20B3CCCA EB12DED0 5CCD8C97 240C8D1C

C : D4163C6A 49313E63 0D1ACCBE 7AD73B3E 3312DE9D DA850D91 03785C3A C611B112
    5D1BCAFC 033755D2 3B8EE05E 15251E4E 636A724F F0A8E584 4AABEAAF 122FC0C4

H : 3312DE9D DA850D91 03785C3A C611B112 5D1BCAFC 033755D2 3B8EE05E 15251E4E
    636A724F F0A8E584 4AABEAAF 122FC0C4

H : 9D DE 12 33 91 0D 85 DA 3A 5C 78 03 12 B1 11 C6
    FC CA 1B 5D D2 55 37 03 5E E0 8E 3B 4E 1E 25 15
    4F 72 6A 63 84 E5 A8 F0 AF EA AB 4A C4 C0 2F 12
```

3.2.5 Final State and Output for Shabal-512

```
A : 1FD517B4 18EE0662 002DA3F7 3C864C42 00BDBC17 D3A91349 84B98D58 DB0A255C
    EA84933C 78858700 4E1BD28E 22E17C53

B : D90A51B3 625DBAFC A04AE0F1 15BBA584 3D510ABB 8DF79E61 BBC7D290 442F3794
    A65F93BA 7FCB1E48 60830747 B3922082 93AC0A65 FFB4D084 7900BFF4 8548658A

C : C6168015 0A3F1FC8 688DD952 8E9E2FED 23EF9578 BCE2A7CB A5D80961 E6C9E632
    9701A5A6 F037B89F 20C6C44E DC7931E7 2BB5AB82 B3ADCD32 9CE25056 22305E98

H : C6168015 0A3F1FC8 688DD952 8E9E2FED 23EF9578 BCE2A7CB A5D80961 E6C9E632
    9701A5A6 F037B89F 20C6C44E DC7931E7 2BB5AB82 B3ADCD32 9CE25056 22305E98

H : 15 80 16 C6 C8 1F 3F 0A 52 D9 8D 68 ED 2F 9E 8E
    78 95 EF 23 CB A7 E2 BC 61 09 D8 A5 32 E6 C9 E6
    A6 A5 01 97 9F B8 37 F0 4E C4 C6 20 E7 31 79 DC
    82 AB B5 2B 32 CD AD B3 56 50 E2 9C 98 5E 30 22
```

3.3 Final States and Outputs when Hashing Message B

The value of message B, expressed as a list of bytes, is the following:

```
M1 : 61 62 63 64 65 66 67 68 69 6A 6B 6C 6D 6E 6F 70
     71 72 73 74 75 76 77 78 79 7A 2D 30 31 32 33 34
     35 36 37 38 39 2D 41 42 43 44 45 46 47 48 49 4A
     4B 4C 4D 4E 4F 50 51 52 53 54 55 56 57 58 59 5A

M2 : 2D 30 31 32 33 34 35 36 37 38 39 2D 61 62 63 64
     65 66 67 68 69 6A 6B 6C 6D 6E 6F 70 71 72 73 74
     75 76 77 78 79 7A
```

Note that for each output length the hash value of message B is given twice: first as a word list directly extracted from the end of the state buffer C, then as a byte list expressed in accordance with its *little-endian* representation.

3.3.1 Final State and Output for Shabal-192

```
A : F9D98DBE 30B70551 86CB5CAF BDB2F590 AF169E21 BD8AF9BE 9EEA9756 F7D08C3A
    C51970D2 26C8004C 5BFD5D4B 24891C29

B : 34E18578 04C53BCB FC371288 11A6D737 61190916 E719D732 66662512 9D6323C1
    0E02D0B3 F982ED56 77580C07 39804591 D2590E21 514A0457 11667C92 712C2FEB
```

```
C : 593CCDF8 F2E993B0 DD79ADFB A855551E 2B63F3B6 24A62526 E88CEC5E 6FD09762
    D678E2F8 2953038A 79AE0F69 76956D22 B4FDE80A 37058CF5 55561711 157B307D

H : 79AE0F69 76956D22 B4FDE80A 37058CF5 55561711 157B307D

H : 69 0F AE 79 22 6D 95 76 0A E8 FD B4 F5 8C 05 37
    11 17 56 55 7D 30 7B 15
```

3.3.2 Final State and Output for Shabal-224

```
A : 894924F9 B9663D4A 3211E95C E3077A9D 12706153 2CE27DCF CE8EC0DB 90F7B2A7
    0AEA318D D66C462E 90837F7A 506E9AC9

B : 9E9BFB65 66B89207 696D88EA 677D16EF 5A9BE34F D3618C82 FBCC3A81 AA0538CC
    A574BC1A 1593FD54 A1D2FB2A E3B31AC6 A46C5A7F CDE04097 FDB835E0 0A706953

C : E3BD6C0F E3B4A56E E9349EB2 29739374 5522513E B4754483 8D7C035E 9236E8EE
    3A11ED4E 8D2DD6C7 B474342A 1AD1A9F4 433DDB52 CF58F15B 565D4C45 F525711D

H : 8D2DD6C7 B474342A 1AD1A9F4 433DDB52 CF58F15B 565D4C45 F525711D

H : C7 D6 2D 8D 2A 34 74 B4 F4 A9 D1 1A 52 DB 3D 43
    5B F1 58 CF 45 4C 5D 56 1D 71 25 F5
```

3.3.3 Final State and Output for Shabal-256

```
A : 08ABA604 9C4035C7 8B73310B B3795EF0 E6B83DEB 7A57B2AB 31D05460 23D8D113
    7630AEDA DCE8C11C A7146FD5 F5A59553

B : DA91E764 394C58F5 B1A9C163 7CADCDD5 25514A39 B6E44767 F91D226C 29C46011
    6B9D04BB CA590BB3 58D41196 668B13D2 97D7D432 2166BAB2 99CD95C1 24247080

C : C539CC9B CA52634F 214754DE 19A73AD2 AAF2D843 91D84323 7C4EFAFB 54D18CAB
    BF349FB4 304C8651 6CC43C53 DE2B54C2 D06FF9C2 9A535C6F 58AD6EFF 7A32F783

H : BF349FB4 304C8651 6CC43C53 DE2B54C2 D06FF9C2 9A535C6F 58AD6EFF 7A32F783

H : B4 9F 34 BF 51 86 4C 30 53 3C C4 6C C2 54 2B DE
    C2 F9 6F D0 6F 5C 53 9A FF 6E AD 58 83 F7 32 7A
```

3.3.4 Final State and Output for Shabal-384

```
A : E0042A7C 232B50B8 3DD8F7C8 6CEA315D E27E7E4E E86814E8 F6FDAC30 6CC5A5C3
    396FDAA7 70EB195E E7B2616E BE5A25FA

B : D2EA74CA 12F9F6AD 75E6BE06 A16ABFF6 060268AB F16FA81F 83DA0DA6 D37D8E46
    BC83E112 3414D903 AD33A01D 1EB62E29 5A8B2929 24FEB781 136123D5 78B55F43

C : 6D085ED6 71C06B61 6D009973 9317C3EB 0E2C0130 0B46DC3E C22786D7 D24409C3
    9A6689A1 977A2DFA 772FEF13 A474444C CECBF13A 24B4FAC5 F073088C A0EBFB38

H : 0E2C0130 0B46DC3E C22786D7 D24409C3 9A6689A1 977A2DFA 772FEF13 A474444C
    CECBF13A 24B4FAC5 F073088C A0EBFB38

H : 30 01 2C 0E 3E DC 46 0B D7 86 27 C2 C3 09 44 D2
    A1 89 66 9A FA 2D 7A 97 13 EF 2F 77 4C 44 74 A4
    3A F1 CB CE C5 FA B4 24 8C 08 73 F0 38 FB EB A0
```

3.3.5 Final State and Output for Shabal-512

```
A : 706F3E32 22946DE1 15E78C72 2CE64CAC 5E568D8A 9C96B1AC 8F9951F0 B0FAA007
    E3443293 15CCF7A7 0D0736D8 4930715B

B : 27D27CC5 240FAAD1 6AE08A5D DD1A5439 BE5864F8 F0671108 F8881886 85D62586
    9E795F12 E6068F6B 0914D0D5 A9F12D15 BBAB1377 14431DCF D98C1192 4E2A5AC7

C : 7F6F7E67 F00AD712 2F6635B3 5168B559 663E65F3 86337D64 4301DADF A5C84C25
    94213EDB 716F8C06 607B7D59 B4224D98 D9601F7A CBDFA81C B9655D17 CFCE5973

H : 7F6F7E67 F00AD712 2F6635B3 5168B559 663E65F3 86337D64 4301DADF A5C84C25
    94213EDB 716F8C06 607B7D59 B4224D98 D9601F7A CBDFA81C B9655D17 CFCE5973

H : 67 7E 6F 7F 12 D7 0A F0 B3 35 66 2F 59 B5 68 51
    F3 65 3E 66 64 7D 33 86 DF DA 01 43 25 4C C8 A5
    DB 3E 21 94 06 8C 6F 71 59 7D 7B 60 98 4D 22 B4
    7A 1F 60 D9 1C A8 DF CB 17 5D 65 B9 73 59 CE CF
```

3.4 Intermediate States for Messages A and B

A complete follow-up of the execution of each Shabal-ℓ_h on both messages A and B is given in Appendix B.

Chapter 4

Design Rationale

Contents

This chapter explains how we came up (ended up) with the hash construct described in previous chapters. Inherent to all ideas underlying a cryptographic construction are arguments of different nature: simplicity; performance; attacks; proofs; intuitions. Certain elements of Shabal relate to intuitive considerations in the sense that although we were not able to conceive attacks on other design choices, we felt that the computational components we eventually adopted in Shabal are definitely a better option than alternate approaches. Other aspects of the hash function, such as the size and nature of the different parts of the internal state, follow a clear methodology with a number of metrics that we chose to optimize. Several reformulations have been adopted in order to simplify the description of Shabal and allow easy-to-code and fast implementations. Finally, from the very beginning we decided to rely on the power of security proofs (see Chapter 5) to properly validate the mode of operation and select various size parameters.

Many design components interact in obvious or more subtle ways and their validation was not carried out independently, even though for the sake of readability and comprehension, we chose to discuss them separately in this document. Finally we point out that the structure of this document is not directly related to the importance of the concepts and ideas we developed while designing Shabal, but rather follow a desire for a clear exposition of our conclusions.

4.1 A Quest for Provably Secure Efficiency

4.1.1 A Short Story about the Mode of Operation of Shabal

It may not be obvious at first sight to understand the design rationale behind the operating mode of Section 2.2; this mode of operation is the result of a long series of works which we sum up in this section.

Before opting for the final mode previously described, we were planning to rely on the mode of operation shown on Figure 4.1 which we retrospectively call Old Mode 1. Old Mode 1 features a double message insertion, and a large keyed permutation. The idea of relying on multiple message insertion is not new (it exists for example in RadioGatún): a direct effect of double insertion is to increase the diffusion of input differences[1], which hardens the search for differential paths. Old Mode 1 also exploits the idea of an *accumulator* (X on Figure 4.1) which collects input bits from all message blocks and prevents internal collisions from happening after one round of message insertion.

Interestingly, this mode turned out to be indifferentiable from a random oracle, a must-have property. Also, provable resistance to collision, preimage and second preimage attacks can be shown in the ideal cipher model. Besides, the core object of Old Mode 1, a keyed permutation, is more efficiently instantiated than any unkeyed permutation defined on the same input domain since the field holding the parameter (key) does not require to be written and given as output. One may thus expect to realize better throughputs than with a basic sponge construction such as [7]. On the other hand, the domain of the keyed permutation still has to be very large to meet satisfactory security bounds and therefore Old Mode 1 leaves the impression that there is room for more advanced improvements.

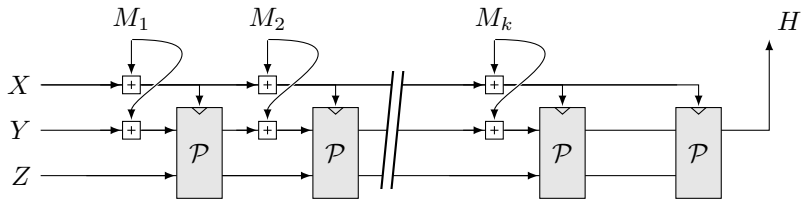

Figure 4.1: Mode of Operation Old Mode 1

We later found how to refine Old Mode 1 into a more efficient mode of operation Old Mode 2 depicted on Figure 4.2. Old Mode 2 makes use of a smaller permutation than Old Mode 1 since a fraction of the input space has been converted into a part of the key space, with now 2 parameters instead of just one. Relying on a keyed permutation with optimally small input space is clearly beneficial to both security and performance, since it is much easier to construct efficient keyed permutations on smaller domains. Like Old Mode 1, Old Mode 2 turned out to be indifferentiable. But in addition to that, interestingly, this mode happens to support provable collision, preimage and second preimage resistance, although the simulators are much more intricate to conceive than in the case of Old Mode 1.

[1] with double insertion, message modifications that one makes to correct differences in one round are also to be corrected in next round and so on.

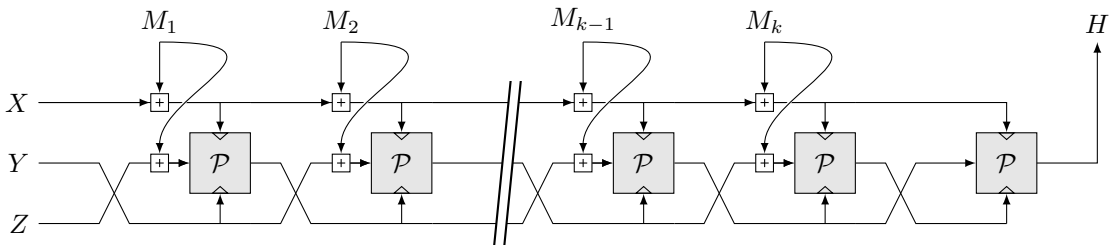

Figure 4.2: Mode of Operation Old Mode 2

It finally became clear to us that the variable X played no significant role in the security of Old Mode 2. Said differently, we found that Old Mode 2 could be reformulated to yield an equivalent mode which does not require to store and update X. At any point in time indeed, an attacker is always able to set X to a prescribed value m by inserting $M = m - X$. Old Mode 2 then replaces variable Y with $Y - X + m$. As a consequence, the mode of operation behaves exactly as if variables (X, Y) were replaced with a single variable $Y - X$. However, if one wants to continue the sequence of atomic blocks *i.e.,* in order to construct the final value for $Y - X$, it is requested to subtract the value X (which is m) from the value of Y (which is equal to the value of Z before variables are swapped). We then studied the mode described in Section 2.2 as an evolution of Old Mode 2, where $Y - X$ was renamed B, m was renamed M and Z was renamed C; A corresponds to a buffer that does not appear on the figures describing Old Mode 1 and 2.

Further options such as the inclusion of a 64-bit counter W or of a series of final rounds emerged later with the search for lightweight tricks that would strengthened security: the presence of a counter improves resistance against all forms of attacks as shown in Section 5.6; the implementation of final rounds arose from various discussions on the expected and actual degree (in the sense of Boolean functions) of the output bits, see Section 4.4.

4.1.2 Security Proofs: An Intuition as to Why Shabal is Secure

The notion of capacity was recently introduced as a security metric for truncated operating modes and sponge constructions. A mode of operation has capacity c if it is indifferentiable from a random oracle up to 2^c evaluations of the inner primitive. In particular, this implies that generating internal collisions must cost at least 2^c evaluations of the primitive, since coming up with an internal collision is enough to distinguish the hash construct from a random oracle.

The operating mode of Shabal is shown to have a capacity of exactly $(\ell_a + \ell_m)/2$ bits which for $\ell_a = 32 \cdot r = 32 \cdot 12 = 384$ gives a concrete capacity of 448 bits. Therefore internal collisions are much more unlikely than standard collisions on the hash output even when $\ell_h = 512$ since 256 is significantly smaller than 448.

In a nutshell, the high capacity of Shabal comes from the fact that large parts of the internal state cannot be controlled by the adversary, either because they contain the output of the inner keyed permutation $\mathcal{P}$ or because they are uncontrollably influenced by the output of $\mathcal{P}$. This phenomenon also plays a major role in the other security notions such as preimage and second preimage resistance, as exemplified by Theorems 4 and 5.

4.2 Designing the Keyed Permutation $\mathcal{P}$

In this section, we explain how the keyed permutation $\mathcal{P}$ was designed. Let us start by saying that many approaches would be equally sound to instantiate $\mathcal{P}$. In this respect, we clearly made very specific choices and several features could possibly have led to other interesting constructions without necessarily decreasing security. Each time we had to make a choice between several

equivalent approaches, we let our decision be dictated by our quest for simplicity and performance.

4.2.1 An NLFSR-based Structure

The keyed permutation $(A, B) \mapsto \mathcal{P}_{M,C}(A, B)$ is basically made upon a nonlinear feedback shift register (NLFSR). Both variables A and B are actually updated as 16-word NLFSRs whose nonlinear feedback functions depend on parameters M and C. However, the specificity of our design resides in that these NLFSRs are not independent from each other (we refer the reader to Figure 2.4 for a view on the two NLFSRs). The two feedback functions interact with one another: B influences the feedback of register A and conversely.

4.2.2 A Permutation

In order to guarantee that message rounds do not cause the internal state to lose entropy, we wanted $\mathcal{P}$ to be a permutation for any fixed choice of parameters M and C. This property is ensured by the NLFSR-based structure used for both registers A and B. Function $\mathcal{P}$ can actually be decomposed into an initial $\lll 17$ rotation applied to input B:

$$B_i \mapsto B_i \lll 17$$

and $16p$ steps of

$$\begin{cases} (A, B) & \mapsto & \pi_{M,C}(A, B) \\ (M_0, \ldots, M_{15}) & \mapsto & (M_1, \ldots, M_{15}, M_0) \\ (C_0, \ldots, C_{15}) & \mapsto & (C_{15}, C_0, , \ldots, C_{14}) \end{cases}$$

for the following elementary step function π:

$$\begin{array}{rccc} \pi_{M,C}: & \{0,1\}^{32r} \times \{0,1\}^{512} & \rightarrow & \{0,1\}^{32r} \times \{0,1\}^{512} \\ & (A_0, \ldots, A_{r-1}, B_0, \ldots, B_{15}) & \mapsto & (A_1, \ldots, A_r, B_1, \ldots, B_{16}) \end{array}$$

where (remind that (o_1, o_2, o_3) stands for a fixed tuple of offsets)

$$\begin{array}{rcl} A_r & = & \mathcal{U}(A_0 \oplus \mathcal{V}(A_{r-1} \lll 15) \oplus C_8) \oplus B_{o_1} \oplus (B_{o_2} \wedge \overline{B_{o_3}}) \oplus M_0 \\ B_{16} & = & (B_0 \lll 1) \oplus \overline{A_r}. \end{array}$$

$\mathcal{P}$ ends with a final transformation of the internal state

$$(A, B, C) \mapsto (A + \sigma(C), B, C)$$

where $\sigma(C)$ is an r-word vector derived from the 16 words of C.

Thus, $\mathcal{P}$ is a permutation if and only if the elementary step function π is a permutation. From the previous description, it appears that, for given values of M and C, $\pi_{M,C}$ can be inverted by using:

$$\begin{array}{rcl} B_0 & = & (B_{16} \oplus \overline{A_r}) \ggg 1 \\ A_0 & = & \mathcal{V}(A_{r-1} \lll 15) \oplus C_8 \oplus \mathcal{U}^{-1}(A_r \oplus B_{o_1} \oplus (B_{o_2} \wedge \overline{B_{o_3}}) \oplus M_0). \end{array}$$

Since any step of $\mathcal{P}$ can be inverted by the previous formula, $\mathcal{P}$ is a permutation.

4.2.3 Register A

The role of register A is to improve the effect of diffusion: if a difference occurs in one word of A, it has to be corrected in the following words (which requires to include a difference in the corresponding message word), otherwise the difference will spread on and lead to an avalanche effect.

The feedback function in register A is defined by:

$$A_{t+r} = \mathcal{U}(A_t \oplus \mathcal{V}(A_{t+r-1} \lll 15) \oplus C_{8-t}) \oplus \mathcal{G}(B_{o_1+t}, B_{o_2+t}, B_{o_3+t}) \oplus M_t, \quad \forall t \geq 0$$

for some function $\mathcal{G}$ whose characteristics are discussed in Section 4.2.5. We first provide more detail on the choices of all elementary operations performed while computing the feedback word.

Introducing A.

For performance reasons, only two taps of register A are involved in the feedback function: the use of A_t is required to ensure that $\mathcal{P}$ is a permutation; A_{t+r-1} has been chosen for the second involved word in order to make that any difference introduced in A by the feedback function immediately impacts the next step. These two words must affect the feedback in a nonlinear manner. Thus, some nonlinear functions, $\mathcal{U}$ and $\mathcal{V}$, are used, whose choice is detailed below. Finally, the feedback function involves a rotated version of A_{t+r-1}. This rotation aims at moving the least-significant bits of the words of A to another position since the least-significant coordinate functions of $\mathcal{U}$ and $\mathcal{V}$ are both linear.

Introducing C.

The words of C are introduced by a XOR, as normally there is no real control over C. But, the words of C are not introduced in the same order as the words of A. The reason is that, at Round t, register C corresponds to the B output of the previous round. Then, at the beginning of Round t, register A has a linear dependency with the last r words of C. Most notably, if an attacker succeeds in finding a differential trail such that the last r words of C and the corresponding words of A have the same differences at the end of Round $(t-1)$, these differences might cancel at Round t if the words of A and C are taken in the same order. That is why, at each round, $C_{8-i \bmod 16}$ is introduced at Step i instead of $C_{i \bmod 16}$.

Introducing M.

M is introduced directly by a XOR, so that any difference in M will affect A. The initial $\lll 17$ rotation applied to B before the first step guarantees that similar difference patterns in B and M do not cancel (this might be possible without the rotation by choosing appropriate linear approximations of $\mathcal{G}$).

Using $\mathcal{U}$ and $\mathcal{V}$ as S-Boxes.

The feedback function of register A uses two simple functions $\mathcal{U}(x) = 3 \times x \bmod 2^{32}$ and $\mathcal{V}(x) = 5 \times x \bmod 2^{32}$ whose goal is to increase both the degree and the nonlinearity. Their presence hardens the search for simple and high-probability differentials of the type "modify-then-correct".

Using two nonlinear functions instead of just one allows to guarantee that inserting two different message blocks will cause at least one difference between the inputs of one of the executions of $\mathcal{U}$ or $\mathcal{V}$ after two rounds. This property is proved in Section 11.3.2 (see Theorem 7) and cannot be derived when a single function, *e.g.,* $\mathcal{U}$, is used.

$\mathcal{U}$ and $\mathcal{V}$ have been chosen to be as simple as possible: they can easily be hard-coded under the form of a bit shift followed by addition, or equivalently as two additions for $\mathcal{U}$ and three additions for $\mathcal{V}$ if the multiplication by a small constant is not available on the hardware platform. The choice of nonlinear functions which can be implemented in software with simple CPU operations avoids the use of look-up tables which would have increased the code size, see Section 12.3.3.

Moreover, the absolute constants 3 and 5 used in both multiplications are invertible modulo 2^{32}, implying that $\mathcal{U}$ and $\mathcal{V}$ are permutations, and so no entropy on x is lost. Another advantage of these two functions is that they cannot transform a symmetric difference, *i.e.,* the all-0 or the all-1 word, into a symmetric difference as proven in Proposition 1 of Section 11.3.2.

4.2.4 Register B

The nonlinear feedback function in register B is defined by:

$$B_{t+16} = (B_t \lll 1) \oplus \overline{A_{t+r}}, \quad \forall t \geq 0.$$

Introducing A.

Register A impacts the feedback of register B by the XOR with A_{t+r}. There is no need to use a more complicated operation since A_{t+r} is a nonlinear function of A and B. The use of the lastly computed word of A for influencing B is very natural since it corresponds to the word in A which has the highest polynomial degree in the message bits.

Introducing B.

The insertion of B_t with the help of a XOR is required to ensure that $\mathcal{P}$ is a permutation. The rotation $B_t \lll 1$ aims at avoiding that the differences appearing in B after one loop of $\mathcal{P}$, *i.e.,* after 16 elementary steps, correspond to the initial differences for differential trails which do not generate any difference in A. Otherwise, the number of conditions required for having a collision on register A after all of the three loops of $\mathcal{P}$ would decrease (a detailed analysis on this is provided in Section 11.3.3).

The Addition of Constant `0xFFFFFFFF`.

Adding the constant `0xFFFFFFFF` is intended to avoid that the all-zero internal state is a trivial fixed point. See also Section 4.6.

4.2.5 Function $\mathcal{G}$

For implementation reasons, we wished $\mathcal{G}$ to use a small number of taps of register B. Three is actually the lowest number of taps as shown below. The offsets (o_1, o_2, o_3) were chosen by an empirical search over all possible triplets, with the goal to maximize resistance against certain differential paths of small Hamming weight. See Section 4.3.

One of the main conditions in the choice of $\mathcal{G}$ is that it must involve simple operations available on a 32-bit processor. For this reason, we chose a bitwise function $\mathcal{G}$, *i.e.,* a function such that the i-th output bit depends on the i-th input bits only, and such that all 32 coordinate functions correspond to the same Boolean function g.

Then, this 3-variable Boolean must satisfy the following conditions:

- g must be balanced, otherwise B_t and B_{t+16} are correlated;
- g must be nonlinear;

These conditions imply that g has degree 2 since the degree of an n-variable balanced function is at most $(n-1)$. It is worth noticing that 3 is the lowest number of variables we could choose for satisfying the previous conditions.

All such functions g are equivalent, up to an affine permutation of the input and up to the addition of an affine function. For all of them, there exist exactly 4 biased approximation by a function of degree 1 and each of them holds with probability 3/4. Here are some examples.

- $g(x_1, x_2, x_3) = x_1 + x_2x_3$. The biased affine approximations of g are x_1, $x_1 + x_2$, $x_1 + x_3$ and $x_1 + x_2 + x_3 + 1$.
- $g(x_1, x_2, x_3) = x_1 + x_2 + x_2x_3$. The biased affine approximations of g are x_1, $x_1 + x_2$, $x_1 + x_3 + 1$ and $x_1 + x_2 + x_3$.
- $g(x_1, x_2, x_3) = x_1x_2 + x_1x_3 + x_2$. The biased affine approximations of g are x_2, $x_1 + x_2$, x_3 and $x_1 + x_3 + 1$.
- $g(x_1, x_2, x_3) = x_1 + x_3 + x_1x_2 + x_1x_3 + x_2x_3$, *i.e.,* The biased affine approximations of g are x_1, $x_2 + 1$, x_3 and $x_1 + x_2 + x_3$.

It appears that the fact that a given input variable, *e.g.,* x_1, is involved in all approximations of degree 1 makes the search for differential trails much harder. An unsuitable value of B_t has to be handled by the attacker, since she cannot use an approximation of $\mathcal{G}$ which does not involve x_1. Any balanced 3-variable Boolean function whose all biased affine approximations involve x_1 is linear in x_1:

$$g(x_1, x_2, x_3) = x_1 + q(x_2, x_3).$$

We have chosen for the quadratic function q a function which is not symmetric, since it seems unsuitable than, when $o_2 > o_3$, $B_{t+o_2} = 0$ implies that $q(B_{t+o_2}, B_{t+o_3}) = 0$ and $q(B_{t+2o_2-o_3}, B_{t+o_2}) = 0$.

4.2.6 The Final Transformation

The final transformation

$$(A, B, C) \mapsto (A + \sigma(C), B, C)$$

applied to the internal state after the $16p$ steps of π aims at strengthening the inverse permutation $\mathcal{P}^{-1}$. Otherwise, $\mathcal{P}^{-1}$ would consist of $16p$ steps of $\pi^{-1}_{M,C}$, but part B of the output of $\pi^{-1}_{M,C}$ is independent from both parameters M and C. Most notably, it follows that, for an r-step $\mathcal{P}$, part B of the input of $\mathcal{P}$ is completely determined by the knowledge of outputs A and B. This unsuitable property may be exploited in a (second)-preimage attack for $p = 1$ and $p = 2$, see Section 11.6 for details.

In order to eliminate this weakness, the final transformation makes part A of the output dependent on C. When computing backwards in a (second)-preimage attack, the C-input of $\mathcal{P}^{-1}$ at Round i actually depends on M_i since the message block has been subtracted before applying $\mathcal{P}^{-1}$. Then, the final transformation makes the A input of the first r elementary step functions $\pi^{-1}_{M_i,C}$ depend on both M_i and C.

To find the vector $\sigma(C)$ involved in this transformation we have searched for those which lead to the highest dependence between the words of the B-part of the output of $\mathcal{P}^{-1}$ and the words of M for $p = 1$, $p = 2$ and $p = 3$. We have restricted our search to the $\sigma(C)$ which can be computed by a simple loop of the form: for i from 0 to $s - 1$,

$$\sigma(C)[i \bmod r] \leftarrow \sigma(C)[i \bmod r] + C[(-1)^e i + \text{offset}].$$

We have then performed an exhaustive search for the size of the loop s, the direction e and the offset for the recommended choice of r, *i.e.,* $r = 12$. Another condition was that each word of A in the output of $\mathcal{P}$ must depend on a different set of the words of C. The vector $\sigma(C)$ that we have chosen can be computed by a loop of size 36, with $e = 0$ and offset $= 3$, *i.e.,* for i from 0 to 11,

$$\sigma(C)[i] = C[i + 3] + C[i + 3 + r] + C[i + 3 + 2r].$$

It is worth noticing that each $A[i]$ then depends on three words of C that are all different for the different i. If a different value for r is to be used, s, e and the offset must be recomputed.

4.3 How We Chose (o_1, o_2, o_3)

4.3.1 The Basic Idea

As explained in the previous section, the update of register A in permutation $\mathcal{P}$ involves some words of B selected by an offset triplet (o_1, o_2, o_3). We now provide details on the method we used to elect such a triplet of offsets.

To determine the best offset triplet (o_1, o_2, o_3), we have looked for triplets which ensure the best diffusion of differences inside the internal state of Shabal. To this purpose, we define a specific criterion. For the sake of readability, we consider in this section the function $\mathcal{R}$ described in Section 2.2 but without specifying the counter w. More precisely, $\mathcal{R}$ is the message round function which takes as input the $(32 + r)$-word internal state and the current message block and outputs

the internal state after the first round is completed (hence we have $\mathcal{R}(M, S_{old}) = S_{new}$, where S is the internal state).

Our main criterion was the following: given a vector δ with small weight, the minimum of the weight of

$$\mathcal{R}(M \oplus \delta, S) \oplus \mathcal{R}(M, S)$$

over all the possible values of M should be maximal. Indeed, the idea behind this criterion is that we want to maximize the number of differences caused by a few input differences (in one round), in order to obtain as fast as possible an uncontrollably large set of differences.

We have scalable security parameters, namely r and p. To study the diffusion of the round function, we need to specify these parameters. Since one of our goals is to design the function which ensures the best diffusion, we are thus searching for the triplets (o_1, o_2, o_3) which ensure the best diffusion even if p and r are the worst parameters for the diffusion. Proceeding this way, the chosen offsets should be a good choice whatever the choice of parameters.

It is clear that $p = 1$ is a bad choice for diffusion and even for security. The role of r in the function is quite different: the bigger r, the larger the capacity and the worst the diffusion. We thus decided to choose $r = 16$ to proceed to the search.

In a phase of analysis, to satisfy this criterion, we have first linearized the $\mathcal{R}$ function. $\mathcal{R}$ thus becomes an affine function: we have

$$\mathcal{R}(M, S) = \psi(M, S) \oplus \alpha$$

where ψ is a linear function and α a constant vector. Thus, we want to maximize the weight of

$$\mathcal{R}(M \oplus \delta, S) \oplus \mathcal{R}(M, S) = \psi(\delta, 0) \ .$$

This analysis gives arguments on the choice of the triplets for the linearized $\mathcal{R}$ function but not for the real function. This first phase enables us to determine a family of good choices of triplets. Once this set is defined, we analyze these triplets over the real round function.

4.3.2 Linearization

There exist different nonlinear computations inside the round function. Firstly, the message insertion is made thanks to an integer addition. In the linearized form, this operation is switched into a bitwise XOR. Secondly, the functions $\mathcal{U}$ and $\mathcal{V}$ which are used to compute A are also nonlinear. We have $\mathcal{U} : x \mapsto 3 \times x \mod 2^{32}$ and $\mathcal{V} : x \mapsto 5 \times x \mod 2^{32}$; we linearized them by replacing $3 \times x$ with $x \oplus (x \ll 1)$ and $5 \times x$ with $x \oplus (x \ll 2)$ (see also Section 6.4). These are clearly the best linear approximations of $\mathcal{U}$ and $\mathcal{V}$. Finally, we need to linearize the computation of $B[i + o_1 \bmod 16] \oplus B[i + o_2 \bmod 16] \wedge \overline{B[i + o_3 \bmod 16]}$. As explained in Section 4.2.5, this computation can be linearized in four different ways (in the following, we denote by $\mathcal{R}_1$, $\mathcal{R}_2$ $\mathcal{R}_3$ and $\mathcal{R}_4$ the linearized round functions associated respectively to the first, the second, the third and the fourth linearized computations):

1. $B[i + o_1 \bmod 16]$
2. $B[i + o_1 \bmod 16] \oplus B[i + o_2 \bmod 16]$
3. $B[i + o_1 \bmod 16] \oplus B[i + o_3 \bmod 16]$
4. $B[i + o_1 \bmod 16] \oplus B[i + o_2 \bmod 16] \oplus B[i + o_3 \bmod 16]$

These four different choices for the linearized round function enable us to determine conditions over the possible triplets. The first one gives arguments for the choices of o_1, and respectively the second one for o_2 and the third one for o_3. The second and the third linearized round functions are exactly the same and the best o_2 is thus the best o_3 as well. This is logical since $B[i + o_2 \bmod 16]$ and $B[i + o_3 \bmod 16]$ play symmetric roles. We thus use the second and the fourth linearized round functions to choose o_2 and o_3.

4.3.3 Search Methods

We have studied the Hamming weight of the output of $\psi_i(\delta) = \mathcal{R}_i(M \oplus \delta, S) \oplus \mathcal{R}_i(M, S),\ 1 \leq i \leq 4$ for δ of low Hamming weight, using two methods. Using brute force search, it is possible to compute $\psi_i(\delta)$ for all δ of weight three. It is possible to go a bit further using some algorithm dedicated to the search for low-weight vectors in a code, as we explain hereafter. It is worth noticing that the main classical algorithms for finding low Hamming weight words in a linear code are dedicated to binary codes while this is not the case here. Then, we used the following techniques: the values of $\psi_i(\delta)$ for all δ of weight w are computed and stored in a list. This list can then be sorted following a lexicographic order. Thanks to this sort, two consecutive elements in the list, namely $\psi_i(\delta_1)$ and $\psi_i(\delta_2)$ are close to each other for the Hamming distance with a higher probability than two random elements in the list. As ψ_i is a linear function, we have

$$\psi_i(\delta_1) \oplus \psi_i(\delta_2) = \psi_i(\delta_1 \oplus \delta_2) \tag{4.1}$$

Thus we have tested this way the weights of ψ_i for *some* elements of weight $2w$. Hence this algorithm is probabilistic. To improve the quality of the result, it is possible to sort the list again using a different lexicographic order and to compute the XOR of consecutive elements lying in the resulting list again. With the recommended settings of Shabal, this algorithm can test many vectors of weight 4.

4.3.4 Results on the Linearized Function

Using the two search methods, we have first studied the function $\mathcal{R}_1$. We found that this function has a minimal output weight of 16 for $o_1 = 12, 13$. The values $9, 10$ and 14 give a weight bigger than 13. Other choices of o_1 leads to a minimum weight less than 10.

We have also studied the functions $\mathcal{R}_2$, $\mathcal{R}_3$ and $\mathcal{R}_4$ to determine a good family of offsets. In fact the study of $\mathcal{R}_2$ and $\mathcal{R}_3$ leads to the same results as mentioned above. We decided to study triplets of offsets such that the minimum weight obtained with $\mathcal{R}_2$ was bigger than 50 given o_1 amongst $9, 10, 12, 13, 14$. This value ensures that it remains sufficiently many offsets to test. As a result, we have pairs of offsets which could play the role of (o_1, o_2) or (o_1, o_3). The results lead to $(12, 2)$, $(12, 5)$, $(12, 7)$, $(12, 9)$, $(13, 3)$, $(13, 6)$, $(13, 7)$, $(14, 4)$, $(14, 9)$. Amongst these pairs, we look at the ones that give the best results for $\mathcal{R}_3$. We select those with a final weight higher than 100. This leads to the triplets: $(14, 9, 3)$, $(14, 11, 4)$, $(13, 8, 6)$, $(13, 9, 6)$, $(13, 12, 6)$,$(13, 8, 7)$, $(14, 9, 8)$.

We remove on purpose the triplet $(13, 12, 6)$ because two consecutive offsets always lead to a bad result with $\mathcal{R}_2$ and the triplets $(14, 9, 8)$, $(13, 8, 7)$, $(13, 8, 6)$ because 8 is the worst possible offset for $\mathcal{R}_1$. Furthermore, 8 is special since it is the half of the number of words in the state buffer B.

The remaining possible triplets are thus $(14, 9, 3)$, $(14, 11, 4)$ and $(13, 9, 6)$.

4.3.5 Final Results on the Real Function for $p = 1$ and $r = 12$

After this analysis, we have studied these chosen triplets on the real (*i.e.,* non-linearized) function. Given a low-weight difference between two messages, the two associated internal states after one round should have a difference with a sufficiently high weight to ensure a good difference diffusion in the internal state. To check whether this property is true for a given triplet, we have computed the minimal weight of $\mathcal{R}(M) \oplus \mathcal{R}(M \oplus \delta)$ for all δ of weight at most three. As this computation is not possible for every message M, we computed the minimum weight over the δ for different random messages. To accelerate the computation, instead of choosing random messages of random length, we chose random messages of constant length of 16 words and we randomly chose the IV of the internal state. This random choice of IV simulates the insertion of a random prefix message before the insertion of differences. We have made this analysis for $r = 12$, which is the recommended r. Nevertheless, we study the round function with p sets to 1 and 2. With $p = 3$, the output of the permutation seems to be random and whatever the small input difference is, the difference of the outputs looks like a random string. This property does not help to fix the best triplet of offsets.

We have repeated these computations about 2^{45} times on all the selected triplets. We have finally chosen the triplet $(13, 9, 6)$ which is from our computations the best one.

4.4 Shabal and Degree

$\mathcal{P}$ is a keyed permutation which takes as input an internal state $S = (A, B, C)$ and a message block M and which outputs A' and B'. Each bit of the output can be expressed as a Boolean function of the bits of the inputs. Nevertheless, it is impossible to write formally these functions since the number of input bits equals 1920.

It has been shown that some properties of the algebraic normal form (ANF) of a Boolean function can be exploited to distinguish it from a random Boolean function (see Section 11.9 for instance). These attacks mainly rely on two properties:

- the sparsity of the ANF of the Boolean function;
- the degree of the Boolean function.

It can be easily seen that the ANF of any output of $\mathcal{P}$ has no reason to be sparse, because of the use of $\mathcal{U}, \mathcal{V}$, of the different rotations and of the function $\phi : (x, y) \mapsto \overline{x} \wedge y$. The second criterion needs to be further investigated. Indeed, $\mathcal{U}$ and $\mathcal{V}$ have special algebraic expressions since they coincide many times with their linear approximations. Furthermore, the choice for function ϕ of a bitwise operation may not ensure the growth of the degrees of the Boolean functions.

However, estimating the degree requires to simplify some operations. We have thus investigated the degrees of the message round functions of two different weakened versions of Shabal:

1. the weakened 1-bit version, named Weakinson-1bit (see Section 6.1),
2. the weakened 32-bit version where $\mathcal{U}$ and $\mathcal{V}$ are linear functions, where all additions and subtractions are replaced by XORs and where the final update loop on A is removed. This variant corresponds to Weakinson-$\oplus$-LinearUV-NoFinalUpdateA (see Section 6.3).

Moreover, for both variants, we do not take into account the design of Shabal, in the sense that B is assumed to be independent from M (while M is added to B in the construction).

The degrees of the outputs of the round function in both variants are obviously lower than the degrees obtained for the full Shabal, in particular because in both weakened versions, $\mathcal{U}$, $\mathcal{V}$ and all additions are linear, implying that the only source of nonlinearity is the quadratic function $\phi : (x, y) \mapsto x \wedge \overline{y}$.

4.4.1 Degree of Weakinson-1bit

The weakened version of Shabal with 1-bit words is of great interest since the Boolean function output by $\mathcal{P}$ can be formally expressed with computer algebra software such as PARI/GP (see http://pari.math.u-bordeaux.fr/).

We have decided to study the degrees of the Boolean functions in the different parameters independently from each other. This leads to Table 4.1 which gives the degrees in the different input variables of the outputs of $\mathcal{P}$, for different values of parameter p. Since the output degrees in A, C and M are the same, they are all given in the same row in Table 4.1.

Inputs	$p = 1$, $\deg(A')$	$p = 1$, $\deg(B')$	$p = 2$, $\deg(A')$	$p = 2$, $\deg(B')$	$p = 3$, $\deg(A')$	$p = 3$, $\deg(B')$
M, A, C	From 1 to 2	From 1 to 2	From 4 to 8	From 2 to 8	From 8 to 14	From 10 to 14
B	From 2 to 5	From 2 to 5	From 7 to 12	From 5 to 12	From 14 to 16	From 12 to 16

Table 4.1: Degrees of the outputs of the message round function in Weakinson-1bit

4.4.2 Degree of Weakinson-⊕-LinearUV-NoFinalUpdateA

For this weakened variant, we compute the degree by modeling the actions of all the different operations on the degree. For instance, to compute the degree of the Boolean function which takes M as input and outputs $P_{M,0}(0,0) = (A', B')$, we replace M by the 512-bit all-1 vector and A, B and C by the all-0 vector. This models the fact that each bit of M has a degree equal to 1 while each bit of A, B and C has a degree equal to 0.

In our model, each operation which appears in the computation of $\mathcal{P}$ updates the vector (A, B). The model firstly assumes that $\deg(\phi(B[i], B[j])) = \deg(B[i]) + \deg(B[j])$. Secondly, it is assumed that $\deg(A[i] \oplus A[j]) = \max(A[i], A[j])$. These are optimistic assumptions on the degree, since we do not take into account the reduction by the polynomials $x_i^2 + x_i$ which are 0 over $\mathbf{F}_2$ for any input variable x_i. The assumption that ϕ ensures the growth of the degree is strongly related to the fact that we use word rotations. Since ϕ is a bitwise operation, if no rotation is used, the Boolean function would only depend on the input bits at the same position in the word, leading to a very weak function. With these rotations, the two polynomials corresponding to the inputs of ϕ should have different monomials. For this reason, it seems realistic to assume that ϕ ensures the growth of the degree.

These different hypotheses lead to Table 4.2. It can be seen that, even for $p = 3$, the obtained degree is not maximal (*i.e.,* less than 512). But, the actions of $\mathcal{U}$, $\mathcal{V}$ and of the different additions modulo 2^{32} are expected to ensure a much higher degree. Moreover, it has been chosen for the final rounds to use 3 consecutive iterations of permutation $\mathcal{P}$. We thus think that the resulting transformation is sufficient to resist all attacks that attempt to distinguish $\mathcal{P}$ from a random keyed permutation.

Inputs	$p = 1$, $\deg(A')$	$p = 1$, $\deg(B')$	$p = 2$, $\deg(A')$	$p = 2$, $\deg(B')$	$p = 3$, $\deg(A')$	$p = 3$, $\deg(B')$
M, A, C	From 1 to 2	From 1 to 2	From 4 to 9	From 2 to 9	From 12 to 32	From 9 to 32
B	From 2 to 5	From 2 to 5	From 8 to 21	From 5 to 21	From 28 to 70	From 21 to 70

Table 4.2: Degrees of the outputs of the message round function in Weakinson-⊕-LinearUV-NoFinalUpdateA

It is worth noticing that both previous approaches aim at giving arguments on the choice of parameter p and of the number of final rounds. Actually, computing the degrees of the underlying Boolean functions in Shabal is not feasible.

4.5 Initial Values

For some initial values before the first message introduction, the scheme may be weaker than for some random-looking initial value. Indeed, some components of A update, *e.g.,* $B[i + o_2 + \text{mod}32] \wedge \overline{B[i + o_3 \bmod 32]}$, preserve symmetry (in the sense that they transform the all-0 and the all-1 words into all-0 or all-1 words); furthermore, $\mathcal{U}$ and $\mathcal{V}$ functions have 0 as a fixed point. Therefore, we have tried to start the message rounds with an almost random initial state, called IV_{ℓ_h}.

To set this IV_{ℓ_h}, we may have stated that one would choose some "natural" values, such as the expression of π in hexadecimal. However, while this kind of setting is widely accepted to be trapdoor-free, it imposes to store in implementations the full values of IV_{ℓ_h}, which for our Shabal implementation is $(1024+32r)$-bit long per IV (that is (A, B, C)). Constrained environments such as low-cost smart cards, RFID or hardware would clearly suffer from this choice.

Instead, we have decided to exhibit some IV_{ℓ_h} that come naturally from our definition of Shabal. More precisely, we have decided that the IV_{ℓ_h} would be obtained after performing some preliminary steps, with initial full-0 state value and message blocks $(M_{-1}, M_0) = \{\ell_h, \ell_h + 1, ..., \ell_h + 31\}$, where ℓ_h is the wanted output length and where the prefix is composed of 32 words of 32 bits. The length of the prefix (that is 32 words) has been chosen so that it is sufficient to make the buffers almost random-looking. Thanks to this prefix, the state is set to a non-pathological value. As one can

see, there are as many IV_{ℓ_h} as the number of possible output lengths, in order to let (if possible) the hash values and security of the different-length Shabal independent.

This kind of IV setting has two principal advantages. First, one may reasonably argue that this is trapdoor-free. Indeed, the IV_{ℓ_h} choice can be verified easily to be honestly generated. Second, this setting is very efficient regarding implementation: on low-power devices, one can simply prefix the message with (M_{-1}, M_0) and start Shabal with all-0 internal state; on the contrary, on non-restricted machines or for faster implementations, one should tabulate the value IV_{ℓ_h} and directly start with message blocks.

Clearly, if the number of IV_{ℓ_h} is too large in practice (even if reconstructible, certain implementations will prefer to tabulate the values), we may consider to limit them to fewer values.

4.6 The Effect of Counter w

We have proposed to use a simple counter w in our generic construction and in Shabal. In fact, the counter is not critical for the indifferentiability proof of Section 5.3. However, w provides a simple way to avoid *fixed points* (see also the addition of the all-one constant when updating $A[i + 16j \bmod r]$ in Section 4.2.4 and Section 11.4).

Furthermore, counter w makes second preimage attacks harder as shown in the proof of Theorem 5. This is due to the fact that an attacker which attempts to connect somewhere on the challenge path also has to collide on the counter. We refer the reader to Chapter 5 for more details.

4.7 Output of the Hash Function

The output of the Shabal hash function is directly given by the construction depicted in Section 2.2, which is proven to be indifferentiable from a random oracle in Section 5.3. We have chosen to output the $\ell_h/32$ last words of B directly after the last keyed permutation is carried out, which are the words with highest degree.

4.8 Nonlinearity

Let us summarize the sources of nonlinearity within Shabal:

(i) the effect of carries due to the insertion of message blocks in both B and C;

(ii) functions $\mathcal{U}(x) = 3 \times x \bmod 2^{32}$ and $\mathcal{V}(x) = 5 \times x \bmod 2^{32}$;

(iii) the quadratic function $(x, y) \mapsto x \wedge \overline{y}$ used when updating the words of of A;

(iv) the final update loop on the buffer A.

These four ingredients excepted, Shabal is linear. As one can see in Chapter 6, we propose weakened versions of Shabal in order to simplify the cryptanalysis of the real function. In these simplified versions, nonlinear effects are typically eliminated to allow a simpler analysis: one at least among $(i), (ii), (iii), (iv)$ is left nonlinear otherwise the cryptanalysis is trivial and uninteresting.

Chapter 5

Security Proofs for the Shabal Construction

Contents

5.1 Introduction

5.1.1 Provable Security for Hash Constructions

Indifferentiability.

Over the past few years, the notion of indifferentiability has become a standard security notion for symmetric primitives, including hash functions and blockciphers. Originally suggested by Maurer, Renner and Holenstein [31], the property of indifferentiability means that the system behaves ideally up to a certain point, provided that its inner ingredients (some simpler primitives) behave ideally. A number of works [7, 14, 13] have recently been derived from this notion, among which a proof that ideal ciphers are equivalent to random oracles [14].

The concept of indifferentiability specifies a security game played between an oracle system Q and a distinguisher $\mathcal{D}$. Q may contain several components, typically a cryptographic construction $\mathcal{C}^{\mathcal{P}}$ which calls some inner primitive $\mathcal{P}$. Construction $\mathcal{C}$ is said to be indifferentiable up to a certain security bound if the system $Q = (\mathcal{C}^{\mathcal{P}}, \mathcal{P})$ can be replaced by a second oracle system $Q' = (\mathcal{H}, \mathcal{S}^{\mathcal{H}})$ with identical interface in such a way that $\mathcal{D}$ cannot tell the difference. Here $\mathcal{H}$ is the idealized version of $\mathcal{C}^{\mathcal{P}}$ (*i.e.,* a random oracle if $\mathcal{C}$ is a hash construction) and $\mathcal{S}$ is a simulator which must behave like $\mathcal{P}$.

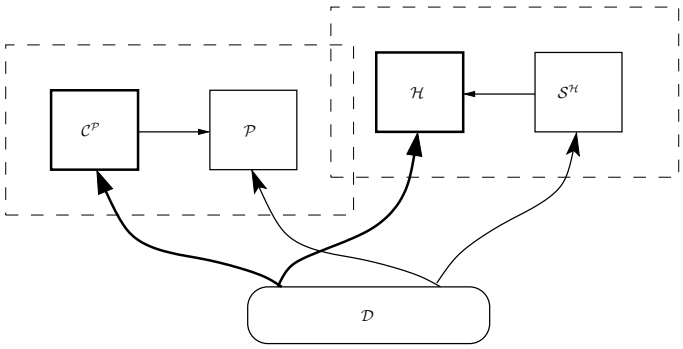

Figure 5.1: The inner primitive $\mathcal{P}$ is assumed ideal. The cryptographic construction $\mathcal{C}^{\mathcal{P}}$ has oracle access to $\mathcal{P}$. The simulator $\mathcal{S}^{\mathcal{H}}$ has oracle access to the random oracle $\mathcal{H}$. The distinguisher interacts either with $Q = (\mathcal{C}^{\mathcal{P}}, \mathcal{P})$ or $Q' = (\mathcal{H}, \mathcal{S}^{\mathcal{H}})$ and has to tell them apart.

In its interaction with the system Q or Q', the distinguisher makes *left calls* to either $\mathcal{C}^{\mathcal{P}}$ or $\mathcal{H}$ and *right calls* to either $\mathcal{P}$ or $\mathcal{S}^{\mathcal{H}}$. We will call N the *total* number of right calls *i.e.,* the number of calls received by $\mathcal{P}$ when $\mathcal{D}$ interacts with Q – regardless of their origin which may be either $\mathcal{C}^{\mathcal{P}}$ or $\mathcal{D}$. We define the advantage of distinguisher $\mathcal{D}$ as

$$\mathsf{Adv}(\mathcal{D}, \mathcal{C}) = \left|\Pr\left[\mathcal{D}^{\mathcal{Q}} = 1 \mid \mathcal{Q} = (\mathcal{C}^{\mathcal{P}}, \mathcal{P})\right] - \Pr\left[\mathcal{D}^{\mathcal{Q}} = 1 \mid \mathcal{Q} = (\mathcal{H}, \mathcal{S}^{\mathcal{H}})\right]\right|$$

where probabilities are taken over the random coins of all parties. Obviously $\mathsf{Adv}(\mathcal{D}, \mathcal{C})$ is a function of N.

Security notions in idealized models.

Hash functions are expected to fulfill three major security properties: collision resistance, preimage resistance and second preimage resistance. Surprisingly enough, up to our knowledge no provable-security framework in idealized proof models has emerged in the cryptographic literature for these notions. We suggest such a framework based on the ideal cipher model in this chapter and subsequently use it to assess the collision, preimage and second preimage resistance of Shabal. Of independent interest, our security models could easily be applied to other constructions.

5.1.2 Summary of Our Security Results

Extending a number of proof techniques (in particular we refine the graph-based simulation approach of [7]), we view the keyed permutation $\mathcal{P}$ as an ideal keyed permutation (*i.e.,* an ideal cipher) and show a number of security properties on the mode of operation of Shabal:

Theorem 2: Shabal behaves like a random oracle up to

$$2^{(\ell_a + \ell_m)/2} = 2^{448}$$

evaluations of $\mathcal{P}$ or $\mathcal{P}^{-1}$;

Theorem 3: Shabal is collision resistant when the collision finder is bounded to $2^{\ell_h/2}$ evaluations of $(\mathcal{P}, \mathcal{P}^{-1})$; internal collisions require no less than

$$2^{(\ell_a+\ell_m)/2} = 2^{448}$$

evaluations of $(\mathcal{P}, \mathcal{P}^{-1})$;

Theorem 4: Shabal is preimage resistant when the preimage finder is limited to

$$\min(2^{\ell_h}, 2^{-(\ell_a+\ell_m-\log(\ell_m+1)-2)}) = \min(2^{\ell_h}, 2^{885}) = 2^{\ell_h}$$

evaluations of $(\mathcal{P}, \mathcal{P}^{-1})$;

Theorem 5: Shabal is second preimage-resistant for κ-bit messages up to

$$2^{\ell_a+\ell_m-\log k^*} = 2^{896-\log k^*}$$

evaluations of $(\mathcal{P}, \mathcal{P}^{-1})$ where $k^* = \lceil(\kappa+1)/\ell_m\rceil$.

5.1.3 Roadmap

Section 5.2 reformulates the mode of operation of Shabal and introduces notation for the proofs. Section 5.3 shows that Shabal essentially behaves as a random oracle. Sections 5.4, 5.5 and 5.6 are dedicated to proving the collision, preimage and second preimage resistance of Shabal assuming that $\mathcal{P}$ behaves as an ideal keyed permutation.

5.2 Reformulating the Mode of Operation of Shabal

Although Shabal is defined with very specific values for parameters, we will consider a formal abstraction of the operating mode where all parameters are left as undefined as possible. For notational convenience, we refer to our generic construction as $\mathcal{C}^{\mathcal{P}}$ throughout our security analysis.

Parameters. Construction $\mathcal{C}^{\mathcal{P}}$ is parameterized by four parameters $\ell_h, \ell_m, \ell_a, \mathsf{E} \in \mathbb{N}$, an initialization vector

$$(a_0, b_0, c_0) \in \{0,1\}^{\ell_a} \times \{0,1\}^{\ell_m} \times \{0,1\}^{\ell_m}$$

and relies on a fixed-length primitive

$$\mathcal{P} : \{0,1\}^{\ell_m} \times \{0,1\}^{\ell_a} \times \{0,1\}^{\ell_m} \times \{0,1\}^{\ell_m} \to \{0,1\}^{\ell_a} \times \{0,1\}^{\ell_m}$$

which we consider to be either a compression function or a keyed permutation over $\{0,1\}^{\ell_a} \times \{0,1\}^{\ell_m}$ with key space $\{0,1\}^{\ell_m} \times \{0,1\}^{\ell_m}$.

Input message. Let $M \in \{0,1\}^*$ be the input of $\mathcal{C}^{\mathcal{P}}$.

Message padding. M is first padded with $1 \,\|\, 0^\ell$ for smallest $\ell \geq 0$ such that $M \,\|\, 1 \,\|\, 0^\ell$ can be split into a list of ℓ_m-bit input blocks

$$M \,\|\, 1 \,\|\, 0^\ell = m_1 \,\|\, m_2 \,\|\, \ldots \,\|\, m_k \,.$$

Initialization. $\mathcal{C}^{\mathcal{P}}$ sets $(m, a, b, c) = (0, a_0, b_0, c_0)$.

Message rounds. For $i = 1$ to k, $\mathcal{C}^{\mathcal{P}}$ executes the two subroutines

$$(m, a, b, c) = \mathsf{Insert}[m_i, i](m, a, b, c) \ , \quad (a, b) = \mathcal{P}(m, a, b, c)$$

where for $\mathfrak{m} \in \{0,1\}^{\ell_m}$ and $w \in \{0,1\}^{64}$

$$\mathsf{Insert}[\mathfrak{m}, w](m, a, b, c) = (\mathfrak{m}, a \oplus (w), c \boxminus m \boxplus \mathfrak{m}, b) \ ,$$

and (w) stands for the 64-bit integer w which most significant bits are completed with 0-bits to yield an ℓ_a-bit integer.

Final rounds. Given the current internal state (m, a, b, c), $\mathcal{C}^{\mathcal{P}}$ now computes for $e = 1$ to E:

$$(m, a, b, c) = \mathsf{Insert}[m_k, k](m, a, b, c) \ , \quad (a, b) = \mathcal{P}(m, a, b, c) \ .$$

Output. The output of $\mathcal{C}^{\mathcal{P}}(M)$ is defined to be the string $b \bmod 2^{\ell_h}$.

The operating mode $\mathcal{C}$ of Shabal is depicted on Fig. 5.2.

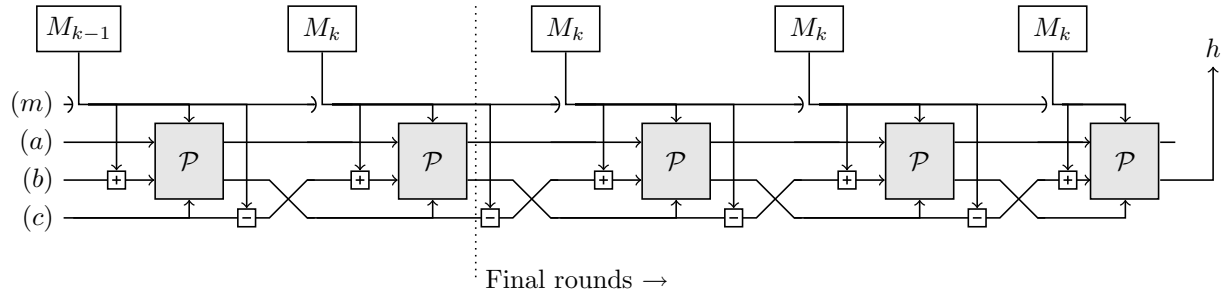

Figure 5.2: A reformulation of the mode of operation of Shabal with a focus on the final rounds. Note that the counter w is omitted on this picture.

5.3 Shabal is Indifferentiable from a Random Oracle

For the sake of completeness, we consider the cases of $\mathcal{P}$ being instantiated either as a random function or as a random keyed permutation. The first case allows us to introduce definitions and proof techniques. We then extend our results to the second case which directly relates to Shabal.

Theorem 1 (Random function)**.** *Assume $\mathcal{P}$ is a random function and let $\mathcal{H}$ be a random oracle. There exists a simulator $\mathcal{S}$ such that for any distinguisher $\mathcal{D}$ totalling at most N right calls,*

$$\mathsf{Adv}(\mathcal{D}, \mathcal{C}) \leq N(2N-1) \cdot 2^{-(\ell_a + \ell_m)} \ .$$

$\mathcal{S}$ makes at most N calls to the random oracle $\mathcal{H}$ and runs in time at most $O(N^2)$.

Theorem 2 (Random keyed permutation)**.** *Assume $\mathcal{P}$ is a random keyed permutation and let $\mathcal{H}$ be a random oracle. There exists a simulator $\mathcal{S}$ such that for any distinguisher $\mathcal{D}$ totalling at most N right calls,*

$$\mathsf{Adv}(\mathcal{D}, \mathcal{C}) \leq N(4N-3) \cdot 2^{-(\ell_a + \ell_m)} \ .$$

$\mathcal{S}$ makes at most N calls to the random oracle $\mathcal{H}$ and runs in time at most $O(N^2)$.

This shows that $\mathcal{C}$ has capacity $(\ell_a + \ell_m)/2$ in the sense of [7]. The remainder of this section is dedicated to a proof of Theorems 1 and 2.

5.3.1 Preliminaries to the Proofs

Our game-based proof technique.

The indifferentiability property is proved by applying game-hopping to progressively construct the simulator $\mathcal{S}$ (as opposed to the proof of [14] which goes the opposite way). The successive games are represented on Fig. 5.3 and are summarized as follows.

(a) We depart from Game 0 which is the original game where the distinguisher interacts with the system $\mathcal{Q} = (\mathcal{C}^{\mathcal{P}}, \mathcal{P})$ where $\mathcal{C}$ is an implementation of the construction[1].

(b) The random function $\mathcal{P}$ is replaced by a simulator $\mathcal{S}$ which merely forwards calls to $\mathcal{P}$ and returns output values to the caller, either $\mathcal{C}$ or $\mathcal{D}$.

(c) $\mathcal{S}$ simulates $\mathcal{P}$ on its own instead of calling it.

(d) $\mathcal{S}$ now makes calls to $\mathcal{H}$ to define output values.

(e) $\mathcal{C}$ is replaced by a temporary simulator $\mathcal{I}$ which executes $\mathcal{C}^{\mathcal{P}}$ on its inputs (thereby calling $\mathcal{S}$ whenever evaluations of $\mathcal{P}$ are required) but ignores its responses and calls $\mathcal{H}$ to return the outputs of $\mathcal{H}$.

(f) $\mathcal{I}$ does not execute $\mathcal{C}^{\mathcal{P}}$ anymore but just returns the outputs of $\mathcal{H}$.

(g) $\mathcal{I}$ is replaced by a direct access to the random oracle $\mathcal{H}$. The distinguisher now interacts with the final system $\mathcal{Q} = (\mathcal{H}, \mathcal{S}^{\mathcal{H}})$.

At each transition, we upper bound the probability gap that $\mathcal{D}$ outputs 1 when interacting with the system before and after the transition is applied. As usual, probability gaps between consecutive games are bounded using the Difference Lemma:

Claim 1 (Difference Lemma)**.** *Let U, V, Ev be three events such that $\Pr[U \wedge \neg\mathsf{Ev}] = \Pr[V \wedge \neg\mathsf{Ev}]$. Then $|\Pr[U] - \Pr[V]| \leq \Pr[\mathsf{Ev}]$.*

We then use triangular inequalities to bound the distinguisher's advantage

$$\mathsf{Adv}(\mathcal{D}, \mathcal{C}) = \left|\Pr\left[\mathcal{D}^{\mathcal{Q}} = 1 \mid \mathcal{Q} = (\mathcal{C}^{\mathcal{P}}, \mathcal{P})\right] - \Pr\left[\mathcal{D}^{\mathcal{Q}} = 1 \mid \mathcal{Q} = (\mathcal{H}, \mathcal{S}^{\mathcal{H}})\right]\right| .$$

Our approach allows crystal-clear indifferentiability proofs and tighter bounds. In particular, this technique readily applies to the sponge construction of Bertoni *et al.* [7] and in particular reveals a secondary term $N2^{-(\ell_a+\ell_m)}$ which would be overlooked by their proof.

Preliminary definitions.

Let $\mathcal{X} = \{0,1\}^{\ell_m} \times \{0,1\}^{\ell_a} \times \{0,1\}^{\ell_m} \times \{0,1\}^{\ell_m}$ be the set of all possible internal states. We distinguish between states reached before and after message insertion as follows. For $x, \tilde{x}, y \in \mathcal{X}$ and $\mathtt{m} \in \{0,1\}^{\ell_m}$, we write

$$\begin{aligned}
x \overset{\mathtt{m},k}{\rightsquigarrow} y \quad & \text{if } y = \mathsf{Insert}[\mathtt{m}, k](x) , \\
x \overset{\star,k}{\rightsquigarrow} y \quad & \text{if } x \overset{\mathtt{m},k}{\rightsquigarrow} y \text{ for some } \mathtt{m} \in \{0,1\}^{\ell_m} , \\
x \overset{k,\tilde{k}}{\sim} \tilde{x} \quad & \text{if there exists } y \in \mathcal{X} \text{ such that } x \overset{\star,k}{\rightsquigarrow} y \text{ and } \tilde{x} \overset{\star,\tilde{k}}{\rightsquigarrow} y , \\
y \xrightarrow{\mathcal{F}} x \quad & \text{if } y = (m, a, b, c),\ x = (m, a', b', c) \text{ where } (a', b') = \mathcal{P}(m, a, b, c) .
\end{aligned}$$

[1] We make no distinction between the construction $\mathcal{C}$ and a program that computes it.

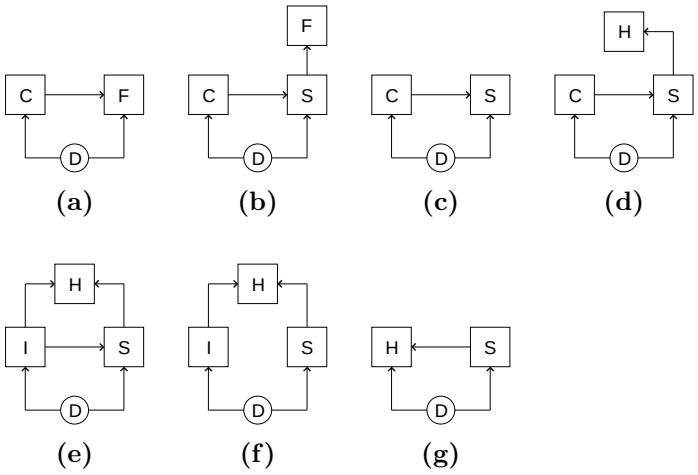

Figure 5.3: Our game-based construction of simulator $\mathcal{S}$.

Definition 1 (0-Paths)**.** *Let $x_0 = (0, a_0, b_0, c_0)$ and $x \in \mathcal{X}$. We call 0-path to x a non-empty list of ℓ_m-bit strings $\mu = \langle m_1, \ldots, m_k \rangle$ such that*

$$x_0 \overset{m_1,1}{\rightsquigarrow} y_1 \xrightarrow{\mathcal{F}} x_1 \overset{m_2,2}{\rightsquigarrow} y_2 \quad \ldots \quad x_{k-2} \overset{m_{k-1},k-1}{\rightsquigarrow} y_{k-1} \xrightarrow{\mathcal{F}} x_{k-1} \overset{m_k,k}{\rightsquigarrow} y_k \xrightarrow{\mathcal{F}} x$$

for some $x_1, \ldots, x_{k-1}, y_1, \ldots, y_k \in \mathcal{X}$.

Definition 2 (e-Paths)**.** *Let $1 \leq e \leq \mathsf{E}$ and $x \in \mathcal{X}$. We call e-path to x a non-empty list of ℓ_m-bit strings $\mu = \langle m_1, \ldots, m_k \rangle$ such that $m_k \neq 0^{\ell_m}$, μ is a 0-path to x_k and*

$$x_k \overset{m_k,k}{\rightsquigarrow} y^1 \xrightarrow{\mathcal{F}} x^1 \quad \ldots \quad y^{e-1} \xrightarrow{\mathcal{F}} x^{e-1} \overset{m_k,k}{\rightsquigarrow} y^e \xrightarrow{\mathcal{F}} x$$

for some $x^k, x^1, \ldots, x^{e-1}, y^1, \ldots, y^e \in \mathcal{X}$.

Let $\mu = \langle m_1, \ldots, m_k \rangle$ be some E-path to $x \in \mathcal{X}$. μ uniquely defines a bitstring $M \in \{0,1\}^*$ such that

$$M \,\|\, 1 \,\|\, 0^\ell = m_1 \,\|\, \ldots \,\|\, m_k$$

for minimal $0 \leq \ell < \ell_m$. Note that M can be the empty bitstring. We then define $\mathsf{unpad}(\mu)$ as being M. Note that μ corresponds to the sequence of message blocks inserted into the internal state by $\mathcal{C}$ when executed on input $M = \mathsf{unpad}(\mu)$. x is then the final internal state reached by the hash function and the final output h is taken as the b-part of x. When μ contains only all-zero blocks, namely $m_i = 0^{\ell_m}$ for $i \in [1, k]$, μ is not considered as a path and $\mathsf{unpad}(\mu)$ is left undefined.

Remark 1. *Note that we have defined $\ell_h = \ell_m$ in the mode of operation $\mathcal{C}$ that we consider. Our results can easily be extended to the general case $0 \leq \ell_h \leq \ell_m$. This convention places the indifferentiability game in the most beneficial setting towards the distinguisher $\mathcal{D}$.*

Hash graphs and graph-based simulators.

Our simulator $\mathcal{S}$ evades the attempts of $\mathcal{D}$ in seeking inconsistencies in the simulation of $\mathcal{P}$. To this end, $\mathcal{S}$ maintains a transcript that collects and organizes information about adversarial queries and outputs responses in a manageable form. The transcript is used to detect inconsistencies, in which case $\mathcal{S}$ aborts execution. Following the approach of [7], we represent the internal states handled by $\mathcal{S}$ as the nodes of a graph $\mathcal{G}$. Graph $\mathcal{G}$ is represented as a tuple of evolving sets $\mathcal{G} = (X, Y, Z) \subseteq \mathcal{X} \times \mathcal{X} \times \mathcal{X}^2$ where $X \cup Y$ is the set of nodes and Z the set of edges of $\mathcal{G}$. Y is the set of queries to $\mathcal{P}$ received by $\mathcal{S}$ and X the set of responses[2] returned by $\mathcal{S}$. X also collects

[2]completed with the m-part and the c-part of their preimage to yield a proper internal state $\in \mathcal{X}$.

all queries to $\mathcal{P}^{-1}$ received by $\mathcal{S}$, their (completed) responses being added to Y. Thus Z contains edges of the form $y \xrightarrow{\mathcal{F}} x$ for $(x, y) \in X \times Y$, an edge $y \xrightarrow{\mathcal{F}} x$ meaning that the internal state y leads to the internal state x when applying the round function

$$y = (m, a, b, c) \to x = (m, a', b', c) \quad \text{where} \quad (a', b') = \mathcal{P}(m, a, b, c) \; .$$

A few natural properties on $\mathcal{G}$ arise from our setting; in particular

- for each and every $x \in X$, there exists an edge $y \xrightarrow{\mathcal{F}} x \in Z$ for some $y \in Y$;
- for each and every $y \in Y$, there exists an edge $y \xrightarrow{\mathcal{F}} x \in Z$ for some $x \in X$;
- after q queries to $(\mathcal{P}, \mathcal{P}^{-1})$ are answered by $\mathcal{S}$, we have

$$|Y| \leq q \; , \quad |X| \leq q \; , \quad |Z| \leq q \; .$$

We will call such a graph a *hash graph*. It is easily seen that given two hash graphs $\mathcal{G}_1 = (X_1, Y_1, Z_1)$ and $\mathcal{G}_2 = (X_2, Y_2, Z_2)$ the componentwise union

$$\mathcal{G} = \mathcal{G}_1 \cup \mathcal{G}_2 = (X_1 \cup X_2, Y_1 \cup Y_2, Z_1 \cup Z_2)$$

is also a hash graph.

Definition 3 (Rooted nodes)**.** *Let $e \in [0, \mathsf{E}]$. An e-path $\mu = \langle m_1, \ldots, m_k \rangle$ to state $x \in \mathcal{X}$ is said to be in graph $\mathcal{G} = (X, Y, Z)$ if all the states along the e-path (including x itself) are nodes of $\mathcal{G}$. More precisely, if the e-path is*

$$x_0 \overset{m_1,1}{\rightsquigarrow} y_1 \xrightarrow{\mathcal{F}} x_1 \quad \ldots \quad y_{k-1} \xrightarrow{\mathcal{F}} x_{k-1} \overset{m_k,k}{\rightsquigarrow} y_k \xrightarrow{\mathcal{F}} x_k \overset{m_k,k}{\rightsquigarrow} y^1 \xrightarrow{\mathcal{F}} x^1 \quad \ldots \quad y^{e-1} \xrightarrow{\mathcal{F}} x^{e-1} \overset{m_k,k}{\rightsquigarrow} y^e \xrightarrow{\mathcal{F}} x$$

then one must have $x_1, \ldots, x_k, x^1, \ldots, x^{e-1}, x \in X$, $y_1, \ldots, y_k, y^1, \ldots, y^e \in Y$, $y_i \xrightarrow{\mathcal{F}} x_i \in Z$ for $i \in [1, k]$, $y^i \xrightarrow{\mathcal{F}} x^i \in Z$ for $i \in [1, e-1]$ and $y^e \xrightarrow{\mathcal{F}} x \in Z$. We will say that a node $x \in X$ is e-rooted in $\mathcal{G}$ if x admits at least one e-path in $\mathcal{G}$. By extension, a node $y \in Y$ is said to be e-rooted if there exists an e-rooted $x \in X$ such that $y \xrightarrow{\mathcal{F}} x \in Z$. By convention, x_0 will always be considered 0-rooted (with a path of length zero) in graph $\mathcal{G}$ regardless of its contents.

Detecting inconsistencies.

We build a simulator $\mathcal{S}$ for $\mathcal{P}$ that makes calls to the random oracle $\mathcal{H}$ and interacts with the distinguisher $\mathcal{D}$. The goal of $\mathcal{S}$ consists in keeping generating associations $y \mapsto \mathcal{P}(y)$ for inputs $y \in \mathcal{X}$ chosen by $\mathcal{D}$ which are consistent with the values output by $\mathcal{H}$. A singularity occurs when maintaining consistency with $\mathcal{H}$ would imply a double definition of $\mathcal{P}(y)$ for some $y \in \mathcal{X}$, a contradiction with $\mathcal{P}$ being well-defined. An example of inconsistency is when two distinct 0-paths $\mu = \langle m_1, m_2, \ldots, m_k \rangle$ and $\mu' = \langle m'_1, m'_2, \ldots, m'_k \rangle \neq \mu$ of identical length k both lead to the same internal state $x \in \mathcal{X}$. Let $M_1 = m_1 \,\|\, m_2 \,\|\, \ldots \,\|\, m_k$ and $M_2 = m'_1 \,\|\, m'_2 \,\|\, \ldots \,\|\, m'_k$. Then it is easily seen that $\mathcal{C}(M_1 \,\|\, M) = \mathcal{C}(M_2 \,\|\, M)$ for any $M \in \{0,1\}^*$. This reveals a strong separation between $\mathcal{C}$ and $\mathcal{H}$ for which such a property cannot be observed.

5.3.2 Proofs of Theorems 1 and 2

Proof of Theorem 1.

We now proceed to construct a sequence of games leading to the security bound claimed in Theorem 1. Recall that $\mathcal{P}$ is a random function in what follows.

Game 0. This is the original game where $\mathcal{D}$ interacts with $\mathcal{C}^{\mathcal{P}}$ and the random function $\mathcal{P}$. Let W_0 denote the event that $\mathcal{D}$ outputs 1 in Game 0, and more generally W_i the event that $\mathcal{D}$ outputs 1 in Game i. By definition of Game 0, we have

$$\Pr[W_0] = \Pr\left[\mathcal{D}^{\mathcal{Q}} = 1 \mid \mathcal{Q} = (\mathcal{C}^{\mathcal{P}}, \mathcal{P})\right] \; .$$

Game 1. $\mathcal{P}$ is replaced by a simulator $\mathcal{S}$ with the same interface as $\mathcal{P}$ which merely forwards calls to $\mathcal{P}$ and returns the responses of $\mathcal{P}$ to either $\mathcal{C}$ or $\mathcal{D}$. Throughout the game, $\mathcal{S}$ constructs the two graphs

$$\mathcal{G}_{\mathcal{C}} = (X_{\mathcal{C}}, Y_{\mathcal{C}}, Z_{\mathcal{C}}) , \qquad \mathcal{G}_{\mathcal{D}} = (X_{\mathcal{D}}, Y_{\mathcal{D}}, Z_{\mathcal{D}}) ,$$

by passively collecting inputs and outputs of $\mathcal{P}$ arising from the requests made respectively by $\mathcal{C}$ or $\mathcal{D}$. $\mathcal{S}$ is depicted on Fig. 5.4. The action of $\mathcal{S}$ does not modify the view of $\mathcal{D}$, meaning that $\Pr[W_1] = \Pr[W_0]$.

Initialization of $\mathcal{S}$

No input, no output

1. set $X_{\mathcal{C}} = Y_{\mathcal{C}} = Z_{\mathcal{C}} = \emptyset$
2. set $X_{\mathcal{D}} = Y_{\mathcal{D}} = Z_{\mathcal{D}} = \emptyset$

Simulation of $\mathcal{P}$

Input: $y = (m, a, b, c) \in \mathcal{X}$, origin $\mathcal{O} =$ either $\mathcal{C}$ or $\mathcal{D}$

Output: (a', b')

1. add node y to $Y_{\mathcal{O}}$
2. call $\mathcal{P}$ to get $(a', b') = \mathcal{P}(m, a, b, c)$
3. add node $x = (m, a', b', c)$ to $X_{\mathcal{O}}$ and edge $y \xrightarrow{\mathcal{F}} x$ to $Z_{\mathcal{O}}$
4. return (a', b') to $\mathcal{O}$

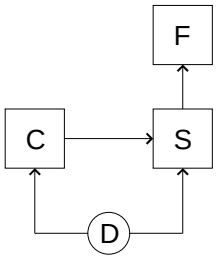

Figure 5.4: Indifferentiability: Simulator $\mathcal{S}$ for $\mathcal{P}$ in Game 1.

Game 2. We slightly modify our simulator to get rid of the random function $\mathcal{P}$ and replace it with a perfect simulation. Every time $\mathcal{S}$ needs to define $\mathcal{P}(y)$ for some $y \in \mathcal{X}$, $\mathcal{S}$ randomly selects the response $\mathcal{P}(y)$. We depict the new simulator on Fig. 5.5. This does not modify the distributions since $\mathcal{P}$ is a random oracle. Hence $\Pr[W_2] = \Pr[W_1]$.

Initialization of $\mathcal{S}$

No input, no output

1. set $X_{\mathcal{C}} = Y_{\mathcal{C}} = Z_{\mathcal{C}} = \emptyset$
2. set $X_{\mathcal{D}} = Y_{\mathcal{D}} = Z_{\mathcal{D}} = \emptyset$

Simulation of $\mathcal{P}$

Input: $y = (m, a, b, c) \in \mathcal{X}$, origin $\mathcal{O} =$ either $\mathcal{C}$ or $\mathcal{D}$

Output: (a', b')

1. add node y to $Y_{\mathcal{O}}$
2. if there exists $y \xrightarrow{\mathcal{F}} x \in Z_{\mathcal{C}} \cup Z_{\mathcal{D}}$
   (a) add x to $X_{\mathcal{O}}$ and edge $y \xrightarrow{\mathcal{F}} x$ to $Z_{\mathcal{O}}$
   (b) return (a', b') where $x = (m, a', b', c)$
3. randomly select $a' \leftarrow \{0,1\}^{\ell_a}$ and $b' \leftarrow \{0,1\}^{\ell_m}$
4. add node $x = (m, a', b', c)$ to $X_{\mathcal{O}}$ and edge $y \xrightarrow{\mathcal{F}} x$ to $Z_{\mathcal{O}}$
5. return (a', b') to $\mathcal{O}$

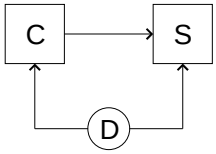

Figure 5.5: Indifferentiability: Simulator $\mathcal{S}$ for $\mathcal{P}$ in Game 2.

Definitions. Let us introduce the following notation, where $e \in [0, \mathsf{E}]$ and $k \in \mathbb{N}$:

$\mathcal{G} = (X, Y, Z)$	the componentwise union of graphs $\mathcal{G}_\mathcal{C}$ and $\mathcal{G}_\mathcal{D}$,
$X^{e,k}$	the subset of X which elements have an e-path of length k ,
$Y^{e,k}$	the subset of Y which elements have an e-path of length k ,
$X^{0,0}$	$\{x_0\}$,
$X^{e,\star}$	$\cup_{k\in\mathbb{N}} X^{e,k}$,
$Y^{e,\star}$	$\cup_{k\in\mathbb{N}} Y^{e,k}$,
$A(q)$	the value of set A after $\mathcal{S}$ has processed the q-th query ,
$\delta\,[cond]$	evaluates to 1 if condition *cond* is met, to 0 otherwise ,
$\mathsf{Event}(q)$	the event Event occurring while $\mathcal{S}$ processes the q-th call to $\mathcal{P}$ (or $(\mathcal{P}, \mathcal{P}^{-1})$) .

Game 3. We now make sure that nodes $y \in Y$ admit at most one E-path in $\mathcal{G}$. To this end, we define the following predicate.

Definition 4 (Collision event Coll)**.** *Given the current graph $\mathcal{G}$ and two states $x, \tilde{x} \in X$, the predicate $\mathsf{Coll}(x, \tilde{x})$ is defined as $\mathsf{Coll}(x, \tilde{x}) = \mathsf{Coll}_0(x, \tilde{x}) \vee \mathsf{Coll}_1(x, \tilde{x})$ where*

- *$\mathsf{Coll}_0(x, \tilde{x})$ evaluates to True if and only if for some $k \in \mathbb{N}$, both $x, \tilde{x} \in X^{0,k}$ and $x \overset{k,k}{\sim} \tilde{x}$;*
- *$\mathsf{Coll}_1(x, \tilde{x})$ evaluates to True if and only if for some $k, \tilde{k} \in \mathbb{N}$, $x \in X^{\mathsf{E}-1,k}$, $\tilde{x} \in X^{\mathsf{E}-1,\tilde{k}}$ and there exists $y \in \mathcal{X}$ such that*

$$x \overset{m_k,k}{\rightsquigarrow} y \quad \text{and} \quad \tilde{x} \overset{\tilde{m}_{\tilde{k}},\tilde{k}}{\rightsquigarrow} y$$

where m_k is the last message block of an $(\mathsf{E}-1)$-path of length k to x and $\tilde{m}_{\tilde{k}}$ is the last block of an $(\mathsf{E}-1)$-path of length $\tilde{k}$ to $\tilde{x}$.

Claim 2. *Assume $y \in Y^{\mathsf{E},\star}$ admits at least two E-paths in $\mathcal{G}$. Then there must exist two rooted nodes $x, \tilde{x} \in X$ such that $\mathsf{Coll}(x, \tilde{x})$ is true.*

Proof. Let us assume that $\mu = \langle m_1, \ldots, m_k \rangle$ and $\tilde{\mu} = \langle \tilde{m}_1, \ldots, \tilde{m}_{\tilde{k}} \rangle$ are two E-paths to y. Let $y^* \in Y$ be the y-node common to μ and $\tilde{\mu}$ with greatest distance from x_0 and such that all the nodes that follow y^* in μ and $\tilde{\mu}$ coincide. We face two cases:

1. either $y^* = y$ in which case $\mathsf{Coll}_1(x, \tilde{x})$ must be true where x (resp. $\tilde{x}$) is the parent of y^* with respect to μ (resp. $\tilde{\mu}$);
2. or the prefix sub-paths of μ and $\tilde{\mu}$ leading to y^* are both 0-paths. Then the parents x and $\tilde{x}$ of y^* with respect to these 0-paths are such that $\mathsf{Coll}_0(x, \tilde{x})$ is true.

In both cases, $\mathsf{Coll}(x, \tilde{x})$ must be true for at least two rooted nodes $x, \tilde{x} \in X_\mathcal{C} \cup X_D$. □

We modify simulator $\mathcal{S}$ to detect a collision whenever a new output value $x \in \mathcal{X}$ is assigned and abort when $\mathsf{Coll}(x, \tilde{x})$ is true for some preexisting rooted $\tilde{x}$. We refer to this event as Abort_1. The upgraded simulator is depicted on Fig. 5.6.

Claim 3. $|\Pr[W_3] - \Pr[W_2]| \leq \Pr[\mathsf{Abort}_1] \leq N(N-1) \cdot 2^{-(\ell_a+\ell_m)}$.

Proof. Obviously, $\Pr[W_3 \wedge \neg\mathsf{Abort}_1] = \Pr[W_2 \wedge \neg\mathsf{Abort}_1]$ so that the Difference Lemma applies. Now consider the input $y = (m, a, b, c) \in \mathcal{X}$ of the q-th simulation of $\mathcal{P}$. Several cases occur:

(i) either y is not rooted in $\mathcal{G}$ in which case for any response state $x = (m, a', b', c) \in \mathcal{X}$ that $\mathcal{S}$ may define, $\mathsf{Coll}(x, \tilde{x})$ evaluates to False for every $\tilde{x} \in X$;

Initialization of $\mathcal{S}$

No input, no output

1. set $X_{\mathcal{C}} = Y_{\mathcal{C}} = Z_{\mathcal{C}} = \emptyset$
2. set $X_{\mathcal{D}} = Y_{\mathcal{D}} = Z_{\mathcal{D}} = \emptyset$

Simulation of $\mathcal{P}$

Input: $y = (m, a, b, c) \in \mathcal{X}$, origin $\mathcal{O} =$ either $\mathcal{C}$ or $\mathcal{D}$

Output: (a', b')

1. add node y to $Y_{\mathcal{O}}$
2. if there exists $y \xrightarrow{\mathcal{F}} x \in Z_{\mathcal{C}} \cup Z_{\mathcal{D}}$
   (a) add x to $X_{\mathcal{O}}$ and edge $y \xrightarrow{\mathcal{F}} x$ to $Z_{\mathcal{O}}$
   (b) return (a', b') where $x = (m, a', b', c)$
3. randomly select $a' \leftarrow \{0,1\}^{\ell_a}$ and $b' \leftarrow \{0,1\}^{\ell_m}$
4. add node $x = (m, a', b', c)$ to $X_{\mathcal{O}}$ and edge $y \xrightarrow{\mathcal{F}} x$ to $Z_{\mathcal{O}}$
5. if $\exists \tilde{x} \in X_{\mathcal{C}} \cup X_{\mathcal{D}}$ such that $\mathsf{Coll}(x, \tilde{x})$ (event Abort_1) then abort
6. return (a', b') to $\mathcal{O}$

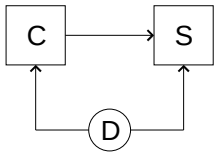

Figure 5.6: Indifferentiability: Simulator $\mathcal{S}$ for $\mathcal{P}$ in Game 3.

(ii) or y is 0-rooted and $\mathsf{Coll}_0(x, \tilde{x})$ may evaluate to True for some 0-rooted $\tilde{x}$;

(iii) or y is $(\mathsf{E}-1)$-rooted and $\mathsf{Coll}_1(x, \tilde{x})$ may evaluate to True for some $(\mathsf{E}-1)$-rooted $\tilde{x}$.

Note that cases (ii) and (iii) are not mutually exclusive. Let us first consider (ii). Assuming $y \in Y^{0,k}$ for some $k \in \mathbb{N}$, let $\tilde{x} = (\tilde{m}, \tilde{a}, \tilde{b}, \tilde{c}) \in X^{0,k}(q-1)$ be fixed and let us pose

$$D(m, c) = \{(m, a', b', c) \mid (a', b') \in \{0,1\}^{\ell_a} \times \{0,1\}^{\ell_m}\} .$$

Then, taking probabilities over the uniformly random selection $x \leftarrow D(m, c)$:

$$\begin{aligned}
\Pr\left[x \overset{k,k}{\sim} \tilde{x}\right] &= \Pr\left[\exists\, \mathfrak{m}, \tilde{\mathfrak{m}} \in \{0,1\}^{\ell_m} : \mathsf{Insert}[\mathfrak{m}, k](x) = \mathsf{Insert}[\tilde{\mathfrak{m}}, k](\tilde{x})\right] \\
&= \Pr\left[\exists\, \mathfrak{m}, \tilde{\mathfrak{m}} \in \{0,1\}^{\ell_m} : \begin{array}{rcl} \mathfrak{m} &=& \tilde{\mathfrak{m}} \\ a' \oplus (k) &=& \tilde{a} \oplus (k) \\ c \boxminus m \boxplus \mathfrak{m} &=& \tilde{c} \boxminus \tilde{m} \boxplus \tilde{\mathfrak{m}} \\ b' &=& \tilde{b} \end{array}\right] \\
&= 2^{-\ell_a} \cdot 2^{-\ell_m} \cdot \delta\left[c \boxminus m = \tilde{c} \boxminus \tilde{m}\right] \leq 2^{-(\ell_a + \ell_m)} .
\end{aligned}$$

Therefore, for any $\tilde{x} \in X^{0,k}(q-1)$

$$\Pr\left[\mathsf{Coll}_0(x, \tilde{x})\right] = \Pr\left[x \overset{k,k}{\sim} \tilde{x}\right] \leq 2^{-(\ell_a + \ell_m)}$$

so that

$$\Pr\left[\mathsf{Coll}_0(x, \tilde{x}) \text{ for some } \tilde{x} \in X^{0,k}(q-1)\right] \leq |X^{0,k}(q-1)| \cdot 2^{-(\ell_a + \ell_m)} \leq (q-1) \cdot 2^{-(\ell_a + \ell_m)} .$$

Let us now consider (iii) and assume that $y \in Y^{\mathsf{E}-1,k}$. Let us fix $\tilde{k} \in \mathbb{N}$ as well as $\tilde{x} = (\tilde{m}, \tilde{a}, \tilde{b}, \tilde{c}) \in X^{\mathsf{E}-1,\tilde{k}}(q-1)$ and $m_k, \tilde{m}_{\tilde{k}} \in \{0,1\}^{\ell_m}$. Taking again probabilities over the distribution $x \leftarrow D(m,c)$, one gets

$$\begin{aligned}
\Pr\left[\exists y \in \mathcal{X} : x \overset{m_k,k}{\rightsquigarrow} y \wedge \tilde{x} \overset{\tilde{m}_{\tilde{k}},\tilde{k}}{\rightsquigarrow} y\right] &= \Pr\left[\mathsf{Insert}[m_k,k](x) = \mathsf{Insert}[\tilde{m}_{\tilde{k}},\tilde{k}](\tilde{x})\right] \\
&= \Pr\left[\begin{array}{rcl} m_k & = & \tilde{m}_{\tilde{k}} \\ a' \oplus (k) & = & \tilde{a} \oplus (\tilde{k}) \\ c & = & \tilde{c} \\ b' & = & \tilde{b} \end{array}\right] \\
&= 2^{-\ell_a} \cdot 2^{-\ell_m} \cdot \delta\left[m_k = \tilde{m}_{\tilde{k}} \wedge c = \tilde{c}\right] \leq 2^{-(\ell_a+\ell_m)} .
\end{aligned}$$

Hence for any $\tilde{x} \in X^{\mathsf{E}-1,\star}(q-1)$

$$\Pr\left[\mathsf{Coll}_1(x,\tilde{x}) \text{ for some } \tilde{x} \in X^{\mathsf{E}-1,\star}(q-1)\right] \leq |X^{\mathsf{E}-1,\star}(q-1)| \cdot 2^{-(\ell_a+\ell_m)} \leq (q-1) \cdot 2^{-(\ell_a+\ell_m)} .$$

Therefore $\Pr\left[\mathsf{Abort}_1(q)\right] \leq 2 \cdot (q-1) \cdot 2^{-(\ell_a+\ell_m)}$ so that

$$\Pr\left[\mathsf{Abort}_1\right] \leq \sum_{q=1}^{N} \Pr\left[\mathsf{Abort}_1(q)\right] \leq \sum_{q=1}^{N} 2 \cdot (q-1) \cdot 2^{-(\ell_a+\ell_m)} = N(N-1) \cdot 2^{-(\ell_a+\ell_m)}$$

as claimed. □

Property 1. *At any moment, for any $y \in Y$, there is at most one E-path to y in $\mathcal{G}$.*

Proof. Assuming that the q-th query $y \in Y$ admits two E-paths, we easily get from the above claim that $\mathsf{Coll}(x,\tilde{x})$ must be true for two preexisting nodes $x, \tilde{x} \in X(q-1)$, meaning that the event Abort_1 must have been realized during a previous call to $\mathcal{S}$. □

Game 4. We now make sure that each query $y \in Y$ received by $\mathcal{S}$ is either rooted at the time of its request or will not be rooted at a later call to $\mathcal{S}$. We define a second predicate as follows.

Definition 5 (Dependence event Dep)**.** *Given the current graph $\mathcal{G}$ and two nodes $x \in X$ and $\tilde{y} \in Y$, $\mathsf{Dep}(x,\tilde{y}) = \mathsf{Dep}_0(x,\tilde{y}) \vee \mathsf{Dep}_1(x,\tilde{y})$ where*

- *$\mathsf{Dep}_0(x,\tilde{y})$ evaluates to True if and only if for some $k \in \mathbb{N}$, $x \in X^{0,k}$ and $x \overset{\star,k+1}{\rightsquigarrow} \tilde{y}$*
- *$\mathsf{Dep}_1(x,\tilde{y})$ evaluates to True if and only if for some $k \in \mathbb{N}$ and $e \in [1, \mathsf{E}-1]$, x admits an e-path $\mu = \langle m_1, \ldots, m_k \rangle$ in $\mathcal{G}$ and $x \overset{m_k,k}{\rightsquigarrow} \tilde{y}$.*

We modify $\mathcal{S}$ to detect that $\mathsf{Dep}(x,\tilde{y})$ evaluates to True for some $\tilde{y} \in Y$ whenever a new output node $x \in \mathcal{X}$ is created, in which case $\mathcal{S}$ aborts. We refer to this event as Abort_2. The new simulator is depicted on Fig. 5.7.

Claim 4. $|\Pr\left[W_4\right] - \Pr\left[W_3\right]| \leq \Pr\left[\mathsf{Abort}_2\right] \leq N(N-1) \cdot 2^{-(\ell_a+\ell_m)}$.

Proof. Let y be the q-th query to $\mathcal{P}$ and assume $y \in Y^{0,k}$. Let us fix $\tilde{y} = (\tilde{m}, \tilde{a}, \tilde{b}, \tilde{c}) \in Y(q-1)$. Taking the following probabilities over $x \leftarrow D(m,c)$, we have

$$\begin{aligned}
\Pr\left[x \overset{\star,k+1}{\rightsquigarrow} \tilde{y}\right] &= \Pr\left[\exists\, \mathfrak{m} \in \{0,1\}^{\ell_m} : \mathsf{Insert}[\mathfrak{m}, k+1](x) = \tilde{y}\right] \\
&= \Pr\left[\exists\, \mathfrak{m} \in \{0,1\}^{\ell_m} : \begin{array}{rcl} \mathfrak{m} & = & \tilde{m} \\ a' \oplus (k+1) & = & \tilde{a} \\ c \boxminus m \boxplus \mathfrak{m} & = & \tilde{b} \\ b' & = & \tilde{c} \end{array}\right] \\
&= 2^{-\ell_a} \cdot 2^{-\ell_m} \cdot \delta\left[c \boxminus m \boxplus \tilde{m} = \tilde{b}\right] \leq 2^{-(\ell_a+\ell_m)} .
\end{aligned}$$

Initialization of $\mathcal{S}$

No input, no output

1. set $X_{\mathcal{C}} = Y_{\mathcal{C}} = Z_{\mathcal{C}} = \emptyset$
2. set $X_{\mathcal{D}} = Y_{\mathcal{D}} = Z_{\mathcal{D}} = \emptyset$

Simulation of $\mathcal{P}$

Input: $y = (m, a, b, c) \in \mathcal{X}$, origin $\mathcal{O} =$ either $\mathcal{C}$ or $\mathcal{D}$

Output: (a', b')

1. add node y to $Y_{\mathcal{O}}$
2. if there exists $y \xrightarrow{\mathcal{F}} x \in Z_{\mathcal{C}} \cup Z_{\mathcal{D}}$
   (a) add x to $X_{\mathcal{O}}$ and edge $y \xrightarrow{\mathcal{F}} x$ to $Z_{\mathcal{O}}$
   (b) return (a', b') where $x = (m, a', b', c)$
3. randomly select $a' \leftarrow \{0,1\}^{\ell_a}$ and $b' \leftarrow \{0,1\}^{\ell_m}$
4. add node $x = (m, a', b', c)$ to $X_{\mathcal{O}}$ and edge $y \xrightarrow{\mathcal{F}} x$ to $Z_{\mathcal{O}}$
5. if $\exists \tilde{x} \in X_{\mathcal{C}} \cup X_{\mathcal{D}}$ such that $\mathsf{Coll}(x, \tilde{x})$ (event Abort_1) then abort
6. if $\exists \tilde{y} \in Y_{\mathcal{C}} \cup Y_{\mathcal{D}}$ such that $\mathsf{Dep}(x, \tilde{y})$ (event Abort_2) then abort
7. return (a', b') to $\mathcal{O}$

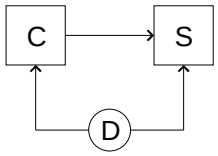

Figure 5.7: Indifferentiability: Simulator $\mathcal{S}$ for $\mathcal{P}$ in Game 4.

Hence for any $\tilde{y} \in Y(q-1)$ one has

$$\Pr\left[\mathsf{Dep}_0(x,\tilde{y})\right] = \Pr\left[x \overset{\star,k+1}{\rightsquigarrow} \tilde{y}\right] \leq 2^{-(\ell_a+\ell_m)} .$$

Let us now assume that y admits an e-path $\mu = \langle m_1, \ldots, m_k \rangle$ for $e \in [1, \mathsf{E}-1]$ and fix $\tilde{y} \in Y(q-1)$. Taking again the following probability over $x \leftarrow D(m,c)$, we get

$$\Pr\left[x \overset{m_k,k}{\rightsquigarrow} \tilde{y}\right] = \Pr\left[\begin{array}{rcl} m_k & = & \tilde{m} \\ a' \oplus (k) & = & \tilde{a} \\ c & = & \tilde{b} \\ b' & = & \tilde{c} \end{array}\right] = 2^{-\ell_a} \cdot 2^{-\ell_m} \cdot \delta\left[c = \tilde{b} \wedge m_k = \tilde{m}\right] \leq 2^{-(\ell_a+\ell_m)} .$$

Hence $\Pr\left[\mathsf{Dep}_1(x,\tilde{y})\right] \leq 2^{-(\ell_a+\ell_m)}$ for any $\tilde{y} \in Y(q-1)$ and overall

$$\Pr\left[\mathsf{Abort}_2(q)\right] = \Pr\left[\exists \tilde{y} \in Y(q-1) : \mathsf{Dep}(x,\tilde{y})\right] \leq 2 \cdot |Y(q-1)| \cdot 2^{-(\ell_a+\ell_m)} \leq 2 \cdot (q-1) \cdot 2^{-(\ell_a+\ell_m)}$$

which gives

$$\Pr\left[\mathsf{Abort}_2\right] \leq \sum_{q=1}^{N} \Pr\left[\mathsf{Abort}_2(q)\right] = \sum_{q=1}^{N} 2 \cdot (q-1) \cdot 2^{-(\ell_a+\ell_m)} = N(N-1) \cdot 2^{-(\ell_a+\ell_m)}$$

as claimed. □

It is easily seen that, unless $\mathcal{S}$ aborts:

Property 2. *Each query $y \in Y$ sent to $\mathcal{S}$ may admit a number of paths (possibly none) at the time it is treated by $\mathcal{S}$ but will admit no new path at a later time during the execution of $\mathcal{S}$.*

Game 5. We now insert the random oracle $\mathcal{H}$ in the game. Instead of defining a completely random response $\mathcal{P}(y)$ for a E-rooted $y \in Y$, $\mathcal{S}$ will rather make a call to $\mathcal{H}$ to let $\mathcal{H}$ define the b-part of $\mathcal{P}(y)$. $\mathcal{S}$ then completes the missing a-part of $\mathcal{P}(y)$ with a random value. Since y has a unique E-path in $\mathcal{G}$ (if any) which can be extracted at the time of its request, this modification is well-defined. When y is not E-rooted, $\mathcal{S}$ defines $\mathcal{P}(y)$ at random as in previous games. The new simulator is depicted on Fig. 5.8. Since the outputs of $\mathcal{H}$ are uniform and independent of $\mathcal{D}$'s view, this does not modify the distributions. Therefore $\Pr[W_5] = \Pr[W_4]$.

Game 6. The program $\mathcal{C}$ is replaced with a temporary simulator $\mathcal{I}$ with identical interface. Whenever $\mathcal{D}$ sends a query $M \in \{0,1\}^*$ to $\mathcal{I}$, $\mathcal{I}$ does two things: first $\mathcal{I}$ executes construction $\mathcal{C}^{\mathcal{P}}$ on M by making calls to $\mathcal{S}$ whenever an evaluation of $\mathcal{P}$ is required; then $\mathcal{I}$ completely ignores the outputs of $\mathcal{S}$ and makes a call to $\mathcal{H}$ to get $h = \mathcal{H}(M)$ and returns h to $\mathcal{D}$. This does not change either the execution of $\mathcal{S}$ or the view of $\mathcal{D}$ so that $\Pr[W_6] = \Pr[W_5]$.

Game 7. We now make sure that each query $y \in Y = Y_{\mathcal{C}} \cup Y_{\mathcal{D}}$ which admits paths in $\mathcal{G} = \mathcal{G}_{\mathcal{C}} \cup \mathcal{G}_{\mathcal{D}}$ admits the same paths in $\mathcal{G}_{\mathcal{D}}$:

Definition 6 (Guess event Guess)**.** *Given the current graphs $\mathcal{G}_{\mathcal{C}}, \mathcal{G}_{\mathcal{D}}$ and a node $y \in Y_{\mathcal{C}} \cup Y_{\mathcal{D}}$, the predicate $\mathsf{Guess}(y)$ evaluates to True if and only if y admits a path in $\mathcal{G} = \mathcal{G}_{\mathcal{C}} \cup \mathcal{G}_{\mathcal{D}}$ but does not admit this path in $\mathcal{G}_{\mathcal{D}}$.*

We modify $\mathcal{S}$ to detect that $\mathsf{Guess}(y)$ evaluates to True for some input query y in which case $\mathcal{S}$ aborts. We refer to this event as Abort_3. The new simulator is as depicted on Fig. 5.9.

Applying the Difference Lemma again, we get

Claim 5. $|\Pr[W_7] - \Pr[W_6]| \leq \Pr[\mathsf{Abort}_3] \leq N \cdot 2^{-(\ell_a+\ell_m)}$.

Initialization of $\mathcal{S}$

No input, no output

1. set $X_{\mathcal{C}} = Y_{\mathcal{C}} = Z_{\mathcal{C}} = \emptyset$
2. set $X_{\mathcal{D}} = Y_{\mathcal{D}} = Z_{\mathcal{D}} = \emptyset$

Simulation of $\mathcal{P}$

Input: $y = (m, a, b, c) \in \mathcal{X}$, origin $\mathcal{O} =$ either $\mathcal{C}$ or $\mathcal{D}$

Output: (a', b')

1. add node y to $Y_{\mathcal{O}}$
2. if there exists $y \xrightarrow{\mathcal{F}} x \in Z_{\mathcal{C}} \cup Z_{\mathcal{D}}$
   (a) add x to $X_{\mathcal{O}}$ and edge $y \xrightarrow{\mathcal{F}} x$ to $Z_{\mathcal{O}}$
   (b) return (a', b') where $x = (m, a', b', c)$
3. if y has an E-path μ in graph $\mathcal{G}_{\mathcal{C}} \cup \mathcal{G}_{\mathcal{D}}$
   (a) compute $M = \mathsf{unpad}(\mu)$
   (b) call $\mathcal{H}$ to get $h = \mathcal{H}(M)$
   (c) set $b' = h$
4. else
   (a) randomly select $b' \leftarrow \{0,1\}^{\ell_m}$
5. randomly select $a' \leftarrow \{0,1\}^{\ell_a}$
6. add node $x = (m, a', b', c)$ to $X_{\mathcal{O}}$ and edge $y \xrightarrow{\mathcal{F}} x$ to $Z_{\mathcal{O}}$
7. if $\exists \tilde{x} \in X_{\mathcal{C}} \cup X_{\mathcal{D}}$ such that $\mathsf{Coll}(x, \tilde{x})$ (event Abort_1) then abort
8. if $\exists \tilde{y} \in Y_{\mathcal{C}} \cup Y_{\mathcal{D}}$ such that $\mathsf{Dep}(x, \tilde{y})$ (event Abort_2) then abort
9. return (a', b') to $\mathcal{O}$

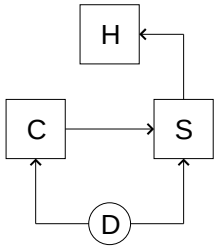

Figure 5.8: Indifferentiability: Simulator $\mathcal{S}$ for $\mathcal{P}$ in Game 5.

Initialization of $\mathcal{S}$

No input, no output

1. set $X_{\mathcal{C}} = Y_{\mathcal{C}} = Z_{\mathcal{C}} = \emptyset$
2. set $X_{\mathcal{D}} = Y_{\mathcal{D}} = Z_{\mathcal{D}} = \emptyset$

Simulation of $\mathcal{P}$

Input: $y = (m, a, b, c) \in \mathcal{X}$, origin $\mathcal{O} =$ either $\mathcal{C}$ or $\mathcal{D}$

Output: (a', b')

1. add node y to $Y_{\mathcal{O}}$
2. if there exists $y \xrightarrow{\mathcal{F}} x \in Z_{\mathcal{C}} \cup Z_{\mathcal{D}}$
   (a) add x to $X_{\mathcal{O}}$ and edge $y \xrightarrow{\mathcal{F}} x$ to $Z_{\mathcal{O}}$
   (b) return (a', b') where $x = (m, a', b', c)$
3. if $\mathsf{Guess}(y)$ (event Abort_3) then abort
4. if y has a path μ in graph $\mathcal{G}_{\mathcal{D}}$
   (a) compute $M = \mathsf{unpad}(\mu)$
   (b) call $\mathcal{H}$ to get $h = \mathcal{H}(M)$
   (c) set $b' = h$
5. else
   (a) randomly select $b' \leftarrow \{0,1\}^{\ell_m}$
6. randomly select $a' \leftarrow \{0,1\}^{\ell_a}$
7. add node $x = (m, a', b', c)$ to $X_{\mathcal{O}}$ and edge $y \xrightarrow{\mathcal{F}} x$ to $Z_{\mathcal{O}}$
8. if $\exists \tilde{x} \in X_{\mathcal{C}} \cup X_{\mathcal{D}}$ such that $\mathsf{Coll}(x, \tilde{x})$ (event Abort_1) then abort
9. if $\exists \tilde{y} \in Y_{\mathcal{C}} \cup Y_{\mathcal{D}}$ such that $\mathsf{Dep}(x, \tilde{y})$ (event Abort_2) then abort
10. return (a', b') to $\mathcal{O}$

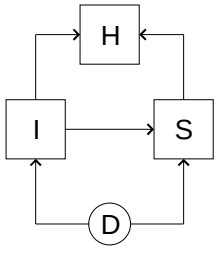

Figure 5.9: Indifferentiability: Simulator $\mathcal{S}$ for $\mathcal{P}$ in Game 7.

Proof. Let $y \in Y = Y_{\mathcal{C}} \cup Y_{\mathcal{D}}$ be a node with an e-path $\mu = \langle m_1, \ldots, m_k \rangle$ in $\mathcal{G}_{\mathcal{C}} \cup \mathcal{G}_{\mathcal{D}}$ such that μ is not an E-path to y in $\mathcal{G}_{\mathcal{D}}$. There must exist a sequence of nodes $x_1, \ldots, x_{k-1} \in X_{\mathcal{C}} \cup X_{\mathcal{D}}$ and $y_1, \ldots, y_{k-1} \in Y_{\mathcal{C}} \cup Y_{\mathcal{D}}$ such that

$$x_0 \overset{m_1,1}{\rightsquigarrow} y_1 \xrightarrow{\mathcal{F}} x_1 \overset{m_2,2}{\rightsquigarrow} y_2 \ \ldots \ \overset{m_k,k}{\rightsquigarrow} y_k \xrightarrow{\mathcal{F}} x_k \overset{m_k,k}{\rightsquigarrow} y^1 \xrightarrow{\mathcal{F}} x^1 \ldots y^{e-1} \xrightarrow{\mathcal{F}} x^{e-1} \overset{m_k,k}{\rightsquigarrow} y \ .$$

Since y does not admit μ as an e-path in $\mathcal{G}_D$, there must exist either $i \in [1,k]$ such that the edge $y_i \xrightarrow{\mathcal{F}} x_i$ does not belong to $Z_{\mathcal{D}}$ and i is maximal or $j \in [1, e-1]$ such that $y^j \xrightarrow{\mathcal{F}} x^j \notin Z_{\mathcal{D}}$ and j is maximal. Let us unify both notation by saying that $y(u) \xrightarrow{\mathcal{F}} x(u) \notin Z_{\mathcal{D}}$ where $u \in [1, k+e-1]$ and u maximal. Then either $y(u) \notin Y_{\mathcal{D}}$ or $x(u) \notin X_{\mathcal{D}}$. If we had $y(u) \in Y_{\mathcal{D}}$ then by unicity of the definitions of $\mathcal{P}$ generated by $\mathcal{S}$ we would have $x(u) \in X_{\mathcal{D}}$, a contradiction. Hence $y(u) \in Y_{\mathcal{C}} \setminus Y_{\mathcal{D}}$. Now since $y(u) \xrightarrow{\mathcal{F}} x(u) \in Z_{\mathcal{C}} \setminus Z_{\mathcal{D}}$, $\mathcal{D}$'s view on $x(u) = (m, a', b', c)$ is m (which $\mathcal{D}$ can choose by itself) and possibly c if $y(u-1) \in Y_{\mathcal{D}}$ since c is equal to the b-value of $x(u-1)$. Hence (a', b') is unknown to $\mathcal{D}$ and uniform over $\{0,1\}^{\ell_a} \times \{0,1\}^{\ell_m}$. Since $x(u) \overset{\star,v}{\rightsquigarrow} y(u+1)$ with $v = u+1$ if $u \in [1, k-1]$ and $v = k$ if $u \in [k, k+e-1]$ (note that $y(u+1) = y$ if $u = k+e-1$), the a-part and c-part of $y(u+1)$ are respectively equal to $a' \oplus (v)$ and b'. Hence these two parts of the query $y(u+1) \in Y_{\mathcal{D}}$ made by $\mathcal{D}$ must collide with the uniformly distributed values chosen by $\mathcal{S}$ (independently of any interaction with $\mathcal{D}$) when processing $\mathcal{P}(y(u))$. This happens with probability $2^{-(\ell_a+\ell_m)}$. Therefore, for any $q \in [1, N]$ and any query $y \in Y(q)$, $\Pr[\mathsf{Guess}(y)] \leq 2^{-(\ell_a+\ell_m)}$ and

$$\Pr[\mathsf{Abort}_3] \leq \Pr[\mathsf{Guess}(y) \text{ for some } y \in Y(N)] \leq N \cdot 2^{-(\ell_a+\ell_m)}$$

which concludes the proof. □

Game 8. We now modify the description of $\mathcal{I}$. Given a query $M \in \{0,1\}^*$, $\mathcal{I}$ simply calls $\mathcal{H}$ and returns $\mathcal{H}(M)$ to $\mathcal{D}$. The simulator $\mathcal{S}$ does not receive inputs from $\mathcal{I}$ anymore, resulting in that the graph $\mathcal{G}_{\mathcal{C}} = (X_{\mathcal{C}}, Y_{\mathcal{C}}, Z_{\mathcal{C}})$ is useless and can be removed completely. Testing for Abort_3 is meaningless and can be removed as well. The new simulator is as depicted on Fig. 5.10. The view of $\mathcal{D}$ is unchanged, so that $\Pr[W_8] = \Pr[W_7]$.

Game 9. The interface $\mathcal{I}$ in Game 7 can be safely removed to let $\mathcal{D}$ interact with $\mathcal{H}$ directly. This does not change any of the distributions. Moreover, the description of this final game complies with the experiment of having $\mathcal{D}$ interact with the system $\mathcal{Q} = (\mathcal{H}, \mathcal{S}^{\mathcal{H}})$ and therefore

$$\Pr[W_9] = \Pr[W_8] = \Pr\left[\mathcal{D}^{\mathcal{Q}} = 1 \mid \mathcal{Q} = (\mathcal{H}, \mathcal{S}^{\mathcal{H}})\right] \ .$$

Conclusion. Summing up, we finally get that $\mathsf{Adv}(\mathcal{D}, \mathcal{C}) \leq \sum_{i=1}^{9} |\Pr[W_i] - \Pr[W_{i-1}]|$ which provides the upper bound stated in Theorem 1. It is easily seen on the final simulator $\mathcal{S}$ (see Fig. 5.10) that $\mathcal{S}$ makes at most N calls to $\mathcal{H}$ and that the extra computation cost due to the extraction of paths and evaluations of predicates is upper bounded by $O(N^2)$.

Proof of Theorem 2.

We now extend the above to the case when $\mathcal{P}$ is a keyed permutation, resulting in that indifferentiability is shown in the ideal cipher model. In addition to queries sent to $\mathcal{P}$, $\mathcal{S}$ also has to simulate queries made to $\mathcal{P}^{-1}$. This does not change essentially the simulators of Games 0–9 previously described and the above proof can be extended to support simulations of $\mathcal{P}^{-1}$ as follows.

Games 0–1. In Game 0 and Game 1, $\mathcal{S}$ simply forwards the $\mathcal{P}^{-1}$ requests to the ideal cipher $\mathcal{P}$ and returns the output without change *i.e.,* there is no simulation.

Initialization of $\mathcal{S}$

No input, no output

1. set $X_{\mathcal{D}} = Y_{\mathcal{D}} = Z_{\mathcal{D}} = \emptyset$

Simulation of $\mathcal{P}$

Input: $y = (a, b, c) \in \mathcal{X}$ (origin is always $\mathcal{D}$)

Output: (a', b')

1. add node y to $Y_{\mathcal{D}}$
2. if there exists $y \xrightarrow{\mathcal{F}} x \in Z_{\mathcal{D}}$
   (a) return (a', b') where $x = (m, a', b', c)$
3. if y has a path μ in graph $\mathcal{G}_{\mathcal{D}}$
   (a) compute $M = \mathsf{unpad}(\mu)$
   (b) call $\mathcal{H}$ to get $h = \mathcal{H}(M)$
   (c) set $b' = h$
4. else
   (a) randomly select $b' \leftarrow \{0,1\}^{\ell_m}$
5. randomly select $a' \leftarrow \{0,1\}^{\ell_a}$
6. add node $x = (m, a', b', c)$ to $X_{\mathcal{D}}$ and edge $y \xrightarrow{\mathcal{F}} x$ to $Z_{\mathcal{D}}$
7. if $\exists \tilde{x} \in X_{\mathcal{D}}$ such that $\mathsf{Coll}(x, \tilde{x})$ (event Abort_1) then abort
8. if $\exists \tilde{y} \in Y_{\mathcal{D}}$ such that $\mathsf{Dep}(x, \tilde{y})$ (event Abort_2) then abort
9. return (a', b') to $\mathcal{D}$

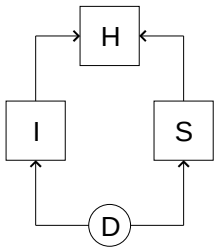

Figure 5.10: Indifferentiability: Simulator $\mathcal{S}$ for $\mathcal{P}$ in Game 8 (and final simulator).

Simulation of $\mathcal{P}^{-1}$

Input: $x = (a, b, c) \in \mathcal{X}$, origin $\mathcal{O} =$ either $\mathcal{C}$ or $\mathcal{D}$

Output: (a', b')

1. add node x to $X_{\mathcal{O}}$
2. if there exists $y \xrightarrow{\mathcal{F}} x \in Z_{\mathcal{C}} \cup Z_{\mathcal{D}}$
   (a) add y to $Y_{\mathcal{O}}$ and edge $y \xrightarrow{\mathcal{F}} x$ to $Z_{\mathcal{O}}$
   (b) return (a', b') where $y = (m, a', b', c)$
3. randomly select $a' \leftarrow \{0,1\}^{\ell_a}$ and $b' \leftarrow \{0,1\}^{\ell_m}$
4. add node $y = (m, a', b', c)$ to $X_{\mathcal{O}}$ and edge $y \xrightarrow{\mathcal{F}} x$ to $Z_{\mathcal{O}}$
5. if $\exists \tilde{x} \in X_{\mathcal{C}} \cup X_{\mathcal{D}}$ such that $y \xrightarrow{\mathcal{F}} \tilde{x} \in Z_{\mathcal{C}} \cup Z_{\mathcal{D}}$ (event Abort_4) then abort
6. return (a', b') to $\mathcal{O}$

Figure 5.11: Indifferentiability: Simulation of $\mathcal{P}^{-1}$ in Game 2.

Game 2. In Game 2, $\mathcal{S}$ simulates $\mathcal{P}^{-1}$ for any input $x \in \mathcal{X}$ by selecting $\mathcal{P}^{-1}(x)$ randomly. To prevent any inconsistency between definitions for $\mathcal{P}$ and $\mathcal{P}^{-1}$, $\mathcal{S}$ detects when the selected response $y \in \mathcal{X}$ assigned to $\mathcal{P}^{-1}(x)$ admits a preexisting definition $y \xrightarrow{\mathcal{F}} \tilde{x}$ for $\tilde{x} \in X_{\mathcal{C}} \cup X_{\mathcal{D}}$ (event Abort_4) in which case $\mathcal{S}$ aborts execution. The new simulation of $\mathcal{P}^{-1}$ is as shown on Fig. 5.11.

Unless event Abort_4 occurs, the view of $\mathcal{D}$ is identical in Game 1 and Game 2 and since Abort_4 occurs with probability at most $2 \cdot (q-1) \cdot 2^{-(\ell_a+\ell_m)}$ when processing the q-th query, it follows that

$$|\Pr[W_2] - \Pr[W_1]| \leq \Pr[\mathsf{Abort}_4] \leq N(N-1) \cdot 2^{-(\ell_a+\ell_m)} .$$

Game 3. In Game 3, we want to ascertain that Property 1 still holds. It is easily seen that Property 1 can be broken only if **(a)** an $\mathcal{P}$-query $y \in \mathcal{X}$ is assigned an image state $x \in \mathcal{X}$ such that $\mathsf{Coll}(x, \tilde{x})$ is true for some preexisting $\tilde{x}$ in $\mathcal{G}_{\mathcal{C}} \cup \mathcal{G}_{\mathcal{D}}$, or **(b)** an $\mathcal{P}^{-1}$-query $x \in \mathcal{X}$ is assigned a preimage state y such that $\mathsf{Dep}(\tilde{x}, y)$ is true for some preexisting node $\tilde{x}$. Case **(a)** is taken care of in the simulation of $\mathcal{P}$ thanks to event Abort_1. We therefore modify the simulation of $\mathcal{P}^{-1}$ to force an abortion (event Abort_5) if $\mathsf{Dep}(\tilde{x}, y)$ evaluates to True for some rooted $\tilde{x} \in X_{\mathcal{C}} \cup X_{\mathcal{D}}$, as depicted on Fig. 5.12.

Claim 6. $|\Pr[W_3] - \Pr[W_2]| \leq \Pr[\mathsf{Abort}_1 \vee \mathsf{Abort}_5] \leq N(N-1) \cdot 2^{-(\ell_a+\ell_m)}$.

Proof. We upper bound $\Pr[\mathsf{Abort}_1 \vee \mathsf{Abort}_5]$ by $\Pr[\mathsf{Abort}_1] + \Pr[\mathsf{Abort}_5]$ and use the previous bound on $\Pr[\mathsf{Abort}_1]$; given fix parameters $\tilde{x} = (\tilde{m}, \tilde{a}, \tilde{b}, \tilde{c}) \in \mathcal{X}(q-1)$ and $m, c \in \{0,1\}^{\ell_m}$, one has

$$\Pr[\mathsf{Dep}(\tilde{x}, y)] \leq 2^{-(\ell_a+\ell_m)}$$

where the probabilities are taken over the random selection $y \leftarrow D(m, c)$. Hence the probability that event $\mathsf{Abort}_5(q)$ occurs is at most $2 \cdot (q-1) \cdot 2^{-(\ell_a+\ell_m)}$ and Abort_5 occurs with probability at most $N(N-1) \cdot 2^{-(\ell_a+\ell_m)}$. □

Game 4. In Game 4, we make sure that Property 2 is verified. To this end, event Abort_2 is added in the simulation of $\mathcal{P}$. It is easily seen that there is no need for an extra abortion event in the simulation of $\mathcal{P}^{-1}$ since the definitions of $\mathcal{P}^{-1}$ cannot create new paths in the graph $\mathcal{G}_{\mathcal{C}} \cup \mathcal{G}_{\mathcal{D}}$ unless Abort_5 occurs.

Games 5–9. Games 5–9 are identical to the proof of Theorem 1 except that $\mathcal{S}$ also contains the simulation of $\mathcal{P}^{-1}$ displayed on Fig. 5.12. Summing all upper bounds on probability gaps, we end up with the claimed indifferentiability bound. $\mathcal{S}$ still makes at most N calls to $\mathcal{H}$ and runs in extra time $O(N^2)$.

5.4 Shabal is Collision Resistant in the Ideal Cipher Model

5.4.1 A Security Model for Collision Resistance in the ICM

We model collision resistance of construction $\mathcal{C}^{\mathcal{P}}$ under the form of a security game played between a collision finder or adversary $\mathcal{A}$ and a challenger $\mathcal{V}$.

Definition 7 (COLL Game)**.** *The game is described as follows:*

1. *$\mathcal{A}$ makes calls to the ideal cipher $(\mathcal{P}, \mathcal{P}^{-1})$*
2. *$\mathcal{A}$ outputs two messages $M_1, M_2 \in \{0,1\}^*$*
3. *$\mathcal{V}$ computes $\mathcal{C}^{\mathcal{P}}(M_1)$ and $\mathcal{C}^{\mathcal{P}}(M_2)$ by calling $\mathcal{P}$*
4. *$\mathcal{V}$ outputs 1 if $\mathcal{C}^{\mathcal{P}}(M_1) = \mathcal{C}^{\mathcal{P}}(M_2)$ or 0 otherwise.*

Simulation of $\mathcal{P}^{-1}$

Input: $x = (m, a, b, c) \in \mathcal{X}$, origin $\mathcal{O} =$ either $\mathcal{C}$ or $\mathcal{D}$

Output: (a', b')

1. add node x to $X_{\mathcal{O}}$
2. if there exists $y \xrightarrow{\mathcal{F}} x \in Z_{\mathcal{C}} \cup Z_{\mathcal{D}}$
   (a) add y to $Y_{\mathcal{O}}$ and edge $y \xrightarrow{\mathcal{F}} x$ to $Z_{\mathcal{O}}$
   (b) return (a', b') where $y = (m, a', b', c)$
3. randomly select $a' \leftarrow \{0,1\}^{\ell_a}$ and $b' \leftarrow \{0,1\}^{\ell_m}$
4. add node $y = (m, a', b', c)$ to $X_{\mathcal{O}}$ and edge $y \xrightarrow{\mathcal{F}} x$ to $Z_{\mathcal{O}}$
5. if $\exists \tilde{x} \in X_{\mathcal{C}} \cup X_{\mathcal{D}}$ such that $y \xrightarrow{\mathcal{F}} \tilde{x} \in Z_{\mathcal{C}} \cup Z_{\mathcal{D}}$ (event Abort_4) then abort
6. if $\exists \tilde{x} \in X_{\mathcal{C}} \cup X_{\mathcal{D}}$ such that $\mathsf{Dep}(\tilde{x}, y)$ (event Abort_5) then abort
7. return (a', b') to $\mathcal{O}$

Figure 5.12: Indifferentiability: Simulation of $\mathcal{P}^{-1}$ in Games 3–9.

We define the success probability $\mathsf{Suc}^{\mathsf{COLL}}(\mathcal{A}, \mathcal{C})$ of $\mathcal{A}$ as the probability that $\mathcal{V}$ outputs 1 when interacting with $\mathcal{A}$ as per the COLL game. $\mathsf{Suc}^{\mathsf{COLL}}(\mathcal{A}, \mathcal{C})$ is a function of the total number N of queries received by the ideal cipher $(\mathcal{P}, \mathcal{P}^{-1})$ throughout the game. Note that $\mathcal{V}$ itself has to make $(k_1 + \mathsf{E}) + (k_2 + \mathsf{E})$ calls to $\mathcal{P}$ when verifying the response (M_1, M_2) of $\mathcal{A}$ if hashing M_1 and M_2 leads to the insertion of respectively k_1 and k_2 message blocks. Thus overall $\mathcal{A}$ makes $N - (k_1 + \mathsf{E}) - (k_2 + \mathsf{E})$ calls to $(\mathcal{P}, \mathcal{P}^{-1})$.

5.4.2 Proving Collision Resistance for Shabal's Mode of Operation

We conceive a simulator $\mathcal{S}$ that simulates $\mathcal{V}$ and $(\mathcal{P}, \mathcal{P}^{-1})$ towards $\mathcal{A}$. An high-level view of $\mathcal{S}$ is as follows. Simulator $\mathcal{S}$

1. simulates the ideal cipher $(\mathcal{P}, \mathcal{P}^{-1})$ and may abort while doing so
2. receives $M_1, M_2 \in \{0,1\}^*$ from $\mathcal{A}$
3. runs its own simulation of $\mathcal{P}$ to compute $\mathcal{C}^{\mathcal{P}}(M_1)$ and $\mathcal{C}^{\mathcal{P}}(M_2)$
4. outputs 0

The underlying proof technique consists in making the simulation of $\mathcal{P}$ abort when $\mathcal{C}^{\mathcal{P}}(M_1) = \mathcal{C}^{\mathcal{P}}(M_2)$. Therefore either $\mathcal{S}$ outputs 0 or the game aborts. Consequently

$$\mathsf{Suc}^{\mathsf{COLL}}(\mathcal{A}, \mathcal{C}) \leq \Pr[\mathcal{S} \text{ aborts}] \ .$$

We state

Theorem 3 (Collision resistance)**.** *There exists a simulator $\mathcal{S}$ as above such that for any collision finder $\mathcal{A}$ playing as per the* COLL *game limited to at most N calls to $(\mathcal{P}, \mathcal{P}^{-1})$,*

$$\Pr[\mathcal{S} \textit{ aborts}] \leq 3 \cdot N(N-1) \cdot 2^{-(\ell_a + \ell_m)} + N(N-1) \cdot 2^{-\ell_h}$$

and $\mathcal{S}$ runs in time at most $O(N^2)$.

5.4.3 Proof of Theorem 3

We build a sequence of games which starts with the COLL security game and ends with a final simulator $\mathcal{S}$. We refer to Section 5.3 for notation and definitions.

Game 0. This is the original COLL game where $\mathcal{A}$ interacts with $\mathcal{V}$ and the ideal cipher $\mathcal{P}$. Let W_0 denote the event that $\mathcal{V}$ outputs 1 in Game 0, and more generally W_i the event that $\mathcal{V}$ outputs 1 in Game i. By definition of Game 0, we have

$$\Pr[W_0] = \mathsf{Suc}^{\mathsf{COLL}}(\mathcal{A}, \mathcal{C}) .$$

Game 1. We replace $\mathcal{V}$ by a first simulator $\mathcal{S}$ which behaves as $\mathcal{V}$ and forwards calls to $(\mathcal{P}, \mathcal{P}^{-1})$. In addition, $\mathcal{S}$ keeps track of the definitions of $\mathcal{P}$ by maintaining the hash graph $\mathcal{G}$ as depicted on Fig. 5.13. The action of $\mathcal{S}$ does not modify the view of $\mathcal{D}$, meaning that $\Pr[W_1] = \Pr[W_0]$.

Initialization of $\mathcal{S}$

No input, no output

1. set $X = Y = Z = \emptyset$

Simulation of $\mathcal{P}$

Input: $y = (m, a, b, c) \in \mathcal{X}$ from $\mathcal{O}$ (either $\mathcal{A}$ or $\mathcal{S}$)

Output: (a', b')

1. add node y to Y
2. call $\mathcal{P}$ to get $(a', b') = \mathcal{P}(m, a, b, c)$
3. add node $x = (m, a', b', c)$ to X and edge $y \xrightarrow{\mathcal{F}} x$ to Z
4. return (a', b') to $\mathcal{O}$

Simulation of $\mathcal{P}^{-1}$

Input: $x = (m, a, b, c) \in \mathcal{X}$ from $\mathcal{A}$

Output: (a', b')

1. add node x to X
2. call $\mathcal{P}^{-1}$ to get $(a', b') = \mathcal{P}^{-1}(m, a, b, c)$
3. add node $y = (m, a', b', c)$ to Y and edge $y \xrightarrow{\mathcal{F}} x$ to Z
4. return (a', b') to $\mathcal{A}$

Completion of $\mathcal{S}$

Input: $M_1, M_2 \in \{0,1\}^*$ from $\mathcal{A}$

1. compute $h_1 = \mathcal{C}^{\mathcal{P}}(M_1, \ell_m)$ and $h_2 = \mathcal{C}^{\mathcal{P}}(M_2, \ell_m)$ by calling $\mathcal{P}$ accordingly
2. if $h_1 = h_2$ then output 1 else output 0

Figure 5.13: Collision resistance: simulator $\mathcal{S}$ in Game 1.

Game 2. We slightly modify our simulator to eliminate the ideal cipher $\mathcal{P}$ and replace it with a perfect simulation. Every time $\mathcal{S}$ needs to define $\mathcal{P}(y)$ for some $y \in \mathcal{X}$ or $\mathcal{P}^{-1}(x)$ for $x \in \mathcal{X}$, $\mathcal{S}$ randomly selects the response. We depict the new simulator on Fig. 5.14. This does not modify the distributions since $\mathcal{P}$ is an ideal cipher. Hence $\Pr[W_2] = \Pr[W_1]$.

Game 3. We now make sure that nodes $x \in X$ which admit one E-path in $\mathcal{G}$ do not collide on their b-part. To this end, we define the following predicate.

Definition 8 (Output collision event $\mathsf{OutputColl}$)**.** *Given the current graph $\mathcal{G}$ and two states $x, \tilde{x} \in X$, the predicate $\mathsf{OutputColl}(x, \tilde{x})$ evaluates to True if and only if both x and $\tilde{x}$ admit an E-path in $\mathcal{G}$ and $b \equiv \tilde{b} \bmod 2^{\ell_h}$ where $x = (m, a, b, c)$ and $\tilde{x} = (\tilde{m}, \tilde{a}, \tilde{b}, \tilde{c})$.*

Initialization of $\mathcal{S}$

No input, no output

1. set $X = Y = Z = \emptyset$

Simulation of $\mathcal{P}$

Input: $y = (m, a, b, c) \in \mathcal{X}$ from $\mathcal{O}$ (either $\mathcal{A}$ or $\mathcal{S}$)

Output: (a', b')

1. add node y to Y
2. if there exists $y \xrightarrow{\mathcal{F}} x = (m, a', b', c) \in Z$ then return (a', b') to $\mathcal{O}$
3. else
   (a) randomly select $a' \leftarrow \{0,1\}^{\ell_a}$ and $b' \leftarrow \{0,1\}^{\ell_m}$
   (b) add node $x = (m, a', b', c)$ to X and edge $y \xrightarrow{\mathcal{F}} x$ to Z
   (c) return (a', b') to $\mathcal{O}$

Simulation of $\mathcal{P}^{-1}$

Input: $x = (m, a, b, c) \in \mathcal{X}$ from $\mathcal{A}$

Output: (a', b')

1. add node x to X
2. if there exists $(m, a', b', c) = y \xrightarrow{\mathcal{F}} x \in Z$ then return (a', b') to $\mathcal{A}$
3. else
   (a) randomly select $a' \leftarrow \{0,1\}^{\ell_a}$ and $b' \leftarrow \{0,1\}^{\ell_m}$
   (b) add node $y = (m, a', b', c)$ to Y and edge $y \xrightarrow{\mathcal{F}} x$ to Z
   (c) return (a', b') to $\mathcal{A}$

Input: $M_1, M_2 \in \{0,1\}^*$ from $\mathcal{A}$

1. compute $h_1 = \mathcal{C}^{\mathcal{P}}(M_1, \ell_m)$ and $h_2 = \mathcal{C}^{\mathcal{P}}(M_2, \ell_m)$ by calling $\mathcal{P}$ accordingly
2. if $h_1 = h_2$ then output 1 else output 0

Figure 5.14: Collision resistance: simulator $\mathcal{S}$ in Game 2.

Initialization of $\mathcal{S}$

No input, no output

1. set $X = Y = Z = \emptyset$

Simulation of $\mathcal{P}$

Input: $y = (m, a, b, c) \in \mathcal{X}$ from $\mathcal{O}$ (either $\mathcal{A}$ or $\mathcal{S}$)

Output: (a', b')

1. add node y to Y
2. if there exists $y \xrightarrow{\mathcal{F}} x = (m, a', b', c) \in Z$ then return (a', b') to $\mathcal{O}$
3. else
   (a) randomly select $a' \leftarrow \{0,1\}^{\ell_a}$ and $b' \leftarrow \{0,1\}^{\ell_m}$
   (b) add node $x = (m, a', b', c)$ to X and edge $y \xrightarrow{\mathcal{F}} x$ to Z
   (c) if $\exists \tilde{x} \in X$ such that $\mathsf{OutputColl}(x, \tilde{x})$ (event Abort_1) then abort
   (d) return (a', b') to $\mathcal{O}$

Simulation of $\mathcal{P}^{-1}$

Input: $x = (m, a, b, c) \in \mathcal{X}$ from $\mathcal{A}$

Output: (a', b')

1. add node x to X
2. if there exists $(m, a', b', c) = y \xrightarrow{\mathcal{F}} x \in Z$ then return (a', b') to $\mathcal{A}$
3. else
   (a) randomly select $a' \leftarrow \{0,1\}^{\ell_a}$ and $b' \leftarrow \{0,1\}^{\ell_m}$
   (b) add node $y = (m, a', b', c)$ to Y and edge $y \xrightarrow{\mathcal{F}} x$ to Z
   (c) return (a', b') to $\mathcal{A}$

Input: $M_1, M_2 \in \{0,1\}^*$ from $\mathcal{A}$

1. compute $h_1 = \mathcal{C}^{\mathcal{P}}(M_1, \ell_m)$ and $h_2 = \mathcal{C}^{\mathcal{P}}(M_2, \ell_m)$ by calling $\mathcal{P}$ accordingly
2. if $h_1 = h_2$ then output 1 else output 0

Figure 5.15: Collision resistance: simulator $\mathcal{S}$ in Game 3.

We modify simulator $\mathcal{S}$ to detect an output collision whenever a new output value $x \in \mathcal{X}$ is assigned and abort when $\mathsf{OutputColl}(x, \tilde{x})$ is true for some preexisting rooted $\tilde{x}$. We refer to this event as Abort_1. The upgraded simulator is depicted on Fig. 5.15.

We state:

Claim 7. *One has* $|\Pr[W_3] - \Pr[W_2]| \leq \Pr[\mathsf{Abort}_1] \leq N(N-1)2^{-\ell_h}$.

Proof. Let us consider the input $y = (m, a, b, c) \in \mathcal{X}$ of the simulation of $\mathcal{P}$ and assume that y is the q-th query to $(\mathcal{P}, \mathcal{P}^{-1})$. Two cases may occur:

(i) either y admits no E-path in $\mathcal{G}$ in which case for any response state assigned by $\mathcal{S}$, $\mathsf{OutputColl}(x, \tilde{x})$ is false for any $\tilde{x} \in X$,

(ii) or y admits an E-path in $\mathcal{G}$.

Let us assume (ii). Let $\tilde{x} = (\tilde{m}, \tilde{a}, \tilde{b}, \tilde{c}) \in X(q-1)$ be fixed and let us pose

$$D(m, c) = \{(m, a', b', c) \mid (a', b') \in \{0,1\}^{\ell_a} \times \{0,1\}^{\ell_m}\} .$$

Then, taking probabilities over the distribution $x \leftarrow D(m, c)$:

$$\Pr[\mathsf{OutputColl}(x, \tilde{x})] = \Pr\left[b' \equiv \tilde{b} \bmod 2^{\ell_h}\right] = 2^{-\ell_m} .$$

Therefore

$$\Pr[\mathsf{Abort}_1(q)] = \Pr[\mathsf{OutputColl}(x, \tilde{x}) \text{ for some } \tilde{x} \in X(q-1)] \leq |X(q-1)| \cdot 2^{-\ell_h} \leq (q-1) \cdot 2^{-\ell_h} .$$

so that

$$\Pr[\mathsf{Abort}_1] \leq \sum_{q=1}^{N} \Pr[\mathsf{Abort}_1(q)] \leq \sum_{q=1}^{N} (q-1) \cdot 2^{-\ell_h} = N(N-1) \cdot 2^{-\ell_h}$$

as announced. □

Game 4. We now ascertain that no pair of internal states $(x, \tilde{x}) \in X^2$ can collide in the sense of paths. Referring to Section 5.3, recall that the predicate $\mathsf{Coll}_0(x, \tilde{x})$ is True when both x and $\tilde{x}$ admit 0-paths of length k and $x \overset{k,k}{\sim} \tilde{x}$. We slightly alter our simulator to detect that Coll_0 evaluates to True during the game, in which case $\mathcal{S}$ aborts. This event is referred to as Abort_2. The new simulator is displayed on Fig. 5.16.

Claim 8. $|\Pr[W_4] - \Pr[W_3]| \leq \Pr[\mathsf{Abort}_2] \leq N(N-1) \cdot 2^{-(\ell_a + \ell_m)}$.

The proof of this claim is identical to the one provided in Section 5.3, Game 3.

Game 5. We now make sure that $\mathcal{A}$ cannot succeed in connecting rooted nodes to non-rooted nodes. This captures the collision-finding strategy where $\mathcal{A}$ applies the operating mode backwards starting from a hash value and later succeeds in finding a path to one of the generated internal states. To ensure this, we proceed in two steps by applying a modification of $\mathcal{S}$ in this game and a second one in Game 6. Recall that the predicate $\mathsf{Dep}_0(x, \tilde{y})$ evaluates to True when x admits a 0-path of length k, $\tilde{y} \in Y$ and $x \overset{\star,k+1}{\rightsquigarrow} \tilde{y}$. We modify $\mathcal{S}$ to detect that $\mathsf{Dep}(x, \tilde{y})$ evaluates to True for some $\tilde{y} \in Y$ whenever a new output node $x \in \mathcal{X}$ is created, in which case $\mathcal{S}$ aborts. We refer to this event as Abort_3. The new simulator is depicted on Fig. 5.17.

Claim 9. $|\Pr[W_5] - \Pr[W_4]| \leq \Pr[\mathsf{Abort}_3] \leq N(N-1) \cdot 2^{-(\ell_a + \ell_m)}$.

The proof of this claim is similar to the one provided in Section 5.3, Game 4.

Initialization of $\mathcal{S}$

No input, no output

1. set $X = Y = Z = \emptyset$

Simulation of $\mathcal{P}$

Input: $y = (m, a, b, c) \in \mathcal{X}$ from $\mathcal{O}$ (either $\mathcal{A}$ or $\mathcal{S}$)

Output: (a', b')

1. add node y to Y
2. if there exists $y \xrightarrow{\mathcal{F}} x = (m, a', b', c) \in Z$ then return (a', b') to $\mathcal{O}$
3. else
   (a) randomly select $a' \leftarrow \{0,1\}^{\ell_a}$ and $b' \leftarrow \{0,1\}^{\ell_m}$
   (b) add node $x = (m, a', b', c)$ to X and edge $y \xrightarrow{\mathcal{F}} x$ to Z
   (c) if $\exists \tilde{x} \in X$ such that $\mathsf{OutputColl}(x, \tilde{x})$ (event Abort_1) then abort
   (d) if $\exists \tilde{x} \in X$ such that $\mathsf{Coll}_0(x, \tilde{x})$ (event Abort_2) then abort
   (e) return (a', b') to $\mathcal{O}$

Simulation of $\mathcal{P}^{-1}$

Input: $x = (m, a, b, c) \in \mathcal{X}$ from $\mathcal{A}$

Output: (a', b')

1. add node x to X
2. if there exists $(m, a', b', c) = y \xrightarrow{\mathcal{F}} x \in Z$ then return (a', b') to $\mathcal{A}$
3. else
   (a) randomly select $a' \leftarrow \{0,1\}^{\ell_a}$ and $b' \leftarrow \{0,1\}^{\ell_m}$
   (b) add node $y = (m, a', b', c)$ to Y and edge $y \xrightarrow{\mathcal{F}} x$ to Z
   (c) return (a', b') to $\mathcal{A}$

Input: $M_1, M_2 \in \{0,1\}^*$ from $\mathcal{A}$

1. compute $h_1 = \mathcal{C}^{\mathcal{P}}(M_1, \ell_m)$ and $h_2 = \mathcal{C}^{\mathcal{P}}(M_2, \ell_m)$ by calling $\mathcal{P}$ accordingly
2. if $h_1 = h_2$ then output 1 else output 0

Figure 5.16: Collision resistance: simulator $\mathcal{S}$ in Game 4.

Initialization of $\mathcal{S}$

No input, no output

1. set $X = Y = Z = \emptyset$

Simulation of $\mathcal{P}$

Input: $y = (m, a, b, c) \in \mathcal{X}$ from $\mathcal{O}$ (either $\mathcal{A}$ or $\mathcal{S}$)

Output: (a', b')

1. add node y to Y
2. if there exists $y \xrightarrow{\mathcal{F}} x = (m, a', b', c) \in Z$ then return (a', b') to $\mathcal{O}$
3. else
   (a) randomly select $a' \leftarrow \{0,1\}^{\ell_a}$ and $b' \leftarrow \{0,1\}^{\ell_m}$
   (b) add node $x = (m, a', b', c)$ to X and edge $y \xrightarrow{\mathcal{F}} x$ to Z
   (c) if $\exists \tilde{x} \in X$ such that $\mathsf{OutputColl}(x, \tilde{x})$ (event Abort_1) then abort
   (d) if $\exists \tilde{x} \in X$ such that $\mathsf{Coll}_0(x, \tilde{x})$ (event Abort_2) then abort
   (e) if $\exists \tilde{y} \in Y$ such that $\mathsf{Dep}_0(x, \tilde{y})$ (event Abort_3) then abort
   (f) return (a', b') to $\mathcal{O}$

Simulation of $\mathcal{P}^{-1}$

Input: $x = (m, a, b, c) \in \mathcal{X}$ from $\mathcal{A}$

Output: (a', b')

1. add node x to X
2. if there exists $(m, a', b', c) = y \xrightarrow{\mathcal{F}} x \in Z$ then return (a', b') to $\mathcal{A}$
3. else
   (a) randomly select $a' \leftarrow \{0,1\}^{\ell_a}$ and $b' \leftarrow \{0,1\}^{\ell_m}$
   (b) add node $y = (m, a', b', c)$ to Y and edge $y \xrightarrow{\mathcal{F}} x$ to Z
   (c) return (a', b') to $\mathcal{A}$

Input: $M_1, M_2 \in \{0,1\}^*$ from $\mathcal{A}$

1. compute $h_1 = \mathcal{C}^{\mathcal{P}}(M_1, \ell_m)$ and $h_2 = \mathcal{C}^{\mathcal{P}}(M_2, \ell_m)$ by calling $\mathcal{P}$ accordingly
2. if $h_1 = h_2$ then output 1 else output 0

Figure 5.17: Collision resistance: simulator $\mathcal{S}$ in Game 5.

Initialization of $\mathcal{S}$

No input, no output

1. set $X = Y = Z = \emptyset$

Simulation of $\mathcal{P}$

Input: $y = (m, a, b, c) \in \mathcal{X}$ from $\mathcal{O}$ (either $\mathcal{A}$ or $\mathcal{S}$)

Output: (a', b')

1. add node y to Y
2. if there exists $y \xrightarrow{\mathcal{F}} x = (m, a', b', c) \in Z$ then return (a', b') to $\mathcal{O}$
3. else
   (a) randomly select $a' \leftarrow \{0,1\}^{\ell_a}$ and $b' \leftarrow \{0,1\}^{\ell_m}$
   (b) add node $x = (m, a', b', c)$ to X and edge $y \xrightarrow{\mathcal{F}} x$ to Z
   (c) if $\exists \tilde{x} \in X$ such that $\mathsf{OutputColl}(x, \tilde{x})$ (event Abort_1) then abort
   (d) if $\exists \tilde{x} \in X$ such that $\mathsf{Coll}_0(x, \tilde{x})$ (event Abort_2) then abort
   (e) if $\exists \tilde{y} \in Y$ such that $\mathsf{Dep}_0(x, \tilde{y})$ (event Abort_3) then abort
   (f) return (a', b') to $\mathcal{O}$

Simulation of $\mathcal{P}^{-1}$

Input: $x = (m, a, b, c) \in \mathcal{X}$ from $\mathcal{A}$

Output: (a', b')

1. add node x to X
2. if there exists $(m, a', b', c) = y \xrightarrow{\mathcal{F}} x \in Z$ then return (a', b') to $\mathcal{A}$
3. else
   (a) randomly select $a' \leftarrow \{0,1\}^{\ell_a}$ and $b' \leftarrow \{0,1\}^{\ell_m}$
   (b) add node $y = (m, a', b', c)$ to Y and edge $y \xrightarrow{\mathcal{F}} x$ to Z
   (c) if $\exists \tilde{x} \in X$ such that $\mathsf{Dep}_0(\tilde{x}, y)$ (event Abort_4) then abort
   (d) return (a', b') to $\mathcal{A}$

Input: $M_1, M_2 \in \{0,1\}^*$ from $\mathcal{A}$

1. compute $h_1 = \mathcal{C}^{\mathcal{P}}(M_1, \ell_m)$ and $h_2 = \mathcal{C}^{\mathcal{P}}(M_2, \ell_m)$ by calling $\mathcal{P}$ accordingly
2. output 0

Figure 5.18: Collision resistance: simulator $\mathcal{S}$ in Game 6 (and final simulator).

Game 6 (final game). We finally complete the modification applied in Game 5 to the case when a connection (see Game 5 above) occurs after a response to $\mathcal{P}^{-1}$ is assigned. When the simulation of $\mathcal{P}^{-1}$ by $\mathcal{S}$ creates a new node y, we make sure that y cannot be connected to a preexisting rooted node $\tilde{x} \in X$. When this happens, $\mathcal{S}$ aborts and we refer to this event as Abort_4. In addition, it is easily seen that no collision is found unless one of the abortion events occurs, so that $\mathcal{S}$ can be modified to always output 0. The new simulator is depicted on Fig. 5.18.

Claim 10. $|\Pr[W_6] - \Pr[W_5]| \leq \Pr[\mathsf{Abort}_4] \leq N(N-1) \cdot 2^{-(\ell_a + \ell_m)}$.

This claim too stems from the results of Section 5.3. Putting it altogether, we get the security bound claimed in Theorem 3.

5.5 Shabal is Preimage Resistant in the Ideal Cipher Model

5.5.1 A Security Model for Preimage Resistance in the ICM

We capture the preimage resistance of construction $\mathcal{C}^{\mathcal{P}}$ as a security game played between an adversary $\mathcal{A}$ and a challenger $\mathcal{V}$.

Definition 9 (PRE Game)**.** *The game is described as follows:*

1. $\mathcal{V}$ *randomly selects* $h \leftarrow \{0,1\}^{\ell_m}$ *and sends* h *to* $\mathcal{A}$
2. $\mathcal{A}$ *makes calls to the ideal cipher* $(\mathcal{P}, \mathcal{P}^{-1})$
3. $\mathcal{A}$ *outputs a message* $M \in \{0,1\}^*$
4. $\mathcal{V}$ *computes* $\mathcal{C}^{\mathcal{P}}(M)$ *by calling* $\mathcal{P}$
5. $\mathcal{V}$ *outputs* 1 *if* $\mathcal{C}^{\mathcal{P}}(M) = h$ *or* 0 *otherwise*

We define the success probability $\mathsf{Suc}^{\mathsf{PRE}}(\mathcal{A}, \mathcal{C})$ of $\mathcal{A}$ as the probability that $\mathcal{V}$ outputs 1 when interacting with $\mathcal{A}$ as per the above game. $\mathsf{Suc}^{\mathsf{PRE}}(\mathcal{A}, \mathcal{C})$ is a function of the total number N of queries received by the ideal cipher $(\mathcal{P}, \mathcal{P}^{-1})$ throughout the game. Note that $\mathcal{V}$ itself has to make $k + \mathsf{E}$ calls to $\mathcal{P}$ when verifying the response M of $\mathcal{A}$ if hashing M leads to the insertion of k message blocks. Thus $\mathcal{A}$ makes $N - (k + \mathsf{E})$ calls to $(\mathcal{P}, \mathcal{P}^{-1})$.

5.5.2 Proving Preimage Resistance for Shabal's Mode of Operation

We conceive a simulator $\mathcal{S}$ that simulates $\mathcal{V}$ and $(\mathcal{P}, \mathcal{P}^{-1})$ towards $\mathcal{A}$. An high-level view of $\mathcal{S}$ is as follows. Simulator $\mathcal{S}$

1. randomly selects $h \leftarrow \{0,1\}^{\ell_m}$
2. sends h to $\mathcal{A}$
3. simulates the ideal cipher $(\mathcal{P}, \mathcal{P}^{-1})$ and may abort while doing so
4. receives $M \in \{0,1\}^*$ from $\mathcal{A}$
5. runs its own simulation of $\mathcal{P}$ to compute $\mathcal{C}^{\mathcal{P}}(M)$
6. outputs 0

Our proof technique is to make the simulation of $\mathcal{P}$ abort upon detection of the event that $\mathcal{C}^{\mathcal{P}}(M) = h$. Therefore either $\mathcal{S}$ outputs 0 or the game aborts. Consequently

$$\mathsf{Suc}^{\mathsf{PRE}}(\mathcal{A}, \mathcal{C}) \leq \Pr[\mathcal{S} \text{ aborts}] \ .$$

We state

Theorem 4 (Preimage resistance). *There exists a simulator $\mathcal{S}$ as above such that for any preimage finder $\mathcal{A}$ playing as per the* PRE *game limited to at most N calls to $(\mathcal{P}, \mathcal{P}^{-1})$,*

$$\Pr[\mathcal{S} \textit{ aborts}] \leq N \cdot 2^{-(\ell_a + \ell_m - \log(\ell_m+1)-2)}$$

and $\mathcal{S}$ runs in time at most $O(N^2)$.

5.5.3 Proof of Theorem 4

We build a sequence of games which starts with the PRE security game and ends with a final simulator $\mathcal{S}$. We first introduce a number of definitions.

Preliminary definitions.

Given a hash graph $\mathcal{G} = (X, Y, Z)$ (see Section 5.3.1 for definitions), recall that by $X^{e,\ell} \subseteq X$ and $Y^{e,\ell} \subseteq Y$ denote the sets of internal states admitting an e-path of length ℓ in $\mathcal{G}$. For a given $h \in \{0,1\}^{\ell_m}$, we define

$$\mathcal{X}[h] = \{(m, a, h, c) \mid (m, a, c) \in \{0,1\}^{\ell_m} \times \{0,1\}^{\ell_a} \times \{0,1\}^{\ell_m}\} .$$

Definition 10 (e-Antipaths). *Let $e \in [1, \mathsf{E}]$. A message block m is said to be an e-antipath (with respect to $h \in \{0,1\}^{\ell_m}$) of index k to state $y \in \mathcal{X}$ if $m \neq 0^{\ell_m}$ and*

$$y \xrightarrow{\mathcal{F}} x^e \overset{m,k}{\rightsquigarrow} y^{e+1} \xrightarrow{\mathcal{F}} x^{e+1} \quad \dots \quad y^{\mathsf{E}-1} \xrightarrow{\mathcal{F}} x^{\mathsf{E}-1} \overset{m,k}{\rightsquigarrow} y^{\mathsf{E}} \xrightarrow{\mathcal{F}} x^{\mathsf{E}}$$

for some $x^e, x^{e+1}, \dots, x^{\mathsf{E}}, y^{e+1}, y^{e+2}, \dots, y^{\mathsf{E}} \in \mathcal{X}$ and $x^{\mathsf{E}} \in \mathcal{X}[h]$. y is said to admit an e-antipath in hash graph $\mathcal{G} = (X, Y, Z)$ if $x^e, \dots, x^{\mathsf{E}} \in X$ and $y, y^{e+1}, \dots, y^{\mathsf{E}} \in Y$.

Definition 11 (0-Antipaths). *Let $y \in \mathcal{X}$. We call 0-antipath (with respect to $h \in \{0,1\}^{\ell_m}$) of index ℓ to y a list of message blocks $\mu = \langle m_{\ell+1}, \dots, m_k \rangle$ such that m_k is a 1-antipath of index k to y_1 and*

$$y \xrightarrow{\mathcal{F}} x_\ell \overset{m_{\ell+1},\ell+1}{\rightsquigarrow} y_{\ell+1} \xrightarrow{\mathcal{F}} x_{\ell+1} \quad \dots \quad y_{k-1} \xrightarrow{\mathcal{F}} x_{k-1} \overset{m_k,k}{\rightsquigarrow} y_k \xrightarrow{\mathcal{F}} x_k \overset{m_k,k}{\rightsquigarrow} y^1$$

for some $x_\ell, \dots, x_k, y_{\ell+1}, \dots, y_k, y^1 \in \mathcal{X}$. y is said to admit a 0-antipath in hash graph $\mathcal{G} = (X, Y, Z)$ if all intermediate states along the antipath are nodes of $\mathcal{G}$.

Let $h \in \{0,1\}^{\ell_m}$ and $\mathcal{G}$ a hash graph. By extension to the above, we will say that an e-antipath of index ℓ to $y \in Y$ is also an e-antipath of index ℓ to $x \in X$ if $y \xrightarrow{\mathcal{F}} x \in Z$. We will denote by $Y_{e,\ell} \subseteq Y$ and $X_{e,\ell} \subseteq X$ the subsets of nodes of $\mathcal{G}$ admitting an e-antipath of index ℓ.

Intuition of the proof.

The preimage finder has no other choice than exploring the hash graph in the hope to form an E-path connecting x_0 to a final state $x^{\mathsf{E}} \in \mathcal{X}[h]$. Regardless of the specific strategies $\mathcal{A}$ may adopt to do so, such a path can only be created by connecting an e-path to a compatible e-antipath. $\mathcal{A}$ makes use of calls to $\mathcal{P}$ to create or lengthen paths and calls to $\mathcal{P}^{-1}$ to create or lengthen antipaths, until two of them (one of each kind) eventually connect. We build our simulator to monitor such attempts and abort the game whenever a path/antipath connection is likely to occur.

The sequence of games.

We now proceed to construct the sequence of games leading to the security bound claimed in Theorem 4.

Game 0. This is the original PRE game where $\mathcal{A}$ interacts with $\mathcal{V}$ and the ideal cipher $\mathcal{P}$. Let W_0 denote the event that $\mathcal{V}$ outputs 1 in Game 0, and more generally W_i the event that $\mathcal{V}$ outputs 1 in Game i. By definition of Game 0, we have

$$\Pr[W_0] = \mathsf{Suc}^{\mathsf{PRE}}(\mathcal{A}, \mathcal{C}) .$$

Game 1. We replace $\mathcal{V}$ by a first simulator $\mathcal{S}$ which behaves as $\mathcal{V}$ and forwards calls to $(\mathcal{P}, \mathcal{P}^{-1})$. In addition, $\mathcal{S}$ keeps track of the definitions of $\mathcal{P}$ by maintaining the graph $\mathcal{G}$ as depicted on Fig. 5.19. The action of $\mathcal{S}$ does not modify the view of $\mathcal{D}$, meaning that $\Pr[W_1] = \Pr[W_0]$.

Initialization of $\mathcal{S}$

No input, no output

1. randomly select $h \leftarrow \{0,1\}^{\ell_m}$
2. set $X = Y = Z = \emptyset$

Simulation of $\mathcal{P}$

Input: $y = (m, a, b, c) \in \mathcal{X}$ from $\mathcal{O}$ (either $\mathcal{A}$ or $\mathcal{S}$)

Output: (a', b')

1. add node y to Y
2. call $\mathcal{P}$ to get $(a', b') = \mathcal{P}(m, a, b, c)$
3. add node $x = (m, a', b', c)$ to X and edge $y \xrightarrow{\mathcal{F}} x$ to Z
4. return (a', b') to $\mathcal{O}$

Simulation of $\mathcal{P}^{-1}$

Input: $x = (m, a, b, c) \in \mathcal{X}$ from $\mathcal{A}$

Output: (a', b')

1. add node x to X
2. call $\mathcal{P}^{-1}$ to get $(a', b') = \mathcal{P}^{-1}(m, a, b, c)$
3. add node $y = (m, a', b', c)$ to Y and edge $y \xrightarrow{\mathcal{F}} x$ to Z
4. return (a', b') to $\mathcal{A}$

Completion of $\mathcal{S}$

Input: $M \in \{0,1\}^*$ from $\mathcal{A}$

1. compute $\mathcal{C}^{\mathcal{P}}(M, \ell_m)$ by calling $\mathcal{P}$ accordingly
2. if $\mathcal{C}^{\mathcal{P}}(M, \ell_m) = h$ then output 1 else output 0

Figure 5.19: Preimage resistance: simulator $\mathcal{S}$ in Game 1.

Game 2. We slightly modify our simulator to get rid of the ideal cipher $\mathcal{P}$ and replace it with a perfect simulation. Every time $\mathcal{S}$ needs to define $\mathcal{P}(y)$ for some $y \in \mathcal{X}$ or $\mathcal{P}^{-1}(x)$ for $x \in \mathcal{X}$, $\mathcal{S}$ randomly selects the response. We depict the new simulator on Fig. 5.20. This does not modify the distributions since $\mathcal{P}$ is an ideal cipher. Hence $\Pr[W_2] = \Pr[W_1]$.

Game 3. We now insert an early-abort condition as follows. $\mathcal{S}$ creates two collections of sets $\{Y[\![\beta]\!] \mid \beta \in \{0,1\}^{\ell_m}\}$ and $\{Y\langle\!\langle\gamma\rangle\!\rangle \mid \gamma \in \{0,1\}^{\ell_m}\}$ where all sets are initially empty at the beginning of the game. Each time $\mathcal{A}$ sends a query $x = (m, a, b, c)$ to $\mathcal{P}^{-1}$, $\mathcal{S}$ selects a response state $y = (m, a', b', c)$ as in Game 2; however, $\mathcal{S}$ now adds y to sets $Y[\![b' \boxminus m]\!]$ and $Y\langle\!\langle b'\rangle\!\rangle$. This operation amounts to sort response states output by the simulation of $\mathcal{P}^{-1}$ according to the two mappings $(m, a', b', c) \mapsto b' \boxminus m$ and $(m, a', b', c) \mapsto b'$. When adding y to $Y[\![b' \boxminus m]\!]$ and $Y\langle\!\langle b'\rangle\!\rangle$, $\mathcal{S}$ checks that these two sets have a limited number B of elements and aborts if this is no longer

Initialization of $\mathcal{S}$

No input, no output

1. randomly select $h \leftarrow \{0,1\}^{\ell_m}$
2. set $X = Y = Z = \emptyset$

Simulation of $\mathcal{P}$

Input: $y = (m, a, b, c) \in \mathcal{X}$ from $\mathcal{O}$ (either $\mathcal{A}$ or $\mathcal{S}$)

Output: (a', b')

1. add node y to Y
2. if there exists $y \xrightarrow{\mathcal{F}} x = (m, a', b', c) \in Z$ then return (a', b') to $\mathcal{O}$
3. else
   (a) randomly select $a' \leftarrow \{0,1\}^{\ell_a}$ and $b' \leftarrow \{0,1\}^{\ell_m}$
   (b) add node $x = (m, a', b', c)$ to X and edge $y \xrightarrow{\mathcal{F}} x$ to Z
   (c) return (a', b') to $\mathcal{O}$

Simulation of $\mathcal{P}^{-1}$

Input: $x = (m, a, b, c) \in \mathcal{X}$ from $\mathcal{A}$

Output: (a', b')

1. add node x to X
2. if there exists $(m, a', b', c) = y \xrightarrow{\mathcal{F}} x \in Z$ then return (a', b') to $\mathcal{A}$
3. else
   (a) randomly select $a' \leftarrow \{0,1\}^{\ell_a}$ and $b' \leftarrow \{0,1\}^{\ell_m}$
   (b) add node $y = (m, a', b', c)$ to Y and edge $y \xrightarrow{\mathcal{F}} x$ to Z
   (c) return (a', b') to $\mathcal{A}$

Completion of $\mathcal{S}$

Input: $M \in \{0,1\}^*$ from $\mathcal{A}$

1. compute $\mathcal{C}^{\mathcal{P}}(M, \ell_m)$ by calling $\mathcal{P}$ accordingly
2. if $\mathcal{C}^{\mathcal{P}}(M, \ell_m) = h$ then output 1 else output 0

Figure 5.20: Preimage resistance: simulator $\mathcal{S}$ in Game 2.

the case. This is as depicted on Fig. 5.21. The bound B is an optimization parameter which will be determined later in the proof.

Initialization of $\mathcal{S}$

No input, no output

1. randomly select $h \leftarrow \{0,1\}^{\ell_m}$
2. set $X = Y = Z = \emptyset$

Simulation of $\mathcal{P}$

Input: $y = (m, a, b, c) \in \mathcal{X}$ from $\mathcal{O}$ (either $\mathcal{A}$ or $\mathcal{S}$)

Output: (a', b')

1. add node y to Y
2. if there exists $y \xrightarrow{\mathcal{F}} x = (m, a', b', c) \in Z$ then return (a', b') to $\mathcal{O}$
3. else
   (a) randomly select $a' \leftarrow \{0,1\}^{\ell_a}$ and $b' \leftarrow \{0,1\}^{\ell_m}$
   (b) add node $x = (m, a', b', c)$ to X and edge $y \xrightarrow{\mathcal{F}} x$ to Z
   (c) return (a', b') to $\mathcal{O}$

Simulation of $\mathcal{P}^{-1}$

Input: $x = (a, b, c) \in \mathcal{X}$ from $\mathcal{A}$

Output: (a', b')

1. add node x to X
2. if there exists $(m, a', b', c) = y \xrightarrow{\mathcal{F}} x \in Z$ then return (a', b') to $\mathcal{A}$
3. else
   (a) randomly select $a' \leftarrow \{0,1\}^{\ell_a}$ and $b' \leftarrow \{0,1\}^{\ell_m}$
   (b) add node $y = (m, a', b', c)$ to Y and edge $y \xrightarrow{\mathcal{F}} x$ to Z
   (c) add node y to $Y[\![b' \boxminus m]\!]$ and $Y\langle\!\langle b' \rangle\!\rangle$
   (d) if $|Y[\![b' \boxminus m]\!]| > B$ or $|Y\langle\!\langle b' \rangle\!\rangle| > B$ (event Abort_1) then abort
   (e) return (a', b') to $\mathcal{A}$

Completion of $\mathcal{S}$

Input: $M \in \{0,1\}^*$ from $\mathcal{A}$

1. compute $\mathcal{C}^{\mathcal{P}}(M, \ell_m)$ by calling $\mathcal{P}$ accordingly
2. if $\mathcal{C}^{\mathcal{P}}(M, \ell_m) = h$ then output 1 else output 0

Figure 5.21: Preimage resistance: simulator $\mathcal{S}$ in Game 3.

We state:

Claim 11. *One has* $|\Pr[W_3] - \Pr[W_2]| \leq \Pr[\mathsf{Abort}_1] \leq \frac{2}{(B+1)!} N^{B+1} 2^{-\ell_m \cdot B}$.

Proof. We recall the following result on multi-collisions [25]: if one picks N random values in $\{0,1\}^{\ell_m}$, the probability that the same value is selected at most B times is

$$\exp\left(-\frac{N^{B+1}}{(B+1)!\,(2^{\ell_m})^B}\right).$$

When assigning the response state $y = (m, a', b', c)$ to $\mathcal{P}^{-1}(x)$, $\mathcal{S}$ uniformly selects b'. This results in that y will be added to $Y[\![\beta]\!]$ for a randomly distributed $\beta \in \{0,1\}^{\ell_m}$. Similarly, y will also be

added to $Y\langle\!\langle\gamma\rangle\!\rangle$ for a randomly distributed $\gamma \in \{0,1\}^{\ell_m}$. This random experiment takes place at most N times throughout the execution of $\mathcal{S}$ resulting in that

$$\left.\begin{array}{r}\Pr\left[\max_\beta |Y[\![\beta]\!](N)| > B\right] \\ \Pr\left[\max_\gamma |Y\langle\!\langle\gamma\rangle\!\rangle(N)| > B\right]\end{array}\right\} = 1 - e^{-\frac{N^{B+1}}{(B+1)!2^{\ell_m \cdot B}}} \leq \frac{N^{B+1}}{(B+1)!2^{\ell_m \cdot B}} ,$$

so that

$$\Pr\left[\mathsf{Abort}_1\right] \leq \Pr\left[\max_\beta |Y[\![\beta]\!](N)| > B\right] + \Pr\left[\max_\gamma |Y\langle\!\langle\gamma\rangle\!\rangle(N)| > B\right]$$

which directly gives the claimed bound. □

Property 3. *Unless $\mathcal{S}$ aborts, $\max_\beta |Y[\![\beta]\!](q)| \leq B$ and $\max_\gamma |Y\langle\!\langle\gamma\rangle\!\rangle(q)| \leq B$ for any $q \in [1, N]$.*

Game 4. We now make sure that $\mathcal{A}$ cannot create an E-path from x_0 to some $x \in \mathcal{X}[h]$ when sending a query to $\mathcal{P}$ during the game. To this end, we define the following predicate.

Definition 12 (Predicate $\mathsf{Connect}$)**.** *Let $\mathcal{G} = (X, Y, Z)$ be the current graph and $(x, y) \in X \times Y$. We define $\mathsf{Connect}(x, y) = \mathsf{Connect}_0(x, y) \vee \mathsf{Connect}_1(x, y)$ where*

- *$\mathsf{Connect}_0(x, y)$ evaluates to True if and only if for some $\ell \in \mathbb{N}$,*

$$x \in X^{0,\ell-1} , \quad y \in Y_{0,\ell+1} \quad \text{and} \quad x \overset{\star,\ell}{\rightsquigarrow} y$$

- *$\mathsf{Connect}_1(x, y)$ evaluates to True if and only if for some $e \in [1, \mathsf{E}]$, $\ell \in \mathbb{N}$ and $m \neq 0^{\ell_m}$,*

$$x \in X^{e,\ell} , \quad y \in Y_{e,\ell} \quad \text{and} \quad x \overset{m,\ell}{\rightsquigarrow} y$$

  where m is an e-antipath of index ℓ to y and also the last block of an e-path of length ℓ to x.

We modify simulator $\mathcal{S}$ to detect a connection whenever a new output value $x \in \mathcal{X}$ is assigned to $\mathcal{P}(y)$ and abort when $\mathsf{Connect}(x, \tilde{y})$ is true for some preexisting $\tilde{y}$. We refer to this event as Abort_2. The upgraded simulator is depicted on Fig. 5.22.

Claim 12. *One has $|\Pr[W_4] - \Pr[W_3]| \leq \Pr[\mathsf{Abort}_2] \leq 2 \cdot N \cdot B \cdot 2^{-(\ell_a+\ell_m)}$.*

Proof. Obviously, $\Pr[W_4 \wedge \neg\mathsf{Abort}_2] = \Pr[W_3 \wedge \neg\mathsf{Abort}_2]$ so that the Difference Lemma applies. Let us consider the q-th query $y = (m, a, b, c)$. Let us first assume that $y \in Y^{0,\ell-1}$ for some integer $\ell \geq 0$. Let $\tilde{y} = (\tilde{m}, \tilde{a}, \tilde{b}, \tilde{c}) \in Y_{0,\ell+1}(q-1)$ be fixed and recall that $D(m, c) = \{(m, a', b', c) \mid (a', b') \in \{0,1\}^{\ell_a} \times \{0,1\}^{\ell_m}\}$. Then taking probabilities over $x \leftarrow D(m, c)$, one gets

$$\begin{aligned}\Pr\left[x \overset{\star,\ell}{\rightsquigarrow} \tilde{y}\right] &= \Pr\left[\exists\, \mathfrak{m} \in \{0,1\}^{\ell_m} : \mathsf{Insert}[\mathfrak{m}, \ell](x) = \tilde{y}\right] \\ &= \Pr\left[\exists\, \mathfrak{m} \in \{0,1\}^{\ell_m} : \begin{array}{rcl} \mathfrak{m} &=& \tilde{m} \\ a' \oplus (\ell) &=& \tilde{a} \\ c \boxminus m \boxplus \mathfrak{m} &=& \tilde{b} \\ b' &=& \tilde{c} \end{array}\right] \\ &\leq 2^{-\ell_a} \cdot 2^{-\ell_m} \cdot \delta\left[c \boxminus m = \tilde{b} \boxminus \tilde{m}\right] .\end{aligned}$$

Therefore

$$\begin{aligned}\Pr\left[\exists \tilde{y} \in Y_{0,\ell+1}(q-1) \text{ with } x \overset{\star,\ell}{\rightsquigarrow} \tilde{y}\right] &\leq 2^{-\ell_a} \cdot 2^{-\ell_m} \cdot |Y_{0,\ell+1}(q-1) \cap Y[\![c \boxminus m]\!](q-1)| \\ &\leq 2^{-\ell_a} \cdot 2^{-\ell_m} \cdot |Y[\![c \boxminus m]\!](q-1)| \\ &\leq 2^{-\ell_a} \cdot 2^{-\ell_m} \cdot \max_\beta |Y[\![\beta]\!](q-1)| \leq B \cdot 2^{-(\ell_a+\ell_m)} .\end{aligned}$$

Initialization of $\mathcal{S}$

No input, no output

1. randomly select $h \leftarrow \{0,1\}^{\ell_m}$
2. set $X = Y = Z = \emptyset$

Simulation of $\mathcal{P}$

Input: $y = (m, a, b, c) \in \mathcal{X}$ from $\mathcal{O}$ (either $\mathcal{A}$ or $\mathcal{S}$)

Output: (a', b')

1. add node y to Y
2. if there exists $y \xrightarrow{\mathcal{F}} x = (m, a', b', c) \in Z$ then return (a', b') to $\mathcal{O}$
3. else
   (a) randomly select $a' \leftarrow \{0,1\}^{\ell_a}$ and $b' \leftarrow \{0,1\}^{\ell_m}$
   (b) add node $x = (m, a', b', c)$ to X and edge $y \xrightarrow{\mathcal{F}} x$ to Z
   (c) if $\exists \tilde{y} \in Y$ such that $\mathsf{Connect}(x, \tilde{y})$ (event Abort_2) then abort
   (d) return (a', b') to $\mathcal{O}$

Simulation of $\mathcal{P}^{-1}$

Input: $x = (m, a, b, c) \in \mathcal{X}$ from $\mathcal{A}$

Output: (a', b')

1. add node x to X
2. if there exists $(m, a', b', c) = y \xrightarrow{\mathcal{F}} x \in Z$ then return (a', b') to $\mathcal{A}$
3. else
   (a) randomly select $a' \leftarrow \{0,1\}^{\ell_a}$ and $b' \leftarrow \{0,1\}^{\ell_m}$
   (b) add node $y = (m, a', b', c)$ to Y and edge $y \xrightarrow{\mathcal{F}} x$ to Z
   (c) add node y to $Y[\![b' \boxminus m]\!]$ and $Y\langle\!\langle b' \rangle\!\rangle$
   (d) if $|Y[\![b' \boxminus m]\!]| > B$ or $|Y\langle\!\langle b' \rangle\!\rangle| > B$ (event Abort_1) then abort
   (e) return (a', b') to $\mathcal{A}$

Completion of $\mathcal{S}$

Input: $M \in \{0,1\}^*$ from $\mathcal{A}$

1. compute $\mathcal{C}^{\mathcal{P}}(M, \ell_m)$ by calling $\mathcal{P}$ accordingly
2. if $\mathcal{C}^{\mathcal{P}}(M, \ell_m) = h$ then output 1 else output 0

Figure 5.22: Preimage resistance: simulator $\mathcal{S}$ in Game 3.

Let us now consider the case where $y \in Y^{e,\ell}$ for some $e \in [1, \mathsf{E}]$ and $\ell \in \mathbb{N}$. Hence the last block of any e-path to y must be m. We now fix $\tilde{y} = (\tilde{m}, \tilde{a}, \tilde{b}, \tilde{c}) \in Y_{e,\ell}(q-1)$ and it holds that

$$\Pr\left[x \overset{m,\ell}{\rightsquigarrow} \tilde{y}\right] = \Pr\left[\begin{array}{rcl} m & = & \tilde{m} \\ a' \oplus (\ell) & = & \tilde{a} \\ c & = & \tilde{b} \\ b' & = & \tilde{c} \end{array}\right] \leq 2^{-\ell_a} \cdot 2^{-\ell_m} \cdot \delta\left[c = \tilde{b}\right] .$$

Hence

$$\begin{array}{rcl} \Pr\left[\exists \tilde{y} \in Y_{e,\ell}(q-1) \text{ with } x \overset{m,\ell}{\rightsquigarrow} \tilde{y}\right] & \leq & 2^{-\ell_a} \cdot 2^{-\ell_m} \cdot |Y_{e,\ell}(q-1) \cap Y\langle\langle c \rangle\rangle(q-1)| \\ & \leq & 2^{-\ell_a} \cdot 2^{-\ell_m} \cdot |Y\langle\langle c \rangle\rangle(q-1)| \\ & \leq & 2^{-\ell_a} \cdot 2^{-\ell_m} \cdot \max_{\gamma} |Y\langle\langle \gamma \rangle\rangle(q-1)| \leq B \cdot 2^{-(\ell_a+\ell_m)} . \end{array}$$

Overall, one must have

$$\Pr\left[\mathsf{Abort}_2(q)\right] = \Pr\left[\exists \tilde{y} \in Y(q-1) \text{ with } \mathsf{Connect}(x, \tilde{y})\right] \leq 2 \cdot B \cdot 2^{-(\ell_a+\ell_m)}$$

so that

$$\Pr\left[\mathsf{Abort}_2\right] \leq \sum_{q=1}^{N} \Pr\left[\mathsf{Abort}_2(q)\right] \leq \sum_{q=1}^{N} 2 \cdot B \cdot 2^{-(\ell_a+\ell_m)} = 2 \cdot N \cdot B \cdot 2^{-(\ell_a+\ell_m)}$$

as claimed. □

Property 4. *Unless $\mathcal{S}$ aborts, the treatment of a request $y \in \mathcal{X}$ to $\mathcal{P}$ by $\mathcal{S}$ can by no means create a connection between a path and an antipath.*

Game 5. We now ascertain that $\mathcal{A}$ is unable to create an E-path from x_0 to some $x \in \mathcal{X}[h]$ by sending adaptively chosen queries to $\mathcal{P}^{-1}$ during the game. We proceed in two steps and start by inserting a new abort condition. $\mathcal{S}$ creates a collection of sets $\{X[\![\lambda]\!] \mid \lambda \in \{0,1\}^{\ell_m}\}$ where all sets are set to $\emptyset$ at the beginning of the game. For each query $y = (m, a, b, c)$ that $\mathcal{A}$ sends to $\mathcal{P}$, $\mathcal{S}$ assigns a response state $x = (m, a', b', c)$ as in Game 4; however, $\mathcal{S}$ now adds x to sets $X[\![b']\!]$. When adding x to $X[\![b']\!]$, $\mathcal{S}$ checks that $|X[\![b']\!]| \leq B$ and aborts if this is not the case: this event is referred to as Abort_3. This is as shown on Fig. 5.23.

Claim 13. *One has* $|\Pr[W_5] - \Pr[W_4]| \leq \Pr[\mathsf{Abort}_3] \leq \frac{1}{(B+1)!} N^{B+1} 2^{-\ell_m \cdot B}$.

Proof. We invoke the same argument based on multi-collisions as in the study of Abort_1. When $\mathcal{S}$ assigns the response state $x = (m, a', b', c)$ to $\mathcal{P}(y)$, $\mathcal{S}$ uniformly selects b' which results in that x will be added to $X[\![\lambda]\!]$ for a randomly distributed $\lambda \in \{0,1\}^{\ell_m}$. Thus,

$$\Pr\left[\max_{\lambda} |X[\![\lambda]\!](N)| > B\right] = 1 - e^{-\frac{N^{B+1}}{(B+1)!2^{\ell_m \cdot B}}} \leq \frac{N^{B+1}}{(B+1)!2^{\ell_m \cdot B}} ,$$

so that

$$\Pr\left[\mathsf{Abort}_3\right] \leq \Pr\left[\max_{\lambda} |X[\![\lambda]\!](N)| > B\right]$$

which provides the desired bound. □

Property 5. *Unless $\mathcal{S}$ aborts,* $\max_{\lambda} |X[\![\lambda]\!](q)| \leq B$ *for any* $q \in [1, N]$.

Initialization of $\mathcal{S}$

No input, no output

1. randomly select $h \leftarrow \{0,1\}^{\ell_m}$
2. set $X = Y = Z = \emptyset$

Simulation of $\mathcal{P}$

Input: $y = (m, a, b, c) \in \mathcal{X}$ from $\mathcal{O}$ (either $\mathcal{A}$ or $\mathcal{S}$)

Output: (a', b')

1. add node y to Y
2. if there exists $y \xrightarrow{\mathcal{F}} x = (m, a', b', c) \in Z$ then return (a', b') to $\mathcal{O}$
3. else
   (a) randomly select $a' \leftarrow \{0,1\}^{\ell_a}$ and $b' \leftarrow \{0,1\}^{\ell_m}$
   (b) add node $x = (m, a', b', c)$ to X and edge $y \xrightarrow{\mathcal{F}} x$ to Z
   (c) if $\exists \tilde{y} \in Y$ such that $\mathsf{Connect}(x, \tilde{y})$ (event Abort_2) then abort
   (d) add node x to $X[\![b']\!]$
   (e) if $|X[\![b']\!]| > B$ (event Abort_3) then abort
   (f) return (a', b') to $\mathcal{O}$

Simulation of $\mathcal{P}^{-1}$

Input: $x = (m, a, b, c) \in \mathcal{X}$ from $\mathcal{A}$

Output: (a', b')

1. add node x to X
2. if there exists $(m, a', b', c) = y \xrightarrow{\mathcal{F}} x \in Z$ then return (a', b') to $\mathcal{A}$
3. else
   (a) randomly select $a' \leftarrow \{0,1\}^{\ell_a}$ and $b' \leftarrow \{0,1\}^{\ell_m}$
   (b) add node $y = (m, a', b', c)$ to Y and edge $y \xrightarrow{\mathcal{F}} x$ to Z
   (c) add node y to $Y[\![b' \boxminus m]\!]$ and $Y\langle\!\langle b' \rangle\!\rangle$
   (d) if $|Y[\![b' \boxminus m]\!]| > B$ or $|Y\langle\!\langle b' \rangle\!\rangle| > B$ (event Abort_1) then abort
   (e) return (a', b') to $\mathcal{A}$

Completion of $\mathcal{S}$

Input: $M \in \{0,1\}^*$ from $\mathcal{A}$

1. compute $\mathcal{C}^{\mathcal{P}}(M, \ell_m)$ by calling $\mathcal{P}$ accordingly
2. if $\mathcal{C}^{\mathcal{P}}(M, \ell_m) = h$ then output 1 else output 0

Figure 5.23: Preimage resistance: simulator $\mathcal{S}$ in Game 5.

Initialization of $\mathcal{S}$

No input, no output

1. randomly select $h \leftarrow \{0,1\}^{\ell_m}$
2. set $X = Y = Z = \emptyset$

Simulation of $\mathcal{P}$

Input: $y = (m, a, b, c) \in \mathcal{X}$ from $\mathcal{O}$ (either $\mathcal{A}$ or $\mathcal{S}$)

Output: (a', b')

1. add node y to Y
2. if there exists $y \xrightarrow{\mathcal{F}} x = (m, a', b', c) \in Z$ then return (a', b') to $\mathcal{O}$
3. else
   (a) randomly select $a' \leftarrow \{0,1\}^{\ell_a}$ and $b' \leftarrow \{0,1\}^{\ell_m}$
   (b) add node $x = (m, a', b', c)$ to X and edge $y \xrightarrow{\mathcal{F}} x$ to Z
   (c) if $\exists \tilde{y} \in Y$ such that $\mathsf{Connect}(x, \tilde{y})$ (event Abort_2) then abort
   (d) add node x to $X[\![b']\!]$
   (e) if $|X[\![b']\!]| > B$ (event Abort_3) then abort
   (f) return (a', b') to $\mathcal{O}$

Simulation of $\mathcal{P}^{-1}$

Input: $x = (m, a, b, c) \in \mathcal{X}$ from $\mathcal{A}$

Output: (a', b')

1. add node x to X
2. if there exists $(m, a', b', c) = y \xrightarrow{\mathcal{F}} x \in Z$ then return (a', b') to $\mathcal{A}$
3. else
   (a) randomly select $a' \leftarrow \{0,1\}^{\ell_a}$ and $b' \leftarrow \{0,1\}^{\ell_m}$
   (b) add node $y = (m, a', b', c)$ to Y and edge $y \xrightarrow{\mathcal{F}} x$ to Z
   (c) add node y to $Y[\![b' \boxminus m]\!]$ and $Y\langle\!\langle b' \rangle\!\rangle$
   (d) if $|Y[\![b' \boxminus m]\!]| > B$ or $|Y\langle\!\langle b' \rangle\!\rangle| > B$ (event Abort_1) then abort
   (e) if $\exists \tilde{x} \in X$ such that $\mathsf{Connect}(\tilde{x}, y)$ (event Abort_4) then abort
   (f) return (a', b') to $\mathcal{A}$

Completion of $\mathcal{S}$

Input: $M \in \{0,1\}^*$ from $\mathcal{A}$

1. output 0

Figure 5.24: Preimage resistance: simulator $\mathcal{S}$ of Game 6 (and final simulator).

Game 6 (final game). We modify simulator $\mathcal{S}$ to detect a connection whenever a new output value $y \in \mathcal{X}$ is assigned to $\mathcal{P}^{-1}(x)$ and abort when $\mathsf{Connect}(\tilde{x}, y)$ is true for some preexisting $\tilde{x}$. We refer to this event as Abort_4. As shown below, the four abortion events introduced in this game and earlier games cover all cases leading to the creation of an E-path in the hash graph. As a result, the final outcome of $\mathcal{S}$ can be safely modified to make $\mathcal{S}$ return a systematic 0. The upgraded simulator is depicted on Fig. 5.24.

Claim 14. *One has* $|\Pr[W_6] - \Pr[W_5]| \leq \Pr[\mathsf{Abort}_4] \leq 2 \cdot N \cdot B \cdot 2^{-(\ell_a+\ell_m)}$.

Proof. We apply the Difference Lemma again. We consider the q-th query $x = (m, a, b, c)$ to $\mathcal{P}^{-1}$. Let us first assume that $x \in X_{0,\ell+1}$ for some integer $\ell \geq 0$. Let $\tilde{x} = (\tilde{m}, \tilde{a}, \tilde{b}, \tilde{c}) \in X^{0,\ell-1}(q-1)$ be fixed and recall that $D(m,c) = \{(m, a', b', c) \mid (a', b') \in \{0,1\}^{\ell_a} \times \{0,1\}^{\ell_m}\}$. Then taking probabilities over $y \leftarrow D(m,c)$, one gets

$$\begin{aligned}
\Pr\left[\tilde{x} \overset{\star,\ell}{\rightsquigarrow} y\right] &= \Pr\left[\exists\, \mathfrak{m} \in \{0,1\}^{\ell_m} : \mathsf{Insert}[\mathfrak{m}, \ell](\tilde{x}) = y\right] \\
&= \Pr\left[\exists\, \mathfrak{m} \in \{0,1\}^{\ell_m} : \begin{array}{rcl} \mathfrak{m} &=& m \\ \tilde{a} \oplus (\ell) &=& a' \\ \tilde{c} \boxminus \tilde{m} \boxplus \mathfrak{m} &=& b' \\ \tilde{b} &=& c \end{array}\right] \\
&\leq 2^{-\ell_a} \cdot 2^{-\ell_m} \cdot \delta\left[c = \tilde{b}\right] .
\end{aligned}$$

Therefore

$$\begin{aligned}
\Pr\left[\exists \tilde{x} \in X_{0,\ell-1}(q-1) \text{ with } \tilde{x} \overset{\star,\ell}{\rightsquigarrow} y\right] &\leq 2^{-\ell_a} \cdot 2^{-\ell_m} \cdot |X_{0,\ell-1}(q-1) \cap X[\![c]\!](q-1)| \\
&\leq 2^{-\ell_a} \cdot 2^{-\ell_m} \cdot |X[\![c]\!](q-1)| \\
&\leq 2^{-\ell_a} \cdot 2^{-\ell_m} \cdot \max_{\lambda} |X[\![\lambda]\!](q-1)| \leq B \cdot 2^{-(\ell_a+\ell_m)} .
\end{aligned}$$

Let us now consider the case where $x \in X_{e,\ell}$ for some $e \in [1, \mathsf{E}]$ and $\ell \in \mathbb{N}$ and let $m \neq 0^{\ell_m}$ be an e-antipath to x. Fixing $\tilde{x} = (\tilde{m}, \tilde{a}, \tilde{b}, \tilde{c}) \in Y^{e,\ell}(q-1)$, one gets

$$\Pr\left[\tilde{x} \overset{\tilde{m},\ell}{\rightsquigarrow} y\right] = \Pr\left[\begin{array}{rcl} \tilde{m} &=& m \\ \tilde{a} \oplus (\ell) &=& a' \\ \tilde{c} &=& b' \\ \tilde{b} &=& c \end{array}\right] \leq 2^{-\ell_a} \cdot 2^{-\ell_m} \cdot \delta\left[c = \tilde{b}\right] .$$

Hence

$$\begin{aligned}
\Pr\left[\exists \tilde{x} \in X^{e,\ell}(q-1) \text{ with } \tilde{x} \overset{\tilde{m},\ell}{\rightsquigarrow} y\right] &\leq 2^{-\ell_a} \cdot 2^{-\ell_m} \cdot |X^{e,\ell}(q-1) \cap X[\![c]\!](q-1)| \\
&\leq 2^{-\ell_a} \cdot 2^{-\ell_m} \cdot |X[\![c]\!](q-1)| \\
&\leq 2^{-\ell_a} \cdot 2^{-\ell_m} \cdot \max_{\lambda} |X[\![\lambda]\!](q-1)| \leq B \cdot 2^{-(\ell_a+\ell_m)} .
\end{aligned}$$

Therefore one has

$$\Pr[\mathsf{Abort}_4(q)] = \Pr[\exists \tilde{x} \in X(q-1) \text{ with } \mathsf{Connect}(\tilde{x}, y)] \leq 2 \cdot B \cdot 2^{-(\ell_a+\ell_m)}$$

so that

$$\Pr[\mathsf{Abort}_4] \leq \sum_{q=1}^{N} \Pr[\mathsf{Abort}_4(q)] \leq \sum_{q=1}^{N} 2 \cdot B \cdot 2^{-(\ell_a+\ell_m)} = 2 \cdot N \cdot B \cdot 2^{-(\ell_a+\ell_m)}$$

as wanted. □

Property 6. *Unless $\mathcal{S}$ aborts, requests to $\mathcal{P}^{-1}$ treated by $\mathcal{S}$ can by no means create a connection between a path and an antipath.*

Conclusion. Summing up, we get the upper bound

$$\begin{aligned}\Pr[\mathcal{S} \text{ aborts}] &\leq \frac{3}{(B+1)!} \cdot N^{B+1} \cdot 2^{-\ell_m \cdot B} + 4 \cdot N \cdot B \cdot 2^{-(\ell_a+\ell_m)} \\ &\leq 4 \cdot \frac{N}{2^{\ell_a+\ell_m}} \cdot \left(B + \frac{N \cdot 2^{\ell_a}}{(B+1)!} \left(\frac{N}{2^{\ell_m}} \right)^{B-1} \right) = f(\ell_a, \ell_m, N, B) \ .\end{aligned}$$

We now choose a particular value for B as a function of ℓ_a, ℓ_m and N based on the following observations: when $N/2^{\ell_m} \leq 1/2$, we see that by setting $B = \ell_m$, the second term is upper bounded by

$$\frac{N \cdot 2^{\ell_a}}{(B+1)!} \left(\frac{N}{2^{\ell_m}} \right)^{B-1} < \frac{2^{\ell_a}}{(\ell_m+1)!} < 1$$

for values of ℓ_a in the practical range $64 \leq \ell_a \leq 1024$ and $\ell_m = 512$. Then

$$\Pr[\mathcal{S} \text{ aborts}] < 4 \cdot \frac{N}{2^{\ell_a+\ell_m}} \cdot (\ell_m + 1) = N \cdot 2^{-(\ell_a+\ell_m-\log(\ell_m+1)-2)}$$

thereby giving a security margin of $\ell_a - \log(\ell_m + 1) - 2$ bits of security.

5.6 Shabal is Second Preimage Resistant in the Ideal Cipher Model

5.6.1 Capturing Second Preimage Resistance in the ICM

Adapting the security model of previous sections, we now consider second preimage resistance in the same spirit. We define a security game played between an adversary $\mathcal{A}$ and a challenger $\mathcal{V}$. The game is described as follows.

Definition 13 (SP Game)**.** *The security game takes as input a parameter $\kappa \in \mathbb{N}$.*

1. *$\mathcal{V}$ randomly selects a κ-bit message $M^* \leftarrow \{0,1\}^\kappa$ and sends M^* to $\mathcal{A}$*
2. *$\mathcal{A}$ makes a series of calls to the ideal cipher $(\mathcal{P}, \mathcal{P}^{-1})$*
3. *$\mathcal{A}$ outputs a message $M \in \{0,1\}^*$*
4. *$\mathcal{V}$ computes $\mathcal{C}^{\mathcal{P}}(M)$ by calling $\mathcal{P}$*
5. *$\mathcal{V}$ outputs 1 if $\mathcal{C}^{\mathcal{P}}(M) = \mathcal{C}^{\mathcal{P}}(M^*)$ or 0 otherwise*

The success probability $\mathsf{Suc}^{\mathsf{SP}}(\mathcal{A}, \mathcal{C})$ of $\mathcal{A}$ is the probability that $\mathcal{V}$ outputs 1 when interacting with $\mathcal{A}$ as per the SP game. In addition to Shabal's parameters, $\mathsf{Suc}^{\mathsf{SP}}(\mathcal{A}, \mathcal{C})$ depends on the total number N of queries received by the ideal cipher $(\mathcal{P}, \mathcal{P}^{-1})$ throughout the game. Define k^* as the block length of M^* *i.e.,* the number of ℓ_m-bit blocks required to encode the padded input message (hence $k^* = \lceil (\kappa+1)/\ell_m \rceil$). Assume that the block length of M is k; then note that $\mathcal{V}$ itself has to make $k + \mathsf{E}$ calls to $\mathcal{P}$ when verifying the response M of $\mathcal{A}$. Thus in total $\mathcal{A}$ makes $N - (k + \mathsf{E})$ calls to $(\mathcal{P}, \mathcal{P}^{-1})$.

5.6.2 Proving Second Preimage Resistance for Shabal's Mode of Operation

We build a simulator $\mathcal{S}$ that simulates $\mathcal{V}$ and $(\mathcal{P}, \mathcal{P}^{-1})$ towards $\mathcal{A}$ in the spirit of the previous section: our simulator $\mathcal{S}$

1. randomly selects $M^* \leftarrow \{0,1\}^\kappa$

2. runs its own simulation of $\mathcal{P}$ to compute $\mathcal{C}^{\mathcal{P}}(M^*)$
3. sends M^* to $\mathcal{A}$
4. simulates the ideal cipher $(\mathcal{P}, \mathcal{P}^{-1})$ towards $\mathcal{A}$; the simulation may provoke the abortion of $\mathcal{S}$
5. receives $M \in \{0,1\}^*$ from $\mathcal{A}$
6. runs its own simulation of $\mathcal{P}$ to compute $\mathcal{C}^{\mathcal{P}}(M)$
7. outputs 0

Our proof technique is to make the simulation of $\mathcal{P}$ abort the game upon detection of the event that $\mathcal{C}^{\mathcal{P}}(M) = \mathcal{C}^{\mathcal{P}}(M^*)$. Consequently either $\mathcal{S}$ outputs 0 or the game aborts and

$$\mathsf{Suc}^{\mathsf{SP}}(\mathcal{A}, \mathcal{C}) \leq \Pr[\mathcal{S} \text{ aborts}] \ .$$

Theorem 5 (Second preimage resistance)**.** *There exists a simulator $\mathcal{S}$ as above such that for any second preimage finder $\mathcal{A}$ playing as per the* SP *game limited to at most N calls to $(\mathcal{P}, \mathcal{P}^{-1})$,*

$$\Pr[\mathcal{S} \textit{ aborts}] \leq 2 \cdot N \cdot 2^{-(\ell_a + \ell_m - \log k^*)}$$

and $\mathcal{S}$ runs in time at most $O(N^2)$.

5.6.3 Proof of Theorem 5

We prove Theorem 4 using game hopping, starting with the SP security game and ending with a full simulator. We make use of the definitions introduced in Section 5.5 and will refer to it for notation.

Intuition of the proof.

Let $m_1^*, \ldots, m_k^*$ be the sequence of inserted message blocks arising from the hashing of $M^* \leftarrow \{0,1\}^\kappa$. Again, the security game between the adversary $\mathcal{A}$ and the simulator $\mathcal{S}$ amounts to detecting certain events which may or may not occur while the hash graph is evolving dynamically. In a nutshell, the second preimage finder $\mathcal{A}$ has only two means to construct a second preimage $M \in \{0,1\}^*$:

(i) a preimage finding approach: $\mathcal{A}$ attempts to connect a path and an antipath with respect to the target output $h = \mathcal{C}^{\mathcal{P}}(M^*)$. This is independent from the input message blocks $m_1^*, \ldots, m_k^*$;

(ii) or by connecting paths or antipaths to nodes hanging from the *challenge path i.e.,* the path created by $\mathcal{C}^{\mathcal{P}}(M^*)$. Doing such connections depends on the values of $m_1^*, \ldots, m_k^*$.

Of course, $\mathcal{A}$ may combine the two approaches into some integrated strategy; we will show that whatever this strategy is, its success probability is upper bounded by the sum of two bounds, the bound on (i) stemming from the proof of preimage resistance provided in the previous section, and a bound on (ii) which we explicit in this section. Again, we build a simulator that detects (i) and (ii) and aborts the game whenever a winning connection is likely to occur. For completeness, the abortion events related to (i) are described and properly discussed in the sequence of games that follows, even though they appear unchanged from Section 5.5.

The sequence of games.

We now proceed to construct the sequence of games leading to the security bound claimed in Theorem 5.

Game 0. This is the original SP game where $\mathcal{A}$ interacts with $\mathcal{V}$ and the ideal cipher $\mathcal{P}$. Let W_0 denote the event that $\mathcal{V}$ outputs 1 in Game 0, and more generally W_i the event that $\mathcal{V}$ outputs 1 in Game i. By definition of Game 0, we have

$$\Pr[W_0] = \mathsf{Suc}^{\mathsf{SP}}(\mathcal{A}, \mathcal{C}) .$$

Game 1. We replace $\mathcal{V}$ by a first simulator $\mathcal{S}$ which behaves as $\mathcal{V}$ and forwards calls to $(\mathcal{P}, \mathcal{P}^{-1})$. $\mathcal{S}$ also constructs the hash graph $\mathcal{G}$ as depicted on Fig. 5.25. The action of $\mathcal{S}$ does not modify the view of $\mathcal{D}$, meaning that $\Pr[W_1] = \Pr[W_0]$.

Initialization of $\mathcal{S}$

No input, no output

1. randomly select $M^* \leftarrow \{0,1\}^\kappa$
2. set $X = Y = Z = \emptyset$

Simulation of $\mathcal{P}$

Input: $y = (m, a, b, c) \in \mathcal{X}$ from $\mathcal{O}$ (either $\mathcal{A}$ or $\mathcal{S}$)

Output: (a', b')

1. add node y to Y
2. call $\mathcal{P}$ to get $(a', b') = \mathcal{P}(m, a, b, c)$
3. add node $x = (m, a', b', c)$ to X and edge $y \xrightarrow{\mathcal{F}} x$ to Z
4. return (a', b') to $\mathcal{O}$

Simulation of $\mathcal{P}^{-1}$

Input: $x = (m, a, b, c) \in \mathcal{X}$ from $\mathcal{A}$

Output: (a', b')

1. add node x to X
2. call $\mathcal{P}^{-1}$ to get $(a', b') = \mathcal{P}^{-1}(m, a, b, c)$
3. add node $y = (m, a', b', c)$ to Y and edge $y \xrightarrow{\mathcal{F}} x$ to Z
4. return (a', b') to $\mathcal{A}$

Completion of $\mathcal{S}$

Input: $M \in \{0,1\}^*$ from $\mathcal{A}$

1. compute $\mathcal{C}^{\mathcal{P}}(M, \ell_m)$ by calling $\mathcal{P}$ accordingly
2. if $\mathcal{C}^{\mathcal{P}}(M, \ell_m) = \mathcal{C}^{\mathcal{P}}(M, \ell_m)$ then output 1 else output 0

Figure 5.25: Second preimage resistance: simulator $\mathcal{S}$ in Game 1.

Game 2. We now modify $\mathcal{S}$ to eliminate the ideal cipher $\mathcal{P}$. Every time $\mathcal{S}$ needs to define $\mathcal{P}(y)$ for some $y \in \mathcal{X}$ or $\mathcal{P}^{-1}(x)$ for $x \in \mathcal{X}$, $\mathcal{S}$ randomly selects the response. We depict the new simulator on Fig. 5.26. In addition, $\mathcal{S}$ now computes $h = \mathcal{C}^{\mathcal{P}}(M^*)$ prior to transmit M^* to $\mathcal{A}$ so that the hash graph contains the E-path $\mu^* = \langle m_1^*, \ldots, m_k^* \rangle$ connecting x_0 to some final state $\in \mathcal{X}[h]$. This does not modify the distributions since $\mathcal{P}$ is an ideal cipher. Hence $\Pr[W_2] = \Pr[W_1]$.

Game 3. $\mathcal{S}$ now aborts when $\mathcal{A}$ succeeds in connecting a path to the challenge path μ^*. Let

$$x_0 \overset{m_1^*,1}{\rightsquigarrow} y_1^* \xrightarrow{\mathcal{F}} x_1^* \overset{m_2^*,2}{\rightsquigarrow} y_2^* \quad \ldots \quad \overset{m_{k^*}^*,k^*}{\rightsquigarrow} y_{k^*}^* \xrightarrow{\mathcal{F}} x_{k^*}^* \overset{m_{k^*}^*,k^*}{\rightsquigarrow} y_{k^*+1}^* \xrightarrow{\mathcal{F}} x_{k^*+1}^* \quad \ldots \quad \overset{m_{k^*}^*,k^*}{\rightsquigarrow} y_{k^*+\mathsf{E}}^* \xrightarrow{\mathcal{F}} x_{k^*+\mathsf{E}}^*$$

be the challenge path in the hash graph $\mathcal{G}$ *i.e.,* the sequence of internal states reached by the operating mode when computing $\mathcal{C}^{\mathcal{P}}(M^*)$. We define the following predicate.

Initialization of $\mathcal{S}$

No input, no output

1. randomly select $M^* \leftarrow \{0,1\}^\kappa$
2. set $X = Y = Z = \emptyset$
3. compute $\mathcal{C}^{\mathcal{P}}(M^*) = h$ using the simulation of $\mathcal{P}$ below

---

Simulation of $\mathcal{P}$

Input: $y = (m, a, b, c) \in \mathcal{X}$ from $\mathcal{O}$ (either $\mathcal{A}$ or $\mathcal{S}$)

Output: (a', b')

1. add node y to Y
2. if there exists $y \xrightarrow{\mathcal{F}} x = (m, a', b', c) \in Z$ then return (a', b') to $\mathcal{O}$
3. else
   (a) randomly select $a' \leftarrow \{0,1\}^{\ell_a}$ and $b' \leftarrow \{0,1\}^{\ell_m}$
   (b) add node $x = (m, a', b', c)$ to X and edge $y \xrightarrow{\mathcal{F}} x$ to Z
   (c) return (a', b') to $\mathcal{O}$

Simulation of $\mathcal{P}^{-1}$

Input: $x = (m, a, b, c) \in \mathcal{X}$ from $\mathcal{A}$

Output: (a', b')

1. add node x to X
2. if there exists $(m, a', b', c) = y \xrightarrow{\mathcal{F}} x \in Z$ then return (a', b') to $\mathcal{A}$
3. else
   (a) randomly select $a' \leftarrow \{0,1\}^{\ell_a}$ and $b' \leftarrow \{0,1\}^{\ell_m}$
   (b) add node $y = (m, a', b', c)$ to Y and edge $y \xrightarrow{\mathcal{F}} x$ to Z
   (c) return (a', b') to $\mathcal{A}$

Completion of $\mathcal{S}$

Input: $M \in \{0,1\}^*$ from $\mathcal{A}$

1. compute $\mathcal{C}^{\mathcal{P}}(M, \ell_m)$ by calling $\mathcal{P}$ accordingly
2. if $\mathcal{C}^{\mathcal{P}}(M, \ell_m) = h$ then output 1 else output 0

Figure 5.26: Second preimage resistance: simulator $\mathcal{S}$ in Game 2.

Definition 14 (Predicate $\mathsf{ConnectChallenge}$). *Let $\mathcal{G} = (X, Y, Z)$ be the current graph and let $x \in X$. The Boolean predicate $\mathsf{ConnectChallenge}(x)$ evaluates to True if and only if for some $\ell \in [0, k^* - 2]$, $x \in X^{0,\ell}$ and $x \overset{\star,\ell+1}{\rightsquigarrow} y^*_{\ell+1}$.*

We modify $\mathcal{S}$ to detect that $\mathsf{ConnectChallenge}(x)$ is realized for some response state x output by the simulation of $\mathcal{P}$, in which case $\mathcal{S}$ aborts. We refer to this abortion event as Abort_1. $\mathcal{S}$ is shown on Fig. 5.27.

Initialization of $\mathcal{S}$

No input, no output

1. randomly select $M^* \leftarrow \{0,1\}^\kappa$
2. set $X = Y = Z = \emptyset$
3. compute $\mathcal{C}^{\mathcal{P}}(M^*) = h$ using the simulation of $\mathcal{P}$ below

---

Simulation of $\mathcal{P}$

Input: $y = (m, a, b, c) \in \mathcal{X}$ from $\mathcal{O}$ (either $\mathcal{A}$ or $\mathcal{S}$)

Output: (a', b')

1. add node y to Y
2. if there exists $y \xrightarrow{\mathcal{F}} x = (m, a', b', c) \in Z$ then return (a', b') to $\mathcal{O}$
3. else
   (a) randomly select $a' \leftarrow \{0,1\}^{\ell_a}$ and $b' \leftarrow \{0,1\}^{\ell_m}$
   (b) add node $x = (m, a', b', c)$ to X and edge $y \xrightarrow{\mathcal{F}} x$ to Z
   (c) if $\mathsf{ConnectChallenge}(x)$ (event Abort_1) then abort
   (d) return (a', b') to $\mathcal{O}$

Simulation of $\mathcal{P}^{-1}$

Input: $x = (a, b, c) \in \mathcal{X}$ from $\mathcal{A}$

Output: (a', b')

1. add node x to X
2. if there exists $(m, a', b', c) = y \xrightarrow{\mathcal{F}} x \in Z$ then return (a', b') to $\mathcal{A}$
3. else
   (a) randomly select $a' \leftarrow \{0,1\}^{\ell_a}$ and $b' \leftarrow \{0,1\}^{\ell_m}$
   (b) add node $y = (m, a', b', c)$ to Y and edge $y \xrightarrow{\mathcal{F}} x$ to Z
   (c) return (a', b') to $\mathcal{A}$

Completion of $\mathcal{S}$

Input: $M \in \{0,1\}^*$ from $\mathcal{A}$

1. compute $\mathcal{C}^{\mathcal{P}}(M, \ell_m)$ by calling $\mathcal{P}$ accordingly
2. if $\mathcal{C}^{\mathcal{P}}(M, \ell_m) = h$ then output 1 else output 0

Figure 5.27: Second preimage resistance: simulator $\mathcal{S}$ in Game 3.

We state:

Claim 15. *One has $|\Pr[W_3] - \Pr[W_2]| \leq \Pr[\mathsf{Abort}_1] \leq N \cdot (k^* - 1) \cdot 2^{-(\ell_a + \ell_m)}$.*

Proof. Let $y = (m, a, b, c) \in Y$ be a query to $\mathcal{P}$ and suppose y is the q-th query to $(\mathcal{P}, \mathcal{P}^{-1})$ for $q \in [1, N]$. Considering the set $D(m, c)$ of all possible response states $x = (m, a', b', c)$, let us assume that $y \in Y^{0,\ell}$ for some $\ell \in [0, k^* - 2]$. Noting $y^*_{\ell+1} = (m^*_{\ell+1}, a^*_{\ell+1}, b^*_{\ell+1}, c^*_{\ell+1})$ and taking

the probabilities over the random choice $x \leftarrow D(m,c)$, we get that

$$\begin{aligned}
\Pr\left[x \overset{\star,\ell+1}{\rightsquigarrow} y^*_{\ell+1}\right] &= \Pr\left[\exists\, \mathfrak{m} \in \{0,1\}^{\ell_m} : \mathsf{Insert}[\mathfrak{m}, \ell+1](x) = y^*_{\ell+1}\right] \\
&= \Pr\left[\exists\, \mathfrak{m} \in \{0,1\}^{\ell_m} : \begin{array}{rcl} \mathfrak{m} &=& m^*_{\ell+1} \\ a' \oplus (\ell+1) &=& a^*_{\ell+1} \\ c \boxminus m \boxplus \mathfrak{m} &=& b^*_{\ell+1} \\ b' &=& c^*_{\ell+1} \end{array}\right] \\
&\leq 2^{-\ell_a} \cdot 2^{-\ell_m} \cdot \delta\left[c \boxminus m = b^*_{\ell+1} \boxminus m^*_{\ell+1}\right] \leq 2^{-(\ell_a+\ell_m)} .
\end{aligned}$$

Hence

$$\begin{aligned}
\Pr\left[\mathsf{Abort}_1(q)\right] = \Pr\left[\mathsf{ConnectChallenge}(x)\right] &\leq \sum_{\ell \in [0,k^*-2], y \in Y^{0,\ell}} \Pr\left[x \overset{\star,\ell+1}{\rightsquigarrow} y^*_{\ell+1}\right] \\
&\leq \left|\{\ell \in [0,k^*-2] : y \in Y^{0,\ell}\}\right| \cdot 2^{-(\ell_a+\ell_m)} \\
&\leq (k^*-1) \cdot 2^{-(\ell_a+\ell_m)} .
\end{aligned}$$

Overall,

$$\Pr\left[\mathsf{Abort}_1\right] \leq \sum_{q=1}^{N} \Pr\left[\mathsf{Abort}_1(q)\right] \leq N \cdot (k^*-1) \cdot 2^{-(\ell_a+\ell_m)}$$

which concludes the proof. □

Game 4. We now make $\mathcal{S}$ abort when $\mathcal{A}$ succeeds in connecting an antipath (with respect to h) to the challenge path μ^*. We define the predicate $\mathsf{ConnectChallenge}$ to nodes $y \in Y$ as follows.

Definition 15. *Let $\mathcal{G} = (X,Y,Z)$ be the current graph and let $y \in Y$. $\mathsf{ConnectChallenge}(y)$ evaluates to True if and only if for some $\ell \in [1,k^*-1]$, $y \in Y_{0,\ell}$ and $x^*_{\ell-1} \overset{\star,\ell}{\rightsquigarrow} y$.*

We modify $\mathcal{S}$ to detect that $\mathsf{ConnectChallenge}(y)$ is realized for some response state y output by the simulation of $\mathcal{P}^{-1}$, in which case $\mathcal{S}$ aborts. We refer to this abortion event as Abort_2. The new simulator is depicted on Fig. 5.28.

We state:

Claim 16. *One has $\left|\Pr[W_4] - \Pr[W_3]\right| \leq \Pr[\mathsf{Abort}_2] \leq N \cdot (k^*-1) \cdot 2^{-(\ell_a+\ell_m)}$.*

Proof. Let $x = (m,a,b,c) \in Y$ be a query to $\mathcal{P}^{-1}$ and suppose that x is the q-th query to $(\mathcal{P},\mathcal{P}^{-1})$ for $q \in [1,N]$. Considering the set $D(m,c)$ of all possible response states $y = (m,a',b',c)$, let us assume that $x \in X_{0,\ell}$ for some $\ell \in [1,k^*-1]$. Noting $x^*_{\ell-1} = (m^*_{\ell-1}, a^*_{\ell-1}, b^*_{\ell-1}, c^*_{\ell-1})$ and taking the probabilities over the random choice $x \leftarrow D(m,c)$, we get that

$$\begin{aligned}
\Pr\left[x^*_{\ell-1} \overset{\star,\ell}{\rightsquigarrow} y\right] &= \Pr\left[\exists\, \mathfrak{m} \in \{0,1\}^{\ell_m} : \mathsf{Insert}[\mathfrak{m}, \ell](x^*_{\ell-1}) = y\right] \\
&= \Pr\left[\exists\, \mathfrak{m} \in \{0,1\}^{\ell_m} : \begin{array}{rcl} \mathfrak{m} &=& m \\ a^*_{\ell-1} \oplus (\ell) &=& a' \\ c^*_{\ell-1} \boxminus m^*_{\ell-1} \boxplus \mathfrak{m} &=& b' \\ b^*_{\ell-1} &=& c \end{array}\right] \\
&\leq 2^{-\ell_a} \cdot 2^{-\ell_m} \cdot \delta\left[c = b^*_{\ell-1}\right] \leq 2^{-(\ell_a+\ell_m)} .
\end{aligned}$$

Hence

$$\begin{aligned}
\Pr\left[\mathsf{Abort}_2(q)\right] = \Pr\left[\mathsf{ConnectChallenge}(y)\right] &\leq \sum_{\ell \in [1,k^*-1], x \in X_{0,\ell}} \Pr\left[x^*_{\ell-1} \overset{\star,\ell}{\rightsquigarrow} y\right] \\
&\leq \left|\{\ell \in [1,k^*-1] : x \in X_{0,\ell}\}\right| \cdot 2^{-(\ell_a+\ell_m)} \\
&\leq (k^*-1) \cdot 2^{-(\ell_a+\ell_m)} .
\end{aligned}$$

Initialization of $\mathcal{S}$

No input, no output

1. randomly select $M^* \leftarrow \{0,1\}^\kappa$
2. set $X = Y = Z = \emptyset$
3. compute $\mathcal{C}^{\mathcal{P}}(M^*) = h$ using the simulation of $\mathcal{P}$ below

---

Simulation of $\mathcal{P}$

Input: $y = (m, a, b, c) \in \mathcal{X}$ from $\mathcal{O}$ (either $\mathcal{A}$ or $\mathcal{S}$)

Output: (a', b')

1. add node y to Y
2. if there exists $y \xrightarrow{\mathcal{F}} x = (m, a', b', c) \in Z$ then return (a', b') to $\mathcal{O}$
3. else
   (a) randomly select $a' \leftarrow \{0,1\}^{\ell_a}$ and $b' \leftarrow \{0,1\}^{\ell_m}$
   (b) add node $x = (m, a', b', c)$ to X and edge $y \xrightarrow{\mathcal{F}} x$ to Z
   (c) if ConnectChallenge(x) (event Abort$_1$) then abort
   (d) return (a', b') to $\mathcal{O}$

Simulation of $\mathcal{P}^{-1}$

Input: $x = (a, b, c) \in \mathcal{X}$ from $\mathcal{A}$

Output: (a', b')

1. add node x to X
2. if there exists $(m, a', b', c) = y \xrightarrow{\mathcal{F}} x \in Z$ then return (a', b') to $\mathcal{A}$
3. else
   (a) randomly select $a' \leftarrow \{0,1\}^{\ell_a}$ and $b' \leftarrow \{0,1\}^{\ell_m}$
   (b) add node $y = (m, a', b', c)$ to Y and edge $y \xrightarrow{\mathcal{F}} x$ to Z
   (c) if ConnectChallenge(y) (event Abort$_2$) then abort
   (d) return (a', b') to $\mathcal{A}$

Completion of $\mathcal{S}$

Input: $M \in \{0,1\}^*$ from $\mathcal{A}$

1. compute $\mathcal{C}^{\mathcal{P}}(M, \ell_m)$ by calling $\mathcal{P}$ accordingly
2. if $\mathcal{C}^{\mathcal{P}}(M, \ell_m) = h$ then output 1 else output 0

Figure 5.28: Second preimage resistance: simulator $\mathcal{S}$ in Game 4.

Finally,

$$\Pr\left[\mathsf{Abort}_2\right] \leq \sum_{q=1}^{N} \Pr\left[\mathsf{Abort}_2(q)\right] \leq N \cdot (k^* - 1) \cdot 2^{-(\ell_a + \ell_m)}$$

as announced. □

Game 5. We now insert an early-abort condition as follows. $\mathcal{S}$ creates two collections of sets $\{Y[\![\beta]\!] \mid \beta \in \{0,1\}^{\ell_m}\}$ and $\{Y\langle\!\langle\gamma\rangle\!\rangle \mid \gamma \in \{0,1\}^{\ell_m}\}$ where all sets are initially empty at the beginning of the game. Each time $\mathcal{A}$ sends a query $x = (m, a, b, c)$ to $\mathcal{P}^{-1}$, $\mathcal{S}$ selects a response state $y = (m, a', b', c)$ as in Game 2; however, $\mathcal{S}$ now adds y to sets $Y[\![b' \boxminus m]\!]$ and $Y\langle\!\langle b'\rangle\!\rangle$. This operation amounts to sort response states output by the simulation of $\mathcal{P}^{-1}$ according to the two mappings $(m, a', b', c) \mapsto b' \boxminus m$ and $(m, a', b', c) \mapsto b'$. When adding y to $Y[\![b' \boxminus m]\!]$ and $Y\langle\!\langle b'\rangle\!\rangle$, $\mathcal{S}$ checks that these two sets have a limited number B of elements and aborts if this is no longer the case. This is as depicted on Fig. 5.29. The bound B is an optimization parameter which is later determined as in the proof of Theorem 4.

We state:

Claim 17. *One has* $|\Pr\left[W_5\right] - \Pr\left[W_4\right]| \leq \Pr\left[\mathsf{Abort}_3\right] \leq \frac{2}{(B+1)!} N^{B+1} 2^{-\ell_m \cdot B}$.

The proof is identical to the proof provided in the previous section.

Property 7. *Unless $\mathcal{S}$ aborts,* $\max_\beta |Y[\![\beta]\!](q)| \leq B$ *and* $\max_\gamma |Y\langle\!\langle\gamma\rangle\!\rangle(q)| \leq B$ *for any* $q \in [1, N]$.

Game 6. We now make sure that $\mathcal{A}$ cannot create an E-path from x_0 to some $x \in \mathcal{X}[h]$ when sending a query to $\mathcal{P}$ during the game. We reuse to the predicate $\mathsf{Connect}$ defined in Section 5.6. We modify simulator $\mathcal{S}$ to detect a connection whenever a new output value $x \in \mathcal{X}$ is assigned to $\mathcal{P}(y)$ and abort when $\mathsf{Connect}(x, \tilde{y})$ is true for some preexisting $\tilde{y}$. We refer to this event as Abort_4. The upgraded simulator is depicted on Fig. 5.30.

Claim 18. *One has* $|\Pr\left[W_6\right] - \Pr\left[W_5\right]| \leq \Pr\left[\mathsf{Abort}_4\right] \leq 2 \cdot N \cdot B \cdot 2^{-(\ell_a + \ell_m)}$.

The proof is identical to the one provided in Section 5.5.

Property 8. *Unless $\mathcal{S}$ aborts, the treatment of a request to $\mathcal{P}$ by $\mathcal{S}$ can by no means create a connection between a path and an antipath.*

Game 7. We now ascertain that $\mathcal{A}$ is unable to create an E-path from x_0 to some $x \in \mathcal{X}[h]$ by sending adaptively chosen queries to $\mathcal{P}^{-1}$ during the game. We proceed in two steps and start by inserting a new abort condition. $\mathcal{S}$ creates a collection of sets $\{X[\![\lambda]\!] \mid \lambda \in \{0,1\}^{\ell_m}\}$ where all sets are set to $\emptyset$ at the beginning of the game. For each query $y = (m, a, b, c)$ that $\mathcal{A}$ sends to $\mathcal{P}$, $\mathcal{S}$ assigns a response state $x = (m, a', b', c)$ as in Game 6; however, $\mathcal{S}$ now adds x to sets $X[\![b']\!]$. When adding x to $X[\![b']\!]$, $\mathcal{S}$ checks that $|X[\![b']\!]| \leq B$ and aborts if this is not the case: this event is referred to as Abort_5.

Claim 19. *One has* $|\Pr\left[W_7\right] - \Pr\left[W_6\right]| \leq \Pr\left[\mathsf{Abort}_5\right] \leq \frac{1}{(B+1)!} N^{B+1} 2^{-\ell_m \cdot B}$.

The proof has already been given in Section 5.5. We claim the following property.

Property 9. *Unless $\mathcal{S}$ aborts,* $\max_\lambda |X[\![\lambda]\!](q)| \leq B$ *for any* $q \in [1, N]$.

Game 8 (final game). We modify simulator $\mathcal{S}$ to detect a connection whenever a new output value $y \in \mathcal{X}$ is assigned to $\mathcal{P}^{-1}(x)$ and abort when $\mathsf{Connect}(\tilde{x}, y)$ is true for some preexisting $\tilde{x}$. We refer to this event as Abort_6. The upgraded simulator is depicted on Fig. 5.31.

Claim 20. *One has* $|\Pr\left[W_8\right] - \Pr\left[W_7\right]| \leq \Pr\left[\mathsf{Abort}_6\right] \leq 2 \cdot N \cdot B \cdot 2^{-(\ell_a + \ell_m)}$.

The proof is similar to the one of Section 5.5. We claim:

Property 10. *Unless $\mathcal{S}$ aborts, requests to $\mathcal{P}^{-1}$ treated by $\mathcal{S}$ can by no means create a connection between a path and an antipath.*

Initialization of $\mathcal{S}$

No input, no output

1. randomly select $M^* \leftarrow \{0,1\}^\kappa$
2. set $X = Y = Z = \emptyset$
3. compute $\mathcal{C}^{\mathcal{P}}(M^*) = h$ using the simulation of $\mathcal{P}$ below

---

Simulation of $\mathcal{P}$

Input: $y = (m, a, b, c) \in \mathcal{X}$ from $\mathcal{O}$ (either $\mathcal{A}$ or $\mathcal{S}$)

Output: (a', b')

1. add node y to Y
2. if there exists $y \xrightarrow{\mathcal{F}} x = (m, a', b', c) \in Z$ then return (a', b') to $\mathcal{O}$
3. else
   (a) randomly select $a' \leftarrow \{0,1\}^{\ell_a}$ and $b' \leftarrow \{0,1\}^{\ell_m}$
   (b) add node $x = (m, a', b', c)$ to X and edge $y \xrightarrow{\mathcal{F}} x$ to Z
   (c) if ConnectChallenge(x) (event Abort$_1$) then abort
   (d) return (a', b') to $\mathcal{O}$

Simulation of $\mathcal{P}^{-1}$

Input: $x = (a, b, c) \in \mathcal{X}$ from $\mathcal{A}$

Output: (a', b')

1. add node x to X
2. if there exists $(m, a', b', c) = y \xrightarrow{\mathcal{F}} x \in Z$ then return (a', b') to $\mathcal{A}$
3. else
   (a) randomly select $a' \leftarrow \{0,1\}^{\ell_a}$ and $b' \leftarrow \{0,1\}^{\ell_m}$
   (b) add node $y = (m, a', b', c)$ to Y and edge $y \xrightarrow{\mathcal{F}} x$ to Z
   (c) if ConnectChallenge(y) (event Abort$_2$) then abort
   (d) add node y to $Y[\![b' \boxminus m]\!]$ and $Y\langle\!\langle b' \rangle\!\rangle$
   (e) if $|Y[\![b' \boxminus m]\!]| > B$ or $|Y\langle\!\langle b' \rangle\!\rangle| > B$ (event Abort$_3$) then abort
   (f) return (a', b') to $\mathcal{A}$

Completion of $\mathcal{S}$

Input: $M \in \{0,1\}^*$ from $\mathcal{A}$

1. compute $\mathcal{C}^{\mathcal{P}}(M, \ell_m)$ by calling $\mathcal{P}$ accordingly
2. if $\mathcal{C}^{\mathcal{P}}(M, \ell_m) = h$ then output 1 else output 0

Figure 5.29: Second preimage resistance: simulator $\mathcal{S}$ in Game 5.

Initialization of $\mathcal{S}$

No input, no output

1. randomly select $M^* \leftarrow \{0,1\}^\kappa$
2. set $X = Y = Z = \emptyset$
3. compute $\mathcal{C}^{\mathcal{P}}(M^*) = h$ using the simulation of $\mathcal{P}$ below

Simulation of $\mathcal{P}$

Input: $y = (m, a, b, c) \in \mathcal{X}$ from $\mathcal{O}$ (either $\mathcal{A}$ or $\mathcal{S}$)

Output: (a', b')

1. add node y to Y
2. if there exists $y \xrightarrow{\mathcal{F}} x = (m, a', b', c) \in Z$ then return (a', b') to $\mathcal{O}$
3. else
   (a) randomly select $a' \leftarrow \{0,1\}^{\ell_a}$ and $b' \leftarrow \{0,1\}^{\ell_m}$
   (b) add node $x = (m, a', b', c)$ to X and edge $y \xrightarrow{\mathcal{F}} x$ to Z
   (c) if ConnectChallenge(x) (event Abort$_1$) then abort
   (d) if $\exists \tilde{y} \in Y$ such that Connect$(x, \tilde{y})$ (event Abort$_4$) then abort
   (e) return (a', b') to $\mathcal{O}$

Simulation of $\mathcal{P}^{-1}$

Input: $x = (m, a, b, c) \in \mathcal{X}$ from $\mathcal{A}$

Output: (a', b')

1. add node x to X
2. if there exists $(m, a', b', c) = y \xrightarrow{\mathcal{F}} x \in Z$ then return (a', b') to $\mathcal{A}$
3. else
   (a) randomly select $a' \leftarrow \{0,1\}^{\ell_a}$ and $b' \leftarrow \{0,1\}^{\ell_m}$
   (b) add node $y = (m, a', b', c)$ to Y and edge $y \xrightarrow{\mathcal{F}} x$ to Z
   (c) if ConnectChallenge(y) (event Abort$_2$) then abort
   (d) add node y to $Y[\![b' \boxminus m]\!]$ and $Y\langle\!\langle b' \rangle\!\rangle$
   (e) if $|Y[\![b' \boxminus m]\!]| > B$ or $|Y\langle\!\langle b' \rangle\!\rangle| > B$ (event Abort$_3$) then abort
   (f) return (a', b') to $\mathcal{A}$

Completion of $\mathcal{S}$

Input: $M \in \{0,1\}^*$ from $\mathcal{A}$

1. compute $\mathcal{C}^{\mathcal{P}}(M, \ell_m)$ by calling $\mathcal{P}$ accordingly
2. if $\mathcal{C}^{\mathcal{P}}(M, \ell_m) = h$ then output 1 else output 0

Figure 5.30: Second preimage resistance: simulator $\mathcal{S}$ in Game 6.

Initialization of $\mathcal{S}$

No input, no output

1. randomly select $M^* \leftarrow \{0,1\}^\kappa$
2. set $X = Y = Z = \emptyset$
3. compute $\mathcal{C}^{\mathcal{P}}(M) = h$ using the simulation of $\mathcal{P}$ below

---

Simulation of $\mathcal{P}$

Input: $y = (a, b, c) \in \mathcal{X}$ from $\mathcal{O}$ (either $\mathcal{A}$ or $\mathcal{S}$)

Output: b'

1. add node y to Y
2. if there exists $y \xrightarrow{\mathcal{F}} x = (m, a', b', c) \in Z$ then return (a', b') to $\mathcal{O}$
3. else
   (a) randomly select $a' \leftarrow \{0,1\}^{\ell_a}$ and $b' \leftarrow \{0,1\}^{\ell_m}$
   (b) add node $x = (m, a', b', c)$ to X and edge $y \xrightarrow{\mathcal{F}} x$ to Z
   (c) if ConnectChallenge(x) (event Abort$_1$) then abort
   (d) if $\exists \tilde{y} \in Y$ such that Connect$(x, \tilde{y})$ (event Abort$_4$) then abort
   (e) add node x to $X[\![b']\!]$
   (f) if $|X[\![b']\!]| > B$ (event Abort$_5$) then abort
   (g) return (a', b') to $\mathcal{O}$

Simulation of $\mathcal{P}^{-1}$

Input: $x = (a, b, c) \in \mathcal{X}$ from $\mathcal{A}$

Output: b'

1. add node x to X
2. if there exists $(m, a', b', c) = y \xrightarrow{\mathcal{F}} x \in Z$ then return (a', b') to $\mathcal{A}$
3. else
   (a) randomly select $a' \leftarrow \{0,1\}^{\ell_a}$ and $b' \leftarrow \{0,1\}^{\ell_m}$
   (b) add node $y = (m, a', b', c)$ to Y and edge $y \xrightarrow{\mathcal{F}} x$ to Z
   (c) if ConnectChallenge(y) (event Abort$_2$) then abort
   (d) add node y to $Y[\![b' \boxminus m]\!]$ and $Y\langle\!\langle b' \rangle\!\rangle$
   (e) if $|Y[\![b' \boxminus m]\!]| > B$ or $|Y\langle\!\langle b' \rangle\!\rangle| > B$ (event Abort$_3$) then abort
   (f) if $\exists \tilde{x} \in X$ such that Connect$(\tilde{x}, y)$ (event Abort$_6$) then abort
   (g) return (a', b') to $\mathcal{A}$

Completion of $\mathcal{S}$

Input: $M \in \{0,1\}^*$ from $\mathcal{A}$

1. compute $\mathcal{C}^{\mathcal{P}}(M, \ell_m)$ by calling $\mathcal{P}$ accordingly
2. output 0

Figure 5.31: Second preimage resistance: final simulator $\mathcal{S}$.

Conclusion. Summing up, we get the bound

$$\Pr\left[\mathcal{S} \text{ aborts}\right] \leq \frac{3}{(B+1)!} N^{B+1} 2^{-\ell_m \cdot B} + 4NB \cdot 2^{-(\ell_a+\ell_m)} + 2 \cdot N \cdot (k^* - 1) \cdot 2^{-(\ell_a+\ell_m)} \ .$$

The first two terms are optimized as in Section 5.5. Noting that once optimized they remain smaller than

$$2 \cdot N \cdot 2^{-(\ell_a+\ell_m)} \ ,$$

we finally get the bound claimed in Theorem 5.

Chapter 6

Weakened Versions of Shabal

Contents

In order to simplify the analysis of our hash function, we propose several weakened versions of Shabal, with names of the Weakinson-XXX form. The weaknesses that might be found on these variants may or may not teach us some things about the full hash function, depending on the techniques used in the attacks. Most of the variants we propose consist in removing the nonlinearity sources depicted in Section 4.8.

Even if we have tried to simplify the cryptanalyst's work, we may have not taken into account some simplifications that would be interesting to study. In case, we encourage the interested reader to consider other variants of Shabal, as far as they do follow the fundamental basis of its design.

6.1 With Smaller Words

First proposed four variants simply consider that words are no more 32-bit words, but respectively 1-, 4-, 8- and 16-bit words. We therefore name these variants as Weakinson-1bit, Weakinson-4bit, Weakinson-8bit and Weakinson-16bit. Amongst these reduced versions, the 1-bit variant is much weaker, as many of the Shabal operations would be meaningless in this context (*e.g.,* the bit rotations, $\mathcal{U}(x) = 3 \times x \bmod 2^{32}$, $\mathcal{V}(x) = 5 \times x \bmod 2^{32}$, additions).

Definitions of the variants Weakinson-1bit, Weakinson-4bit, Weakinson-8bit and Weakinson-16bit strictly follows the standard definition (see Section 2.3), except that operations that were modulo 2^{32} are replaced by operations that are modulo 2^1, 2^4, 2^8 or 2^{16} ($x \lll y$ is replaced by $x \lll (y \mod (1, 4, 8, 16))$ respectively). For the counter and the prefix blocks (M_{-1}, M_0), we simply consider them constructed on words of 1 (respectively 4,8,16) bits. Thus, for example, in Weakinson-1bit, the counter "loops" each 4 message blocks, while (M_{-1}, M_0) is made of alternating bits 0 and 1.

The padding rule is unchanged but also applies on *small words*. In the case of a message whose bitlength is a multiple of the block length (which is the case in the following examples) the padding thus consists of a full extra block. The first word of this block has value `0x0080` (resp. `0x80`, `0x8`, `1`) for Weakinson-16bit (resp. Weakinson-8bit, Weakinson-4bit, Weakinson-1bit), and this first word is followed by 15 other words whose value is 0.

Finally, to behave as a restriction of full Shabal over smaller words, the hash value corresponds to the last $\ell_h/32$ words of the state buffer C. As a consequence, for all *small word* variants

(Weakinson-1bit,...,Weakinson-16bit), the exact output length is not equal to the expected one (*e.g.,* 64 bits instead of 256 for Weakinson-8bit).

Pattern for Weakinson-1bit($0_1{}^{16}$) is as follows, with $l_h = 256$:

```
A : .......1 .......0 .......1 .......1 .......1 .......1 .......0 .......0
    .......1 .......0 .......0 .......1

B : .......1 .......0 .......0 .......1 .......1 .......1 .......0 .......0
    .......1 .......1 .......1 .......1 .......0 .......0 .......0 .......1

C : .......1 .......0 .......0 .......1 .......1 .......1 .......0 .......1
    .......0 .......0 .......0 .......1 .......0 .......0 .......1 .......0

H : .......0 .......0 .......0 .......1 .......0 .......0 .......1 .......0
```

Pattern for Weakinson-4bit($0_4{}^{16}$) is as follows, with $l_h = 256$:

```
A : .......C .......E .......1 .......6 .......1 .......F .......5 .......5
    .......8 .......E .......8 .......4

B : .......B .......E .......9 .......B .......4 .......8 .......D .......8
    .......4 .......0 .......5 .......3 .......9 .......7 .......4 .......8

C : .......6 .......F .......4 .......F .......E .......5 .......C .......E
    .......6 .......3 .......F .......F .......9 .......C .......0 .......D

H : .......6 .......3 .......F .......F .......9 .......C .......0 .......D
```

Pattern for Weakinson-8bit($0_8{}^{16}$) is as follows, with $l_h = 256$:

```
A : ......F1 ......EB ......E1 ......3A ......C6 ......58 ......01 ......87
    ......D5 ......10 ......8D ......52

B : ......DE ......54 ......51 ......7E ......FF ......95 ......2E ......56
    ......2E ......12 ......6E ......5E ......C5 ......59 ......25 ......26

C : ......C8 ......92 ......E2 ......5C ......99 ......ED ......A5 ......FF
    ......31 ......5D ......24 ......E0 ......DC ......3F ......69 ......D9

H : ......31 ......5D ......24 ......E0 ......DC ......3F ......69 ......D9
```

Pattern for Weakinson-16bit($0_{16}{}^{16}$) is the following, with $l_h = 256$:

```
A : ....076C ....40DC ....8788 ....AD4C ....F1EF ....69BF ....870A ....40ED
    ....3315 ....0EA5 ....7114 ....F084

B : ....A681 ....A393 ....9AFA ....CACF ....29E5 ....94AF ....40F7 ....51F0
    ....C032 ....6A05 ....598D ....60DF ....AC4C ....D942 ....432E ....39CC

C : ....87A9 ....30D0 ....AA7D ....18FC ....794D ....1071 ....2783 ....EA43
    ....8CCB ....BFFB ....55AF ....D177 ....1671 ....944F ....7EA4 ....0B5D

H : ....8CCB ....BFFB ....55AF ....D177 ....1671 ....944F ....7EA4 ....0B5D
```

6.2 With Linear Message Introduction

Second proposed variant is a version of Shabal for which some nonlinearity sources have been removed. In order to simplify cryptanalysis and notably search for differential paths, we propose

to replace the additions/subtractions of message blocks in (B, C) by XORs. More precisely, Weakinson-$\oplus$ follows the definition given in Section 2.3, except that the *add* step is replaced by

$$B \leftarrow B \oplus M_i,$$

and the *sub* step is replaced by

$$C \leftarrow C \oplus M_i,$$

where $\oplus$ is computed word per word on buffers B, C and M_i.

Pattern for Weakinson-$\oplus({0_{32}}^{16})$ is as follows, with $l_h = 256$:

```
A : 5A922744 6C5F4CDE 36712DDA 243281AD 2A4745B6 B0484606 41E736FE 3804B831
    EC790220 ADC41C4A 6E14A40C FD73D2FB

B : 66AD540B 5ADCE9DF 19BA13EA F639BB26 CC62A3F2 195E37E4 49218138 6DF780E4
    EAB93E0E A92796AC E0173209 587AB49C 258A4ED4 EAE17311 3BE23745 B0210272

C : 492A6656 72639EE5 00EB60D9 A59DF01E FE2EE212 8189980B E14CFF5E 990045F8
    A9B3E792 3F0A6E76 71651EF3 62BA3EDD 4BF8C75D 6E387998 AA95829C AB08C0C6

H : A9B3E792 3F0A6E76 71651EF3 62BA3EDD 4BF8C75D 6E387998 AA95829C AB08C0C6
```

6.3 With $\mathcal{U}(x) = x$ and $\mathcal{V}(x) = x$

In order to simplify the update of the A buffer, we propose a variant called Weakinson-SimpleUV, where the $\mathcal{U}$ and $\mathcal{V}$ functions are replaced by identity. We expect this variant to propose a simpler framework for cryptanalysis, without totally removing the A memory effect.

Pattern for Weakinson-SimpleUV$({0_{32}}^{16})$ is as follows, with $l_h = 256$:

```
A : 97B5AC21 07FABC4E 124079E3 5EE4374B 308FF84D 36F1F76B E256DF9C D5191AB2
    37799815 A0244AB4 8091CABD E683AB20

B : 7B13E5F6 2BC07FC4 6D134194 BF615661 1AD65E53 CA80EC67 5EFD063E 8D3C4E19
    F6A58A22 7E70FF7F 15F72B44 35198E6E 4F409255 4955C79C 8A5E526A D09F129E

C : E45B7968 415606C5 623DEAB2 CAA7C4C3 AD8156FC 312D55EE 463275AE D011A532
    8051063F 47936F2C C5B7D1B0 AE9222A8 1224C272 3B6BB168 30A959E0 7CE5CCA4

H : 8051063F 47936F2C C5B7D1B0 AE9222A8 1224C272 3B6BB168 30A959E0 7CE5CCA4
```

6.4 With $\mathcal{U}(x) = (x \ll 1) \oplus x$ and $\mathcal{V}(x) = (x \ll 2) \oplus x$

Another version that we propose is a variant called Weakinson-LinearUV, where the $\mathcal{U}$ and $\mathcal{V}$ functions are replaced by their linearized counterparts, that is $\mathcal{U}(x) = (x \ll 1) \oplus x$ and $\mathcal{V}(x) = (x \ll 2) \oplus x$ (remember for example that normally, $\mathcal{U}$ is defined by $\mathcal{U}(x) = (x \ll 1) + x \bmod 2^{32}$).

Pattern for Weakinson-LinearUV$({0_{32}}^{16})$ is as follows, with $l_h = 256$:

```
A : 7D2C8738 F05B4D6D 285269AD C84D795F 12B047FD 10E216D7 8841EBFA 36264ABE
    4611AD57 7738084E F781D82E 8E6D4ECD

B : 536E59C1 D2A8024C E90C42A2 E94F7F95 CE7E2A0A BEFC757B F362487B 96524FFD
    D0C5C174 758695F7 E9DEA919 5161C9C1 260A0E37 3E2792B6 6B34F09A 4026F817

C : EB688602 0320F5D9 44CB8021 4AE599E9 76ABD9F6 13F6D196 DA89469C 1726D214
    DD4E3CD9 FE604991 6B3143F8 A736F8E3 F5CBD4C9 ECC16C73 3E01E463 DE1C29BA

H : DD4E3CD9 FE604991 6B3143F8 A736F8E3 F5CBD4C9 ECC16C73 3E01E463 DE1C29BA
```

6.5 Without the Last Update Loop on A

One can also consider to remove the last loop on updating of A in the permutation. More precisely, we would remove the 36 updates of form $A[j \bmod r] \leftarrow A[j \bmod r] + C[j + 3 \bmod 16]$. As shown in the analysis of Section 11.6, this results in a much weaker permutation.

Pattern for Weakinson-NoFinalUpdateA($0_{32}{}^{16}$) is as follows, with $l_h = 256$:

```
A : E9C8136E 53AF87C2 2AC08B96 35924295 2C1E7E0A A08A0106 A1A16363 E70CC268
    B6D84B88 2EA7E106 69890460 EBDB103E

B : 60C088C4 FD32344D 55F6AFC7 8159C310 0A838854 76385AFD 4AB18F25 51D586B2
    C370B22D 75B471C2 5B8381FA 9D1C54E0 DAF7088D A1E92D63 7DF687DD FA7A8419

C : B2F4BD83 27099457 E2EEFE89 08154CEC 8CEDEA10 C8D599A5 320C880B AC21D064
    EFCD7C6A 81F426FB 11576938 347955BF C45598B6 728E0694 D4D34ABD D9D1880E

H : EFCD7C6A 81F426FB 11576938 347955BF C45598B6 728E0694 D4D34ABD D9D1880E
```

6.6 Other Non-described Variants

In fact, the number of variants one may consider is almost infinite. The closer they are to the real Shabal, the more interesting the cryptanalysis is. Here, we enumerate some possible modifications, without explicitly giving some test patterns for them.

One may notably:

1. remove the counter w;
2. reduce r (which reduces the security margin of the construction);
3. increase r, whose effect remains unsure when it becomes large (as the diffusion is decreased);
4. reduce p (normally, the security is better if p is larger, but for some differential attacks, increasing p might be a way to decrease some probabilities, as it was the case with 82-round SHA, see [9]); we note that Weakinson-NoFinalUpdateA with $p = 1$ and $p = 2$ has been studied in Section 11.6.

Changing the offsets or the rotate values is more tricky, and so is not considered as an educative variant.

Of course, any non-trivial combination of the previously depicted variants is possible, and may be the subject of study by the community. Thus, the names of the variants follow our denomination strategy, with for example Weakinson-⊕-16bit-SimpleUV. These combinations allow very weak versions of Shabal. Clearly, the goal is to attack variants that are as close as possible to full Shabal.

Chapter 7

Implementation Tricks: How to Speed Up Codes on Your Platform

Contents

Shabal was meant to be efficient when implemented on common 32-bit and 64-bit general purpose hardware (without needlessly sacrificing performance on smaller systems or implementations with dedicated hardware). Nevertheless, efficient implementation on a given platform requires a proper mapping of the Shabal algorithm structure to the features of that platform. We here list some points which are worth taking into account when implementing Shabal.

7.1 Desktop and Server Systems

Desktop and server systems are generic polyvalent computers, using one or a few central processors with a handful of 32-bit or 64-bit registers, and clocked at frequencies measured in gigahertz. That market is dominated by processors compatible with the so-called "x86" instruction set, initially created by Intel. The two now common variants for this instruction set consist of, respectively, about seven 32-bit user registers, or fifteen 64-bit registers. However, other architectures are still widely deployed as well, for instance SPARC and Power systems.

7.1.1 Cache Issues

On big systems, cache issues tend to dominate computation time, because the main memory is quite slower than the CPU. This is an increasing trend, since RAM speed benefits little from increased transistor density, contrary to CPU cores.

Briefly stated, a hash function implementation provides maximum performance only when it fits within a fraction of the CPU level 1 caches. In a complete application, hashing is just a part of the overall data processing; thus, the hash function implementation shall use only a small part of the L1 caches, because other procedures down the data path use some cache as well, and are typically interleaved with the hash function implementation. Desktop and server processors

usually feature about 32 or 64 kilobytes of L1 cache for data, and about the same amount for code.

Shabal implies a very low pressure on the data cache. The state of Shabal fits in less than 300 bytes. Elementary operations are word-based primitives implemented natively by most CPU; none of them is likely to benefit from table-based code. This economy of L1 cache is one of the strong points of Shabal, performance-wise.

The code L1 cache, however, may become an issue if not taken into account during implementation. A common optimization technique is *loop unrolling*: when a sequence of instructions is to be executed several times in a row, then it may be worthwhile to duplicate that sequence. Loop unrolling saves some or all of the cost of the loop management itself, at the expense of a greater L1 cache consumption. In Shabal, an obvious candidate for loop unrolling is the permutation, which repeats the same sequence p times. When fully unrolled, this sequence fits in roughly 7 kilobytes of code with $p = 3$ (this depends on the target architecture and the compiler). This should be small enough to fit in the L1 cache along with the rest of the application code which lies in the critical path. Conversely, unrolling two successive rounds of Shabal (which would transform the "swap" operation of B and C into a mere compilation-time data routing problem) appears not to be worthwhile, because it would double L1 cache consumption.

Note that this effect of cache consumption is often overlooked in benchmarks, which run the measured function "alone".

7.1.2 Precomputations

A number of computations do not depend on the actual data. For instance, in the permutation, the indices of the accessed state element are always the same. The value of $i + 16j \bmod r$ depends on i and j, but *not* on the input data. This value may thus be computed in advance, at compilation time. Precomputing these indices is natural and immediate when loop unrolling is applied: by unrolling the loops on i and j, all indices become, at the syntax level, constant expressions which the compiler computes directly.

Some programming language implementations may perform such unrolling automatically; however, this is an optimization feature which can rarely be finely tuned by the programmer. As we saw above, some loops are worth unrolling, but not all, and which unrolling level should be applied depends on the overall application structure and usage, of which the compiler knows little or none. Therefore, it is often necessary to apply unrolling "manually", *i.e.,* by duplicating the sequence by hand, directly in source code. Metalanguages (*e.g.,* the C preprocessor, when targeting the C programming language) can be used, to some extent, to perform this unrolling operation at compilation time.

Another possible and quite different precomputation is related to the Shabal prefix. The input data is prefixed by 32 words (two full blocks), which value depend on the intended output length, but is independent of the message data. Instead of prefixing the input message, the Shabal implementation may directly initialize its internal state to the values it should contain *after* processing the prefix blocks. Such a precomputed internal state uses about 176 bytes per intended output length. Depending on the implementation technique, these 176 bytes may be counted against the data or the code cache; either way, the cost is small, and substantially increases throughput when Shabal is primarily used on very small messages.

7.1.3 Machine Code Generation

The CPU executes *machine code*. On desktop and server processors, programmers very rarely input machine code (or assembly, which is a direct translation of machine code). Optimization rules for laying out machine code instructions (in particular choosing in which CPU registers data should be stored) are complex, arcane, and more suited to automatic machine code generation. Indeed, modern CPU have been designed so that *compilers* (in particular C compilers) may perform a good job at machine code generation. Using a programming language such as C also increases

portability, since optimization rules and actual instructions change between processor brands and generations.

Optimization of a Shabal implementation is thus mostly a matter of giving the compiler as much information as possible on what operations shall occur. Precomputation of indices for state access is an important step in that process. When most loops are unrolled (namely, the i and j loops in the permutation, and the i loop in the message input), then spatial layout of state elements becomes irrelevant: each of the state words (A, B, C and W) and of the current message block words (M) are accessed independently, and which word is accessed is known at compile time. It turns out that it helps the compiler to explicitly state that fact, by first "copying" the full state to so many local variables with no "array" semantics. The machine code generation system knows how to optimize away unneeded copies (when the architecture supports machine opcodes with memory operands); and by making explicit copies to local variables, the programmer informs the compiler that the array semantics (ordered sequence of slots) need not be maintained. Furthermore, local variables which addresses are never taken are known never to be accessed through indirections, which again helps the compiler.

The core permutation in Shabal uses three rotations of 32-bit words, by 1, 15 and 17 bits respectively. Some architectures feature explicit instructions for rotations (*e.g.,* the `rol` opcode on x86 processors); for other systems, logical shifts and Boolean combinations must be used. Regardless of the architecture features, usual programming language (*e.g.,* C) lack standard operators for expressing such a rotation. Some compilers provide ad hoc extensions. However, it turns out that most modern compilers recognize the "rotation construction" (two shifts and a Boolean bitwise OR) and know how to use the specific rotation instructions of the processor, if available and worthwhile.

The permutation includes multiplications by 3 and 5 (modulo 2^{32}). Multiplication by 3 can be implemented with two additions, or a logical shift and an addition. Multiplication by 5 is a matter of three additions, or one shift and one addition. On some platforms, multiplications by 3 or 5 can be performed with a single efficient opcode primarily designed for memory array access. Since the optimal representation of such a multiplication varies between architectures, it is recommended to express the operation as a raw multiplication, so that the compiler may choose the best code sequence for this operation.

7.1.4 Parallelism

Although Shabal is inherently a sequential algorithm, it has some limited support for local parallelism. Namely:

- The decoding of a message block into 16 words may be performed in parallel, limited mostly by the input memory bus width and speed.
- The addition (to B) and subtraction (from C) of message words can be performed in parallel.
- The rotation by seventeen bits of all words of B, at the beginning of the permutation, can be performed parallely.
- The additions of words of C to A at the end of the permutation modify the various A words independently from each other.
- The swap of B and C is also a routing problem which can be performed in parallel.

The easiest way to exploit this parallelism is to let the compiler perform its job. Loop unrolling and the use of local variables help the compiler detect which code chunks may be computed independently of each other, and thus be scheduled to operate simultaneously on distinct parts of the processor.

Modern processors have special units meant for SIMD computations. An example is the SSE2 unit which is found on recent x86-compatible processors. Preliminary implementation experiments

have not shown those units to be worthwhile for Shabal implementation, mostly because transferring the data to and from the SIMD unit proved to be too expensive, with regards to the gains obtained by parallel execution.

7.2 Embedded and Small Systems

Recent embedded and small systems tend to align on the use of 32-bit processors, mostly MiPS and ARM cores. Even smart cards gradually abandon 8-bit and 16-bit cores. Shabal uses only 32-bit words and simple operations (bitwise Boolean operations, and additions modulo 2^{32}). Shabal does not use complex operations such as multiplications; modular multiplication is efficient on desktop and server systems, but many embedded systems lack efficient hardware support for multiplication (as was explained above, the multiplications by 3 and 5, which are part of the core permutation, are usually translated to additions or other simpler operations).

The w counter, stored in the W buffer, is nominally defined as a 64-bit value; however, that counter is initialized at 0 and is incremented for each data block. Thus, the 32 higher bits of W remain equal to zero as long as the total input data size is less than 2^{32} 512-bit blocks, *i.e.,* about 275 gigabytes[1]. This amount of data far exceeds what a typical embedded system may ever process; this allows for a 32-bit only handling of W. Even if a full 64-bit W must be maintained, then it is easy to manually handle the carry: if the increment of the lower 32-bit of W yields the value 0, then a carry should be propagated to the higher bits. Note that besides being incremented for each input block, the main use of W is to be split into two 32-bit words, combined with two state words at each round. Thus, even if the host platform supports 64-bit values natively, it may be a good idea to keep W as two separate 32-bit words.

If Shabal must be implemented on a very small, 8-bit or 16-bit CPU, then carry propagation must be applied to all 32-bit additions. On such an architecture, 32-bit words are split into several chunks of length 8 or 16 bits; thus, a rotation by 16 bits, being a swap of the high and low halves, is a mere problem of data routing which can be solved with little to no runtime cost. Assuming the 16-bit rotation to be essentially free, we can see that all word rotations used in Shabal can be simplified to left or right rotations by 1 bit, which are often more efficient on small CPU than generic n-bit shifts or rotations.

The initial state for Shabal is defined from the processing of the prefix, which depends only on the intended output size. Performance-wise, this step is usually replaced by a precomputed IV, which is the internal Shabal state *after* processing of the prefix blocks. However, on platforms where code space is a very scarce resource, that IV could be replaced by explicit processing of the prefix, which may use a few less code bytes, at the expense of some extra clock cycles for each hashed message. Another trade-off between code size and computing speed is the amount of loop unrolling which is applied when implementing the permutation.

7.3 ASIC and FPGA

Dedicated hardware can be used to implement Shabal. The most complex operations will be modular additions, which require carry propagation. Most support packages already feature ready-to-use optimized adders; carry propagation over n bits can be performed with a circuit of depth $\log n$. Bitwise Boolean operations are easy; rotations are mere data routing with no or very little runtime cost. Cost on FPGA and ASIC is measured in propagation delays (which depend on the circuit depth) and space (number of logic gates needed for the overall circuit).

The opportunities for local parallelism described in Section 7.1.4 can be exploited on dedicated hardware, at the expense of additional gates; however, the cost is dominated by the main double-loop in the permutation. In that core permutation, we see that the computation of each new value for a word $A[i + 16j \bmod r]$ depends on the value which was computed immediately before for

[1] In the formal description, the counter initial value is -1, but we are assuming prefix preprocessing, hence the actual initial value for W is 1.

$A[i - 1 + 16j \bmod r]$. This effectively prohibits parallelism. This suggests a design which has a single unit performing the update of a word of A, invoked 48 times per input data block. It is easily seen that the accessed elements of A, B, C and M are regular enough to allow for a simple shift-register based indexing: these state variables are stored in big registers (12×32 bits for A, 16×32 bits for B, C and M) which are rotated by 1 word (32 bits) at each iteration. Updates on $B[i]$ can be performed in parallel of the next iteration, since the new value of $B[i]$ will not be used immediately (neither of o_1, o_2 or o_3 is equal to 15).

The core iteration, which is invoked 48 times, contains the multiplications by 3 and 5, which are cascaded and thus amount to 4 serially linked additions. Addition and subtraction of input words may use up to 32 additions per block; even when routed through a single unit, this amounts to less than 15% of the computation time. Thus, using several adders to perform parallel computations does not seem to provide much benefits. The additions of C words to A words amount to 36 additional additions, which can be performed mostly concurrently with the subtraction of message blocks from C and the beginning of the processing of the next block; yet again, a shift register for C and an adder unit will be used for this operation.

This means that Shabal can be implemented in dedicated hardware with only seven 32-bit adder units.[2] The rest of the design mostly consists of data routing and bitwise computations which should contribute little to the overall cost, with regards to the additions.

We thus claim that Shabal is quite space efficient when implemented in dedicated hardware.

[2]Sharing the same adder for addition of message words to B, and subtraction from C, seems overly complex, hence the two extra adders.

Part 2.B.2

A Statement of the Algorithm's Estimated Computational Efficiency and Memory Requirements in Hardware and Software

Chapter 8

Computational Efficiency And Memory Requirements In Hardware and Software

Contents

In the sequel, we present a statement of Shabal's estimated computational efficiency and memory requirement in hardware and software across a variety of platforms. On the software side, the presentation includes measurement of the efficiency on both high-end (PCs) and low-end (router) software platforms as well as 8-bit processors. On the hardware side, we give a rough gate count estimate for ASIC or FPGA. The software measurements give an estimate of Shabal efficiency on the reference platform. They can also be compared to other hash function performance that are detailed in Section 12.3.4.

In Chapter 12, one can found a comparison (on several aspects) of Shabal with several other hash functions. In Appendix A, the interested reader can also find some simple implementations on various environments, including recent smart cards.

8.1 High-End Software Platforms

A *high-end* software platform is, basically, a modern desktop or server PC. That market is dominated by x86-compatible processors. We tested the optimized implementation of Shabal on five such architectures:

- a quadri-core Intel Xeon X3220 CPU clocked at 2.4 GHz, in 64-bit mode ("AMD64" architecture);
- the same quadri-core Intel Xeon, this time used in 32-bit mode ("i386" architecture);
- an AMD Athlon64 3200+ CPU clocked at 2 GHz, in 64-bit mode;
- the same AMD Athlon64, in 32-bit mode;
- a VIA C7 CPU clocked at 2 GHz (32-bit mode only).

All systems run Linux, and the GNU compiler GCC is used (version 4.2.3) with optimization flags `-O2 -fomit-frame-pointer`.

The Xeon CPU should provide performance very similar to what is expected from the reference platform: compared to the Intel Core2 Duo, the Xeon has more cores and more cache, but this should not impact our measures since we use a single core, and the complete test data and code fits in the L1 cache of all tested processors.

The AMD processor is representative of the products of Intel's direct competitor AMD. Although that CPU is relatively old (that specific hardware was manufactured in 2005), newer AMD cores exhibit similar timings per clock cycle.

The C7 CPU is an x86-compatible CPU designed for low power consumption. It does not implement the 64-bit instruction set, and delivers less computing power per clock cycle, for a much reduced energy cost.

Code size is the following:

- In 64-bit mode, compiled code size is 21456 bytes. This includes precomputed IV for four output sizes (224, 256, 384 and 512 bits); each IV uses 176 bytes. The main update function uses 7360 bytes of code, while the finalizing function (which handles padding and the extra invocations of the permutation) uses 12960 bytes of code.
- In 32-bit mode, compiled code size is 24048 bytes. The main update function size is 8080 bytes, while the finalizing function totalizes 14768 bytes.

The code uses no precomputed table besides the IV, which is 176-byte long. The hash state size, including the buffer for the current partial block and other "administrative" variables, is less than 300 bytes. Even counting the copy of the state that the code may perform during its computation (since the optimized code formally specifies the use of local variables for the state words), the data L1 cache consumption remains very low.

We measured hashing bandwidth, assuming that both code and data are already in the innermost CPU caches. The input data is assumed to be split into individual messages which are hashed independently, each with its padding and finalization. Thus, the short messages emphasize padding cost, while long messages measure asymptotic speed of the core update mechanism. In the Table 8.1, we list the bandwidth achieved on our five test platforms, for messages of individual sizes 16, 64, 256, 1024 and 8192 bytes; we also give a measure for a unique message which length exceeds a hundred megabytes. Figures are in megabytes per second. Accuracy is roughly 2%.

	Bandwidth per unit message size (MB/s)					
Platform	16	64	256	1024	8192	long
Xeon, 64-bit	12.86	39.71	98.69	156.52	189.04	194.52
Xeon, 32-bit	10.02	31.36	76.92	119.84	144.32	147.90
Athlon64, 64-bit	9.17	28.44	70.83	112.79	137.31	140.91
Athlon64, 32-bit	6.88	21.44	52.84	83.62	100.16	103.64
C7	4.25	13.48	33.55	54.12	65.79	67.79

Table 8.1: Shabal performance on high-end software platforms

From these figures, we may estimate processing efficiency of Shabal on the reference platform (64-bit mode) at about 1.54 clock cycle per input bit (790 cycles per 512-bit block), with a fixed additional cost of about 2200 clock cycles per message (for the finalizing function). Each message consists of at least one block (when padding is applied), thus the minimal cost for a message is close to 3000 clock cycles (this figure is stable for all message sizes from 0 to 511 bits).

8.2 Low-End Software Platforms

We chose a "typical" low-end software platform: a broadband/WiFi router from Linksys, using a Broadcom BCM3302 CPU clocked at 200 MHz. This is a MiPS-compatible core integrated

with network hardware. This platform should be viewed as representative of common low-cost network-intensive hardware. Our test machine runs a reduced version of Linux. The C compiler is again GCC, version 4.2.4.

Code size is now 21036 bytes (7852 bytes for the update function, 12768 bytes for the finalizing function), which is quite comparable to what was obtained on x86 processors. We list achieved bandwidth in the Table 8.2.

	Bandwidth per unit message size (MB/s)					
Platform	16	64	256	1024	8192	long
Broadcom MiPS	0.33	0.51	1.64	3.70	5.63	6.24

Table 8.2: Shabal performance on low-end software platforms

8.3 Smartcard Platforms

For test purposes, we have developed prototypes on several smartcard architectures, from low-cost 8-bit CPUs to 32-bit high-end architectures. In our RAM consumption, the message buffer is not counted since it is a part of the user memory. Our estimates state for 2048-bit messages (*i.e.,* 4 blocks), which is not the typical case where Shabal reaches its best performance. As for code size, we do not take into account IV tables, since we preferred to implement the prefix approach (*i.e.,* the Shabal mode where the IV is not stored but reconstructed during the execution), which is one of the advantages of Shabal on constrained environments. Due to lack of time, our implementations are not fully optimized at the moment of the submission of this document. Furthermore, for intellectual property reasons, we are not allowed to provide the source codes in assembly in this document.

From our experiment, it appears that the function is relatively easy to implement. On Implementation 1, on a recent 8-bit smartcard with arithmetic coprocessor, we have obtained a full code of about 1.2 kilobytes, using around a 256-byte array in CPU-RAM and the coprocessor RAM. The hashing costs around 215 000 cycles, which could be reduced to 160 000 cycles, using IV approach. As a comparison, a fairly optimized SHA-1 costs about 120 000 cycles.

On Implementation 2, on a classical 32-bit processor, our implementation uses 300 bytes in RAM, and the code takes 2kB of ROM. The 2048-bit message hashing takes about 60 000 cycles, which is nearly 2.5 slower than an optimized version of SHA-1 under the same platform. Using IV approach, execution timing would be reduced to 50 000 cycles.

Finally, on Implementation 3, on a recent 8-bit 8051 smartcard, our code stands on 1.2 kB, consumes 192 bytes of RAM, and hashing takes about 750 000 cycles. This is about 3 times more than the 250 000 cycles needed for the optimized version of SHA-1 on the same smartcard architecture.

Once again, these implementations are only *early prototypes*, and so conclusions are hard to make at this point, when comparing with implementations that have been extensively optimized. We intend to deliver more precise implementations during the NIST SHA-3 competition, to offer more accurate sources of comparison. We note however that (not surprisingly) Shabal is much more efficient on 32-bit platforms than on 8-bit platforms.

8.4 Dedicated Hardware

As was pointed out in Section 7.3, an implementation of Shabal on a dedicated ASIC or FPGA spends most of its time in the core loop, which is run 48 times per input block. The core loop contains mostly four 32-bit adders which are serially cascaded (the result of each is used in the next adder). The circuit depth for an optimized 32-bit adder should be roughly equal to 7 or 8 gates. Assuming a design where three gates can be traversed per clock cycle, and taking into

account the extra operations (mostly one bitwise exclusive OR in the critical path, and the costs for the additions and subtractions of message blocks), we may estimate a latency of 700 clock cycles per 512-bit input block (we count 3 clock cycles per adder in the inner iteration, plus 32 adder invocations for message word addition and subtraction). This yields an asymptotic efficiency of roughly 1.4 clock cycles per input bit, which is rather close to what is achieved with a generic-purpose CPU.

An hardware implementation uses seven 32-bit adder units. Such a unit uses about 800 gates, hence 5600 gates for the adders alone. We need shift registers to hold state variables and the current block, which together account for 1984 data bits; at least 6000 gates are needed for that. Some extra gates are needed for the exclusive OR operations, the w counter update, and the general data routing and handling. A very rough estimate of the gate count for our circuit will be around 20000 gates.

Part 2.B.3

A Series of Known Answer Tests and Monte Carlo Tests

Chapter 9

Known Answer Tests and Monte Carlo Tests

The *Known Answer Tests* (KAT) and *Monte Carlo Tests* (MCT) have been generated according to the format specified by the NIST, using the generation code provided by the NIST.

The results are provided in the enclosed files, *i.e.,* `ShortMsgKAT_`ℓ_h`.txt`, `LongMsgKAT_`ℓ_h`.txt`, `ExtremelyLongMsgKAT_`ℓ_h`.txt` and `MonteCarlo_`ℓ_h`.txt`, with ℓ_h being equal to the standard output lengths for the submission process (224, 256, 384 and 512). It has been verified that the exact same files are produced by both the reference and the optimized implementations, on various 32-bit and 64-bit platforms and compilers.

Part 2.B.4

A Statement of the Expected Strength

Chapter 10

Statement of the Expected Strength

Contents

In this chapter, we list our security claims, concerning collision resistance, one-wayness and second-preimage resistance of Shabal. In all following statements, the supported message length is supposed to be at most 2^{64} blocks, *i.e.,* 2^{73} bits. Note that, as currently described, Shabal can accommodate longer messages (in fact, messages of arbitrary length), however we do not claim anything regarding the security for messages longer than this bound.

10.1 Collision Resistance

In order to quantify the collision resistance of Shabal, we define a family of Shabal variants, in which the initial values of both the internal state and the counter can be arbitrarily fixed and are viewed as parameters.

Let $M, (M_1, ..., M_k)$ be a k-uple of message blocks. We define $\mathsf{Shabal}^*(M, S_0, W_0)$ as the words C_i for $16 - \ell_h/32 \leq i \leq 15$ of state S_{k+3}, which is defined by the following relations:

$$\forall i \in \{1, .., k\}, (S_i, W_i) = \mathcal{R}(M_i, S_{i-1}, W_{i-1}) \tag{10.1}$$

$$\forall i \in \{k+1, k+2, k+3\}, (S_i, W_i) = \mathcal{F}(M_k, S_{i-1}, W_{i-1}) \tag{10.2}$$

Informally, $\mathsf{Shabal}^*(M, S, W)$ is a version of $\mathsf{Shabal}(M)$ with the IV set to S_0, and the initial value of the counter set to W_0. The collision resistance of Shabal is defined as the resistance to the following type of adversaries:

1. The challenger draws random message blocks M_{-1}, M_0 and sends them to the adversary;
2. The adversary outputs M, M' and wins the game if $M' \neq M$ and $\mathsf{Shabal}^*(M_{-1}||M_0||M, 0, -1) = \mathsf{Shabal}^*(M_{-1}||M_0||M', 0, -1)$.

As collisions exist for all hash functions, we randomize the security game by randomizing the prefix used for Shabal.

The existence of deterministic adversaries that can output collisions for a given hash function with probability 1 can also be dealt with by taking account of *human ignorance* [37]. We then define the notion of collision resistance of Shabal as its resistance to *known* collision search algorithms, defined by their ability to output two distinct messages M and M' such that $\mathcal{H}(M) = \mathcal{H}(M')$.

Security Claim 1. *For any $\ell_h \in \{192, 224, 256, 384, 512\}$, finding a collision for Shabal of message digest ℓ_h bits requires at least $2^{\ell_h/2}$ calls to the message round function.*

10.2 Preimage Resistance

We define the preimage resistance of Shabal as its resistance to all known adversaries of the type described below.

1. The challenger draws a random $H \in \{0,1\}^{\ell_h}$ and sends it to the adversary;
2. The adversary outputs a message M and wins the game if $\mathsf{Shabal}(M) = H$.

We now claim the following security against preimage attacks.

Security Claim 2. *For any $\ell_h \in \{192, 224, 256, 384, 512\}$, any preimage attack against Shabal with ℓ_h-bit message digests requires at least 2^{ℓ_h} calls to the message round function.*

10.3 Second-preimage Resistance

We define the notion of second-preimage resistance as the resistance to all known adversaries of the type described below, with a parameter k.

1. The challenger draws a random $M \in \{0,1\}^{2^k}$, and sends it to the adversary;
2. The adversary outputs M' and wins the game if $\mathsf{Shabal}(M) = \mathsf{Shabal}(M')$ and $M' \neq M$.

We now claim the following security against second-preimage attacks.

Security Claim 3. *For any $\ell_h \in \{192, 224, 256, 384, 512\}$, any second-preimage attack against Shabal-ℓ_h for messages shorter than 2^k bits requires at least $2^{\ell_h - k}$ calls to the message round function.*

10.4 Resistance to Length-extension Attacks

The well-known Merkle-Damgård construction has an undesirable property called *length extension*. It means that once an attacker has one collision, *i.e.,* two messages M_1 and M_2 with $|M_1| = |M_2|$ such that $\mathcal{H}(M_1) = \mathcal{H}(M_2)$, then for any suffix M it also holds that $\mathcal{H}(M_1 \| \text{pad}(M_1) \| M) = \mathcal{H}(M_2 \| \text{pad}(M_2) \| M)$.

The length-extension attack can be extended to a more general setting, in the following sense: given $\mathcal{H}(M)$, an attacker can compute $\mathcal{H}(M || \text{pad}(M) || M')$ for any M', even if she does not know M.

The question then arises whether such an attack can be applied to Shabal. In other words, can an adversary generate a large number of distinct collisions, with the cost of only one collision search? We can consider the following security game:

1. The challenger draws random message blocks M_{-1}, M_0 and sends them to the adversary;

2. The adversary outputs M, M' and wins the game if $\mathsf{Shabal}^*(M_{-1}||M_0||M||T, 0, -1)$ $= \mathsf{Shabal}^*(M_{-1}||M_0||M'||T, 0, -1)$ for all possible suffixes T.

The message extension attack can be applied to the Shabal hash function only if an internal collision occurs before the three final rounds. However, the complexity for finding an internal collision in Shabal-ℓ_h (with $\ell_h \in \{192, 224, 256, 384, 512\}$) is expected to require at least $2^{512/2}$ calls to the message round function, as the cost of an internal collision is expected to be the same for the five output lengths.

We then claim the following resistance to length-extension attacks.

Security Claim 4. *For any $\ell_h \in \{192, 224, 256, 384, 512\}$, any length-extension attack against Shabal-ℓ_h requires at least 2^{256} calls to the message round function.*

10.5 Strength of a Subset of the Output Bits

This section claims the security of truncated versions of Shabal. The idea is not new and consists in building a variant of a hash function by simply truncating the output and keeping only the first bits output by the entire function. Here we go even further, by stating that we can not only *truncate* the output but also extract *any substring* of the output bitstring. Informally, our claim thus says that all bits resulting from a Shabal computation are equally strong.

Security Claim 5. *For any $\ell_h \in \{192, 224, 256, 384, 512\}$ and any $\ell \leq \ell_h$, any ℓ-bit hash function specified by taking a fixed subset of the output bits of Shabal-ℓ_h meets the above requirements with ℓ replacing ℓ_h.*

10.6 PRF HMAC-Shabal

In this section we suggest the use of Shabal in the HMAC construction and claim the following security bound about the security of HMAC-Shabalviewed as a PRF (Pseudo-Random Function) family.

Security Claim 6. *For any $\ell_h \in \{192, 224, 256, 384, 512\}$, any distinguishing attack against HMAC-Shabal-ℓ_h requires at least $2^{\ell_h/2}$ calls to the message round function.*

An argument for this claim is provided by the fact that distinguishing Shabal-ℓ_h from a random function requires at least $2^{\ell_h/2}$ calls to the message round function.

Part 2.B.5

An Analysis of the Algorithm with Respect to Known Attacks

Chapter 11

Shabal: Resistance against Known Attacks

Contents

We study in this chapter the resistance of Shabal with respect to known attacks, especially to collision and (second)-preimage attacks. The structure of Shabal has some similarities with other *sponge-like* hash functions that have been proposed in the literature such as Panama [17] and more recently RadioGatún [5] or Grindahl [29]. The security analysis of Shabal with respect to known attacks stems from the security analysis made on *sponge-like* hash functions.

More precisely, Section 11.1 first presents the best collision attack, and (second)-preimage attacks, which originates from Shabal generic construction. These attacks have been exhibited by the security proofs given in Section 5. Generic attacks for internal collisions are also described in

Section 11.2. Section 11.3 then focuses on differential attacks and the search for some particular differential trails is investigated. Some attacks against weakened versions of Shabal are presented in Sections 11.5 and 11.6. Section 11.7 evaluates the applicability of length extension and multi-collision attacks. In Section 11.8, we show that the slide attacks presented in [24] cannot be directly applied to Shabal. We finally explain in Section 11.10 the provenance of constants that are used in Shabal, as requested by NIST. In our analysis, we always consider that $16p \equiv 0 \bmod r$.

11.1 Known Attacks Identified by the Security Proofs

11.1.1 Collision Attacks

We refer to the security proof of Section 5.4 which establishes that a generic bound on the ability to generate collisions in Shabal is

$$\min(2^{\ell_h/2}, 2^{(\ell_a+\ell_m)/2}) \ .$$

It appears that this bound is tight in the sense that there exist generic collision attacks which meet the given bound. Such attacks are precisely the ones that optimize one of the abortion probabilities of the simulator $\mathcal{S}$ in the COLL game. These attacks are divided into two categories: the ones that generate internal collisions and the default, trivial collision-finding attacks. We discuss the first category later in this chapter. Attacks in the second category just amount to hashing L random messages until two of them collide. This is expected to succeed as soon as L is close enough to the birthday bound $2^{\ell_h/2}$.

11.1.2 Second-preimage Attacks

We refer to Sections 5.5 and 5.6 for more details on the notation and definitions that we use here. The security bound provided by Theorem 5 can be reached by a generic attack which attempts to realize the ConnectChallenge predicate:

(i) either by creating paths from the initial state x_0 which connect to one of the internal states reached by the hashing of the input message $M^* \in \{0,1\}^\kappa$,

(ii) or by creating antipaths with respect to the target hash value h which connect to one of these internal states.

Adopting one of the above approaches or both of them simultaneously leads to an attack cost (in terms of evaluations of $\mathcal{P}$ or $\mathcal{P}^{-1}$) close to

$$2^{\ell_a+\ell_m-\log k^*}$$

where $k^* = \lceil (\kappa+1)/\ell_m \rceil$ is the number of message blocks inserted while hashing M^*. We now describe the generic attack based on (i) in more details. The generic attacks based on (ii) or on a combination of (i) and (ii) are easily expressed in a similar fashion.

The basic principle of the attack is as follows. The attacker computes the k^* internal states reached after each message round during the computation of $\mathcal{H}(M^*)$. Then, she chooses L messages $M^i = (m_1^i, \ldots, m_{k^*-1}^i)$ such that m_1^i is randomly chosen and all other m_j^i for $2 \leq j \leq k^*-1$ are such that the B-part of the internal state reached after the j-th round is equal to the B-part of the corresponding internal state obtained for M^*. The attack then succeeds if there exists an ℓ, $2 \leq \ell \leq k^*-1$, such that one of the internal states reached after the ℓ-th round for some M^i corresponds to the internal state reached after the ℓ-th round for M^*. In this case, $(m_1^i, \ldots, m_\ell^i, m_{\ell+1}^*, \ldots, m_{k^*}^*)$ has the same hash value as M^*.

More precisely, with the notation of Sections 5.5 and 5.6, the second-preimage finder $\mathcal{A}$ proceeds as follows. On input M^*, $\mathcal{A}$ first hashes M^* and stores all the internal states

$$x_0 \overset{m_1^*,1}{\rightsquigarrow} y_1^* \overset{\mathcal{F}}{\longrightarrow} x_1^* \overset{m_2^*,2}{\rightsquigarrow} y_2^* \quad \ldots \quad \overset{m_{k^*}^*,k^*}{\rightsquigarrow} y_{k^*}^* \overset{\mathcal{F}}{\longrightarrow} x_{k^*}^* \overset{m_{k^*}^*,k^*}{\rightsquigarrow} y_{k^*+1}^* \overset{\mathcal{F}}{\longrightarrow} x_{k^*+1}^* \quad \ldots \quad \overset{m_{k^*}^*,k^*}{\rightsquigarrow} y_{k^*+3}^* \overset{\mathcal{F}}{\longrightarrow} x_{k^*+3}^*$$

successively reached during the computation. Here $m_1^*, \ldots, m_{k^*}^*$ stand for the message blocks inserted while hashing M^* in chronological order. Now for each $\ell \in [1, k^* - 1]$, $\mathcal{A}$ computes

$$\gamma_\ell = b_{\ell+1}^* \boxminus m_{\ell+1}^*$$

where $b_{\ell+1}^*$ is the B-part of internal state $y_{\ell+1}^*$. Now $\mathcal{A}$ generates L lists of internal states as follows. For each $i \in [1, L]$, $\mathcal{A}$

1. picks a random message block $m_1^i \leftarrow \{0,1\}^{\ell_m}$;
2. inserts m_1^i to x_0 to get y_1^i and applies the round function to y_1^i to get $x_1^i = (m_1^i, a_1^i, b_1^i, c_1^i)$, *i.e.,* $x_1^i = \mathcal{R}(m_1^i, x_0, 1)$;
3. computes $m_2^i = c_1^i \boxminus \gamma_2$
4. inserts m_2^i to x_1^i and applies the round function to get x_2^i, *i.e.,* $x_2^i = \mathcal{R}(m_2^i, x_1^i, 2)$;
5. computes $m_3^i = c_2^i \boxminus \gamma_3$;
6. inserts m_3^i, and so forth until the list $X^i = (x_0, x_1^i, \ldots, x_{k^*-1}^i)$ is completed.

Overall, this costs $L \cdot (k^* - 1)$ evaluations of $\mathcal{P}$. Note that for any $i \in [1, L]$ and $\ell \in [2, k^* - 1]$,

$$c_{\ell-1}^i \boxminus m_\ell^i = \gamma_\ell \ . \tag{11.1}$$

Now $\mathcal{A}$ scans all the lists $X^1, \ldots, X^L$ with the hope that for some (i, ℓ) with $2 \leq \ell \leq k^* - 1$, it holds that

$$x_\ell^i \overset{\star,\ell+1}{\rightsquigarrow} y_{\ell+1}^* \ .$$

If this is the case, then the predicate $\mathsf{ConnectChallenge}(x_\ell^i)$ evaluates to True and $\mathcal{A}$ succeeds in creating a path from x_0 to one of the internal states on the target path *i.e.,* the sequence of states reached by the hashing of M^*. A second preimage is then put together by joining the two lists of message blocks

$$(m_1^i, \ldots, m_\ell^i) \quad \text{and} \quad (m_{\ell+1}^*, \ldots, m_{k^*}^*)$$

and $\mathcal{A}$ outputs the string $M \in \{0,1\}^\kappa$ whose padded value gives this list of blocks. It follows from the analysis of Section 5.6 (Game 3) that if Eq. (11.1) is fulfilled then

$$\Pr\left[x_\ell^i \overset{\star,\ell+1}{\rightsquigarrow} y_{\ell+1}^*\right] = 2^{-(\ell_a + \ell_m)}$$

for fixed (i, ℓ). This results in that the attack has success probability

$$L \cdot (k^* - 1) \cdot 2^{-(\ell_a + \ell_m)}$$

or equivalently has constant and substantial probability as soon as L is close enough to

$$2^{\ell_a + \ell_m - \log(k^* - 1)} \ .$$

11.1.3 Preimage Attacks

Generic preimage attacks arise from the probability bounds of Section 5.5 to where we refer the reader again for definitions. An example of such an attack is found when maximizing the probability that the abortion event Abort_2 occurs at some point when playing the PRE security game. We will focus on this strategy in what follows, leaving as an exercise to the reader to describe the dual approach which resides in provoking event Abort_4 and which outcome is identical.

The strategy of the attacker is as follows:

- create L final internal states whose B-parts correspond to the target hash value;

- for each of these final internal states, randomly choose the last message block m_k^i, $1 \leq i \leq L$, and compute backwards the final rounds and the last message round;
- determine the most frequent value β taken by the B-part of the previously computed internal states;
- randomly select L' messages M^i of $(k-2)$ blocks. For each of them, determine m_{k-1}^i such that the B-part of the internal state reached after the $(k-1)$-th round during the computation of $\mathcal{H}(M^i \| m_{k-1}^i)$ is equal to β.

A preimage of h can then found if both lists of internal states with B-part equal to β intersect.

With the notation used in Section 5.5, the preimage finder $\mathcal{A}$ creates random 0-antipaths with respect to the target value $h \in \{0,1\}^{\ell_m}$. This amounts to depart from a number of random final states $\tilde{x}_1, \ldots, \tilde{x}_L$ (where L is an adjustment parameter), all of which have a B-part equal to h, and apply the mode of operation backwards until the final rounds and the last message round are performed. To this end, $\mathcal{A}$ fixes the length parameter $k \geq \lceil (\ell_a + \ell_m)/\ell_m \rceil$ to some small value and uses k to parameterize the final rounds. Let $(\tilde{y}_1, \ldots \tilde{y}_L)$ be the corresponding internal states collected by $\mathcal{A}$. Now $\mathcal{A}$ lists

$$\beta_1, \ldots, \beta_L$$

where for each $i \in [1, L]$, $\beta_i = \tilde{b}_i \boxminus \tilde{m}_i$ where $\tilde{m}_i$ is the M-part of $\tilde{x}_i$ and $\tilde{b}_i$ is the B-part of $\tilde{y}_i$. Note that all of these values depend on an output of $\mathcal{P}^{-1}$ $(\tilde{b}_i)$ and therefore should follow a random distribution. $\mathcal{A}$ sorts this list and sets β to a value with maximal number of occurrences in the list. If the number of occurrences of β is smaller than a fixed bound U, then $\mathcal{A}$ increases L to extend the collection of states $\tilde{y}_i, i \in [1, L]$ until the bound is reached. Let then $I \subseteq [1, L]$ be the set of indices i for which $\beta_i = \beta$ (*i.e.*, $|I| = U$). Starting from x_0, $\mathcal{A}$ now looks for internal states that can be reached in exactly $(k-1)$ rounds that have maximal compatibility with $\{\tilde{y}_i, i \in I\}$ as follows. For $j = 1$ to some L', $\mathcal{A}$ applies $k-2$ rounds of the mode of operation with random message blocks, thereby collecting L' internal states $(x_1^0, \ldots, x_{L'}^0)$. For each $j \in [1, L']$, $\mathcal{A}$ defines $m_j = c_j^0 \boxminus \beta$ where c_j^0 is the C-part of x_j^0, inserts m_j to x_j^0 to get y_j and applies $\mathcal{P}$ to get an internal state x_j. Note that for $j \in [1, L']$, x_j is reached in exactly $(k-1)$ rounds and that if $x_j = (m_j, a_j, b_j, c_j)$ then $c_j \boxminus m_j = \beta$. Therefore as shown in the proof of Theorem 4,

$$\Pr\left[x_j \overset{\tilde{m}_i, k-1}{\rightsquigarrow} \tilde{y}_i\right] = 2^{-(\ell_a + \ell_m)}$$

and

$$\Pr\left[\exists (i,j) \in I \times [1, L'] : \Pr\left[x_j \overset{\tilde{m}_i, k-1}{\rightsquigarrow} \tilde{y}_i\right]\right] = L' \cdot |I| \cdot 2^{-(\ell_a + \ell_m)} = L' \cdot U \cdot 2^{-(\ell_a + \ell_m)} .$$

When a such a connection occurs, $\mathcal{A}$ succeeds in creating a complete path from x_0 to some final state in $\mathcal{X}[h]$ and a preimage is found by appending $\tilde{m}_i$ to the list of message blocks leading to x_j. The overall cost of the attack is $N = k \cdot L' + 3 \cdot L$ evaluations of $\mathcal{P}$ or $\mathcal{P}^{-1}$ and the success probability, when L, L', U are optimized as a function of N, is upper bounded by

$$N \cdot 2^{-(\ell_a + \ell_m - \log(\ell_m + 1) - 2)} ,$$

as shown in Section 5.5.

11.2 Internal Collisions

11.2.1 Generic Internal Collision Attack

There exist several ways to generate internal collisions; all strategies require of the order of $2^{(\ell_a + \ell_m)/2}$ iterations of $\mathcal{P}$ or $\mathcal{P}^{-1}$ in accordance with the security proof of Section 5.4.

One of these attacks consists in randomly choosing L messages of the same length and in computing the internal states reached at the end of the message rounds. For each message, an

additionnal block is then chosen such that the additionnal message round leads to an internal state whose B-part equals a given constant γ. Thus, L different random messages have been obtained which lead to a list of L internal states whose B-parts are equal to γ. An internal collision can then be found if two internal states in the list have the same A and C-parts.

We now describe the previous strategy, referring to the proof of Theorem 3 for definitions and notation. The collision finder favors the abortion event Abort_1 by attempting to create a pair $(x, \tilde{x})$ of X-nodes of the hash graph $\mathcal{G}$ such that

$$x, \tilde{x} \in X^{0,k} \quad \text{and} \quad x \overset{k,k}{\sim} \tilde{x}$$

meaning that both $x, \tilde{x}$ admit 0-path of length k and that there exist message blocks $\mathfrak{m}, \tilde{\mathfrak{m}}$ such that

$$x \overset{\mathfrak{m},k}{\rightsquigarrow} y \quad \text{and} \quad \tilde{x} \overset{\tilde{\mathfrak{m}},k}{\rightsquigarrow} y$$

for some possible internal state $y \in X$ which is not necessarily a node of $\mathcal{G}$. If the collision finder is lucky enough to generate such a pair $(x, \tilde{x})$ then it is easy to extend the two paths leading respectively to x and $\tilde{x}$ with a common suffix path starting from y. Any suffix path will lead to a collision with same-length colliding messages, and therefore many pairs of colliding messages can be generated.

The collision finder $\mathcal{A}$ proceeds as follows. An attack parameter is $k \geq 2$ and $\gamma \in \{0,1\}^{\ell_m}$. $\mathcal{A}$ generates L lists of $k-1$ random message blocks and for each one of them, stores the internal state $x_i^0 = (m_i^0, a_i^0, b_i^0, c_i^0), i \in [1, L]$ reached by inserting the listed message blocks. Now for each $i \in [1, L]$, $\mathcal{A}$ computes

$$m_i = b_i^0 \boxminus \gamma \,,$$

inserts the message block m_i to x_i^0, applies the round function and obtains some new state x_i. Note at this stage that for any $i \in [1, L]$, we have $c_i \boxminus m_i = \gamma$ where $x_i = (m_i, a_i, b_i, c_i)$. Since the a and b parts of x_i are outputs of the keyed permutation $\mathcal{P}$, it is easily seen that it is enough to have

$$(a_i, b_i) = (a_j, b_j)$$

for some $i \neq j \in [1, L]$ to provide a colliding pair of states (x_i, x_j) in the sense that $x_i \overset{k,k}{\sim} x_j$. If such a pair is found, then $\mathcal{A}$ picks an arbitrary non-zero block m_{k+1}, inserts it to both x_i and x_j and applies the final rounds which will lead to the same final state. This is expected to work as soon as the number of trials L is close enough to the birthday bound $2^{(\ell_a+\ell_m)/2}$.

11.2.2 One-block Internal Collisions

The particular structure of the message round function obviously guarantees that it is collision-free.

Theorem 6. *Let M, M' be two distinct message blocks for Shabal hash function. For any possible value for the Shabal internal state, (A, B, C), and for any possible value for the counter w, we have:*

$$\mathcal{R}(M, A, B, C, w) \neq \mathcal{R}(M', A, B, C, w).$$

Proof. This comes from the fact that $\mathcal{R}(M, A, B, C, w) = (A', B', C', w+1)$ with $B' = C - M$, implying that there is no collision on part B of the internal state. □

This implies that if a pair of messages M and M' which differs on a single block leads to an internal collision, then M and M' do not differ on their last block. However, this property obviously does not imply that any two distinct one-block messages lead to different hash values.

11.3 Differential Attacks

Most of the collision attacks against hash functions that have been published for now consist in finding a set of message pairs that are to follow a differential trail (*i.e.,* a sequence of differences in internal states) that ends to a non-difference and next in estimating the probability of the trail. Thus, the first step in order to mount a differential attack against Shabal is to get a trail with non-zero probability. There is no systematic method to find a trail unless a simple backtracking process is used. Thus, the classical method consists in searching for some particular differential trails which can be handled more easily, such as truncated differential trails, symmetric differential trails or differential trails without any difference in register A.

11.3.1 Truncated Differential

At Asiacrypt 2007, Peyrin [35] found a collision attack against Grindahl using truncated differential trails. A truncated differential trail is a binary differential trail where each bit means that there is a difference or not in an input word. This approach has also been adopted by Bouillaguet and Fouque [11] for the analysis of a reduced version of RadioGatún. It enabled the authors to discover differential trails with better properties than differential trails obtained through a backtracking algorithm given by the authors of RadioGatún.

Truncated differential trails for Shabal then correspond to differential trails for Weakinson-1bit, *i.e.,* for the weakened version of Shabal with 1-bit words. The existence of such trails is discussed in Section 11.5.

Truncated differential trails can be used for breaking Grindahl because Grindahl is a byte-oriented hash function, using a simple message round function. However, it seems highly improbable that these differential trails can be exploited to derive differential trails on the complete version of Shabal since Shabal deals with 32-bit words, reducing the probability of a truncated difference cancelation. Indeed, most operations contributing to the security of $\mathcal{P}$ in Shabal disappear when the word length is reduced to 1: rotations and the nonlinear functions $\mathcal{U}$ and $\mathcal{V}$ are replaced by the identity function in Weakinson-1bit. Thus, differential trails found on Weakinson-1bit do not take into account properties of these operations and it seems unlikely they could be adapted into a differential trail on Shabal. Therefore, Shabal should be immune against this kind of attacks.

11.3.2 Differential Trails without any Input Difference for $\mathcal{U}$ and $\mathcal{V}$

A good strategy for finding a differential trail might be to search for trails which do not cause any input difference for both nonlinear functions $\mathcal{U}$ and $\mathcal{V}$ since $\mathcal{U}$ and $\mathcal{V}$ are the only components in $\mathcal{P}$ whose algebraic degree exceeds 2.

Therefore, it is important to ensure that any input difference leads to a difference on the inputs of either $\mathcal{U}$ or $\mathcal{V}$. This is guaranteed by the following result.

Theorem 7. *Let M_0, M'_0, M_1, M'_1 be 4 message blocks for Shabal hash function, with $M_0 \neq M'_0$. Let (A, B, C) and w be any possible value for the Shabal internal state and for the counter. During the simultaneous computations of $\mathcal{R}(M_1, \mathcal{R}(M_0, A, B, C, w)))$ and $\mathcal{R}(M'_1, \mathcal{R}(M'_0, A, B, C, w))$, there is at least one difference between the inputs of one of the $\mathcal{U}$ or one of the $\mathcal{V}$ functions.*

Proof. Since we insert a difference in the first message block, there is a difference in the B part of the internal state before the $\mathcal{P}$ function. Then one of the following three cases happens:

1. A difference occurs in one of the first $(16p-1)$ new values of A computed during one of the $\mathcal{P}$ computations. Then the inputs of the following $\mathcal{V}$ function are different.

2. During the first $\mathcal{P}$ computation, a difference occurs on the last computed word of A at Step $16p$. But, this word is modified neither by the final transformation in $\mathcal{P}$ (because there is no difference in C), nor by the message insertion, nor by the counter addition. It is then

used as the input of $\mathcal{V}$ in the first step of the next computation of $\mathcal{P}$ because r divides the number of steps $16p$.

3. No difference occurs in the A values of the first $\mathcal{P}$ function. Then, there is no collision in register B at the end of the first $\mathcal{P}$ since the difference Δ'_j between the j-th words of B satisfies $\Delta'_j = \Delta_j \lll 3$ where Δ_j is the initial difference after the $\lll 17$ rotation. Therefore, there is a difference on the C input of the second $\mathcal{P}$ function, and no difference in the initial A values. As a consequence, either a difference occurs on an intermediate A value, which is an input of $\mathcal{V}$, or the difference on C implies differences on the inputs of $\mathcal{U}$.

In all these cases, we found a difference between the inputs of a given $\mathcal{U}$ or $\mathcal{V}$ function, which proves the theorem. □

It is worth noticing from the proof that the previous property does not hold if only one of the functions $\mathcal{U}$ or $\mathcal{V}$ is used.

11.3.3 Differential Trails without any Difference in A

Since a difference in register A propagates very fast to B, another good strategy for finding a differential trail may be to search for trails which do not cause any difference in A.

Let $S = (A^0, B^0, C^0)$ be a given internal state of Shabal. For a fixed message block M, we want to find another message block M' so that, for both message insertions, register A always contains the same value during the whole computation of $\mathcal{P}$.

Let δ_t denote the difference between the t-th message words, $0 \leq t \leq 15$, and let Δ_t denote the difference between the t-th words in register B after the $\lll_{17}$ rotation:

$$B_t \oplus \Delta_t = B'_t.$$

Using that

$$\begin{aligned} B_t &= (M_t + B_t^0) \lll 17 \\ B'_t &= \left((\delta_t \oplus M_t) + B_t^0\right) \lll 17 \end{aligned}$$

we deduce that, for any t, $0 \leq t \leq 15$, we have

$$\delta_t = M_t \oplus \left[\left((\Delta_t \ggg 17) \oplus (M_t + B_t^0)\right) - B_t^0\right]. \tag{11.2}$$

Now, we denote by A_{12+t} (resp. by B_{16+t}), $0 \leq t < 16p$, the new word in A (resp. in B) computed at Step t. We want to find a condition for having a collision in all A_{12+t}, $0 \leq t < 16p$. For any t, $0 \leq t < 16p$, we have

$$A_{12+t} = (\overline{B_{6+t}} \wedge B_{9+t}) \oplus B_{13+t} \oplus M_t \oplus \mathrm{C}st_t$$

where $\mathrm{C}st_t$ only depends on $A^0 \oplus W^0$, B^0, C^0 and M, and is the same for both message blocks because all A_t collide. Thus, a collision for A_{12+t} corresponds to the following condition

$$(\overline{B_{6+t}} \wedge B_{9+t}) \oplus B_{13+t} = ((\overline{B_{6+t}} \oplus \Delta_{6+t}) \wedge (B_{9+t} + \Delta_{9+t})) \oplus (B_{13+t} \oplus \Delta_{13+t}) \oplus \delta_t,$$

or equivalently to

$$(\Delta_{6+t} \wedge \Delta_{9+t}) \oplus (\Delta_{6+t} \wedge B_{9+t}) \oplus (\Delta_{9+t} \wedge (B_{6+t} \oplus \mathbf{1})) \oplus \Delta_{13+t} = \delta_t. \tag{11.3}$$

This condition leads to the following lower bound on the Hamming weight of any differential trail which does not generate any difference in register A.

Theorem 8. *Let M and M' be two message blocks for Shabal hash function. Let S and w be any possible values for the Shabal internal state and counter. If there is no difference in register A during the simultaneous computations of $\mathcal{R}(M, S, w)$ and $\mathcal{R}(M', S, w)$, then M and M' differ on at least 7 words.*

Proof. Equation (11.3) implies that, if $\Delta_{6+t} = \Delta_{9+t} = 0$ for some t, then $\Delta_{13+t} = \delta_t$. Therefore, either both δ_t and δ_{13+t} vanish or none of them vanishes. We have then performed an exhaustive search over all 2^{16} possible truncated differential (*i.e.,* the binary vector whose bits mean that there is a difference or not in an input word). It shows that the Hamming weight of any truncated differential satisfying the previous condition is at least 7. For instance, the vector with nonzero differences in words 0, 1, 2, 4, 7, 9 and 11 fulfills the previous condition. More precisely, 48 such patterns of Hamming weight 7 exist, corresponding to the rotated versions of 3 patterns only. □

11.3.4 Symmetric Differential Trails

Symmetric differential trails, *i.e.,* with all δ_t and Δ_t in $\{0, \mathbf{1}\}$, are usually used for mounting a differential attacks since their search is much simpler than the search for a general differential trail. For instance, symmetric trails suppress the impacts of all rotations. Such differential trails have been investigated in [5] and [28] for analyzing the security of RadioGatún.

To find such differential trails, the adversary first has to study the propagation of such differences through the elementary functions used in Shabal. One can first notice that $\mathcal{U}$ and $\mathcal{V}$ are the only elementary functions in $\mathcal{P}$ which cannot transform an all-1 difference to either an all-0 or all-1 difference as stated in the following proposition.

Proposition 1. *For any $x \in \{0,1\}^{32}$, we have*

$$\begin{aligned} \mathcal{U}(x \oplus \mathbf{1}) \oplus \mathcal{U}(x) &\notin \{0, \mathbf{1}\} \\ \mathcal{V}(x \oplus \mathbf{1}) \oplus \mathcal{V}(x) &\notin \{0, \mathbf{1}\} \end{aligned}$$

Proof. For any $x = (x_0, \ldots, x_{31}) \in \{0,1\}^{32}$, we have:

$$\begin{aligned} \mathcal{U}(x) &= 3 \sum_{i=0}^{31} x_i 2^i \bmod 2^{32} \\ &\equiv 3x_0 + 6x_1 \bmod 4 \\ &\equiv x_0 + 2(x_0 \oplus x_1) \bmod 4. \end{aligned}$$

Therefore, $y = \mathcal{U}(x)$ satisfies $y_0 = x_0$ and $y_1 = x_0 \oplus x_1$. It follows that

$$\mathcal{U}(x \oplus \mathbf{1}) \oplus \mathcal{U}(x) \equiv 1 \bmod 4,$$

implying that $\mathcal{U}(x \oplus \mathbf{1}) \oplus \mathcal{U}(x) \notin \{0, \mathbf{1}\}$.

Similarly,

$$\begin{aligned} \mathcal{V}(x) &= 5 \sum_{i=0}^{31} x_i 2^i \bmod 2^{32} \\ &\equiv x_0 + 2x_1 + 4(x_0 \oplus x_2) \bmod 8, \end{aligned}$$

implying that

$$\mathcal{V}(x \oplus \mathbf{1}) \oplus \mathcal{V}(x) \equiv 3 \bmod 8.$$

□

Proposition 1 ensures that no symmetric trail that involves a difference in the inputs of $\mathcal{U}$ or $\mathcal{V}$ can be found. Theorem 7 then implies that there is no symmetric trail for two Shabal rounds or more, in the following sense.

Theorem 9. *Let M_0, M'_0, M_1, M'_1 be four message blocks for Shabal hash function, such that $M_0 \neq M'_0$ and all words in $(M_0 \oplus M'_0)$ and in $(M_1 \oplus M'_1)$ are symmetric, i.e., equal either to 0 or to the all-1 word. Let (A, B, C) and w be an internal state and a counter value for Shabal such that all words in $(B + M_0) \oplus (B + M'_0)$ and $(C - M_0 + M_1) \oplus (C - M'_0 + M'_1)$ are symmetric. Then, there is an elementary step during the simultaneous computations of $\mathcal{R}(M_1, \mathcal{R}(M_0, A, B, C, w))$ and $\mathcal{R}(M'_1, \mathcal{R}(M'_0, A, B, C, w))$ such that at least one difference (i.e., XOR) between the values of A or B is not symmetric.*

Since there is no symmetric trail for two Shabal rounds or more, the best goal for an attacker is to find a symmetric trail on one message block, starting from colliding states, and targeting a given symmetric difference in the outputs of the round function.

In the case of symmetric differential trails, the conditions exhibited in Section 11.3.3 can be simplified since we have that the differences Δ_i between the i-th words in register B after the ($\lll 17$) rotation are equal to the differences δ_i between the i-th messages words (this obvious property can be deduced from (11.2)). Then, it follows from (11.3) that there is no difference in A_{t+12} for all $0 \leq t < 16p$ if and only if

$$(\Delta_{6+t} \wedge \Delta_{9+t}) \oplus (\Delta_{6+t} \wedge B_{9+t}) \oplus (\Delta_{9+t} \wedge (B_{6+t} \oplus \mathbf{1})) \oplus \Delta_{13+t} \oplus \Delta_t. \tag{11.4}$$

This comes from the fact that $\Delta_{16+t} = \Delta_t$ for all t. Now, by applying (11.4) for $0 \leq t \leq 15$, we can see that the value of each pair $(\Delta_{t+6 \bmod 16}, \Delta_{t+9 \bmod 16})$, $0 \leq t \leq 15$, provides a bitwise condition relating the input difference Δ and some words of the initial state of B. Moreover, (11.4) applied to $16 \leq t < 16p$ and combined with $B_{16+t} = (B_t \lll 1) \oplus A_{12+t} \oplus \mathbf{1}$ leads to 3 additional bitwise relations between the successive values in register A. Table 11.1 summarizes these bitwise relations. Now, an exhaustive search on all 2^{16} symmetric differential trails can be performed.

$\Delta_{t+6 \bmod 16}$	$\Delta_{t+9 \bmod 16}$	Condition	Relations
0	0	$\Delta_t \oplus \Delta_{t+13} = 0$	
1	0	$B_{9+t} = \Delta_t \oplus \Delta_{t+13}$	$A_{21+t} = A_{37+t} = A_{53+t} = \mathbf{1}$
0	1	$B_{6+t} = \Delta_t \oplus \Delta_{t+13} \oplus \mathbf{1}$	$A_{18+t} = A_{34+t} = A_{50+t} = \mathbf{1}$
1	1	$B_{6+t} \oplus B_{9+t} = \Delta_t \oplus \Delta_{t+13} \oplus \mathbf{1}$	$A_{18+t} \oplus A_{21+t} = A_{34+t} \oplus A_{37+t} = A_{50+t} \oplus A_{53+t} = 0$

Table 11.1: Conditions derived from (11.4) for symmetric differential trails

All trails with $(\Delta_{t+6 \bmod 16}, \Delta_{t+9 \bmod 16}) = (0, 0)$ and $\Delta_t + \Delta_{t+13} \neq 0$ are eliminated. For the remaining trails, we check whether the conditions imposed on the B variables are consistent. It follows that all trails which fulfill these conditions are such that the number of t, $0 \leq t \leq 15$, such that $(\Delta_{t+6 \bmod 16}, \Delta_{t+9 \bmod 16}) \neq (0, 0)$ is at least 11. The minimal number of Boolean relations on A derived from Table 11.1 is then $3 \times 11 \times 32$, resulting in 1054 equations for the 3-loop version of Shabal. For any given initial state of Shabal this number must be compared with the size of the message block, *i.e.,* 512 bits. It follows that, at least 544 relations cannot be fulfilled by a deterministic algorithm. We then expect that symmetric differential trails exist for at most a fraction 2^{-544} of the possible internal states.

11.4 Fixed Points

In all previously considered differential attacks, internal collisions are obtained by considering pairs of messages with the same length. However, internal collisions may be searched for messages of different length. A strategy in such attacks consists in exploiting the existence of fixed points for the message round function. The use of a counter at each message round then avoids the existence of trivial fixed point for Shabal.

11.5 Generic Attacks against Weakinson-1bit

When considering weakened versions of Shabal, generic attacks become practical. Thus it becomes possible to find collisions. Such collisions can then be used to derive differential trails. However, it seems highly unlikely that these differential trails can be exploited to derive differential trails on the complete version of Shabal, as explained in Section 11.3.1. In the case of Weakinson-1bit, it is possible to perform an exhaustive search over all possible one-block message differences. In

particular, we have found that a differential trail which does not cause any difference in register A during the first loop can be found for roughly 56 % of the possible pairs (B, M).

11.6 (Second)-preimage Attack against Weakinson-NoFinalUpdateA

We now exhibit a preimage attack and a second-preimage attack against Weakinson-NoFinalUpdateA, *i.e.,* the weakened variant of Shabal without the last update loop in $\mathcal{P}$. On this weakened variant, the attack is faster than the generic attack for $p = 1$, and has the same complexity as the generic attack for $p = 2$. The attack mainly relies on the following weakness of Weakinson-NoFinalUpdateA with $p = 1$: given the outputs of $\mathcal{P}$, the attacker is able to choose a message block M such that part B of $(\mathcal{P}^{-1}_{M,C}(A', B') - M)$ has any prescribed value.

11.6.1 Attack against Weakinson-NoFinalUpdateA with $p = 1$

We first describe the second-preimage attack. Let $\mathcal{M}$ be a k-block message. As we have a counter, we search for another message $\mathcal{M}'$ of the same length as $\mathcal{M}$. We split $\mathcal{M}'$ into three parts: the first k_1 blocks, 2 intermediate blocks and the last $(k - k_1 - 2)$ blocks. Let us now randomly choose N_1 vectors $(M_1, \ldots, M_{k_1})$ of k_1 message blocks. We compute N_1 internal states S_{k_1} obtained for each of these messages from the initial state. Similarly, we randomly choose N_2 vectors $(M_{k_1+3}, \ldots, M_k)$ of $k-k_1-2$ message blocks. From the final internal state obtained when $\mathcal{M}$ is hashed, we compute backwards the internal states S_{k_1+2}, before the insertion of M_{k_1+3} for these N_2 messages. Now, we can use the 2 intermediate blocks M_{k_1+1} and M_{k_1+2} to find a collision of the 16 words on the B-part of S_{k_1+1}, which means that finding a collision on the rest of the internal state is enough to find a message which has the same hash value as $\mathcal{M}$.

Let β be a target value in $\{0,1\}^{512}$ for the B-part of S_{k_1+1}. For each of the N_1 values of $S_{k_1} = (A_{k_1}, B_{k_1}, C_{k_1})$, we choose $M_{k_1+1} = C_{k_1} - \beta$, implying that the B-part of S_{k_1+1}, which corresponds to $C_{k_1} - M_{k_1+1}$, equals β.

The difference now with Shabal, is that we are able to go backwards. Let $A_0, \ldots, A_{11}$ and $B_0, \ldots, B_{15}$ be the values in registers A and B of S_{k_1+1}, and let $A_{16}, \ldots, A_{27}$ and $B_{16}, \ldots, B_{31}$ be the values of A and B after applying $\mathcal{P}$. These outputs are known since they are included in the internal state S_{k_1+2}. By definition, we have, for any $0 \leq i \leq 15$,

$$(B_i \lll 1) = B_{i+16} \oplus A_{i+12} \oplus \mathbf{1}. \tag{11.5}$$

This means that B_i is entirely determined by S_{k_1+2}, for i from 4 to 15. We can then choose the words $M_{k_1+2,i}$ of index i in the message block M_{k_1+2} for $4 \leq i \leq 15$ so that:

$$B_i = (\beta_i + M_{k_1+2,i}) \lll 17.$$

For finding the values of $M_{k_1+2,i}$, $0 \leq i \leq 3$, which lead to the expected values β_i, $0 \leq i \leq 3$, we now compute the values of A_{12}, A_{13}, A_{14} and A_{15} with the following equation for i from 12 to 15:

$$A_{i+12} = \overline{B_{i+6}} B_{i+9} \oplus B_{i+13} \oplus M_{k_1+2,i} \oplus \mathcal{U}(A_i \oplus \mathcal{V}(A_{i+11} \lll 15) \oplus (C_{8-i} + M_{k_1+2,8-i}))$$

or equivalently,

$$A_i = \mathcal{V}(A_{i+11} \lll 15) \oplus (C_{8-i} + M_{k_1+2,8-i}) \oplus \mathcal{U}^{-1}(A_{i+12} \oplus \overline{B_{i+6}} B_{i+9} \oplus B_{i+13} \oplus M_{k_1+2,i}).$$

Computing $A_{12}, \ldots, A_{15}$ actually involves $A_{24}, \ldots, A_{27}$, some B_i for $i \geq 18$ and $C_9, \ldots, C_{12}$ which are known since they can be deduced from S_{k_1+2} and $M_{k_1+2,9}, \ldots, M_{k_1+2,12}$. Then, we can compute $B_0, \ldots B_3$ from (11.5), and we choose $M_{k_1+2,0}, \ldots, M_{k_1+2,3}$ such that

$$B_i = (\beta_i + M_{k_1+2,i}) \lll 17, \ \forall 0 \leq i \leq 3.$$

Finally we have found N_1 prefixes $M_1, \ldots, M_{k_1}, M_{k_1+1}$ and N_2 suffixes $M_{k_1+2}, M_{k_1+3}, \ldots, M_k$ which lead to two sets of internal states S_{k_1+1} whose B-parts are all equal to a given value β. For $N_1 = N_2 = 2^{32\times\frac{16+12}{2}} = 2^{32\times 14}$, we then find a collision between both sets of internal states. Therefore, a message $\mathcal{M}'$ with the same hash value as $\mathcal{M}$ has been found within $2^{32\times 14}$ calls to the message round function, which is better than the generic second-preimage attack for a hash length $\ell_h = 512$.

A preimage attack can be mounted by the same method. It consists in randomly choosing a final internal state whose part C is (partially) determined by the targeted hash value. Then, the previously described attack enables to find a message which leads to this final internal state.

11.6.2 Attack against Weakinson-NoFinalUpdateA with $p = 2$

For Weakinson-NoFinalUpdateA with $p = 2$, the same method can be used. But, for $p = 2$, we are able to fix only 12 words of the B-part of S_{k_1+1}. Actually, if we consider the backward computation from all N_2 values for S_{k_1+2}, the known variables corresponding to S_{k_1+2} are now $A_{32}, \ldots, A_{43}$ and $B_{32}, \ldots, B_{47}$. As in the previous case, all B_i, for i from 20 to 31, are completely determined by S_{k_1+2} and B_{i+16} using (11.5). This does not require any condition on M_{k_1+2}. Now, we assign $M_{k_1+2,12}$, $M_{k_1+2,13}$, $M_{k_1+2,14}$ and $M_{k_1+2,15}$ to some fixed arbitrary values, *e.g.,* 0. Then, we want that the input of the first elementary step of $\mathcal{P}$ at round $(k_1 + 2)$ satisfies

$$B_i = (\beta_i + M_{k_1+2,i}) \lll 17, \ \forall 12 \le i \le 15.$$

Several intermediate values of A_i, B_i and M_i can now be deduced by using the following relations:

$$\begin{aligned} A_{i+12} &= \overline{B_{i+6}} B_{i+9} \oplus B_{i+13} \oplus M_{k_1+2,i} \oplus \mathcal{U}(A_i \oplus \mathcal{V}(A_{i+11} \lll 15) \\ &\quad \oplus (C_{8-i} + M_{k_1+2,8-i})) \qquad (11.6) \\ (B_i \lll 1) &= B_{i+16} \oplus A_{i+12} \oplus \mathbf{1}. \qquad (11.7) \end{aligned}$$

Actually, we have the following deduction sequence from the knowledge of B_{13}, B_{14}, B_{15}: from (11.7) with $13 \le i \le 15$, we obtain A_{25}, A_{26}, A_{27}. From (11.6) with $25 \le i \le 27$, we obtain $M_{k_1+2,i}$ for $9 \le i \le 11$. Thus, we obtain the prescribed values for words 9 to 11 of part B of S_{k_1+1} if and only if

$$B_i = (\beta_i + M_{k_1+2,i}) \lll 17, \ \forall 9 \le i \le 11.$$

Now, the values of $B_9, B_{10}, B_{11}, B_{12}$ determine $A_{21}, A_{22}, A_{23}, A_{24}$ by applying (11.7) with $9 \le i \le 12$.

On the other hand, the knowledge of $M_{k_1+2,i}$ for $9 \le i \le 15$ leads to $A_{28}, A_{29}, A_{30}, A_{31}$ by applying (11.6) with $28 \le i \le 31$. Then, $A_{28}, A_{29}, A_{30}, A_{31}$ determine $B_{16}, B_{17}, B_{18}, B_{19}$ by (11.7) with $16 \le i \le 19$.

Now, $A_{23}, A_{24}, A_{25}, A_{26}, A_{27}$ determine $A_{12}, A_{13}, A_{14}, A_{15}$ by (11.6) with $12 \le i \le 15$, since B_j for $j \ge 18$ and $M_{k_1+2,j}$ for $9 \le j \le 15$ are known. ¿From $A_{12}, A_{13}, A_{14}, A_{15}$ and $B_{16}, B_{17}, B_{18}, B_{19}$, we obtain B_0, B_1, B_2, B_3 by applying (11.7) with $0 \le i \le 3$. Therefore, for

$$M_{k_1+2,i} = (B_i \ggg 17) - \beta_i, \ \ \forall 0 \le i \le 3,$$

we obtain the prescribed values for the first 4 words of part B of S_{k_1+1}.

Message blocks $M_{k_1+2,i}$ for $5 \le i \le 8$ can be deduced from (11.6) for $21 \le i \le 24$ since $A_{21}, A_{22}, A_{23}, A_{24}$ and $M_{k_1+2,0}, M_{k_1+2,1}, M_{k_1+2,2}, M_{k_1+2,3}$ are known. The knowledge of A_{27}, A_{28}, $M_{k_1+2,0}$ and $M_{k_1+2,8}$ determines A_{16} by applying (11.6) with $i = 16$. From A_{16} and B_{20}, we deduce B_4 by (11.7) with $i = 4$. We finally choose $M_{k_1+2,4}$ so that

$$B_4 = (\beta_4 + M_{k_1+2,4}) \lll 17.$$

A message block M_{k_1+2} has then been found so that 12 words in part B of S_{k_1+1} are equal to the corresponding words in β (*i.e.,* all words except words 5 to 8). A (second)-preimage can

then be found as soon as a collision on the remaining $16+4+r$ words of the internal state can be found. This requires $2^{32\times 16}$ calls to the message round function, *i.e.,* the same complexity as the generic second-preimage attack for a hash length $\ell_h = 512$.

This attack does not apply to Shabal: the final transformation in $\mathcal{P}$, *i.e.,* the last update loop, has been chosen in order to eliminate this weakness as explained in Section 4.2.6.

11.7 Generic Attacks Against Merkle-Damgård-Based Hash Functions

Most practical hash functions, such as SHA-1 or SHA-2, are iterated hash functions based on the well-known Merkle-Damgård construction. Due to certain structural weaknesses of the Merkle-Damgård (MD) construction, MD-based hash functions are vulnerable to some generic attacks such as length-extension attacks [32] or multicollision attacks [25]. In this section, we investigate the applicability of these attacks on Shabal.

11.7.1 Length-extension Attacks

The well-known Merkle-Damgård construction has an undesirable property called *length extension.* It means that once an attacker has one collision, *i.e.,* two distinct messages M_1 and M_2 with $|M_1| = |M_2| = k\dot{\ell}_m$ $(k > 0)$ such that $H(M_1) = H(M_2)$, then for any suffix M it also holds that $H(M_1\|M) = H(M_2\|M)$.

The message extension attack can be applied to the Shabal hash function only if an internal collision occurs before the final rounds, or in the final rounds but before the second call to the message round function. In the latter case, the internal collision can be transformed into an internal collision before the final rounds by appending the same block message to the two messages leading to an internal collision. Thus, for simplicity reasons, we consider only the case where the internal collision occurs before the three final rounds. Once an adversary has found two distinct messages M_1 and M_2 such that $|M_1| = |M_2| = k\dot{\ell}_m$ $(k > 0)$ and $\mathsf{Shabal}(M_1) = \mathsf{Shabal}(M_2)$, it becomes possible to extend the collision. Indeed for every suffix M, we then have $\mathsf{Shabal}(M_1\|M) = \mathsf{Shabal}(M_2\|M)$. Note that it is necessary that $|M_1| = |M_2|$ due to the use of a counter in Shabal. As explained in Section 11.2, the complexity for finding an internal collision in Shabal-ℓ_h (with $\ell_h \in \{192, 224, 256, 384, 512\}$) is expected to require the order of $2^{(\ell_a+\ell_m)/2}$ iterations of the message round function. Thus, the complexity of any length-extension attacks is expected to require at least 2^{256} calls to the message round function, independently of ℓ_h. For more details about recent investigations on these attacks, see [29].

11.7.2 Multi-Collisions

The multi-collision attack [25] applies to iterative hash functions and exploits the fact that the complexity for finding 2^u messages which have the same hash value corresponds to the complexity for finding u internal collisions (from u prescribed internal states). The 2^u-collision attack against a hash function H actually consists in finding u pairs of messages (M_i, M_i'), $1 \leq i \leq u$, of the same length k_i such that both inserting M_i and inserting M_i' from the internal state S_{i-1} lead to the same internal state S_i. From such pairs, 2^u messages of length $k_1 + \ldots + k_u$ can be constructed by concatenating the previous u messages and choosing either M_i or M_i' for each $1 \leq i \leq u$. The complexity for finding a 2^u-multicollision then corresponds to the complexity of finding u internal collisions.

As explained in Section 11.2, the complexity for finding an internal collision in Shabal-ℓ_h (with $\ell_h \in \{192, 224, 256, 384, 512\}$) is expected to require the order of $2^{(\ell_a+\ell_m)/2}$ calls to the message round function. Thus, in Shabal-ℓ_h, $\ell_h \in \{192, 224, 256, 384, 512\}$, the complexity for finding a 2^u-multi-collision is expected to require at least $u \cdot 2^{(\ell_a+\ell_m)/2}$ calls to the message round function.

11.8 Slide Attacks

Slide attacks apply on hash functions (see *e.g.,* [24]) but there is no obvious way to transform them into practical attacks. In the hash function setting, a slide property would allow to detect a non-random behavior of the hash function. We have shown in Section 5 that the $\mathcal{P}$-based construction (see Section 2.2) is indifferentiable from a random oracle up to $2^{(\ell_a+\ell_m)/2} > 2^{256}$ calls to the message round function. Since Shabal is a particular instantiation of the $\mathcal{P}$-based construction, there is no slide property due to the operating mode of the hash function Shabal. Thus, there is no obvious slide attack against the Shabal hash function.

11.9 Algebraic Distinguishers and Cube Attacks

Algebraic distinguishers consist in computing some coefficients of the algebraic normal form of a Boolean function, and in checking whether these binary coefficients are randomly distributed or not. They rely on the fact that the coefficient of a monomial of degree d in the algebraic normal form corresponds to the sum modulo 2 of the values of the function when the input varies in a d-dimensional linear space. Therefore, a distinguishing attack can be mounted if the attacker has access to such 2^d evaluations of a Boolean function related to the considered primitive. This basic principle has been used for a long time for block ciphers in the so-called higher-order differential attacks. It has been introduced by Saarinen [39] for chosen-IV attacks against stream ciphers and been developed in [21]. Finally, Fischer, Khazaei and Meier [22] and Dinur and Shamir [20] have recently shown how such key-recovery attacks can be mounted based on the same technique. Moreover, [20] exhibits an algorithm for finding the monomials which must be considered in the attack, even if the algebraic normal form of the studied Boolean function is not known to the attacker.

Such a distinguishing attack may apply in the context of hash functions. For n-bit messages, each bit of the hash value can be seen as a Boolean function $(m_1, \ldots, m_n) \mapsto h(m_1, \ldots, m_n)$ in n variables. It is clearly suitable that this function behaves like a random function. This means that, for any subset of the input bits, $I \in \{1, \ldots, n\}$, the superpoly of I in h behaves like a random polynomial, where the superpoly of of I in h corresponds to the $(n - |I|)$-variable function

$$(m_i, i \notin I) \mapsto \bigoplus_{(m_i, i \in I) \in \{0,1\}^{|I|}} h(m_1, \ldots, m_n).$$

Thus, the fact that h has a high degree and is not sparse is a priori sufficient to resist such attacks. The three final rounds in Shabal are expected to ensure that each coordinate of final internal state is a random-looking polynomial of the message bits (see Section 4.4 for some details on the degree).

11.10 Attacks Taking Advantage of The Chosen Constants

To prevent the existence of possible "trapdoors", the provenance of constants or tables used in Shabal have been justified. There is no table used in Shabal. The only constants specified in Shabal-ℓ_h where $\ell_h \in \{192, 224, 256, 384, 512\}$, are the initial values IV_{ℓ_h} of the state (A, B, C). Rationale of this choice is given in Section 4.5.

11.11 Differential Attack on HMAC-Shabal

At Crypto 2007, Fouque *et al.* [23] proposed a new attack to take into account differential collision paths in some hash functions such as MD4 and MD5 to recover some key bits of HMAC-MD4. The idea is to find differential collision paths that depend on bits of the IV. This kind of attack has been extended to MD4 at Eurocrypt 2008 by Wang *et al.* in [40]. These attacks use the fact that

differential paths with high probability exist and are easy to compute. This is the case for MD4 and the pseudo-collision on MD5 exhibited by den Boer and Bosselaers [19]. These attacks can be applied to HMAC-Shabal if it is possible to find such differential paths. Moreover, to recover the outer key of HMAC, such paths need to be constrained to use only one message block with constrained variation since the last call to the hash function has many zeroes. Finally, it is worth saying that distinguishing attacks are also possible using differential paths. But all these attacks rely on the fact that such differential paths can be easily found.

Pseudo-Random Function.

It is well-known that a good Pseudo-Random Function allows to construct a secure MAC algorithm. For instance, Bellare at Crypto 2006, in [2], proved that if the compression function of the underlying hash function behaves as a good PRF, and that a related PRF (by inverting the message and the key space) is secure under a specific related-key attack, then HMAC is a good MAC. Then, if the compression function of Shabal is a good PRF, then the HMAC-Shabal will be a good MAC. Finally, since no preservation property for PRF has been provided for the mode of operation of Shabal here, we cannot argue that Shabal$(k||M)$ is a secure MAC if the compression function of Shabal is a PRF.

Part 2.B.6

A Statement that Lists and Describes the Advantages and Limitations of the Algorithm

Chapter 12

Advantages and Disadvantages of Shabal

Contents

In this chapter, we try to present a first comparison of Shabal with other hash functions. Our comparison is made by exhibiting advantages and disadvantages of Shabal, as well as by gathering measures on some implementations of Shabal and other hash functions.

12.1 Simplicity of Design

One aim of Shabal is to be secure while keeping a simple design. This simplifies both study and implementation. Each element of Shabal was carefully weighted with regards to security and implementation cost; whenever possible, we favored the simplest choice.

The final result is that our function — or at least, our mode — is really simple to describe. The pictures of Figures 2.1, 2.2 and 2.3 give an overview of the design, and the only non-natural part of Shabal is certainly the permutation description which is described in Section 2.3.2 (see also Figure 2.4). The rest of the function is relatively easy to understand and to remember.

12.2 Provable Security

Our construction is based on a generic construction (see Section 2.2) which is provably indifferentiable in the ideal cipher model from a random oracle, as well as provably one-way and second-preimage resistant (see Chapter 5). This fact is one of the key points of Shabal, and thus, we consider this proof as one of the main advantages of Shabal compared to several other hash functions. Of course, the proof is made in the ideal cipher model, which is an idealized model. It has notably been shown to be inadequate for certain reasons [12], but it is however widely believed that a proof of a non-pathological[1] scheme gives a certain confidence in its real-world security.

[1] *i.e.,* , a scheme that was not made to show a difference between the ideal cipher model and the standard model.

This explains why we have been driven by the security proof during the design of Shabal. However, since we know that the standard model is different from the ideal cipher model, we have added to the design some extra elements that were not requested by the proof (*i.e.,* the scheme was provable, even without these elements). These are notably the use of a block counter w and the double insertion of the message both in B and via the keyed permutation. Furthermore, we have added some tools to provide a security margin: this is the role of A, which can be extended or shrunk via the parameter r.

12.3 Software Implementation Considerations

Shabal was primarily designed for software implementation, even if we tried not to limit to this context (see Chapters 7 and 8). Let us describe here the advantages and disadvantages of Shabal when implemented either on a computer or on a power-limited device.

12.3.1 Word Size

The design is built around elementary operations on 32-bit words, since those operations are natively provided by most platforms. This obviously includes computers (high-end servers, desktop systems...) but also handheld devices, many embedded systems and even the recent generation of smart cards. Indeed, we wanted to design a function which runs efficiently on the most constrained devices. Many high-end computers now provide efficient operations on 64-bit words as well, but these are not available on smaller systems, hence we refrained from using words wider than 32 bits. This contrasts with other hash function designs such as SHA-512 and RadioGatún[64].

12.3.2 Very Few Requested Instructions to Code Shabal

In order to further ease the efficient implementation of Shabal on constrained devices, we tried to limit the number of distinct primitive operations which Shabal uses. Namely, we use additions, subtractions, bitwise Boolean combinations ($\oplus$ and $\wedge$, and bitwise negation), multiplications by 3 and 5, and rotations by a constant amount. All these operations work with 32-bit operands. Bitwise negation is easily achieved with an exclusive or ($\oplus$) with the all-ones constant. Multiplications by 3 and 5 can be implemented with mere additions (respectively 2 and 3), or by combining an addition and a logical shift. Should it be needed, subtraction can be implemented by ways of an addition with the two's complement of the second operand, which is a matter of flipping the bits and inserting an initial carry into the adder.

All those operations are provided natively by most platforms, either as one opcode, or possibly a small sequence (the word rotation is the operation which is most likely not to be implemented as a unique opcode). Sticking to a very small set of very common operations enhances portability and efficiency on a wide variety of platforms; it also helps when size is severely constrained, because units for those operations may then be shared by several invocations.

12.3.3 No S-Box

Shabal uses no S-Box. S-Boxes are a popular design element in cryptographic algorithms because they can be finely tuned to offer the optimal algebraic properties to defeat various types of attacks. However, S-Boxes are expensive to implement, not only in dedicated hardware, but also in software: S-Box access is a memory read by address, which exercises the caches and has a high latency, typically higher than numerical operations on modern processors. Moreover, small S-Boxes can only handle a few bits at a time, thus processing a full 32-bit word with S-Boxes will require several accesses; wider S-Boxes imply prohibitive costs in terms of code size. Besides, S-Box access works over the data cache in a typical CPU, and the data cache is a scarce resource (especially when we are processing huge amounts of data, which is why hash function performance matters in the first place).

In our design, the nonlinear functions $\mathcal{U}$ and $\mathcal{V}$ take the role of S-Boxes, albeit with weaker algebraic properties, but they are much less expensive to implement on most platforms.

12.3.4 Speed Measures

In this section, we give a comparison of Shabal with several other hash functions, on the architectures detailed in Chapter 8. This comparison is given in two tables, Table 12.1 and 12.2. In the tables, Shabal refers to implementations for all lengths $\ell_h \in \{192, 224, 256, 284, 512\}$, as the execution hardly depends on the output length. MD5 and SHA-1 times are just given for comparison, as these functions are known to be broken, respectively in practice and in theory.

	Bandwidth (MB/s), for long messages				
Platform	Shabal	MD5	SHA-1	SHA-256	SHA-512
Xeon, 64-bit	194.52	412.98	318.62	143.55	191.06
Xeon, 32-bit	147.90	419.43	232.41	127.22	45.19
Athlon64, 64-bit	140.91	336.60	239.67	115.21	151.23
Athlon64, 32-bit	103.64	338.72	174.31	108.68	32.42
C7	67.79	139.09	63.61	29.18	11.53
Broadcom MiPS	6.24	10.20	6.15	2.90	1.50

Table 12.1: Shabal performance compared with other hash functions (1)

	Bandwidth (MB/s), for long messages			
Platform	Shabal	WHIRLPOOL	RADIOGATÚN[32]	RADIOGATÚN[64]
Xeon, 64-bit	194.52	38.90	258.11	506.48
Xeon, 32-bit	147.90	26.42	181.99	55.92
Athlon64, 64-bit	140.91	21.79	225.58	443.69
Athlon64, 32-bit	103.64	15.68	143.55	126.03
C7	67.79	6.45	75.40	52.22
Broadcom MiPS	6.24	0.15	1.36	0.55

Table 12.2: Shabal performance compared with other hash functions (2)

The hash function implementations were extracted from the open-source `sphlib` library. Most notably, all these functions (including the Shabal optimized code) were implemented with the same optimization goals and efforts, with the same programming tools, by the same programmer. As such, relative performance of two functions on the same platform should be viewed as intrinsic to the functions themselves. Note that on the Broadcom MiPS platform, a specifically optimized SHA-256 implementation was used, with less loop unrolling; if the exact same C code was used on other platforms, the SHA-256 speed would drop to 1.76 MB/s.

In the tables, one can see the effect of our choice of sticking to 32-bit words: the performance hit is minimal when switching from a 64-bit platform to a 32-bit platform (there is a performance hit because the 64-bit instruction set on x86 CPU offers more registers than the 32-bit instruction set). Conversely, functions which rely on 64-bit operations (SHA-512, WHIRLPOOL and RADIOGATÚN[64]) greatly suffer from being run on a 32-bit only platform.

Our goal was to be as efficient as possible on all types of platforms; this goal is mostly achieved. Indeed, we can compute the ratio of the fastest bandwidth divided by the slowest bandwidth on our set of platforms; this ratio may be viewed as a crude measure for the portability, performance-wise, of the design. We get the following figures:

- on high-end platforms only (all the x86-compatible systems): 2.9 for Shabal, 4.9 for SHA-256, 17.3 for SHA-512, 6.5 for WHIRLPOOL, 3.4 for RADIOGATÚN[32] and 9.7 for RADIOGATÚN[64];

- on all considered platforms (including the Broadcom): 31.2 for Shabal, 49.5 for SHA-256, 127 for SHA-512, 259 for WHIRLPOOL, 190 for RADIOGATÚN[32] and 920 for RADIOGATÚN[64].

That Shabal exhibits the smallest ratios underlines our efforts for designs which are efficient on a wide range of systems.

From these measures, comparing Shabal with the five non-broken hash functions, we see that Shabal is third on Xeon (64-bit), fourth on Athlon64, second on Xeon 32-bit and C7, and first on the Broadcom MiPS. In terms of performance portability, Shabal wins. We would like to underline that even if the security proof (see Chapter 5) influenced the design of Shabal, possibly at the expense of raw performance, the achieved bandwidth remains decent, and is actually quite good on some low-end platforms, such as the Broadcom MiPS, which are industrially significant (hash function performance matters much more on limited systems which perform network-based cryptography all day long than on high-end desktop systems where I/O bandwidth is more limited than computing power).

12.3.5 Code Size

Code size is a critical part of the performance of a hash function and it is rarely well measured by benchmarks. Benchmarks (such as the one we presented in the previous section) run the function "alone", with the full processor caches at its disposal, which is not very representative of actual usage. Yet, on some platforms, the dramatic effect of code size can even be shown with a benchmark. In our previous example, we see that WHIRLPOOL, RADIOGATÚN[32] and RADIOGATÚN[64] are much slower than expected on the Boradcom MiPS platform (when compared to, for instance, SHA-512). This is because their code size exceeds the limited L1 code cache on that architecture (8 kB), implying expensive cache misses for each message block.

We measured the code size of the implementations of the functions on our three target architecture types. Tables 12.3, 12.4 and 12.5 list the code and data sizes. The "code" column measures the total code size for the complete implementation; the "update" column contains only those parts of the code which are on the execution path when hashing streamed data (*i.e.,* without initialization and finalization). The "state" is the size of the state structure which is maintained for a given hash computation; this is mutable data. The "data" column contains the size of the constant data tables which are accessed during main processing: these tables are not modified but they contribute to the data L1 cache pressure (among our test functions, only WHIRLPOOL uses such tables; precomputed IV are not included since they are used only during initialization of the hash computation).

	Code and data size			
Function	code	update	state	data
MD5	2816	2128	88	-
SHA-1	5348	4592	92	-
SHA-256	10240	9728	104	-
SHA-512	1840	1696	200	-
WHIRLPOOL	3840	3472	136	16464
RADIOGATÚN[32]	41072	18336	388	-
RADIOGATÚN[64]	41840	20368	776	-
Shabal	21456	7360	248	-

Table 12.3: Code and data cache consumption of various hash functions, on x86 64-bit architecture.

The internal mutable state size of Shabal is not very big for a function which offers a 512-bit output; it can still be a limitation on some constrained environments, especially smart cards, where RAM is a very scarce resource (contrary to ROM, which may be used for static tables but not for the function running state). Yet, on that specific subject, Shabal fares much better than RADIOGATÚN, and not much worse than SHA-512.

	Code and data size			
Function	code	update	state	data
MD5	2400	2096	88	-
Sha-1	6960	6640	92	-
Sha-256	14352	13968	104	-
Sha-512	6000	5440	200	-
Whirlpool	7536	6928	136	16464
RadioGatún[32]	49120	23088	388	-
RadioGatún[64]	106944	54880	776	-
Shabal	24048	8080	248	-

Table 12.4: Code and data cache consumption of various hash functions, on x86 32-bit architecture.

	Code and data size			
Function	code	update	state	data
MD5	4192	3616	88	-
Sha-1	8132	7500	92	-
Sha-256	8896	8232	104	-
Sha-512	7060	6268	200	-
Whirlpool	15004	14292	136	16464
RadioGatún[32]	41476	21692	388	-
RadioGatún[64]	109052	55004	776	-
Shabal	21036	7852	248	-

Table 12.5: Code and data cache consumption of various hash functions, on MiPS architecture.

Acknowledgments

Acknowledgments

Shabal is the result of a wonderful collaboration between a number of researchers in the area of cryptography. The story of this collaboration begins in 2005 when the French National Research Agency[2] accepts to fund the research project on hash functions Saphir.

Saphir was initiated by France Telecom and four partners: Cryptolog International, DCSSI, Gemalto and LIENS. The high-level objective of the project was specifically to study hash functions and, when NIST announced the opening of the SHA-3 competition, we decided to conceive a new hash function and submit it to both NIST and public scrutiny: Shabal. We have had the pleasure to welcome in our research meetings several partners of the upcoming project Saphir2 who substantially helped us shape Shabal.

I would like to thank all of those who contributed to launch the Saphir project: Pierre-Alain Fouque, Henri Gilbert, Marc Girault, Helena Handschuh, Gwenaëlle Martinet, Guillaume Poupard, Alexandre Stern, Julien Stern. I would also like to thank the French National Research Agency, ANR, for having financially and morally supported our project. I have to thank Marc Mouffron who permitted his team to collaborate and contribute to Shabal.

A big thank you to the Shabal team for their work and spirit: Emmanuel Bresson, Anne Canteaut, Benoît Chevallier-Mames, Christophe Clavier, Thomas Fuhr, Aline Gouget, Thomas Icart, María Naya-Plasencia, Pascal Paillier, Thomas Pornin, Jean-René Reinhard, Céline Thuillet, Marion Videau. And special thanks go to Benoît Chevallier-Mames who coordinated the edition of this document and the day-to-day work on Shabal.

I would like to thank all the people who contributed at various occasions to our brainstorming sessions on Shabal. This notably includes Sébastien Canard, Christophe de Cannière, Hervé Chabanne, Phong Q. Nguyen, Gilles Piret, David Pointcheval, Damien Vergnaud and Sébastien Zimmer. A few people were more permanent members at the beginning of the project, and we want to thank them warmly for their support: Pierre-Alain Fouque, Gaëtan Leurent and Thomas Peyrin.

I also thank Amine Dogui, David Vigilant and Karine Villegas, who gave us appreciated support in programming Shabal in various assembly languages dedicated to smart cards. Thanks also go to CFSSI, ENS and Gemalto who often hosted our brainstorming sessions.

Finally, I thank all the people who helped the Shabal team come to completion in their tasks and submit this hash algorithm.

Longue vie à Shabal !

Jean-François Misarsky
Head of the Saphir Project

[2] ANR: Agence Nationale de la Recherche - The French National Research Agency
http://www.agence-nationale-recherche.fr/Intl

Bibliography

Bibliography

[1] E. Andreeva, G. Neven, B. Preneel, and T. Shrimpton. Seven property preserving hashing: ROX. In Advances in Cryptology — ASIACRYPT 2007, volume 4833 of LNCS, pages 130–146. Springer, 2007.

[2] M. Bellare. New proofs for NMAC and HMAC: Security without collision resistance. In Advances in Cryptology — CRYPTO 2006, volume 4117 of LNCS, pages 602–619. Springer, 2006.

[3] M. Bellare and T. Ristenpart. Multi-property-preserving hash domain extension and the EMD transform. In Advances in Cryptology — ASIACRYPT 2006, volume 4284 of LNCS, pages 299–314. Springer, 2006.

[4] M. Bellare and P. Rogaway. Random oracles are practical: a paradigm for designing efficient protocols. In Proceedings of the first Annual Conference on Computer and Communications Security, pages 62–73. ACM, 1993.

[5] G. Bertoni, J. Daemen, G. Van Assche, and M. Peeters. RadioGatún, a belt-and-mill hash function. *Second Cryptographic Hash Workshop*, Santa Barbara, USA, August 2006.

[6] G. Bertoni, J. Daemen, M. Peeters, and G. Van Assche. Sponge functions. *Ecrypt Hash Workshop*, Barcelona, Spain, May 2007. http://sponge.noekeon.org/,.

[7] G. Bertoni, J. Daemen, M. Peeters, and G. Van Assche. On the indifferentiability of the sponge construction. In Advances in Cryptology — EUROCRYPT 2008, volume 4965 of LNCS, pages 181–197. Springer, 2008.

[8] E. Biham. New techniques for cryptanalysis of hash functions and improved attacks on Snefru. In Fast Software Encryption – FSE 2008, volume 5086 of LNCS, pages 444–461. Springer, 2008.

[9] E. Biham and R. Chen. Near-collisions of SHA-0. In Advances in Cryptology — CRYPTO 2004, volume 3152 of LNCS, pages 290–305. Springer, 2004.

[10] E. Biham and A. Shamir. Differential cryptanalysis of Snefru, Khafre, REDOC-II, LOKI and Lucifer. In Advances in Cryptology — CRYPTO'91, volume 576 of LNCS, pages 156–171. Springer, 1991.

[11] C. Bouillaguet and P.-A. Fouque. Analysis of radiogatún using algebraic techniques. In Selected Areas in Cryptography – SAC 2008, LNCS. Springer, 2008.

[12] R. Canetti, O. Goldreich, and S. Halevi. The random oracle methodology, revisited. Journal of the ACM, 51(4):557–594, 2004.

[13] J.-S. Coron, Y. Dodis, C. Malinaud, and P. Puniya. Merkle-Damgård revisited: how to construct a hash function. In Advances in Cryptology — CRYPTO 2005, volume 3621 of LNCS, pages 430–448. Springer, 2005.

[14] J.-S. Coron, J. Patarin, and Y. Seurin. The random oracle model and the ideal cipher model are equivalent. In Advances in Cryptology — CRYPTO 2008, volume 5157 of LNCS, pages 1–20. Springer, 2008.

[15] J. Daemen. Cipher and Hash Function Design. Strategies based on linear and differential cryptanalysis. PhD thesis, Katholieke Universiteit Leuven, 1995.

[16] J. Daemen and G. Van Assche. Producing collisions for PANAMA instantaneously. In Fast Software Encryption – FSE 2007, volume 4593 of LNCS, pages 1–18. Springer, 2007.

[17] J. Daemen and C. Clapp. Fast hashing and stream encryption with PANAMA. In Fast Software Encryption – FSE 1998, LNCS, pages 60–74. Springer, 1998.

[18] I. Damgård. A design principle for hash functions. In Advances in Cryptology — CRYPTO'89, volume 435 of LNCS, pages 416–427. Springer, 1989.

[19] B. den Boer and A. Bosselaers. Collisions for the compressin function of MD5. In Advances in Cryptology — EUROCRYPT'93, volume 765 of LNCS, pages 293–304. Springer, 1993.

[20] I. Dinur and A¿ Shamir. Cube attacks on tweakable black box polynomials. IACR ePrint Archive: Report 2008/385, 2008.

[21] H. Englund, T. Johansson, and M. S. Turan. A framework for chosen IV statistical analysis of stream ciphers. In Progress in Cryptology — INDOCRYPT 2007, volume 4859 of LNCS, pages 268–281. Springer, 2007.

[22] S. Fischer, S. Khazaei, and W. Meier. Chosen IV statistical analysis for key recovery attacks on stream ciphers. In AFRICACRYPT 2008, volume 5023 of LNCS, pages 236–245. Springer, 2008.

[23] P .-A. Fouque, G. Leurent, and P.Q. Nguyen. Full key-recovery attacks on HMAC/NMAC-MD4 and NMAC-MD5. In Advances in Cryptology — CRYPTO 2007, volume 4622 of LNCS, pages 13–30. Springer, 2007.

[24] M. Gorski, S. Lucks, and T. Peyrin. Slide attacks on a class of hash functions. In Advances in Cryptology — ASIACRYPT 2008, LNCS. Springer. To appear.

[25] A. Joux. Multicollisions in iterated hash functions. Application to cascaded constructions. In Advances in Cryptology — CRYPTO 2004, volume 3152 of LNCS, pages 306–316. Springer, 2004.

[26] J. Kelsey and B. Schneier. Second preimages on n-bit hash functions for much less than 2^n work. In Advances in Cryptology — EUROCRYPT 2005, volume 3494 of LNCS, pages 474–490, 2005.

[27] D. Khovratovitch. Cryptanalysis of hash functions with structures. http://lj.streamclub.ru/papers/papers_en.html, 2008.

[28] D. Khovratovitch and A. Biryukov. Two attacks on RADIOGATÚN. http://lj.streamclub.ru/papers/papers_en.html, 2008.

[29] L.R. Knudsen, C. Rechberger, and S.S. Thomsen. The Grindahl hash functions. In Fast Software Encryption – FSE 2007, volume 4593 of LNCS, pages 39–57. Springer, 2007.

[30] S. Lucks. A failure-friendly design principle for hash functions. In Advances in Cryptology — ASIACRYPT 2005, volume 3788 of LNCS, pages 474–494. Springer, 2005.

[31] U. Maurer, R. Renner, and C. Holenstein. Indifferentiability, impossibility results on reductions, and applications to the random oracle methodology. In Theory of cryptography – TCC 2004, volume 2951 of LNCS, pages 21–39. Springer, 2004.

[32] A. Menezes, P. C. van Oorschot, and S. A. Vanstone. Handbook of Applied Cryptography. CRC Press, 1996.

[33] R. Merkle. One way hash functions and DES. In Advances in Cryptology — CRYPTO'89, volume 435 of LNCS, pages 428–446. Springer, 1989.

[34] R. C. Merkle. A fast software one-way hash function. Journal of Cryptology, 3(1):43–58, 1990.

[35] T. Peyrin. Cryptanalysis of Grindahl. In Advances in Cryptology — ASIACRYPT 2007, volume 4833 of LNCS, pages 551–567. Springer, 2007.

[36] V. Rijmen, B. Van Rompay, B. Preneel, and J. Vandewalle. Producing collisions for PANAMA. In Fast Software Encryption — FSE 2001, volume 2355 of LNCS, pages 37–51. Springer, 2002.

[37] P. Rogaway. Formalizing human ignorance. In Progress in Cryptology — Vietcrypt 2006, volume 4341 of LNCS, pages 211–228. Springer, 2006.

[38] P. Rogaway and T. Shrimpton. Cryptographic hash-function basics: Definitions, implications, and separations for preimage resistance, second-preimage resistance, and collision resistance. In Fast Software Encryption – FSE 2004, volume 3017 of LNCS, pages 371–388. Springer, 2004.

[39] M.-J. O. Saarinen. Chosen-IV statistical attacks on eStream ciphers. In SECRYPT 2006 - International Conference on Security and Cryptography, pages 260–266. INSTICC Press, 2006.

[40] L. Wang, K. Ohta, and N. Kunihiro. New key-recovery attacks on HMAC/NMAC-MD4 and NMAC-MD5. In Advances in Cryptology — EUROCRYPT 2008, volume 4965 of LNCS, pages 237–253. Springer, 2008.

Appendixes

Appendix A

Basic Implementations

Contents

A.1 A Basic Implementation in C

A.1.1 shabal.h

```
/*
 * Implementation of the Shabal hash function (header file). This header
 * is used for both the "reference" and the "optimized" implementations.
 *
 * (c) 2008 SAPHIR project. This software is provided 'as-is', without
 * any express or implied warranty. In no event will the authors be held
 * liable for any damages arising from the use of this software.
 *
 * Permission is granted to anyone to use this software for any purpose,
 * including commercial applications, and to alter it and redistribute it
 * freely, subject to no restriction.
 *
 * Technical remarks and questions can be addressed to:
 * <thomas.pornin@cryptolog.com>
 */

#ifndef SHABAL_H__
#define SHABAL_H__

#include <limits.h>

#if defined __STDC__ && __STDC_VERSION__ >= 199901L
#include <stdint.h>
#ifdef UINT32_MAX
typedef uint32_t shabal_word32;
#else
typedef uint_fast32_t shabal_word32;
#endif
#else
#if ((UINT_MAX >> 11) >> 11) >= 0x3FF
typedef unsigned int shabal_word32;
#else
typedef unsigned long shabal_word32;
#endif
#endif
```

```
#define SHABAL_BLOCK_SIZE   16
#ifndef SHABAL_PARAM_R
#define SHABAL_PARAM_R      12
#endif

/*
 * Input messages and hash outputs are sequences of bits, stored
 * in arrays of C bytes.
 */
typedef unsigned char BitSequence;

#ifdef ULLONG_MAX
typedef unsigned long long DataLength;
#else
typedef unsigned long DataLength;
#endif

/*
 * Status codes.
 */
typedef enum {
        SUCCESS = 0,
        FAIL = 1,
        BAD_HASHBITLEN = 2
} HashReturn;

/*
 * State structure for Shabal. A structure instance holds the running
 * state for a hash computation. The function implementation is
 * thread-safe and reentrant, as long as the different has computations
 * use distinct instances of this structure. State structures are
 * meant to be allocated by the caller; allocation can be performed
 * in any suitable memory area, including local variables, global
 * variables, and the heap. The structure contains no pointer value
 * and thus can be moved, duplicated (with memcpy()) or serialized
 * at will. In particular, making a copy of a structure instance takes
 * a "snapshot" of the hash computation at that point, which can be
 * resumed later on with alternate subsequence input data.
 *
 * The "hashbitlen" field contains the intended output length, in bits.
 * It is initialized in the Init() function. The other fields are private
 * and should not be accessed externally.
 */
typedef struct {
        BitSequence buffer[SHABAL_BLOCK_SIZE * 4];
        size_t buffer_ptr;
        unsigned last_byte_significant_bits;
        int hashbitlen;
        shabal_word32 A[SHABAL_PARAM_R];
        shabal_word32 B[SHABAL_BLOCK_SIZE];
        shabal_word32 C[SHABAL_BLOCK_SIZE];
        shabal_word32 Whigh, Wlow;
} hashState;

/*
 * Initialize a state structure for a new hash computation. The intended
 * output length (in bits) is provided as the "hashbitlen" parameter;
 * supported output bit lengths are 192, 224, 256, 384 and 512. The
 * "state" pointer should reference a state structure allocated with
 * the proper alignment.
 *
 * Returned value is SUCCESS (0) on success, or a non-zero error code
 * otherwise (namely BAD_HASHBITLEN if the specified "hashbitlen" is
 * not one of the supported output lengths). On failure, the structure
 * contents are indeterminate.
 */
```

```
HashReturn Init(hashState *state, int hashbitlen);

/*
 * Continue a running hash computation. The state structure is provided.
 * The additional input data is a sequence of bits; the "data"
 * parameter points to the start of that sequence, and the "databitlen"
 * contains the sequence length, expressed in bits.
 *
 * The bits within a byte are ordered from most significant to least
 * significant. The input bit sequence MUST begin with the most
 * significant bit of the first byte pointed to by "data". The bit
 * sequence length MUST be a multiple of 8 if this call is not the
 * last Update() call performed for this hash computation. In other
 * words, the input message chunks MUST consist of entire and aligned
 * bytes, except for the very last input byte, which may be "partial".
 *
 * Returned value is SUCCESS (0) on success, or a non-zero error code
 * otherwise.
 */
HashReturn Update(hashState *state,
        const BitSequence *data, DataLength databitlen);

/*
 * Terminate a running hash computation. The state structure is provided.
 * The hash output is written out in the buffer pointed to by "hashval";
 * the hash output length was specified when the structure was initialized.
 * The same bit ordering conventions than for input data are used in the
 * hash data output; since all supported lengths are multiple of 8, the
 * hash output necessarily consists of an integral number of bytes.
 *
 * After this call, the state structure contents are indeterminate. If
 * the structure is to be used for a new hash computation, then it
 * shall be initialized again with Init().
 *
 * Returned value is SUCCESS (0) on success, or a non-zero error code
 * otherwise.
 */
HashReturn Final(hashState *state, BitSequence *hashval);

/*
 * Perform a complete hash computation. This combines calls to Init(),
 * Update() and Final(), with a state structure which this function
 * allocates and releases itself. This function can thus be called
 * independantly of any other runnimg hash computation. The parameters
 * are:
 * - hashbitlen    the output length, in bits (192, 224, 256, 384 or 512)
 * - data          pointer to the input message start
 * - databitlen    input message length, in bits
 * - hashval       pointer to the buffer which receives the hash output
 *
 * Returned value is SUCCESS (0) on success, or a non-zero error code
 * otherwise.
 */
HashReturn Hash(int hashbitlen, const BitSequence *data,
        DataLength databitlen, BitSequence *hashval);

#endif
```

A.1.2 shabal.c

```
/*
 * Implementation of the Shabal hash function (reference code). This
 * code is meant for illustration of the function internals rather than
 * optimization for speed.
 *
```

```
 * (c) 2008 SAPHIR project. This software is provided 'as-is', without
 * any express or implied warranty. In no event will the authors be held
 * liable for any damages arising from the use of this software.
 *
 * Permission is granted to anyone to use this software for any purpose,
 * including commercial applications, and to alter it and redistribute it
 * freely, subject to no restriction.
 *
 * Technical remarks and questions can be addressed to:
 * <thomas.pornin@cryptolog.com>
 */

#include <stddef.h>
#include <string.h>
#include "shabal.h"

/*
 * The values of "p" and "e" are defined in the Shabal specification.
 * Manual overrides should be performed only for research purposes,
 * since this would alter the hash output for a given input.
 */
#ifndef SHABAL_PARAM_P
#define SHABAL_PARAM_P   3
#endif
#ifndef SHABAL_PARAM_E
#define SHABAL_PARAM_E   3
#endif

#define sM    SHABAL_BLOCK_SIZE
#define nR    SHABAL_PARAM_R
#define u32   shabal_word32

/*
 * This macro truncates its argument to 32 bits (reduction modulo 2^32).
 */
#define T32(x)   ((x) & (u32)0xFFFFFFFFUL)

#define O1   13
#define O2    9
#define O3    6

/*
 * Swap the B and C state words.
 */
static void
swap_bc(hashState *state)
{
        int i;

        for (i = 0; i < sM; i ++) {
                u32 t;

                t = state->B[i];
                state->B[i] = state->C[i];
                state->C[i] = t;
        }
}

/*
 * Decode the currently accumulated data block (64 bytes) into the m[]
 * array of 16 words.
 */
static void
decode_block(hashState *state, u32 *m)
{
        int i, j;
```

```
        for (i = 0, j = 0; i < sM; i ++, j += 4) {
                m[i] = (u32)state->buffer[j]
                        + ((u32)state->buffer[j + 1] << 8)
                        + ((u32)state->buffer[j + 2] << 16)
                        + ((u32)state->buffer[j + 3] << 24);
        }
}

/*
 * Add the message block m[] to the B state words.
 */
static void
input_block_add(hashState *state, u32 *m)
{
        int i;

        for (i = 0; i < sM; i ++)
                state->B[i] = T32(state->B[i] + m[i]);
}

/*
 * Subtract the message block m[] from the C state words.
 */
static void
input_block_sub(hashState *state, u32 *m)
{
        int i;

        for (i = 0; i < sM; i ++)
                state->C[i] = T32(state->C[i] - m[i]);
}

/*
 * Combine the block counter with the A state words.
 */
static void
xor_counter(hashState *state)
{
        state->A[0] ^= state->Wlow;
        state->A[1] ^= state->Whigh;
}

/*
 * Increment the block counter, for the next block.
 */
static void
incr_counter(hashState *state)
{
        /*
         * The counter is over two 32-bit words. We manually propagate
         * the carry when needed.
         */
        if ((state->Wlow = T32(state->Wlow + 1)) == 0)
                state->Whigh ++;
}

/*
 * Apply the core Shabal permutation. The message block is provided
 * in the m[] parameter.
 */
static void
apply_perm(hashState *state, u32 *m)
{
        /*
         * We use some local macros to access the A words and compute
         * the U and V functions with an uncluttered syntax. Note that
         * this method performs explicit modular reductions of the
```

```
         * access indices, with either 16 or 12 as divisor. Using 12
         * as divisor, in particular, hurts performance badly (but
         * divisions by 16 are not free either, since we use the "int"
         * type which is signed, and thus prevents, in all generality,
         * the use of a simple bitwise mask by the compiler).
         */
#define xA     (state->A[(i + sM * j) % nR])
#define xAm1   (state->A[(i + sM * j + (nR - 1)) % nR])
#define U(x)   T32((u32)3 * (u32)(x))
#define V(x)   T32((u32)5 * (u32)(x))

        int i, j;

        for (i = 0; i < sM; i ++) {
                u32 t;

                t = state->B[i];
                state->B[i] = T32(t << 17) | (t >> 15);
        }

        for (j = 0; j < SHABAL_PARAM_P; j ++) {
                for (i = 0; i < sM; i ++) {
                        u32 tB;

                        xA = U(xA ^ V(T32(xAm1 << 15) | (xAm1 >> 17))
                                        ^ state->C[(8 + sM - i) % sM])
                                ^ state->B[(i + O1) % sM]
                                ^ (state->B[(i + O2) % sM]
                                        & ~state->B[(i + O3) % sM])
                                ^ m[i];
                        tB = state->B[i];
                        state->B[i] = T32(((tB << 1) | (tB >> 31)) ^ ~xA);
                }
        }

        for (j = 0; j < (3 * nR); j ++)
                state->A[(3 * nR - 1 - j) % nR] = T32(
                        state->A[(3 * nR - 1 - j) % nR]
                        + state->C[(3 * nR * sM + 6 - j) % sM]);

#undef xA
#undef xAm1
#undef U
#undef V
}

/* see shabal.h */
HashReturn
Init(hashState *state, int hashbitlen)
{
        int i, j;
        u32 m[sM];

        /*
         * First, check that the output length is one of the supported
         * lengths, and set the "hashbitlen" field (mandated by NIST API).
         * The 224-, 256-, 384- and 512-bit output lengths are NIST
         * requirements for the SHA-3 competition; the 192-bit output is
         * a "bonus".
         */
        switch (hashbitlen) {
        case 192: case 224: case 256: case 384: case 512:
                break;
        default:
                return BAD_HASHBITLEN;
        }
        state->hashbitlen = hashbitlen;
```

```
        /*
         * Initialize the state (all zero, except the counter which is
         * set to -1).
         */
        for (i = 0; i < nR; i ++)
                state->A[i] = 0;
        for (i = 0; i < sM; i ++) {
                state->B[i] = 0;
                state->C[i] = 0;
        }
        state->Wlow = (u32)0xFFFFFFFFUL;
        state->Whigh = (u32)0xFFFFFFFFUL;

        /*
         * We compute the first two blocks, corresponding to the prefix.
         * We process them immediately.
         */
        for (j = 0; j < (2 * sM); j += sM) {
                for (i = 0; i < sM; i ++)
                        m[i] = hashbitlen + j + i;
                input_block_add(state, m);
                xor_counter(state);
                apply_perm(state, m);
                input_block_sub(state, m);
                swap_bc(state);
                incr_counter(state);
        }

        /*
         * We set the fields for input buffer management.
         */
        state->buffer_ptr = 0;
        state->last_byte_significant_bits = 0;
        return SUCCESS;
}

/* see shabal.h */
HashReturn
Update(hashState *state, const BitSequence *data, DataLength databitlen)
{
        unsigned char *buffer;
        size_t len, ptr;

        /*
         * "last_byte_significant_bits" contains the number of bits
         * which are part of the input data in the last buffered
         * byte. In the state structure, buffer bytes from 0 to
         * buffer_ptr-1 are "full", and the next byte (buffer[buffer_ptr])
         * contains the extra non-integral byte, if any. All calls to
         * Update() shall provide a data bit length multiple of 8,
         * except possibly the last call. Hence, we know that upon
         * entry of this function, last_byte_significant_bits is
         * necessarily zero (if this code was used according to its
         * specification).
         *
         * We process input data by blocks. When we are finished,
         * we have strictly less than a full block in the buffer
         * (buffer_ptr is at most 63, and buffer[buffer_ptr] contains
         * between 0 and 7 significant bits).
         */
        state->last_byte_significant_bits = (unsigned)databitlen & 0x07;
        len = (size_t)(databitlen >> 3);
        buffer = state->buffer;
        ptr = state->buffer_ptr;
        while (len > 0) {
                size_t clen;
```

```
                clen = (sizeof state->buffer) - ptr;
                if (clen > len)
                        clen = len;
                memcpy(buffer + ptr, data, clen);
                ptr += clen;
                data += clen;
                len -= clen;
                if (ptr == sizeof state->buffer) {
                        u32 m[sM];

                        decode_block(state, m);
                        input_block_add(state, m);
                        xor_counter(state);
                        apply_perm(state, m);
                        input_block_sub(state, m);
                        swap_bc(state);
                        incr_counter(state);
                        ptr = 0;
                }
        }
        if (state->last_byte_significant_bits != 0)
                buffer[ptr] = *data;
        state->buffer_ptr = ptr;
        return SUCCESS;
}

/* see shabal.h */
HashReturn
Final(hashState *state, BitSequence *hashval)
{
        unsigned char *buffer;
        size_t ptr;
        unsigned lbsb;
        u32 m[sM];
        int i;

        /*
         * Complete the last block with the padding. Since the buffer data
         * is always short of a full block by at least one bit, we always
         * have room for the padding in that block.
         */
        buffer = state->buffer;
        ptr = state->buffer_ptr;
        lbsb = state->last_byte_significant_bits;
        buffer[ptr] = (buffer[ptr] & ~(0x7F >> lbsb)) | (0x80U >> lbsb);
        memset(buffer + ptr + 1, 0, (sizeof state->buffer) - (ptr + 1));

        /*
         * Now, process the last (padded) block. We add three extra
         * permutations, and we optimize away the unnecessary message
         * additions and subtractions. There is no increment of the
         * counter either in those last rounds (we keep the counter
         * value used for the last data block). We transfer the swap to
         * the next round, because there needs be no final swap.
         *
         * Note that we reuse the decoded final block in several calls
         * to apply_perm().
         */
        decode_block(state, m);
        input_block_add(state, m);
        xor_counter(state);
        apply_perm(state, m);
        for (i = 0; i < SHABAL_PARAM_E; i ++) {
                swap_bc(state);
                xor_counter(state);
                apply_perm(state, m);
```

```
        }

        /*
         * The ouput consists of the B words, truncated to the requested
         * length (in the formal description, the C words are used, but
         * we omitted the last swap, so we have the words in B).
         */
        for (i = sM - (state->hashbitlen >> 5); i < sM; i ++) {
                u32 tmp;

                tmp = state->B[i];
                *hashval ++ = tmp & 0xFF;
                *hashval ++ = (tmp >> 8) & 0xFF;
                *hashval ++ = (tmp >> 16) & 0xFF;
                *hashval ++ = (tmp >> 24) & 0xFF;
        }
        return SUCCESS;
}

/* see shabal.h */
HashReturn
Hash(int hashbitlen, const BitSequence *data,
        DataLength databitlen, BitSequence *hashval)
{
        hashState st;
        HashReturn r;

        /*
         * Here, we do not test the output of Update() because we
         * know that in this implementation, Update() always returns
         * SUCCESS.
         */
        r = Init(&st, hashbitlen);
        if (r != SUCCESS)
                return r;
        Update(&st, data, databitlen);
        return Final(&st, hashval);
}
```

Appendix B

Detailed Test Patterns

Contents

Please consider the environment before printing this chapter.

B.1 Intermediate States for Shabal-192 (Message A)

```
init

A : 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
    00000000 00000000 00000000 00000000

B : 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
    00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000

C : 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
    00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000

W : FFFFFFFF FFFFFFFF


block number = -1 : message block

M : 000000C0 000000C1 000000C2 000000C3 000000C4 000000C5 000000C6 000000C7
    000000C8 000000C9 000000CA 000000CB 000000CC 000000CD 000000CE 000000CF


block number = -1 : add M to B

A : 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
    00000000 00000000 00000000 00000000
```

```
B : 000000C0 000000C1 000000C2 000000C3 000000C4 000000C5 000000C6 000000C7
    000000C8 000000C9 000000CA 000000CB 000000CC 000000CD 000000CE 000000CF

C : 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
    00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000

W : FFFFFFFF FFFFFFFF


block number = -1 : xor counter W into A

A : FFFFFFFF FFFFFFFF 00000000 00000000 00000000 00000000 00000000 00000000
    00000000 00000000 00000000 00000000

B : 000000C0 000000C1 000000C2 000000C3 000000C4 000000C5 000000C6 000000C7
    000000C8 000000C9 000000CA 000000CB 000000CC 000000CD 000000CE 000000CF

C : 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
    00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000

W : FFFFFFFF FFFFFFFF


block number = -1 : permutation - rotate B

A : FFFFFFFF FFFFFFFF 00000000 00000000 00000000 00000000 00000000 00000000
    00000000 00000000 00000000 00000000

B : 01800000 01820000 01840000 01860000 01880000 018A0000 018C0000 018E0000
    01900000 01920000 01940000 01960000 01980000 019A0000 019C0000 019E0000

C : 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
    00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000

W : FFFFFFFF FFFFFFFF


block number = -1 : permutation (j = 0)

A : F78A4200 10145AFD AE842D9A 7DA2B6D1 E125063B 87057989 D33D3495 15A94983
    24F95064 DD75BF25 2CE50A5D 109E76BE

B : 028800C2 F8D8F4B6 29CC61C2 DCBC40CD 1DCAF9C4 7BEE8676 2FDACB6A E94AB67C
    D826AF9B 21AE40DA D032F5A2 EC4D8941 0B45BDFF ECDFA502 5243D265 8161492E

C : 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
    00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000

W : FFFFFFFF FFFFFFFF


block number = -1 : permutation (j = 1)

A : B11AE006 5117AAEA F864E13E 8395DEC5 7473DED3 F4F3C55C EE1065D0 0FBCE63D
    5D489898 3D28BAC9 DF45C1AD C5245735

B : 6E986FA3 0EF5A5BE CF36EA1B A553B940 992294EF 350A49DA 7F0FA886 E84EC433
    FEA840CE EDB4D4A1 A7FEF584 A4F133B9 9D075AD2 D2B370A6 B5683EE5 F2818B9F

C : 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
    00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000

W : FFFFFFFF FFFFFFFF


block number = -1 : permutation (j = 2)
```

```
A : 6A2E9EF8 2A8ED2CE 5B7DEB2B 76F5EC56 C761D844 6FFDD495 A6E50D15 CB7EBB78
    56F43E26 A9D57A1C 5FD697A6 E72A8ACB

B : 58BCBB94 EC47A15F AEE93484 DFCBC8F7 A79448D8 BF65BE85 5A9D45D8 59979BCE
    C5CEA626 4B6B8229 16E719E3 7D6323F4 9305747C F34C64AE CAF91592 FDD6620B

C : 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
    00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000

W : FFFFFFFF FFFFFFFF


block number = -1 : permutation - add C to A

A : 6A2E9EF8 2A8ED2CE 5B7DEB2B 76F5EC56 C761D844 6FFDD495 A6E50D15 CB7EBB78
    56F43E26 A9D57A1C 5FD697A6 E72A8ACB

B : 58BCBB94 EC47A15F AEE93484 DFCBC8F7 A79448D8 BF65BE85 5A9D45D8 59979BCE
    C5CEA626 4B6B8229 16E719E3 7D6323F4 9305747C F34C64AE CAF91592 FDD6620B

C : 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
    00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000

W : FFFFFFFF FFFFFFFF


block number = -1 : subtract M from C

A : 6A2E9EF8 2A8ED2CE 5B7DEB2B 76F5EC56 C761D844 6FFDD495 A6E50D15 CB7EBB78
    56F43E26 A9D57A1C 5FD697A6 E72A8ACB

B : 58BCBB94 EC47A15F AEE93484 DFCBC8F7 A79448D8 BF65BE85 5A9D45D8 59979BCE
    C5CEA626 4B6B8229 16E719E3 7D6323F4 9305747C F34C64AE CAF91592 FDD6620B

C : FFFFFF40 FFFFFF3F FFFFFF3E FFFFFF3D FFFFFF3C FFFFFF3B FFFFFF3A FFFFFF39
    FFFFFF38 FFFFFF37 FFFFFF36 FFFFFF35 FFFFFF34 FFFFFF33 FFFFFF32 FFFFFF31

W : FFFFFFFF FFFFFFFF


block number = -1 : swap B with C

A : 6A2E9EF8 2A8ED2CE 5B7DEB2B 76F5EC56 C761D844 6FFDD495 A6E50D15 CB7EBB78
    56F43E26 A9D57A1C 5FD697A6 E72A8ACB

B : FFFFFF40 FFFFFF3F FFFFFF3E FFFFFF3D FFFFFF3C FFFFFF3B FFFFFF3A FFFFFF39
    FFFFFF38 FFFFFF37 FFFFFF36 FFFFFF35 FFFFFF34 FFFFFF33 FFFFFF32 FFFFFF31

C : 58BCBB94 EC47A15F AEE93484 DFCBC8F7 A79448D8 BF65BE85 5A9D45D8 59979BCE
    C5CEA626 4B6B8229 16E719E3 7D6323F4 9305747C F34C64AE CAF91592 FDD6620B

W : FFFFFFFF FFFFFFFF


block number = 0 : increment counter W

A : 6A2E9EF8 2A8ED2CE 5B7DEB2B 76F5EC56 C761D844 6FFDD495 A6E50D15 CB7EBB78
    56F43E26 A9D57A1C 5FD697A6 E72A8ACB

B : FFFFFF40 FFFFFF3F FFFFFF3E FFFFFF3D FFFFFF3C FFFFFF3B FFFFFF3A FFFFFF39
    FFFFFF38 FFFFFF37 FFFFFF36 FFFFFF35 FFFFFF34 FFFFFF33 FFFFFF32 FFFFFF31

C : 58BCBB94 EC47A15F AEE93484 DFCBC8F7 A79448D8 BF65BE85 5A9D45D8 59979BCE
    C5CEA626 4B6B8229 16E719E3 7D6323F4 9305747C F34C64AE CAF91592 FDD6620B

W : 00000000 00000000
```

```
block number = 0 : message block

M : 000000D0 000000D1 000000D2 000000D3 000000D4 000000D5 000000D6 000000D7
    000000D8 000000D9 000000DA 000000DB 000000DC 000000DD 000000DE 000000DF


block number = 0 : add M to B

A : 6A2E9EF8 2A8ED2CE 5B7DEB2B 76F5EC56 C761D844 6FFDD495 A6E50D15 CB7EBB78
    56F43E26 A9D57A1C 5FD697A6 E72A8ACB

B : 00000010 00000010 00000010 00000010 00000010 00000010 00000010 00000010
    00000010 00000010 00000010 00000010 00000010 00000010 00000010 00000010

C : 58BCBB94 EC47A15F AEE93484 DFCBC8F7 A79448D8 BF65BE85 5A9D45D8 59979BCE
    C5CEA626 4B6B8229 16E719E3 7D6323F4 9305747C F34C64AE CAF91592 FDD6620B

W : 00000000 00000000


block number = 0 : xor counter W into A

A : 6A2E9EF8 2A8ED2CE 5B7DEB2B 76F5EC56 C761D844 6FFDD495 A6E50D15 CB7EBB78
    56F43E26 A9D57A1C 5FD697A6 E72A8ACB

B : 00000010 00000010 00000010 00000010 00000010 00000010 00000010 00000010
    00000010 00000010 00000010 00000010 00000010 00000010 00000010 00000010

C : 58BCBB94 EC47A15F AEE93484 DFCBC8F7 A79448D8 BF65BE85 5A9D45D8 59979BCE
    C5CEA626 4B6B8229 16E719E3 7D6323F4 9305747C F34C64AE CAF91592 FDD6620B

W : 00000000 00000000


block number = 0 : permutation - rotate B

A : 6A2E9EF8 2A8ED2CE 5B7DEB2B 76F5EC56 C761D844 6FFDD495 A6E50D15 CB7EBB78
    56F43E26 A9D57A1C 5FD697A6 E72A8ACB

B : 00200000 00200000 00200000 00200000 00200000 00200000 00200000 00200000
    00200000 00200000 00200000 00200000 00200000 00200000 00200000 00200000

C : 58BCBB94 EC47A15F AEE93484 DFCBC8F7 A79448D8 BF65BE85 5A9D45D8 59979BCE
    C5CEA626 4B6B8229 16E719E3 7D6323F4 9305747C F34C64AE CAF91592 FDD6620B

W : 00000000 00000000


block number = 0 : permutation (j = 0)

A : 40EDD726 1A8754BC 7AF69209 DD22532D 04BFEA68 4E68A17B 6E3AE516 89451CE8
    237E61BE 50533EB1 77550A53 4EE347AA

B : 20C6148A C0A0482A 20FEE685 BD139D7F FB001597 B1D75E84 91851AE9 76FAE317
    DCC19E41 AFECC14E 88EAF5AC B15CB855 BF5228D9 E538AB43 85496DF6 229DACD2

C : 58BCBB94 EC47A15F AEE93484 DFCBC8F7 A79448D8 BF65BE85 5A9D45D8 59979BCE
    C5CEA626 4B6B8229 16E719E3 7D6323F4 9305747C F34C64AE CAF91592 FDD6620B

W : 00000000 00000000


block number = 0 : permutation (j = 1)

A : DCB0D1FF 9807FEF5 55FCAED1 DDFCB641 2546AE31 EFC27512 31D29DFA 9459E7A6
    9E7C5606 FC6CB0E4 5699952C 8FDD2936
```

```
B : 5EA811AF 428F0F99 F1F49AC3 7C94CC73 978382D6 603DF212 8A6C5F00 9DD710E7
    9ACC1283 38218397 BBD6BA77 40BA3915 A41D007D DA4CDC6A C4BFB9E8 2E9D41FD

C : 58BCBB94 EC47A15F AEE93484 DFCBC8F7 A79448D8 BF65BE85 5A9D45D8 59979BCE
    C5CEA626 4B6B8229 16E719E3 7D6323F4 9305747C F34C64AE CAF91592 FDD6620B

W : 00000000 00000000


block number = 0 : permutation (j = 2)

A : A26F4FDE 24426C48 949686DD 723E1870 D02F6DE4 7F2B95C6 80DE631A DADF6026
    ABCC94F0 E7106561 2495F432 6F985A6C

B : 61550878 89EF2B75 A1660C46 7EF3855B 7297B58C 1BC67793 7FB1C723 B66FC640
    1A48B71C F0976D17 088CE80A A454EDF3 1C096BF4 AC76224B 5215781C CD5D2669

C : 58BCBB94 EC47A15F AEE93484 DFCBC8F7 A79448D8 BF65BE85 5A9D45D8 59979BCE
    C5CEA626 4B6B8229 16E719E3 7D6323F4 9305747C F34C64AE CAF91592 FDD6620B

W : 00000000 00000000


block number = 0 : permutation - add C to A

A : FD749ED4 B798E530 33904B6F 46BDA85E 076934B4 454B4058 77F74527 FB4CF465
    62931DA9 E778C8DB 22B3998E AC15CFB9

B : 61550878 89EF2B75 A1660C46 7EF3855B 7297B58C 1BC67793 7FB1C723 B66FC640
    1A48B71C F0976D17 088CE80A A454EDF3 1C096BF4 AC76224B 5215781C CD5D2669

C : 58BCBB94 EC47A15F AEE93484 DFCBC8F7 A79448D8 BF65BE85 5A9D45D8 59979BCE
    C5CEA626 4B6B8229 16E719E3 7D6323F4 9305747C F34C64AE CAF91592 FDD6620B

W : 00000000 00000000


block number = 0 : subtract M from C

A : FD749ED4 B798E530 33904B6F 46BDA85E 076934B4 454B4058 77F74527 FB4CF465
    62931DA9 E778C8DB 22B3998E AC15CFB9

B : 61550878 89EF2B75 A1660C46 7EF3855B 7297B58C 1BC67793 7FB1C723 B66FC640
    1A48B71C F0976D17 088CE80A A454EDF3 1C096BF4 AC76224B 5215781C CD5D2669

C : 58BCBAC4 EC47A08E AEE933B2 DFCBC824 A7944804 BF65BDB0 5A9D4502 59979AF7
    C5CEA54E 4B6B8150 16E71909 7D632319 930573A0 F34C63D1 CAF914B4 FDD6612C

W : 00000000 00000000


block number = 0 : swap B with C

A : FD749ED4 B798E530 33904B6F 46BDA85E 076934B4 454B4058 77F74527 FB4CF465
    62931DA9 E778C8DB 22B3998E AC15CFB9

B : 58BCBAC4 EC47A08E AEE933B2 DFCBC824 A7944804 BF65BDB0 5A9D4502 59979AF7
    C5CEA54E 4B6B8150 16E71909 7D632319 930573A0 F34C63D1 CAF914B4 FDD6612C

C : 61550878 89EF2B75 A1660C46 7EF3855B 7297B58C 1BC67793 7FB1C723 B66FC640
    1A48B71C F0976D17 088CE80A A454EDF3 1C096BF4 AC76224B 5215781C CD5D2669

W : 00000000 00000000


block number = 1 : increment counter W
```

```
A : FD749ED4 B798E530 33904B6F 46BDA85E 076934B4 454B4058 77F74527 FB4CF465
    62931DA9 E778C8DB 22B3998E AC15CFB9

B : 58BCBAC4 EC47A08E AEE933B2 DFCBC824 A7944804 BF65BDB0 5A9D4502 59979AF7
    C5CEA54E 4B6B8150 16E71909 7D632319 930573A0 F34C63D1 CAF914B4 FDD6612C

C : 61550878 89EF2B75 A1660C46 7EF3855B 7297B58C 1BC67793 7FB1C723 B66FC640
    1A48B71C F0976D17 088CE80A A454EDF3 1C096BF4 AC76224B 5215781C CD5D2669

W : 00000001 00000000


block number = 1 : message block

M : 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
    00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000


block number = 1 : add M to B

A : FD749ED4 B798E530 33904B6F 46BDA85E 076934B4 454B4058 77F74527 FB4CF465
    62931DA9 E778C8DB 22B3998E AC15CFB9

B : 58BCBAC4 EC47A08E AEE933B2 DFCBC824 A7944804 BF65BDB0 5A9D4502 59979AF7
    C5CEA54E 4B6B8150 16E71909 7D632319 930573A0 F34C63D1 CAF914B4 FDD6612C

C : 61550878 89EF2B75 A1660C46 7EF3855B 7297B58C 1BC67793 7FB1C723 B66FC640
    1A48B71C F0976D17 088CE80A A454EDF3 1C096BF4 AC76224B 5215781C CD5D2669

W : 00000001 00000000


block number = 1 : xor counter W into A

A : FD749ED5 B798E530 33904B6F 46BDA85E 076934B4 454B4058 77F74527 FB4CF465
    62931DA9 E778C8DB 22B3998E AC15CFB9

B : 58BCBAC4 EC47A08E AEE933B2 DFCBC824 A7944804 BF65BDB0 5A9D4502 59979AF7
    C5CEA54E 4B6B8150 16E71909 7D632319 930573A0 F34C63D1 CAF914B4 FDD6612C

C : 61550878 89EF2B75 A1660C46 7EF3855B 7297B58C 1BC67793 7FB1C723 B66FC640
    1A48B71C F0976D17 088CE80A A454EDF3 1C096BF4 AC76224B 5215781C CD5D2669

W : 00000001 00000000


block number = 1 : permutation - rotate B

A : FD749ED5 B798E530 33904B6F 46BDA85E 076934B4 454B4058 77F74527 FB4CF465
    62931DA9 E778C8DB 22B3998E AC15CFB9

B : 7588B179 411DD88F 67655DD2 9049BF97 90094F28 7B617ECB 8A04B53A 35EEB32F
    4A9D8B9D 02A096D7 32122DCE 4632FAC6 E741260A C7A3E698 296995F2 C259FBAC

C : 61550878 89EF2B75 A1660C46 7EF3855B 7297B58C 1BC67793 7FB1C723 B66FC640
    1A48B71C F0976D17 088CE80A A454EDF3 1C096BF4 AC76224B 5215781C CD5D2669

W : 00000001 00000000


block number = 1 : permutation (j = 0)

A : 4938D355 5288CA69 C3588F79 A6C6D366 AE98E42B 3AC0111E FBC3BCCF 3E91F2FB
    716B0E50 99F3F6A7 6EE059FF BB628852

B : F2A96EA1 F4E8814E 1608C053 0ECBF7E8 71758585 33FD1377 10352945 AAB36B5A
```

```
    1BAFE695 634D24F6 F53BFD9C C8F88221 784560BF 2230F8A7 6E745B62 DD8ADBC0

C : 61550878 89EF2B75 A1660C46 7EF3855B 7297B58C 1BC67793 7FB1C723 B66FC640
    1A48B71C F0976D17 088CE80A A454EDF3 1C096BF4 AC76224B 5215781C CD5D2669

W : 00000001 00000000


block number = 1 : permutation (j = 1)

A : 0C9CDDB2 AE1600E6 3EDE3006 940E8573 C3090C7B C3D810AC 12683C17 F034D340
    D5985BF2 7954156D 527D532E 25F0ED34

B : B7270521 E3A38111 57CE7E2F 39B038AF C88CAF07 E151CC7C 8DE8FE5B 8F69C47E
    C43CEF67 9773B6F5 2B5634C0 FA007ECF CC7C32FA 78461E1D 317F752C B4DE9B3E

C : 61550878 89EF2B75 A1660C46 7EF3855B 7297B58C 1BC67793 7FB1C723 B66FC640
    1A48B71C F0976D17 088CE80A A454EDF3 1C096BF4 AC76224B 5215781C CD5D2669

W : 00000001 00000000


block number = 1 : permutation (j = 2)

A : 89F5EF16 949BFB86 B4F5B8A9 13F6118A F9C835AD CBDD8F73 840DC321 3EFCE7E4
    CC7B611A 06F8AF7A 67210CF8 E30A4E95

B : 643CF641 FD921CA9 85FD62BB C64FDEEE E7134EE6 A9C79C80 50DBBBE1 F2DA6688
    8E4E149D 1AC51D67 2D5E555E 3503E584 AB7CFB10 098B6CBF FA20195F 75488717

C : 61550878 89EF2B75 A1660C46 7EF3855B 7297B58C 1BC67793 7FB1C723 B66FC640
    1A48B71C F0976D17 088CE80A A454EDF3 1C096BF4 AC76224B 5215781C CD5D2669

W : 00000001 00000000


block number = 1 : permutation - add C to A

A : 7A9B88CD 8492257E 07217DFC 87235D0F FC88A7B1 BA130493 1A5AD340 68A1A357
    A6339AA8 AFE28816 1FF513ED BD5E75DE

B : 643CF641 FD921CA9 85FD62BB C64FDEEE E7134EE6 A9C79C80 50DBBBE1 F2DA6688
    8E4E149D 1AC51D67 2D5E555E 3503E584 AB7CFB10 098B6CBF FA20195F 75488717

C : 61550878 89EF2B75 A1660C46 7EF3855B 7297B58C 1BC67793 7FB1C723 B66FC640
    1A48B71C F0976D17 088CE80A A454EDF3 1C096BF4 AC76224B 5215781C CD5D2669

W : 00000001 00000000


block number = 1 : subtract M from C

A : 7A9B88CD 8492257E 07217DFC 87235D0F FC88A7B1 BA130493 1A5AD340 68A1A357
    A6339AA8 AFE28816 1FF513ED BD5E75DE

B : 643CF641 FD921CA9 85FD62BB C64FDEEE E7134EE6 A9C79C80 50DBBBE1 F2DA6688
    8E4E149D 1AC51D67 2D5E555E 3503E584 AB7CFB10 098B6CBF FA20195F 75488717

C : 61550878 89EF2B75 A1660C46 7EF3855B 7297B58C 1BC67793 7FB1C723 B66FC640
    1A48B71C F0976D17 088CE80A A454EDF3 1C096BF4 AC76224B 5215781C CD5D2669

W : 00000001 00000000


block number = 1 : swap B with C

A : 7A9B88CD 8492257E 07217DFC 87235D0F FC88A7B1 BA130493 1A5AD340 68A1A357
```

```
    A6339AA8 AFE28816 1FF513ED BD5E75DE

B : 61550878 89EF2B75 A1660C46 7EF3855B 7297B58C 1BC67793 7FB1C723 B66FC640
    1A48B71C F0976D17 088CE80A A454EDF3 1C096BF4 AC76224B 5215781C CD5D2669

C : 643CF641 FD921CA9 85FD62BB C64FDEEE E7134EE6 A9C79C80 50DBBBE1 F2DA6688
    8E4E149D 1AC51D67 2D5E555E 3503E584 AB7CFB10 098B6CBF FA20195F 75488717

W : 00000001 00000000


block number = 2 : increment counter W

A : 7A9B88CD 8492257E 07217DFC 87235D0F FC88A7B1 BA130493 1A5AD340 68A1A357
    A6339AA8 AFE28816 1FF513ED BD5E75DE

B : 61550878 89EF2B75 A1660C46 7EF3855B 7297B58C 1BC67793 7FB1C723 B66FC640
    1A48B71C F0976D17 088CE80A A454EDF3 1C096BF4 AC76224B 5215781C CD5D2669

C : 643CF641 FD921CA9 85FD62BB C64FDEEE E7134EE6 A9C79C80 50DBBBE1 F2DA6688
    8E4E149D 1AC51D67 2D5E555E 3503E584 AB7CFB10 098B6CBF FA20195F 75488717

W : 00000002 00000000


block number = 2 : message block

M : 00000080 00000000 00000000 00000000 00000000 00000000 00000000 00000000
    00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000


block number = 2 : add M to B

A : 7A9B88CD 8492257E 07217DFC 87235D0F FC88A7B1 BA130493 1A5AD340 68A1A357
    A6339AA8 AFE28816 1FF513ED BD5E75DE

B : 615508F8 89EF2B75 A1660C46 7EF3855B 7297B58C 1BC67793 7FB1C723 B66FC640
    1A48B71C F0976D17 088CE80A A454EDF3 1C096BF4 AC76224B 5215781C CD5D2669

C : 643CF641 FD921CA9 85FD62BB C64FDEEE E7134EE6 A9C79C80 50DBBBE1 F2DA6688
    8E4E149D 1AC51D67 2D5E555E 3503E584 AB7CFB10 098B6CBF FA20195F 75488717

W : 00000002 00000000


block number = 2 : xor counter W into A

A : 7A9B88CF 8492257E 07217DFC 87235D0F FC88A7B1 BA130493 1A5AD340 68A1A357
    A6339AA8 AFE28816 1FF513ED BD5E75DE

B : 615508F8 89EF2B75 A1660C46 7EF3855B 7297B58C 1BC67793 7FB1C723 B66FC640
    1A48B71C F0976D17 088CE80A A454EDF3 1C096BF4 AC76224B 5215781C CD5D2669

C : 643CF641 FD921CA9 85FD62BB C64FDEEE E7134EE6 A9C79C80 50DBBBE1 F2DA6688
    8E4E149D 1AC51D67 2D5E555E 3503E584 AB7CFB10 098B6CBF FA20195F 75488717

W : 00000002 00000000


block number = 2 : permutation - rotate B

A : 7A9B88CF 8492257E 07217DFC 87235D0F FC88A7B1 BA130493 1A5AD340 68A1A357
    A6339AA8 AFE28816 1FF513ED BD5E75DE

B : 11F0C2AA 56EB13DE 188D42CC 0AB6FDE7 6B18E52F EF26378C 8E46FF63 8C816CDF
    6E383491 DA2FE12E D0141119 DBE748A9 D7E83812 449758EC F038A42A 4CD39ABA
```

```
C : 643CF641 FD921CA9 85FD62BB C64FDEEE E7134EE6 A9C79C80 50DBBBE1 F2DA6688
    8E4E149D 1AC51D67 2D5E555E 3503E584 AB7CFB10 098B6CBF FA20195F 75488717

W : 00000002 00000000


block number = 2 : permutation (j = 0)

A : C943C410 4B9134EF 31EFBB0F AC63BE5F 30AA2521 16A218E4 E80EABE4 0B834617
    F5341AC3 D0B751D5 16B0FA0E 8F3DF915

B : BFCBED60 5599C245 5710D79A 04C5B506 19641080 37118802 0B7CAADC ED7E6057
    D6BB8C1E 9B176C77 496727C2 C70C97B9 996C4BCA 3D407AC8 2E610CA5 CA3B74D4

C : 643CF641 FD921CA9 85FD62BB C64FDEEE E7134EE6 A9C79C80 50DBBBE1 F2DA6688
    8E4E149D 1AC51D67 2D5E555E 3503E584 AB7CFB10 098B6CBF FA20195F 75488717

W : 00000002 00000000


block number = 2 : permutation (j = 1)

A : 20491CA1 EB28B829 8E525199 B896DDA0 A7D79EBC 223E4E9D 210ADBB1 1C599CC5
    682D693A 1F3CFE7A E90C388B D8C6DAA5

B : 820A468E 2D6D029B 82E25781 65698F76 A51AB7C5 8EE01181 000A92CC FDC5E5F5
    72C1FB63 22F99F39 E363E1E2 C9700D2C 6AF0F6D6 A74144F2 82373D04 77D08A93

C : 643CF641 FD921CA9 85FD62BB C64FDEEE E7134EE6 A9C79C80 50DBBBE1 F2DA6688
    8E4E149D 1AC51D67 2D5E555E 3503E584 AB7CFB10 098B6CBF FA20195F 75488717

W : 00000002 00000000


block number = 2 : permutation (j = 2)

A : 33E0DA7D EA775049 5B8219CC CEF0326D 0CE5C43A 9DF42A97 DC70AC5F A729892C
    3A5C06D9 FE6F5621 51A2662D 00C48A61

B : DAF409D0 3F639964 C66B562C 87C5FAC2 862A4A09 08488CB5 A468C3AB CA840679
    1699CD03 27F8EB1A E5489065 CA366C8A 1042148A 4F12203B AA33E3DB 109A60B8

C : 643CF641 FD921CA9 85FD62BB C64FDEEE E7134EE6 A9C79C80 50DBBBE1 F2DA6688
    8E4E149D 1AC51D67 2D5E555E 3503E584 AB7CFB10 098B6CBF FA20195F 75488717

W : 00000002 00000000


block number = 2 : permutation - add C to A

A : A47D2606 E1449080 0C673FB4 9FE96A68 3B5890C7 7792845B 9E8F82EF AB60FD26
    288A31D3 1F4DB4B4 1FBA8CD3 791EB4FF

B : DAF409D0 3F639964 C66B562C 87C5FAC2 862A4A09 08488CB5 A468C3AB CA840679
    1699CD03 27F8EB1A E5489065 CA366C8A 1042148A 4F12203B AA33E3DB 109A60B8

C : 643CF641 FD921CA9 85FD62BB C64FDEEE E7134EE6 A9C79C80 50DBBBE1 F2DA6688
    8E4E149D 1AC51D67 2D5E555E 3503E584 AB7CFB10 098B6CBF FA20195F 75488717

W : 00000002 00000000


block number = 2 : subtract M from C

A : A47D2606 E1449080 0C673FB4 9FE96A68 3B5890C7 7792845B 9E8F82EF AB60FD26
    288A31D3 1F4DB4B4 1FBA8CD3 791EB4FF
```

```
B : DAF409D0 3F639964 C66B562C 87C5FAC2 862A4A09 08488CB5 A468C3AB CA840679
    1699CD03 27F8EB1A E5489065 CA366C8A 1042148A 4F12203B AA33E3DB 109A60B8

C : 643CF5C1 FD921CA9 85FD62BB C64FDEEE E7134EE6 A9C79C80 50DBBBE1 F2DA6688
    8E4E149D 1AC51D67 2D5E555E 3503E584 AB7CFB10 098B6CBF FA20195F 75488717

W : 00000002 00000000


block number = 2 : swap B with C

A : A47D2606 E1449080 0C673FB4 9FE96A68 3B5890C7 7792845B 9E8F82EF AB60FD26
    288A31D3 1F4DB4B4 1FBA8CD3 791EB4FF

B : 643CF5C1 FD921CA9 85FD62BB C64FDEEE E7134EE6 A9C79C80 50DBBBE1 F2DA6688
    8E4E149D 1AC51D67 2D5E555E 3503E584 AB7CFB10 098B6CBF FA20195F 75488717

C : DAF409D0 3F639964 C66B562C 87C5FAC2 862A4A09 08488CB5 A468C3AB CA840679
    1699CD03 27F8EB1A E5489065 CA366C8A 1042148A 4F12203B AA33E3DB 109A60B8

W : 00000002 00000000


n0_final = 0 : add M to B

A : A47D2606 E1449080 0C673FB4 9FE96A68 3B5890C7 7792845B 9E8F82EF AB60FD26
    288A31D3 1F4DB4B4 1FBA8CD3 791EB4FF

B : 643CF641 FD921CA9 85FD62BB C64FDEEE E7134EE6 A9C79C80 50DBBBE1 F2DA6688
    8E4E149D 1AC51D67 2D5E555E 3503E584 AB7CFB10 098B6CBF FA20195F 75488717

C : DAF409D0 3F639964 C66B562C 87C5FAC2 862A4A09 08488CB5 A468C3AB CA840679
    1699CD03 27F8EB1A E5489065 CA366C8A 1042148A 4F12203B AA33E3DB 109A60B8

W : 00000002 00000000


n0_final = 0 : xor counter W into A

A : A47D2604 E1449080 0C673FB4 9FE96A68 3B5890C7 7792845B 9E8F82EF AB60FD26
    288A31D3 1F4DB4B4 1FBA8CD3 791EB4FF

B : 643CF641 FD921CA9 85FD62BB C64FDEEE E7134EE6 A9C79C80 50DBBBE1 F2DA6688
    8E4E149D 1AC51D67 2D5E555E 3503E584 AB7CFB10 098B6CBF FA20195F 75488717

C : DAF409D0 3F639964 C66B562C 87C5FAC2 862A4A09 08488CB5 A468C3AB CA840679
    1699CD03 27F8EB1A E5489065 CA366C8A 1042148A 4F12203B AA33E3DB 109A60B8

W : 00000002 00000000


n0_final = 0 : permutation - rotate B

A : A47D2604 E1449080 0C673FB4 9FE96A68 3B5890C7 7792845B 9E8F82EF AB60FD26
    288A31D3 1F4DB4B4 1FBA8CD3 791EB4FF

B : EC82C879 3953FB24 C5770BFA BDDD8C9F 9DCDCE26 3901538F 77C2A1B7 CD11E5B4
    293B1C9C 3ACE358A AABC5ABC CB086A07 F62156F9 D97E1316 32BFF440 0E2EEA91

C : DAF409D0 3F639964 C66B562C 87C5FAC2 862A4A09 08488CB5 A468C3AB CA840679
    1699CD03 27F8EB1A E5489065 CA366C8A 1042148A 4F12203B AA33E3DB 109A60B8

W : 00000002 00000000


n0_final = 0 : permutation (j = 0)
```

```
A : 6F790FAB A4000660 457B3F54 96B16EA7 79837F24 938A78F3 D5933692 6CA2E17B
    5E570DE2 8C5E25B4 E031024C 50192595

B : 9446B9F6 055D9F72 4B9918A7 BE11F2DC BDE71C96 1E772012 C5E98A03 097ED5ED
    F3DECB25 063DB15F 4AB648CA 39F60E65 7CC45DA7 E903DFB2 DFFB282B 7513447A

C : DAF409D0 3F639964 C66B562C 87C5FAC2 862A4A09 08488CB5 A468C3AB CA840679
    1699CD03 27F8EB1A E5489065 CA366C8A 1042148A 4F12203B AA33E3DB 109A60B8

W : 00000002 00000000


n0_final = 0 : permutation (j = 1)

A : 1C290251 4185CEBF BE63FCF7 5123632C CE9E9872 8731B9CB 39B5BAD2 61418961
    B1A3A821 F0B5BB96 2E98F6F6 21871755

B : CF4DBDA1 87AAA701 B9D331B5 7862DE48 35926EF3 33A4044D 5AB41D0E CC854370
    046B6BE5 B20153FE D4F0929C DD308019 C8E9DCC3 AAC9F951 79BC157A 7498FE6A

C : DAF409D0 3F639964 C66B562C 87C5FAC2 862A4A09 08488CB5 A468C3AB CA840679
    1699CD03 27F8EB1A E5489065 CA366C8A 1042148A 4F12203B AA33E3DB 109A60B8

W : 00000002 00000000


n0_final = 0 : permutation (j = 2)

A : F16BE812 B54B9216 05295BC3 55F0AA85 918A9119 3A83095E B944B03B A257427F
    C9469D61 5093FAE2 EFCB8BBF 94F698D6

B : 3B777D9B E71DA24C 9289C8A8 8EA8364D 65B0CA0B 2DFC6573 4FBE9E20 3305D39B
    66A3B92C A17E515C EF5A6AFD E7C9BDB3 A76ADB19 FAFFF7BE E34C5EB4 82389BFD

C : DAF409D0 3F639964 C66B562C 87C5FAC2 862A4A09 08488CB5 A468C3AB CA840679
    1699CD03 27F8EB1A E5489065 CA366C8A 1042148A 4F12203B AA33E3DB 109A60B8

W : 00000002 00000000


n0_final = 0 : permutation - add C to A

A : 5402B016 26ABFA79 9BE7A217 6AF8A837 F46EF30C B23B2A3A 28E9C16E F273ECBB
    E5C70B26 FD9A2678 6F1F23C9 C8DBD0C1

B : 3B777D9B E71DA24C 9289C8A8 8EA8364D 65B0CA0B 2DFC6573 4FBE9E20 3305D39B
    66A3B92C A17E515C EF5A6AFD E7C9BDB3 A76ADB19 FAFFF7BE E34C5EB4 82389BFD

C : DAF409D0 3F639964 C66B562C 87C5FAC2 862A4A09 08488CB5 A468C3AB CA840679
    1699CD03 27F8EB1A E5489065 CA366C8A 1042148A 4F12203B AA33E3DB 109A60B8

W : 00000002 00000000


n0_final = 0 : subtract M from C

A : 5402B016 26ABFA79 9BE7A217 6AF8A837 F46EF30C B23B2A3A 28E9C16E F273ECBB
    E5C70B26 FD9A2678 6F1F23C9 C8DBD0C1

B : 3B777D9B E71DA24C 9289C8A8 8EA8364D 65B0CA0B 2DFC6573 4FBE9E20 3305D39B
    66A3B92C A17E515C EF5A6AFD E7C9BDB3 A76ADB19 FAFFF7BE E34C5EB4 82389BFD

C : DAF40950 3F639964 C66B562C 87C5FAC2 862A4A09 08488CB5 A468C3AB CA840679
    1699CD03 27F8EB1A E5489065 CA366C8A 1042148A 4F12203B AA33E3DB 109A60B8

W : 00000002 00000000
```

```
n0_final = 0 : swap B with C

A : 5402B016 26ABFA79 9BE7A217 6AF8A837 F46EF30C B23B2A3A 28E9C16E F273ECBB
    E5C70B26 FD9A2678 6F1F23C9 C8DBD0C1

B : DAF40950 3F639964 C66B562C 87C5FAC2 862A4A09 08488CB5 A468C3AB CA840679
    1699CD03 27F8EB1A E5489065 CA366C8A 1042148A 4F12203B AA33E3DB 109A60B8

C : 3B777D9B E71DA24C 9289C8A8 8EA8364D 65B0CA0B 2DFC6573 4FBE9E20 3305D39B
    66A3B92C A17E515C EF5A6AFD E7C9BDB3 A76ADB19 FAFFF7BE E34C5EB4 82389BFD

W : 00000002 00000000


n0_final = 1 : add M to B

A : 5402B016 26ABFA79 9BE7A217 6AF8A837 F46EF30C B23B2A3A 28E9C16E F273ECBB
    E5C70B26 FD9A2678 6F1F23C9 C8DBD0C1

B : DAF409D0 3F639964 C66B562C 87C5FAC2 862A4A09 08488CB5 A468C3AB CA840679
    1699CD03 27F8EB1A E5489065 CA366C8A 1042148A 4F12203B AA33E3DB 109A60B8

C : 3B777D9B E71DA24C 9289C8A8 8EA8364D 65B0CA0B 2DFC6573 4FBE9E20 3305D39B
    66A3B92C A17E515C EF5A6AFD E7C9BDB3 A76ADB19 FAFFF7BE E34C5EB4 82389BFD

W : 00000002 00000000


n0_final = 1 : xor counter W into A

A : 5402B014 26ABFA79 9BE7A217 6AF8A837 F46EF30C B23B2A3A 28E9C16E F273ECBB
    E5C70B26 FD9A2678 6F1F23C9 C8DBD0C1

B : DAF409D0 3F639964 C66B562C 87C5FAC2 862A4A09 08488CB5 A468C3AB CA840679
    1699CD03 27F8EB1A E5489065 CA366C8A 1042148A 4F12203B AA33E3DB 109A60B8

C : 3B777D9B E71DA24C 9289C8A8 8EA8364D 65B0CA0B 2DFC6573 4FBE9E20 3305D39B
    66A3B92C A17E515C EF5A6AFD E7C9BDB3 A76ADB19 FAFFF7BE E34C5EB4 82389BFD

W : 00000002 00000000


n0_final = 1 : permutation - rotate B

A : 5402B014 26ABFA79 9BE7A217 6AF8A837 F46EF30C B23B2A3A 28E9C16E F273ECBB
    E5C70B26 FD9A2678 6F1F23C9 C8DBD0C1

B : 13A1B5E8 32C87EC7 AC598CD6 F5850F8B 94130C54 196A1091 875748D1 0CF39508
    9A062D33 D6344FF1 20CBCA91 D915946C 29142084 40769E24 C7B75467 C1702134

C : 3B777D9B E71DA24C 9289C8A8 8EA8364D 65B0CA0B 2DFC6573 4FBE9E20 3305D39B
    66A3B92C A17E515C EF5A6AFD E7C9BDB3 A76ADB19 FAFFF7BE E34C5EB4 82389BFD

W : 00000002 00000000


n0_final = 1 : permutation (j = 0)

A : B2B7AC93 3280F0EF B7D1A386 134AD19A DFA84728 2E1F3273 F925D5FE 3BC0A039
    09CA6539 80F200DF 72A143BF 9869ED12

B : F93A70E0 00D3F200 80E2EB89 F43A3B69 0871A07E E334ECAE 0874BBA2 DDD875D6
    C239C0A1 D36560C3 CCC92962 D5BD3A34 1F601264 4D923358 C740F4B6 6E556C0C

C : 3B777D9B E71DA24C 9289C8A8 8EA8364D 65B0CA0B 2DFC6573 4FBE9E20 3305D39B
    66A3B92C A17E515C EF5A6AFD E7C9BDB3 A76ADB19 FAFFF7BE E34C5EB4 82389BFD
```

```
W : 00000002 00000000


n0_final = 1 : permutation (j = 1)

A : 0E22DD0D 5A7ED8B2 555E9A44 68F49ABD 6F4B6A35 7BF14653 269337DB 6CB49E0E
    18CB4BF6 A18475A2 6D809BBD 99766B2F

B : 8C380710 D2D21616 38F39361 5F2FBD22 F7D7F4F5 98125300 82961306 DD397F7D
    75AEA3B1 034BE6CA 3333377E 3C71112B AE74B102 1F2ADF1C 57ED2149 4FE1B9E9

C : 3B777D9B E71DA24C 9289C8A8 8EA8364D 65B0CA0B 2DFC6573 4FBE9E20 3305D39B
    66A3B92C A17E515C EF5A6AFD E7C9BDB3 A76ADB19 FAFFF7BE E34C5EB4 82389BFD

W : 00000002 00000000


n0_final = 1 : permutation (j = 2)

A : B18E580E 6100E1AB B3089D3F 953C78F7 69DD127D 7E11BDC3 06E6CF36 CD828F57
    03E5CBA8 A40C93A9 622FA53C 8FC52BE0

B : 260A3DC2 E9E62340 385A3EBF 2978F492 A1DE4E1A AEDBB855 49DB44CD D0B179F3
    7D7FAAE0 87798FA8 9F7F5E35 4A9F52FE A0F35652 65A6D26E 320A1851 EFF9A7CD

C : 3B777D9B E71DA24C 9289C8A8 8EA8364D 65B0CA0B 2DFC6573 4FBE9E20 3305D39B
    66A3B92C A17E515C EF5A6AFD E7C9BDB3 A76ADB19 FAFFF7BE E34C5EB4 82389BFD

W : 00000002 00000000


n0_final = 1 : permutation - add C to A

A : AA38E80B A994046A C3229CBC 5AD13E73 ADC3B862 85DDBE95 BD7F2851 9F25611C
    AD5D9343 17CBF1F9 2CAA53C9 B22A93B1

B : 260A3DC2 E9E62340 385A3EBF 2978F492 A1DE4E1A AEDBB855 49DB44CD D0B179F3
    7D7FAAE0 87798FA8 9F7F5E35 4A9F52FE A0F35652 65A6D26E 320A1851 EFF9A7CD

C : 3B777D9B E71DA24C 9289C8A8 8EA8364D 65B0CA0B 2DFC6573 4FBE9E20 3305D39B
    66A3B92C A17E515C EF5A6AFD E7C9BDB3 A76ADB19 FAFFF7BE E34C5EB4 82389BFD

W : 00000002 00000000


n0_final = 1 : subtract M from C

A : AA38E80B A994046A C3229CBC 5AD13E73 ADC3B862 85DDBE95 BD7F2851 9F25611C
    AD5D9343 17CBF1F9 2CAA53C9 B22A93B1

B : 260A3DC2 E9E62340 385A3EBF 2978F492 A1DE4E1A AEDBB855 49DB44CD D0B179F3
    7D7FAAE0 87798FA8 9F7F5E35 4A9F52FE A0F35652 65A6D26E 320A1851 EFF9A7CD

C : 3B777D1B E71DA24C 9289C8A8 8EA8364D 65B0CA0B 2DFC6573 4FBE9E20 3305D39B
    66A3B92C A17E515C EF5A6AFD E7C9BDB3 A76ADB19 FAFFF7BE E34C5EB4 82389BFD

W : 00000002 00000000


n0_final = 1 : swap B with C

A : AA38E80B A994046A C3229CBC 5AD13E73 ADC3B862 85DDBE95 BD7F2851 9F25611C
    AD5D9343 17CBF1F9 2CAA53C9 B22A93B1

B : 3B777D1B E71DA24C 9289C8A8 8EA8364D 65B0CA0B 2DFC6573 4FBE9E20 3305D39B
    66A3B92C A17E515C EF5A6AFD E7C9BDB3 A76ADB19 FAFFF7BE E34C5EB4 82389BFD
```

```
C : 260A3DC2 E9E62340 385A3EBF 2978F492 A1DE4E1A AEDBB855 49DB44CD D0B179F3
    7D7FAAE0 87798FA8 9F7F5E35 4A9F52FE A0F35652 65A6D26E 320A1851 EFF9A7CD

W : 00000002 00000000


n0_final = 2 : add M to B

A : AA38E80B A994046A C3229CBC 5AD13E73 ADC3B862 85DDBE95 BD7F2851 9F25611C
    AD5D9343 17CBF1F9 2CAA53C9 B22A93B1

B : 3B777D9B E71DA24C 9289C8A8 8EA8364D 65B0CA0B 2DFC6573 4FBE9E20 3305D39B
    66A3B92C A17E515C EF5A6AFD E7C9BDB3 A76ADB19 FAFFF7BE E34C5EB4 82389BFD

C : 260A3DC2 E9E62340 385A3EBF 2978F492 A1DE4E1A AEDBB855 49DB44CD D0B179F3
    7D7FAAE0 87798FA8 9F7F5E35 4A9F52FE A0F35652 65A6D26E 320A1851 EFF9A7CD

W : 00000002 00000000


n0_final = 2 : xor counter W into A

A : AA38E809 A994046A C3229CBC 5AD13E73 ADC3B862 85DDBE95 BD7F2851 9F25611C
    AD5D9343 17CBF1F9 2CAA53C9 B22A93B1

B : 3B777D9B E71DA24C 9289C8A8 8EA8364D 65B0CA0B 2DFC6573 4FBE9E20 3305D39B
    66A3B92C A17E515C EF5A6AFD E7C9BDB3 A76ADB19 FAFFF7BE E34C5EB4 82389BFD

C : 260A3DC2 E9E62340 385A3EBF 2978F492 A1DE4E1A AEDBB855 49DB44CD D0B179F3
    7D7FAAE0 87798FA8 9F7F5E35 4A9F52FE A0F35652 65A6D26E 320A1851 EFF9A7CD

W : 00000002 00000000


n0_final = 2 : permutation - rotate B

A : AA38E809 A994046A C3229CBC 5AD13E73 ADC3B862 85DDBE95 BD7F2851 9F25611C
    AD5D9343 17CBF1F9 2CAA53C9 B22A93B1

B : FB3676EE 4499CE3B 91512513 6C9B1D50 9416CB61 CAE65BF8 3C409F7D A736660B
    7258CD47 A2B942FC D5FBDEB4 7B67CF93 B6334ED5 EF7DF5FF BD69C698 37FB0471

C : 260A3DC2 E9E62340 385A3EBF 2978F492 A1DE4E1A AEDBB855 49DB44CD D0B179F3
    7D7FAAE0 87798FA8 9F7F5E35 4A9F52FE A0F35652 65A6D26E 320A1851 EFF9A7CD

W : 00000002 00000000


n0_final = 2 : permutation (j = 0)

A : EB3858E3 3571FF05 07F3E3A5 F01ACEC6 7C2B8559 782CE3DD 553117F5 DC8BA789
    EC309F7C 79D58C9A 4E25C57F 79B079C2

B : 9725D95D 0BA369BF D9D4169F 615F3855 ABF9EC65 121FABD3 D24FD6F0 6D189461
    F77EFA0D C358F69C 1A2D87E9 7080191B 78A13AB7 1475EB05 82DF916B 601339DB

C : 260A3DC2 E9E62340 385A3EBF 2978F492 A1DE4E1A AEDBB855 49DB44CD D0B179F3
    7D7FAAE0 87798FA8 9F7F5E35 4A9F52FE A0F35652 65A6D26E 320A1851 EFF9A7CD

W : 00000002 00000000


n0_final = 2 : permutation (j = 1)

A : 5C8DD649 1612AD22 789B27A7 2AB3CED6 46180EC5 FE6C7E91 A5AC461A 87B17270
    CCE4F762 85DF8D82 3DC305DA F80F92C3
```

```
B : D421E125 EB0EF9F6 B70F6574 FAFC24D3 64E8D056 5E1F25DB 66A357C4 DDC145FE
    4D8FDDAD 6F5CBFE4 B33FD78A 344C031F 48A58454 29785764 5FEC9B32 B868FE39

C : 260A3DC2 E9E62340 385A3EBF 2978F492 A1DE4E1A AEDBB855 49DB44CD D0B179F3
    7D7FAAE0 87798FA8 9F7F5E35 4A9F52FE A0F35652 65A6D26E 320A1851 EFF9A7CD

W : 00000002 00000000


n0_final = 2 : permutation (j = 2)

A : 3F7A1D06 AEE6E8BA 33DFA929 684A2EF8 2D4ABF45 6CDA81F1 52EE20E5 592A3656
    BC0F4857 A15C35CC F42A4EA9 825D3AC4

B : F91EEE5E 99DCC78C 82F72599 8CACD775 09544255 ED275CF3 0166F95E 2C375AFA
    49AAFBE0 4D9C01C6 CB6E700F CE4DCF97 D2BBBF00 0C5364FB B40C8732 0D733948

C : 260A3DC2 E9E62340 385A3EBF 2978F492 A1DE4E1A AEDBB855 49DB44CD D0B179F3
    7D7FAAE0 87798FA8 9F7F5E35 4A9F52FE A0F35652 65A6D26E 320A1851 EFF9A7CD

W : 00000002 00000000


n0_final = 2 : permutation - add C to A

A : A38C0C63 17C2CAE8 3248572C 1C89CAD5 176ED597 B242B8AD 73298C22 7ADF1817
    00D909DA 61AD8518 90266914 9DC1F617

B : F91EEE5E 99DCC78C 82F72599 8CACD775 09544255 ED275CF3 0166F95E 2C375AFA
    49AAFBE0 4D9C01C6 CB6E700F CE4DCF97 D2BBBF00 0C5364FB B40C8732 0D733948

C : 260A3DC2 E9E62340 385A3EBF 2978F492 A1DE4E1A AEDBB855 49DB44CD D0B179F3
    7D7FAAE0 87798FA8 9F7F5E35 4A9F52FE A0F35652 65A6D26E 320A1851 EFF9A7CD

W : 00000002 00000000


n0_final = 2 : subtract M from C

A : A38C0C63 17C2CAE8 3248572C 1C89CAD5 176ED597 B242B8AD 73298C22 7ADF1817
    00D909DA 61AD8518 90266914 9DC1F617

B : F91EEE5E 99DCC78C 82F72599 8CACD775 09544255 ED275CF3 0166F95E 2C375AFA
    49AAFBE0 4D9C01C6 CB6E700F CE4DCF97 D2BBBF00 0C5364FB B40C8732 0D733948

C : 260A3D42 E9E62340 385A3EBF 2978F492 A1DE4E1A AEDBB855 49DB44CD D0B179F3
    7D7FAAE0 87798FA8 9F7F5E35 4A9F52FE A0F35652 65A6D26E 320A1851 EFF9A7CD

W : 00000002 00000000


n0_final = 2 : swap B with C  (final state)

A : A38C0C63 17C2CAE8 3248572C 1C89CAD5 176ED597 B242B8AD 73298C22 7ADF1817
    00D909DA 61AD8518 90266914 9DC1F617

B : 260A3D42 E9E62340 385A3EBF 2978F492 A1DE4E1A AEDBB855 49DB44CD D0B179F3
    7D7FAAE0 87798FA8 9F7F5E35 4A9F52FE A0F35652 65A6D26E 320A1851 EFF9A7CD

C : F91EEE5E 99DCC78C 82F72599 8CACD775 09544255 ED275CF3 0166F95E 2C375AFA
    49AAFBE0 4D9C01C6 CB6E700F CE4DCF97 D2BBBF00 0C5364FB B40C8732 0D733948

W : 00000002 00000000


Hash value (word array):
```

```
H : CB6E700F CE4DCF97 D2BBBF00 0C5364FB B40C8732 0D733948


Hash value (byte array):

H : 0F 70 6E CB 97 CF 4D CE 00 BF BB D2 FB 64 53 0C
    32 87 0C B4 48 39 73 0D
```

B.2 Intermediate States for Shabal-192 (Message B)

```
init

A : 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
    00000000 00000000 00000000 00000000

B : 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
    00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000

C : 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
    00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000

W : FFFFFFFF FFFFFFFF


block number = -1 : message block

M : 000000C0 000000C1 000000C2 000000C3 000000C4 000000C5 000000C6 000000C7
    000000C8 000000C9 000000CA 000000CB 000000CC 000000CD 000000CE 000000CF


block number = -1 : add M to B

A : 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
    00000000 00000000 00000000 00000000

B : 000000C0 000000C1 000000C2 000000C3 000000C4 000000C5 000000C6 000000C7
    000000C8 000000C9 000000CA 000000CB 000000CC 000000CD 000000CE 000000CF

C : 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
    00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000

W : FFFFFFFF FFFFFFFF


block number = -1 : xor counter W into A

A : FFFFFFFF FFFFFFFF 00000000 00000000 00000000 00000000 00000000 00000000
    00000000 00000000 00000000 00000000

B : 000000C0 000000C1 000000C2 000000C3 000000C4 000000C5 000000C6 000000C7
    000000C8 000000C9 000000CA 000000CB 000000CC 000000CD 000000CE 000000CF

C : 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
    00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000

W : FFFFFFFF FFFFFFFF


block number = -1 : permutation - rotate B

A : FFFFFFFF FFFFFFFF 00000000 00000000 00000000 00000000 00000000 00000000
    00000000 00000000 00000000 00000000
```

```
B : 01800000 01820000 01840000 01860000 01880000 018A0000 018C0000 018E0000
    01900000 01920000 01940000 01960000 01980000 019A0000 019C0000 019E0000

C : 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
    00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000

W : FFFFFFFF FFFFFFFF


block number = -1 : permutation (j = 0)

A : F78A4200 10145AFD AE842D9A 7DA2B6D1 E125063B 87057989 D33D3495 15A94983
    24F95064 DD75BF25 2CE50A5D 109E76BE

B : 028800C2 F8D8F4B6 29CC61C2 DCBC40CD 1DCAF9C4 7BEE8676 2FDACB6A E94AB67C
    D826AF9B 21AE40DA D032F5A2 EC4D8941 0B45BDFF ECDFA502 5243D265 8161492E

C : 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
    00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000

W : FFFFFFFF FFFFFFFF


block number = -1 : permutation (j = 1)

A : B11AE006 5117AAEA F864E13E 8395DEC5 7473DED3 F4F3C55C EE1065D0 0FBCE63D
    5D489898 3D28BAC9 DF45C1AD C5245735

B : 6E986FA3 0EF5A5BE CF36EA1B A553B940 992294EF 350A49DA 7F0FA886 E84EC433
    FEA840CE EDB4D4A1 A7FEF584 A4F133B9 9D075AD2 D2B370A6 B5683EE5 F2818B9F

C : 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
    00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000

W : FFFFFFFF FFFFFFFF


block number = -1 : permutation (j = 2)

A : 6A2E9EF8 2A8ED2CE 5B7DEB2B 76F5EC56 C761D844 6FFDD495 A6E50D15 CB7EBB78
    56F43E26 A9D57A1C 5FD697A6 E72A8ACB

B : 58BCBB94 EC47A15F AEE93484 DFCBC8F7 A79448D8 BF65BE85 5A9D45D8 59979BCE
    C5CEA626 4B6B8229 16E719E3 7D6323F4 9305747C F34C64AE CAF91592 FDD6620B

C : 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
    00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000

W : FFFFFFFF FFFFFFFF


block number = -1 : permutation - add C to A

A : 6A2E9EF8 2A8ED2CE 5B7DEB2B 76F5EC56 C761D844 6FFDD495 A6E50D15 CB7EBB78
    56F43E26 A9D57A1C 5FD697A6 E72A8ACB

B : 58BCBB94 EC47A15F AEE93484 DFCBC8F7 A79448D8 BF65BE85 5A9D45D8 59979BCE
    C5CEA626 4B6B8229 16E719E3 7D6323F4 9305747C F34C64AE CAF91592 FDD6620B

C : 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
    00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000

W : FFFFFFFF FFFFFFFF


block number = -1 : subtract M from C
```

```
A : 6A2E9EF8 2A8ED2CE 5B7DEB2B 76F5EC56 C761D844 6FFDD495 A6E50D15 CB7EBB78
    56F43E26 A9D57A1C 5FD697A6 E72A8ACB

B : 58BCBB94 EC47A15F AEE93484 DFCBC8F7 A79448D8 BF65BE85 5A9D45D8 59979BCE
    C5CEA626 4B6B8229 16E719E3 7D6323F4 9305747C F34C64AE CAF91592 FDD6620B

C : FFFFFF40 FFFFFF3F FFFFFF3E FFFFFF3D FFFFFF3C FFFFFF3B FFFFFF3A FFFFFF39
    FFFFFF38 FFFFFF37 FFFFFF36 FFFFFF35 FFFFFF34 FFFFFF33 FFFFFF32 FFFFFF31

W : FFFFFFFF FFFFFFFF


block number = -1 : swap B with C

A : 6A2E9EF8 2A8ED2CE 5B7DEB2B 76F5EC56 C761D844 6FFDD495 A6E50D15 CB7EBB78
    56F43E26 A9D57A1C 5FD697A6 E72A8ACB

B : FFFFFF40 FFFFFF3F FFFFFF3E FFFFFF3D FFFFFF3C FFFFFF3B FFFFFF3A FFFFFF39
    FFFFFF38 FFFFFF37 FFFFFF36 FFFFFF35 FFFFFF34 FFFFFF33 FFFFFF32 FFFFFF31

C : 58BCBB94 EC47A15F AEE93484 DFCBC8F7 A79448D8 BF65BE85 5A9D45D8 59979BCE
    C5CEA626 4B6B8229 16E719E3 7D6323F4 9305747C F34C64AE CAF91592 FDD6620B

W : FFFFFFFF FFFFFFFF


block number = 0 : increment counter W

A : 6A2E9EF8 2A8ED2CE 5B7DEB2B 76F5EC56 C761D844 6FFDD495 A6E50D15 CB7EBB78
    56F43E26 A9D57A1C 5FD697A6 E72A8ACB

B : FFFFFF40 FFFFFF3F FFFFFF3E FFFFFF3D FFFFFF3C FFFFFF3B FFFFFF3A FFFFFF39
    FFFFFF38 FFFFFF37 FFFFFF36 FFFFFF35 FFFFFF34 FFFFFF33 FFFFFF32 FFFFFF31

C : 58BCBB94 EC47A15F AEE93484 DFCBC8F7 A79448D8 BF65BE85 5A9D45D8 59979BCE
    C5CEA626 4B6B8229 16E719E3 7D6323F4 9305747C F34C64AE CAF91592 FDD6620B

W : 00000000 00000000


block number = 0 : message block

M : 000000D0 000000D1 000000D2 000000D3 000000D4 000000D5 000000D6 000000D7
    000000D8 000000D9 000000DA 000000DB 000000DC 000000DD 000000DE 000000DF


block number = 0 : add M to B

A : 6A2E9EF8 2A8ED2CE 5B7DEB2B 76F5EC56 C761D844 6FFDD495 A6E50D15 CB7EBB78
    56F43E26 A9D57A1C 5FD697A6 E72A8ACB

B : 00000010 00000010 00000010 00000010 00000010 00000010 00000010 00000010
    00000010 00000010 00000010 00000010 00000010 00000010 00000010 00000010

C : 58BCBB94 EC47A15F AEE93484 DFCBC8F7 A79448D8 BF65BE85 5A9D45D8 59979BCE
    C5CEA626 4B6B8229 16E719E3 7D6323F4 9305747C F34C64AE CAF91592 FDD6620B

W : 00000000 00000000


block number = 0 : xor counter W into A

A : 6A2E9EF8 2A8ED2CE 5B7DEB2B 76F5EC56 C761D844 6FFDD495 A6E50D15 CB7EBB78
    56F43E26 A9D57A1C 5FD697A6 E72A8ACB

B : 00000010 00000010 00000010 00000010 00000010 00000010 00000010 00000010
    00000010 00000010 00000010 00000010 00000010 00000010 00000010 00000010
```

```
C : 58BCBB94 EC47A15F AEE93484 DFCBC8F7 A79448D8 BF65BE85 5A9D45D8 59979BCE
    C5CEA626 4B6B8229 16E719E3 7D6323F4 9305747C F34C64AE CAF91592 FDD6620B

W : 00000000 00000000


block number = 0 : permutation - rotate B

A : 6A2E9EF8 2A8ED2CE 5B7DEB2B 76F5EC56 C761D844 6FFDD495 A6E50D15 CB7EBB78
    56F43E26 A9D57A1C 5FD697A6 E72A8ACB

B : 00200000 00200000 00200000 00200000 00200000 00200000 00200000 00200000
    00200000 00200000 00200000 00200000 00200000 00200000 00200000 00200000

C : 58BCBB94 EC47A15F AEE93484 DFCBC8F7 A79448D8 BF65BE85 5A9D45D8 59979BCE
    C5CEA626 4B6B8229 16E719E3 7D6323F4 9305747C F34C64AE CAF91592 FDD6620B

W : 00000000 00000000


block number = 0 : permutation (j = 0)

A : 40EDD726 1A8754BC 7AF69209 DD22532D 04BFEA68 4E68A17B 6E3AE516 89451CE8
    237E61BE 50533EB1 77550A53 4EE347AA

B : 20C6148A C0A0482A 20FEE685 BD139D7F FB001597 B1D75E84 91851AE9 76FAE317
    DCC19E41 AFECC14E 88EAF5AC B15CB855 BF5228D9 E538AB43 85496DF6 229DACD2

C : 58BCBB94 EC47A15F AEE93484 DFCBC8F7 A79448D8 BF65BE85 5A9D45D8 59979BCE
    C5CEA626 4B6B8229 16E719E3 7D6323F4 9305747C F34C64AE CAF91592 FDD6620B

W : 00000000 00000000


block number = 0 : permutation (j = 1)

A : DCB0D1FF 9807FEF5 55FCAED1 DDFCB641 2546AE31 EFC27512 31D29DFA 9459E7A6
    9E7C5606 FC6CB0E4 5699952C 8FDD2936

B : 5EA811AF 428F0F99 F1F49AC3 7C94CC73 978382D6 603DF212 8A6C5F00 9DD710E7
    9ACC1283 38218397 BBD6BA77 40BA3915 A41D007D DA4CDC6A C4BFB9E8 2E9D41FD

C : 58BCBB94 EC47A15F AEE93484 DFCBC8F7 A79448D8 BF65BE85 5A9D45D8 59979BCE
    C5CEA626 4B6B8229 16E719E3 7D6323F4 9305747C F34C64AE CAF91592 FDD6620B

W : 00000000 00000000


block number = 0 : permutation (j = 2)

A : A26F4FDE 24426C48 949686DD 723E1870 D02F6DE4 7F2B95C6 80DE631A DADF6026
    ABCC94F0 E7106561 2495F432 6F985A6C

B : 61550878 89EF2B75 A1660C46 7EF3855B 7297B58C 1BC67793 7FB1C723 B66FC640
    1A48B71C F0976D17 088CE80A A454EDF3 1C096BF4 AC76224B 5215781C CD5D2669

C : 58BCBB94 EC47A15F AEE93484 DFCBC8F7 A79448D8 BF65BE85 5A9D45D8 59979BCE
    C5CEA626 4B6B8229 16E719E3 7D6323F4 9305747C F34C64AE CAF91592 FDD6620B

W : 00000000 00000000


block number = 0 : permutation - add C to A

A : FD749ED4 B798E530 33904B6F 46BDA85E 076934B4 454B4058 77F74527 FB4CF465
    62931DA9 E778C8DB 22B3998E AC15CFB9
```

```
B : 61550878 89EF2B75 A1660C46 7EF3855B 7297B58C 1BC67793 7FB1C723 B66FC640
    1A48B71C F0976D17 088CE80A A454EDF3 1C096BF4 AC76224B 5215781C CD5D2669

C : 58BCBB94 EC47A15F AEE93484 DFCBC8F7 A79448D8 BF65BE85 5A9D45D8 59979BCE
    C5CEA626 4B6B8229 16E719E3 7D6323F4 9305747C F34C64AE CAF91592 FDD6620B

W : 00000000 00000000


block number = 0 : subtract M from C

A : FD749ED4 B798E530 33904B6F 46BDA85E 076934B4 454B4058 77F74527 FB4CF465
    62931DA9 E778C8DB 22B3998E AC15CFB9

B : 61550878 89EF2B75 A1660C46 7EF3855B 7297B58C 1BC67793 7FB1C723 B66FC640
    1A48B71C F0976D17 088CE80A A454EDF3 1C096BF4 AC76224B 5215781C CD5D2669

C : 58BCBAC4 EC47A08E AEE933B2 DFCBC824 A7944804 BF65BDB0 5A9D4502 59979AF7
    C5CEA54E 4B6B8150 16E71909 7D632319 930573A0 F34C63D1 CAF914B4 FDD6612C

W : 00000000 00000000


block number = 0 : swap B with C

A : FD749ED4 B798E530 33904B6F 46BDA85E 076934B4 454B4058 77F74527 FB4CF465
    62931DA9 E778C8DB 22B3998E AC15CFB9

B : 58BCBAC4 EC47A08E AEE933B2 DFCBC824 A7944804 BF65BDB0 5A9D4502 59979AF7
    C5CEA54E 4B6B8150 16E71909 7D632319 930573A0 F34C63D1 CAF914B4 FDD6612C

C : 61550878 89EF2B75 A1660C46 7EF3855B 7297B58C 1BC67793 7FB1C723 B66FC640
    1A48B71C F0976D17 088CE80A A454EDF3 1C096BF4 AC76224B 5215781C CD5D2669

W : 00000000 00000000


block number = 1 : increment counter W

A : FD749ED4 B798E530 33904B6F 46BDA85E 076934B4 454B4058 77F74527 FB4CF465
    62931DA9 E778C8DB 22B3998E AC15CFB9

B : 58BCBAC4 EC47A08E AEE933B2 DFCBC824 A7944804 BF65BDB0 5A9D4502 59979AF7
    C5CEA54E 4B6B8150 16E71909 7D632319 930573A0 F34C63D1 CAF914B4 FDD6612C

C : 61550878 89EF2B75 A1660C46 7EF3855B 7297B58C 1BC67793 7FB1C723 B66FC640
    1A48B71C F0976D17 088CE80A A454EDF3 1C096BF4 AC76224B 5215781C CD5D2669

W : 00000001 00000000


block number = 1 : message block

M : 64636261 68676665 6C6B6A69 706F6E6D 74737271 78777675 302D7A79 34333231
    38373635 42412D39 46454443 4A494847 4E4D4C4B 5251504F 56555453 5A595857


block number = 1 : add M to B

A : FD749ED4 B798E530 33904B6F 46BDA85E 076934B4 454B4058 77F74527 FB4CF465
    62931DA9 E778C8DB 22B3998E AC15CFB9

B : BD201D25 54AF06F3 1B549E1B 503B3691 1C07BA75 37DD3425 8ACABF7B 8DCACD28
    FE05DB83 8DACAE89 5D2C5D4C C7AC6B60 E152BFEB 459DB420 214E6907 582FB983

C : 61550878 89EF2B75 A1660C46 7EF3855B 7297B58C 1BC67793 7FB1C723 B66FC640
```

```
    1A48B71C F0976D17 088CE80A A454EDF3 1C096BF4 AC76224B 5215781C CD5D2669

W : 00000001 00000000


block number = 1 : xor counter W into A

A : FD749ED5 B798E530 33904B6F 46BDA85E 076934B4 454B4058 77F74527 FB4CF465
    62931DA9 E778C8DB 22B3998E AC15CFB9

B : BD201D25 54AF06F3 1B549E1B 503B3691 1C07BA75 37DD3425 8ACABF7B 8DCACD28
    FE05DB83 8DACAE89 5D2C5D4C C7AC6B60 E152BFEB 459DB420 214E6907 582FB983

C : 61550878 89EF2B75 A1660C46 7EF3855B 7297B58C 1BC67793 7FB1C723 B66FC640
    1A48B71C F0976D17 088CE80A A454EDF3 1C096BF4 AC76224B 5215781C CD5D2669

W : 00000001 00000000


block number = 1 : permutation - rotate B

A : FD749ED5 B798E530 33904B6F 46BDA85E 076934B4 454B4058 77F74527 FB4CF465
    62931DA9 E778C8DB 22B3998E AC15CFB9

B : 3A4B7A40 0DE6A95E 3C3636A9 6D22A076 74EA380F 684A6FBA 7EF71595 9A511B95
    B707FC0B 5D131B59 BA98BA58 D6C18F58 7FD7C2A5 68408B3B D20E429C 7306B05F

C : 61550878 89EF2B75 A1660C46 7EF3855B 7297B58C 1BC67793 7FB1C723 B66FC640
    1A48B71C F0976D17 088CE80A A454EDF3 1C096BF4 AC76224B 5215781C CD5D2669

W : 00000001 00000000


block number = 1 : permutation (j = 0)

A : AE92BFA7 A5A41F0D 97E9BFE0 D5D29125 D7E45AD4 AD76451A A07288D3 1CEAF9A7
    EC0CCA2F 95DD5CBB 8F04308E 5CED88C8

B : A70EFF9C CF91415F 5F09EFC0 FEA428A2 C1CFD535 821D6591 A2635C06 D7B73173
    7DFCCDC7 D00495F6 05CABBC0 0E916986 AEC2C512 8ADAF684 CC0AC526 CC200E64

C : 61550878 89EF2B75 A1660C46 7EF3855B 7297B58C 1BC67793 7FB1C723 B66FC640
    1A48B71C F0976D17 088CE80A A454EDF3 1C096BF4 AC76224B 5215781C CD5D2669

W : 00000001 00000000


block number = 1 : permutation (j = 1)

A : 16432E52 F2DFD20D 94B0EB8C F25C60A2 5B7A200A C3588ED6 658076F8 583B6AD6
    3613F4F2 B196A6CA F9AAD41D 5A9869F3

B : F908641C 7150CBBA 27B148F8 226E31BD 4A73A166 4A539216 429393EF 0A09F4EB
    12454A23 AD29061F 60DA63F3 10814C51 F90055D0 29129C20 026A034A 3F8489E0

C : 61550878 89EF2B75 A1660C46 7EF3855B 7297B58C 1BC67793 7FB1C723 B66FC640
    1A48B71C F0976D17 088CE80A A454EDF3 1C096BF4 AC76224B 5215781C CD5D2669

W : 00000001 00000000


block number = 1 : permutation (j = 2)

A : 9A1BAE76 63F344EA 3AB9815C FFF0A9EF 857327D2 CAE3E932 C69632F6 453E3FD7
    EB8672D9 56EBC058 7203658F 89F9EF17

B : 185313EA 0F470A20 31E0B3D0 40CD2940 F1031345 08AB9F39 4061597D 141CBFC6
```

```
    5E064C6B 6F4E1AF2 F8DD0AEF 9BC3588A E6792687 FB3107E7 89289CE4 090F0328

C : 61550878 89EF2B75 A1660C46 7EF3855B 7297B58C 1BC67793 7FB1C723 B66FC640
    1A48B71C F0976D17 088CE80A A454EDF3 1C096BF4 AC76224B 5215781C CD5D2669

W : 00000001 00000000


block number = 1 : permutation - add C to A

A : 8AC1482D 53E96EE2 8CE546AF 731DF574 883399D6 B9195E52 5CE34315 6EE2FB4A
    C53EAC67 FFD598F4 2AD76C84 644E1660

B : 185313EA 0F470A20 31E0B3D0 40CD2940 F1031345 08AB9F39 4061597D 141CBFC6
    5E064C6B 6F4E1AF2 F8DD0AEF 9BC3588A E6792687 FB3107E7 89289CE4 090F0328

C : 61550878 89EF2B75 A1660C46 7EF3855B 7297B58C 1BC67793 7FB1C723 B66FC640
    1A48B71C F0976D17 088CE80A A454EDF3 1C096BF4 AC76224B 5215781C CD5D2669

W : 00000001 00000000


block number = 1 : subtract M from C

A : 8AC1482D 53E96EE2 8CE546AF 731DF574 883399D6 B9195E52 5CE34315 6EE2FB4A
    C53EAC67 FFD598F4 2AD76C84 644E1660

B : 185313EA 0F470A20 31E0B3D0 40CD2940 F1031345 08AB9F39 4061597D 141CBFC6
    5E064C6B 6F4E1AF2 F8DD0AEF 9BC3588A E6792687 FB3107E7 89289CE4 090F0328

C : FCF1A617 2187C510 34FAA1DD 0E8416EE FE24431B A34F011E 4F844CAA 823C940F
    E21180E7 AE563FDE C247A3C7 5A0BA5AC CDBC1FA9 5A24D1FC FBC023C9 7303CE12

W : 00000001 00000000


block number = 1 : swap B with C

A : 8AC1482D 53E96EE2 8CE546AF 731DF574 883399D6 B9195E52 5CE34315 6EE2FB4A
    C53EAC67 FFD598F4 2AD76C84 644E1660

B : FCF1A617 2187C510 34FAA1DD 0E8416EE FE24431B A34F011E 4F844CAA 823C940F
    E21180E7 AE563FDE C247A3C7 5A0BA5AC CDBC1FA9 5A24D1FC FBC023C9 7303CE12

C : 185313EA 0F470A20 31E0B3D0 40CD2940 F1031345 08AB9F39 4061597D 141CBFC6
    5E064C6B 6F4E1AF2 F8DD0AEF 9BC3588A E6792687 FB3107E7 89289CE4 090F0328

W : 00000001 00000000


block number = 2 : increment counter W

A : 8AC1482D 53E96EE2 8CE546AF 731DF574 883399D6 B9195E52 5CE34315 6EE2FB4A
    C53EAC67 FFD598F4 2AD76C84 644E1660

B : FCF1A617 2187C510 34FAA1DD 0E8416EE FE24431B A34F011E 4F844CAA 823C940F
    E21180E7 AE563FDE C247A3C7 5A0BA5AC CDBC1FA9 5A24D1FC FBC023C9 7303CE12

C : 185313EA 0F470A20 31E0B3D0 40CD2940 F1031345 08AB9F39 4061597D 141CBFC6
    5E064C6B 6F4E1AF2 F8DD0AEF 9BC3588A E6792687 FB3107E7 89289CE4 090F0328

W : 00000002 00000000


block number = 2 : message block

M : 3231302D 36353433 2D393837 64636261 68676665 6C6B6A69 706F6E6D 74737271
```

```
    78777675 00807A79 00000000 00000000 00000000 00000000 00000000 00000000


block number = 2 : add M to B

A : 8AC1482D 53E96EE2 8CE546AF 731DF574 883399D6 B9195E52 5CE34315 6EE2FB4A
    C53EAC67 FFD598F4 2AD76C84 644E1660

B : 2F22D644 57BCF943 6233DA14 72E7794F 668BA980 0FBA6B87 BFF3BB17 F6B00680
    5A88F75C AED6BA57 C247A3C7 5A0BA5AC CDBC1FA9 5A24D1FC FBC023C9 7303CE12

C : 185313EA 0F470A20 31E0B3D0 40CD2940 F1031345 08AB9F39 4061597D 141CBFC6
    5E064C6B 6F4E1AF2 F8DD0AEF 9BC3588A E6792687 FB3107E7 89289CE4 090F0328

W : 00000002 00000000


block number = 2 : xor counter W into A

A : 8AC1482F 53E96EE2 8CE546AF 731DF574 883399D6 B9195E52 5CE34315 6EE2FB4A
    C53EAC67 FFD598F4 2AD76C84 644E1660

B : 2F22D644 57BCF943 6233DA14 72E7794F 668BA980 0FBA6B87 BFF3BB17 F6B00680
    5A88F75C AED6BA57 C247A3C7 5A0BA5AC CDBC1FA9 5A24D1FC FBC023C9 7303CE12

C : 185313EA 0F470A20 31E0B3D0 40CD2940 F1031345 08AB9F39 4061597D 141CBFC6
    5E064C6B 6F4E1AF2 F8DD0AEF 9BC3588A E6792687 FB3107E7 89289CE4 090F0328

W : 00000002 00000000


block number = 2 : permutation - rotate B

A : 8AC1482F 53E96EE2 8CE546AF 731DF574 883399D6 B9195E52 5CE34315 6EE2FB4A
    C53EAC67 FFD598F4 2AD76C84 644E1660

B : AC885E45 F286AF79 B428C467 F29EE5CE 5300CD17 D70E1F74 762F7FE7 0D01ED60
    EEB8B511 74AF5DAD 478F848F 4B58B417 3F539B78 A3F8B449 4793F780 9C24E607

C : 185313EA 0F470A20 31E0B3D0 40CD2940 F1031345 08AB9F39 4061597D 141CBFC6
    5E064C6B 6F4E1AF2 F8DD0AEF 9BC3588A E6792687 FB3107E7 89289CE4 090F0328

W : 00000002 00000000


block number = 2 : permutation (j = 0)

A : 4CD8A476 C183CB00 EB15E699 0DA9B1DE A0EB57D0 D9904284 85488359 EC89FDEE
    87262F39 56573E60 21B4EFC4 458269AA

B : 9E013C8D 418AFC55 5661DB7C 14EC44B4 F9153201 88738392 96E98368 0975D8D1
    A5A8BAE5 40F67AC5 51541925 2CCCFE7B CD806D79 798D5C6C 9BCDF666 CA1F822E

C : 185313EA 0F470A20 31E0B3D0 40CD2940 F1031345 08AB9F39 4061597D 141CBFC6
    5E064C6B 6F4E1AF2 F8DD0AEF 9BC3588A E6792687 FB3107E7 89289CE4 090F0328

W : 00000002 00000000


block number = 2 : permutation (j = 1)

A : CE92474C 4BEC74B0 28DB9B47 8DB33C30 0DC11FA4 5B09B805 738552C8 07D8C9EC
    2DA81812 A9666E56 FD06BA39 000E0201

B : 8E853EFD 8409F2EE D62A2B37 85D0A413 207D83EE 467E968C 2F2A4317 ED1A4C5C
    7A3CCD78 35FF7EC5 758C56F2 2BD53F39 693E3AA8 57ECFF22 BBE141FA 6C18324E
```

```
C : 185313EA 0F470A20 31E0B3D0 40CD2940 F1031345 08AB9F39 4061597D 141CBFC6
    5E064C6B 6F4E1AF2 F8DD0AEF 9BC3588A E6792687 FB3107E7 89289CE4 090F0328

W : 00000002 00000000


block number = 2 : permutation (j = 2)

A : CB962C61 AAB5F2E1 BD48C6D2 7B006B1B BEC24D1C A7A67B33 B8CFC3C6 551FF849
    32227BC2 388DD364 E2ABB1A3 8FEAD9F3

B : 35DDA664 DC262415 6781BB5F 9B0AAE06 7492D442 D9B72006 1CE3BF03 5ECB0C5D
    B5442813 33A77946 AC2891DD FD4A79C4 1FA1F16D 68ABD2DF 6A96CDA9 A8254290

C : 185313EA 0F470A20 31E0B3D0 40CD2940 F1031345 08AB9F39 4061597D 141CBFC6
    5E064C6B 6F4E1AF2 F8DD0AEF 9BC3588A E6792687 FB3107E7 89289CE4 090F0328

W : 00000002 00000000


block number = 2 : permutation - add C to A

A : B135B153 9A854097 D06C7812 766B154C 1CBB394A 0F02EECD 40108811 C03F1085
    22CFBD52 6E10599B 55D673B5 5251DB43

B : 35DDA664 DC262415 6781BB5F 9B0AAE06 7492D442 D9B72006 1CE3BF03 5ECB0C5D
    B5442813 33A77946 AC2891DD FD4A79C4 1FA1F16D 68ABD2DF 6A96CDA9 A8254290

C : 185313EA 0F470A20 31E0B3D0 40CD2940 F1031345 08AB9F39 4061597D 141CBFC6
    5E064C6B 6F4E1AF2 F8DD0AEF 9BC3588A E6792687 FB3107E7 89289CE4 090F0328

W : 00000002 00000000


block number = 2 : subtract M from C

A : B135B153 9A854097 D06C7812 766B154C 1CBB394A 0F02EECD 40108811 C03F1085
    22CFBD52 6E10599B 55D673B5 5251DB43

B : 35DDA664 DC262415 6781BB5F 9B0AAE06 7492D442 D9B72006 1CE3BF03 5ECB0C5D
    B5442813 33A77946 AC2891DD FD4A79C4 1FA1F16D 68ABD2DF 6A96CDA9 A8254290

C : E621E3BD D911D5ED 04A77B99 DC69C6DF 889BACE0 9C4034D0 CFF1EB10 9FA94D55
    E58ED5F6 6ECDA079 F8DD0AEF 9BC3588A E6792687 FB3107E7 89289CE4 090F0328

W : 00000002 00000000


block number = 2 : swap B with C

A : B135B153 9A854097 D06C7812 766B154C 1CBB394A 0F02EECD 40108811 C03F1085
    22CFBD52 6E10599B 55D673B5 5251DB43

B : E621E3BD D911D5ED 04A77B99 DC69C6DF 889BACE0 9C4034D0 CFF1EB10 9FA94D55
    E58ED5F6 6ECDA079 F8DD0AEF 9BC3588A E6792687 FB3107E7 89289CE4 090F0328

C : 35DDA664 DC262415 6781BB5F 9B0AAE06 7492D442 D9B72006 1CE3BF03 5ECB0C5D
    B5442813 33A77946 AC2891DD FD4A79C4 1FA1F16D 68ABD2DF 6A96CDA9 A8254290

W : 00000002 00000000


n0_final = 0 : add M to B

A : B135B153 9A854097 D06C7812 766B154C 1CBB394A 0F02EECD 40108811 C03F1085
    22CFBD52 6E10599B 55D673B5 5251DB43
```

```
B : 185313EA 0F470A20 31E0B3D0 40CD2940 F1031345 08AB9F39 4061597D 141CBFC6
    5E064C6B 6F4E1AF2 F8DD0AEF 9BC3588A E6792687 FB3107E7 89289CE4 090F0328

C : 35DDA664 DC262415 6781BB5F 9B0AAE06 7492D442 D9B72006 1CE3BF03 5ECB0C5D
    B5442813 33A77946 AC2891DD FD4A79C4 1FA1F16D 68ABD2DF 6A96CDA9 A8254290

W : 00000002 00000000


n0_final = 0 : xor counter W into A

A : B135B151 9A854097 D06C7812 766B154C 1CBB394A 0F02EECD 40108811 C03F1085
    22CFBD52 6E10599B 55D673B5 5251DB43

B : 185313EA 0F470A20 31E0B3D0 40CD2940 F1031345 08AB9F39 4061597D 141CBFC6
    5E064C6B 6F4E1AF2 F8DD0AEF 9BC3588A E6792687 FB3107E7 89289CE4 090F0328

C : 35DDA664 DC262415 6781BB5F 9B0AAE06 7492D442 D9B72006 1CE3BF03 5ECB0C5D
    B5442813 33A77946 AC2891DD FD4A79C4 1FA1F16D 68ABD2DF 6A96CDA9 A8254290

W : 00000002 00000000


n0_final = 0 : permutation - rotate B

A : B135B151 9A854097 D06C7812 766B154C 1CBB394A 0F02EECD 40108811 C03F1085
    22CFBD52 6E10599B 55D673B5 5251DB43

B : 27D430A6 14401E8E 67A063C1 5280819A 268BE206 3E721157 B2FA80C2 7F8C2839
    98D6BC0C 35E4DE9C 15DFF1BA B1153786 4D0FCCF2 0FCFF662 39C91251 0650121E

C : 35DDA664 DC262415 6781BB5F 9B0AAE06 7492D442 D9B72006 1CE3BF03 5ECB0C5D
    B5442813 33A77946 AC2891DD FD4A79C4 1FA1F16D 68ABD2DF 6A96CDA9 A8254290

W : 00000002 00000000


n0_final = 0 : permutation (j = 0)

A : BA7F4347 4863AD24 0217DF60 0D93104E 2415EBE3 3762C22A 0E26C6D8 45A1EC10
    C78897F8 F58997A7 1C3FD0C5 06D77D1B

B : 69A07B7E A8B6F184 5EFF78F9 34F39108 96FDD010 B4791F7B 942C38A2 4546439D
    09DA101E 61BFD560 C87FCC4E 9B02EDE9 DF9F255C A803BE1F 8E7A043D FECCCB8D

C : 35DDA664 DC262415 6781BB5F 9B0AAE06 7492D442 D9B72006 1CE3BF03 5ECB0C5D
    B5442813 33A77946 AC2891DD FD4A79C4 1FA1F16D 68ABD2DF 6A96CDA9 A8254290

W : 00000002 00000000


n0_final = 0 : permutation (j = 1)

A : 9E3B324B 4BAF7BF8 8A1A81C7 10A836D8 3A6B0F26 2BED5D03 FFE01EEF F099D74B
    F33A942E DEA5482D C56A0714 6721E434

B : 7BA6E426 B2B967DB E0552F40 7A23F6F8 213ECBF0 49A88925 12CD89AE 12529CF1
    7270ED88 772F2EC7 E51AE6A5 D95212F4 7AAABA60 8415DEC3 1CEBE96B F2FFBFAF

C : 35DDA664 DC262415 6781BB5F 9B0AAE06 7492D442 D9B72006 1CE3BF03 5ECB0C5D
    B5442813 33A77946 AC2891DD FD4A79C4 1FA1F16D 68ABD2DF 6A96CDA9 A8254290

W : 00000002 00000000


n0_final = 0 : permutation (j = 2)
```

```
A : D5362471 109DB504 9A4A190F 81D6045F 155D65AF B057CF0A F8488BD1 3F7BFE60
    6F2DFF5B 00BE2E79 D45D77A9 6099636C

B : 4F7ED0DF 1DF96A3E DB74F48D 91434BB7 68B44C6E 7C3358B1 402EF5AC 5A8CC242
    0E434140 A1F66D7B CD82B965 72202476 65877464 F76A6C01 12755A80 7A99E3CC

C : 35DDA664 DC262415 6781BB5F 9B0AAE06 7492D442 D9B72006 1CE3BF03 5ECB0C5D
    B5442813 33A77946 AC2891DD FD4A79C4 1FA1F16D 68ABD2DF 6A96CDA9 A8254290

W : 00000002 00000000


n0_final = 0 : permutation - add C to A

A : 15B08ECB DAB02117 B8D33009 70D24C6A B75862A2 100C71C3 E1CD4932 700A0A9F
    664E3382 4A371C3B 4A67E3D4 943C81F5

B : 4F7ED0DF 1DF96A3E DB74F48D 91434BB7 68B44C6E 7C3358B1 402EF5AC 5A8CC242
    0E434140 A1F66D7B CD82B965 72202476 65877464 F76A6C01 12755A80 7A99E3CC

C : 35DDA664 DC262415 6781BB5F 9B0AAE06 7492D442 D9B72006 1CE3BF03 5ECB0C5D
    B5442813 33A77946 AC2891DD FD4A79C4 1FA1F16D 68ABD2DF 6A96CDA9 A8254290

W : 00000002 00000000


n0_final = 0 : subtract M from C

A : 15B08ECB DAB02117 B8D33009 70D24C6A B75862A2 100C71C3 E1CD4932 700A0A9F
    664E3382 4A371C3B 4A67E3D4 943C81F5

B : 4F7ED0DF 1DF96A3E DB74F48D 91434BB7 68B44C6E 7C3358B1 402EF5AC 5A8CC242
    0E434140 A1F66D7B CD82B965 72202476 65877464 F76A6C01 12755A80 7A99E3CC

C : 03AC7637 A5F0EFE2 3A488328 36A74BA5 0C2B6DDD 6D4BB59D AC745096 EA5799EC
    3CCCB19E 3326FECD AC2891DD FD4A79C4 1FA1F16D 68ABD2DF 6A96CDA9 A8254290

W : 00000002 00000000


n0_final = 0 : swap B with C

A : 15B08ECB DAB02117 B8D33009 70D24C6A B75862A2 100C71C3 E1CD4932 700A0A9F
    664E3382 4A371C3B 4A67E3D4 943C81F5

B : 03AC7637 A5F0EFE2 3A488328 36A74BA5 0C2B6DDD 6D4BB59D AC745096 EA5799EC
    3CCCB19E 3326FECD AC2891DD FD4A79C4 1FA1F16D 68ABD2DF 6A96CDA9 A8254290

C : 4F7ED0DF 1DF96A3E DB74F48D 91434BB7 68B44C6E 7C3358B1 402EF5AC 5A8CC242
    0E434140 A1F66D7B CD82B965 72202476 65877464 F76A6C01 12755A80 7A99E3CC

W : 00000002 00000000


n0_final = 1 : add M to B

A : 15B08ECB DAB02117 B8D33009 70D24C6A B75862A2 100C71C3 E1CD4932 700A0A9F
    664E3382 4A371C3B 4A67E3D4 943C81F5

B : 35DDA664 DC262415 6781BB5F 9B0AAE06 7492D442 D9B72006 1CE3BF03 5ECB0C5D
    B5442813 33A77946 AC2891DD FD4A79C4 1FA1F16D 68ABD2DF 6A96CDA9 A8254290

C : 4F7ED0DF 1DF96A3E DB74F48D 91434BB7 68B44C6E 7C3358B1 402EF5AC 5A8CC242
    0E434140 A1F66D7B CD82B965 72202476 65877464 F76A6C01 12755A80 7A99E3CC

W : 00000002 00000000
```

```
n0_final = 1 : xor counter W into A

A : 15B08EC9 DAB02117 B8D33009 70D24C6A B75862A2 100C71C3 E1CD4932 700A0A9F
    664E3382 4A371C3B 4A67E3D4 943C81F5

B : 35DDA664 DC262415 6781BB5F 9B0AAE06 7492D442 D9B72006 1CE3BF03 5ECB0C5D
    B5442813 33A77946 AC2891DD FD4A79C4 1FA1F16D 68ABD2DF 6A96CDA9 A8254290

C : 4F7ED0DF 1DF96A3E DB74F48D 91434BB7 68B44C6E 7C3358B1 402EF5AC 5A8CC242
    0E434140 A1F66D7B CD82B965 72202476 65877464 F76A6C01 12755A80 7A99E3CC

W : 00000002 00000000


n0_final = 1 : permutation - rotate B

A : 15B08EC9 DAB02117 B8D33009 70D24C6A B75862A2 100C71C3 E1CD4932 700A0A9F
    664E3382 4A371C3B 4A67E3D4 943C81F5

B : 4CC86BBB 482BB84C 76BECF03 5C0D3615 A884E925 400DB36E 7E0639C7 18BABD96
    50276A88 F28C674E 23BB5851 F389FA94 E2DA3F43 A5BED157 9B52D52D 8521504A

C : 4F7ED0DF 1DF96A3E DB74F48D 91434BB7 68B44C6E 7C3358B1 402EF5AC 5A8CC242
    0E434140 A1F66D7B CD82B965 72202476 65877464 F76A6C01 12755A80 7A99E3CC

W : 00000002 00000000


n0_final = 1 : permutation (j = 0)

A : E00AE65C 41E3AD21 962642C3 6DBBC826 7A0AE7C0 E561DCCB 104A79A2 F13D48ED
    BCD491BC 47007981 CA03134A DDCD05B0

B : 6C2A38A6 99125EC5 C7D291CC B2C855AA D4FCCA74 9A8545E8 13B9F5D3 3FB7CC3E
    E365BB53 5DE748E3 728A5C17 C5210F66 DA416724 F561F071 5F7C1767 9806974C

C : 4F7ED0DF 1DF96A3E DB74F48D 91434BB7 68B44C6E 7C3358B1 402EF5AC 5A8CC242
    0E434140 A1F66D7B CD82B965 72202476 65877464 F76A6C01 12755A80 7A99E3CC

W : 00000002 00000000


n0_final = 1 : permutation (j = 1)

A : 87492C4D 9D4CAD67 A7274C1F 73A47CAB 52B9484B EEF47C12 785C3BCA 6286CE3D
    E02D1461 C6E1C1C8 C2823D6C C7EA98EB

B : ACC058BC 9EF93721 9EF03CA8 EDED87AC B62B7F77 0C14B5E6 1A0E2935 477AFF68
    BE7DA515 D97DC35E BDCC0BCE 06199D99 19C479FD FBC8630E 395BEAFB AD741F5B

C : 4F7ED0DF 1DF96A3E DB74F48D 91434BB7 68B44C6E 7C3358B1 402EF5AC 5A8CC242
    0E434140 A1F66D7B CD82B965 72202476 65877464 F76A6C01 12755A80 7A99E3CC

W : 00000002 00000000


n0_final = 1 : permutation (j = 2)

A : 5A296E6B B453D5A8 1D363EEA 60DC971D 057EF2FC B7071E8D F33FE465 CA4C815C
    1E2E0224 59253DB5 9C2E569B D43BEEA3

B : 6712B5A5 3AFA6FFE 29704ABF 760A3998 C9806F7B 5385419B D6D5937F 11D69632
    867A4728 FA0367CF 77580C07 39804591 D2590E21 514A0457 11667C92 712C2FEB

C : 4F7ED0DF 1DF96A3E DB74F48D 91434BB7 68B44C6E 7C3358B1 402EF5AC 5A8CC242
    0E434140 A1F66D7B CD82B965 72202476 65877464 F76A6C01 12755A80 7A99E3CC
```

```
W : 00000002 00000000


n0_final = 1 : permutation - add C to A

A : D826C264 D20E6759 AECD6DDA 8EF5DBD6 6BE8E4C1 7D7D7D1A 2F6314CF B37324FA
    7C1E3493 35A43FC7 B1C288C8 F462F834

B : 6712B5A5 3AFA6FFE 29704ABF 760A3998 C9806F7B 5385419B D6D5937F 11D69632
    867A4728 FA0367CF 77580C07 39804591 D2590E21 514A0457 11667C92 712C2FEB

C : 4F7ED0DF 1DF96A3E DB74F48D 91434BB7 68B44C6E 7C3358B1 402EF5AC 5A8CC242
    0E434140 A1F66D7B CD82B965 72202476 65877464 F76A6C01 12755A80 7A99E3CC

W : 00000002 00000000


n0_final = 1 : subtract M from C

A : D826C264 D20E6759 AECD6DDA 8EF5DBD6 6BE8E4C1 7D7D7D1A 2F6314CF B37324FA
    7C1E3493 35A43FC7 B1C288C8 F462F834

B : 6712B5A5 3AFA6FFE 29704ABF 760A3998 C9806F7B 5385419B D6D5937F 11D69632
    867A4728 FA0367CF 77580C07 39804591 D2590E21 514A0457 11667C92 712C2FEB

C : 1D4DA0B2 E7C4360B AE3BBC56 2CDFE956 004CE609 0FC7EE48 CFBF873F E6194FD1
    95CBCACB A175F302 CD82B965 72202476 65877464 F76A6C01 12755A80 7A99E3CC

W : 00000002 00000000


n0_final = 1 : swap B with C

A : D826C264 D20E6759 AECD6DDA 8EF5DBD6 6BE8E4C1 7D7D7D1A 2F6314CF B37324FA
    7C1E3493 35A43FC7 B1C288C8 F462F834

B : 1D4DA0B2 E7C4360B AE3BBC56 2CDFE956 004CE609 0FC7EE48 CFBF873F E6194FD1
    95CBCACB A175F302 CD82B965 72202476 65877464 F76A6C01 12755A80 7A99E3CC

C : 6712B5A5 3AFA6FFE 29704ABF 760A3998 C9806F7B 5385419B D6D5937F 11D69632
    867A4728 FA0367CF 77580C07 39804591 D2590E21 514A0457 11667C92 712C2FEB

W : 00000002 00000000


n0_final = 2 : add M to B

A : D826C264 D20E6759 AECD6DDA 8EF5DBD6 6BE8E4C1 7D7D7D1A 2F6314CF B37324FA
    7C1E3493 35A43FC7 B1C288C8 F462F834

B : 4F7ED0DF 1DF96A3E DB74F48D 91434BB7 68B44C6E 7C3358B1 402EF5AC 5A8CC242
    0E434140 A1F66D7B CD82B965 72202476 65877464 F76A6C01 12755A80 7A99E3CC

C : 6712B5A5 3AFA6FFE 29704ABF 760A3998 C9806F7B 5385419B D6D5937F 11D69632
    867A4728 FA0367CF 77580C07 39804591 D2590E21 514A0457 11667C92 712C2FEB

W : 00000002 00000000


n0_final = 2 : xor counter W into A

A : D826C266 D20E6759 AECD6DDA 8EF5DBD6 6BE8E4C1 7D7D7D1A 2F6314CF B37324FA
    7C1E3493 35A43FC7 B1C288C8 F462F834

B : 4F7ED0DF 1DF96A3E DB74F48D 91434BB7 68B44C6E 7C3358B1 402EF5AC 5A8CC242
    0E434140 A1F66D7B CD82B965 72202476 65877464 F76A6C01 12755A80 7A99E3CC
```

```
C : 6712B5A5 3AFA6FFE 29704ABF 760A3998 C9806F7B 5385419B D6D5937F 11D69632
    867A4728 FA0367CF 77580C07 39804591 D2590E21 514A0457 11667C92 712C2FEB

W : 00000002 00000000


n0_final = 2 : permutation - rotate B

A : D826C266 D20E6759 AECD6DDA 8EF5DBD6 6BE8E4C1 7D7D7D1A 2F6314CF B37324FA
    7C1E3493 35A43FC7 B1C288C8 F462F834

B : A1BE9EFD D47C3BF2 E91BB6E9 976F2286 98DCD168 B162F866 EB58805D 8484B519
    82801C86 DAF743EC 72CB9B05 48ECE440 E8C8CB0E D803EED4 B50024EA C798F533

C : 6712B5A5 3AFA6FFE 29704ABF 760A3998 C9806F7B 5385419B D6D5937F 11D69632
    867A4728 FA0367CF 77580C07 39804591 D2590E21 514A0457 11667C92 712C2FEB

W : 00000002 00000000


n0_final = 2 : permutation (j = 0)

A : E8944762 7BE8E366 597E5149 BE805B7B A478B79D D5559665 BD70C24D 5C30AF3E
    7DB340CD AF64BBCF F4EF6D81 399AAEBF

B : DE9DE86C DBED9EA7 4D46E3AC 1FC6F977 6A3EEAB3 486F9957 943E3D09 AAC63AF2
    874C863F E575C3E9 EE87A474 57BC99C0 C6FA2E80 3410C130 CC81E763 CE4E4EE3

C : 6712B5A5 3AFA6FFE 29704ABF 760A3998 C9806F7B 5385419B D6D5937F 11D69632
    867A4728 FA0367CF 77580C07 39804591 D2590E21 514A0457 11667C92 712C2FEB

W : 00000002 00000000


n0_final = 2 : permutation (j = 1)

A : 3EA1B235 DD7CC0AD 6AEBF1AA 542AD5EB 565684BF F19959DE 6D3D928A F45B85F1
    AD5D43E8 94694935 8FBADF63 34677334

B : 71831A04 B792148C A202DCEC 48018DC3 86DF6971 FB498464 58395A8F 9E14F92E
    CFC741B5 E868B881 481B46BC 04AC1994 245D2641 66472441 0BC1A3B2 9738E7C9

C : 6712B5A5 3AFA6FFE 29704ABF 760A3998 C9806F7B 5385419B D6D5937F 11D69632
    867A4728 FA0367CF 77580C07 39804591 D2590E21 514A0457 11667C92 712C2FEB

W : 00000002 00000000


n0_final = 2 : permutation (j = 2)

A : D922DEAA 2DCAD210 A701A6BF AC069AC0 B6099E6C 067D8D76 16677DEE 8032A1F5
    03B85B77 04743B88 BD2AAF8A C4F50011

B : 593CCDF8 F2E993B0 DD79ADFB A855551E 2B63F3B6 24A62526 E88CEC5E 6FD09762
    D678E2F8 2953038A 79AE0F69 76956D22 B4FDE80A 37058CF5 55561711 157B307D

C : 6712B5A5 3AFA6FFE 29704ABF 760A3998 C9806F7B 5385419B D6D5937F 11D69632
    867A4728 FA0367CF 77580C07 39804591 D2590E21 514A0457 11667C92 712C2FEB

W : 00000002 00000000


n0_final = 2 : permutation - add C to A

A : F9D98DBE 30B70551 86CB5CAF BDB2F590 AF169E21 BD8AF9BE 9EEA9756 F7D08C3A
    C51970D2 26C8004C 5BFD5D4B 24891C29
```

```
B : 593CCDF8 F2E993B0 DD79ADFB A855551E 2B63F3B6 24A62526 E88CEC5E 6FD09762
    D678E2F8 2953038A 79AE0F69 76956D22 B4FDE80A 37058CF5 55561711 157B307D

C : 6712B5A5 3AFA6FFE 29704ABF 760A3998 C9806F7B 5385419B D6D5937F 11D69632
    867A4728 FA0367CF 77580C07 39804591 D2590E21 514A0457 11667C92 712C2FEB

W : 00000002 00000000


n0_final = 2 : subtract M from C

A : F9D98DBE 30B70551 86CB5CAF BDB2F590 AF169E21 BD8AF9BE 9EEA9756 F7D08C3A
    C51970D2 26C8004C 5BFD5D4B 24891C29

B : 593CCDF8 F2E993B0 DD79ADFB A855551E 2B63F3B6 24A62526 E88CEC5E 6FD09762
    D678E2F8 2953038A 79AE0F69 76956D22 B4FDE80A 37058CF5 55561711 157B307D

C : 34E18578 04C53BCB FC371288 11A6D737 61190916 E719D732 66662512 9D6323C1
    0E02D0B3 F982ED56 77580C07 39804591 D2590E21 514A0457 11667C92 712C2FEB

W : 00000002 00000000


n0_final = 2 : swap B with C  (final state)

A : F9D98DBE 30B70551 86CB5CAF BDB2F590 AF169E21 BD8AF9BE 9EEA9756 F7D08C3A
    C51970D2 26C8004C 5BFD5D4B 24891C29

B : 34E18578 04C53BCB FC371288 11A6D737 61190916 E719D732 66662512 9D6323C1
    0E02D0B3 F982ED56 77580C07 39804591 D2590E21 514A0457 11667C92 712C2FEB

C : 593CCDF8 F2E993B0 DD79ADFB A855551E 2B63F3B6 24A62526 E88CEC5E 6FD09762
    D678E2F8 2953038A 79AE0F69 76956D22 B4FDE80A 37058CF5 55561711 157B307D

W : 00000002 00000000


Hash value :

H : 79AE0F69 76956D22 B4FDE80A 37058CF5 55561711 157B307D


Hash value (byte array):

H : 69 0F AE 79 22 6D 95 76 0A E8 FD B4 F5 8C 05 37
    11 17 56 55 7D 30 7B 15
```

B.3 Intermediate States for Shabal-224 (Message A)

```
init

A : 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
    00000000 00000000 00000000 00000000

B : 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
    00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000

C : 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
    00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000

W : FFFFFFFF FFFFFFFF
```

```
block number = -1 : message block

M : 000000E0 000000E1 000000E2 000000E3 000000E4 000000E5 000000E6 000000E7
    000000E8 000000E9 000000EA 000000EB 000000EC 000000ED 000000EE 000000EF


block number = -1 : add M to B

A : 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
    00000000 00000000 00000000 00000000

B : 000000E0 000000E1 000000E2 000000E3 000000E4 000000E5 000000E6 000000E7
    000000E8 000000E9 000000EA 000000EB 000000EC 000000ED 000000EE 000000EF

C : 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
    00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000

W : FFFFFFFF FFFFFFFF


block number = -1 : xor counter W into A

A : FFFFFFFF FFFFFFFF 00000000 00000000 00000000 00000000 00000000 00000000
    00000000 00000000 00000000 00000000

B : 000000E0 000000E1 000000E2 000000E3 000000E4 000000E5 000000E6 000000E7
    000000E8 000000E9 000000EA 000000EB 000000EC 000000ED 000000EE 000000EF

C : 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
    00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000

W : FFFFFFFF FFFFFFFF


block number = -1 : permutation - rotate B

A : FFFFFFFF FFFFFFFF 00000000 00000000 00000000 00000000 00000000 00000000
    00000000 00000000 00000000 00000000

B : 01C00000 01C20000 01C40000 01C60000 01C80000 01CA0000 01CC0000 01CE0000
    01D00000 01D20000 01D40000 01D60000 01D80000 01DA0000 01DC0000 01DE0000

C : 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
    00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000

W : FFFFFFFF FFFFFFFF


block number = -1 : permutation (j = 0)

A : E0F553CA 360FB9F0 13F563A2 1751BF18 97E7BD38 130C3A22 98E64161 83D3B87F
    F2FB6BE4 D594E9DE 8CA0C9A1 E61F0AC1

B : 024800E2 FB28F276 182C49FA 2B1FCF5D 6B8842C7 EF67C5DD 6481BE9E 7FB04780
    0EA4941B 29CF1621 70F7365E 1A4CF53E 1CBAAC35 CA44460F EFB29C5D EB1240E7

C : 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
    00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000

W : FFFFFFFF FFFFFFFF


block number = -1 : permutation (j = 1)

A : 257C1CF3 FAC54BC3 B544997B B9426917 594FC7A1 D13345E4 17044F51 13DC1230
    4D00378B E315C946 A6A4D823 AE0E1AAC
```

```
B : 67772EA5 A9D983AA 1B7E3DF8 B711623E 65EF4DFA C225BD02 90585AE0 AE916A53
    C7CACB3A 56A4987E AB550A38 72247C94 9FC56034 BA443604 379E8815 3A076C00

C : 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
    00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000

W : FFFFFFFF FFFFFFFF


block number = -1 : permutation (j = 2)

A : 01340BD4 73381D1D F97F7508 28F07BF4 8E842C24 D8A8596F D2D4F99A D0763C3C
    632406CB 965BA4DD 014EB6D9 3E7F3B64

B : EC9906C8 F21851C0 C0A747BA 21DAD58B 35156FDF 088C98E7 26303F36 8A2D50AC
    FEEE45AE 8A1E966C 7B811214 CBC13AEB A351395D 1D2C372B 918C590C B58E1C9B

C : 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
    00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000

W : FFFFFFFF FFFFFFFF


block number = -1 : permutation - add C to A

A : 01340BD4 73381D1D F97F7508 28F07BF4 8E842C24 D8A8596F D2D4F99A D0763C3C
    632406CB 965BA4DD 014EB6D9 3E7F3B64

B : EC9906C8 F21851C0 C0A747BA 21DAD58B 35156FDF 088C98E7 26303F36 8A2D50AC
    FEEE45AE 8A1E966C 7B811214 CBC13AEB A351395D 1D2C372B 918C590C B58E1C9B

C : 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
    00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000

W : FFFFFFFF FFFFFFFF


block number = -1 : subtract M from C

A : 01340BD4 73381D1D F97F7508 28F07BF4 8E842C24 D8A8596F D2D4F99A D0763C3C
    632406CB 965BA4DD 014EB6D9 3E7F3B64

B : EC9906C8 F21851C0 C0A747BA 21DAD58B 35156FDF 088C98E7 26303F36 8A2D50AC
    FEEE45AE 8A1E966C 7B811214 CBC13AEB A351395D 1D2C372B 918C590C B58E1C9B

C : FFFFFF20 FFFFFF1F FFFFFF1E FFFFFF1D FFFFFF1C FFFFFF1B FFFFFF1A FFFFFF19
    FFFFFF18 FFFFFF17 FFFFFF16 FFFFFF15 FFFFFF14 FFFFFF13 FFFFFF12 FFFFFF11

W : FFFFFFFF FFFFFFFF


block number = -1 : swap B with C

A : 01340BD4 73381D1D F97F7508 28F07BF4 8E842C24 D8A8596F D2D4F99A D0763C3C
    632406CB 965BA4DD 014EB6D9 3E7F3B64

B : FFFFFF20 FFFFFF1F FFFFFF1E FFFFFF1D FFFFFF1C FFFFFF1B FFFFFF1A FFFFFF19
    FFFFFF18 FFFFFF17 FFFFFF16 FFFFFF15 FFFFFF14 FFFFFF13 FFFFFF12 FFFFFF11

C : EC9906C8 F21851C0 C0A747BA 21DAD58B 35156FDF 088C98E7 26303F36 8A2D50AC
    FEEE45AE 8A1E966C 7B811214 CBC13AEB A351395D 1D2C372B 918C590C B58E1C9B

W : FFFFFFFF FFFFFFFF


block number = 0 : increment counter W
```

```
A : 01340BD4 73381D1D F97F7508 28F07BF4 8E842C24 D8A8596F D2D4F99A D0763C3C
    632406CB 965BA4DD 014EB6D9 3E7F3B64

B : FFFFFF20 FFFFFF1F FFFFFF1E FFFFFF1D FFFFFF1C FFFFFF1B FFFFFF1A FFFFFF19
    FFFFFF18 FFFFFF17 FFFFFF16 FFFFFF15 FFFFFF14 FFFFFF13 FFFFFF12 FFFFFF11

C : EC9906C8 F21851C0 C0A747BA 21DAD58B 35156FDF 088C98E7 26303F36 8A2D50AC
    FEEE45AE 8A1E966C 7B811214 CBC13AEB A351395D 1D2C372B 918C590C B58E1C9B

W : 00000000 00000000


block number = 0 : message block

M : 000000F0 000000F1 000000F2 000000F3 000000F4 000000F5 000000F6 000000F7
    000000F8 000000F9 000000FA 000000FB 000000FC 000000FD 000000FE 000000FF


block number = 0 : add M to B

A : 01340BD4 73381D1D F97F7508 28F07BF4 8E842C24 D8A8596F D2D4F99A D0763C3C
    632406CB 965BA4DD 014EB6D9 3E7F3B64

B : 00000010 00000010 00000010 00000010 00000010 00000010 00000010 00000010
    00000010 00000010 00000010 00000010 00000010 00000010 00000010 00000010

C : EC9906C8 F21851C0 C0A747BA 21DAD58B 35156FDF 088C98E7 26303F36 8A2D50AC
    FEEE45AE 8A1E966C 7B811214 CBC13AEB A351395D 1D2C372B 918C590C B58E1C9B

W : 00000000 00000000


block number = 0 : xor counter W into A

A : 01340BD4 73381D1D F97F7508 28F07BF4 8E842C24 D8A8596F D2D4F99A D0763C3C
    632406CB 965BA4DD 014EB6D9 3E7F3B64

B : 00000010 00000010 00000010 00000010 00000010 00000010 00000010 00000010
    00000010 00000010 00000010 00000010 00000010 00000010 00000010 00000010

C : EC9906C8 F21851C0 C0A747BA 21DAD58B 35156FDF 088C98E7 26303F36 8A2D50AC
    FEEE45AE 8A1E966C 7B811214 CBC13AEB A351395D 1D2C372B 918C590C B58E1C9B

W : 00000000 00000000


block number = 0 : permutation - rotate B

A : 01340BD4 73381D1D F97F7508 28F07BF4 8E842C24 D8A8596F D2D4F99A D0763C3C
    632406CB 965BA4DD 014EB6D9 3E7F3B64

B : 00200000 00200000 00200000 00200000 00200000 00200000 00200000 00200000
    00200000 00200000 00200000 00200000 00200000 00200000 00200000 00200000

C : EC9906C8 F21851C0 C0A747BA 21DAD58B 35156FDF 088C98E7 26303F36 8A2D50AC
    FEEE45AE 8A1E966C 7B811214 CBC13AEB A351395D 1D2C372B 918C590C B58E1C9B

W : 00000000 00000000


block number = 0 : permutation (j = 0)

A : 4389B954 7C2E65DF 7AC0A60E 4D3E5671 7E2ECC0C 30113495 2288EF2A DA590830
    F8F8D0FD 866100F7 129EA35A B74146B4

B : 3D7D89CC 64237EF2 265812A5 119D3DC4 819133F3 CFAECB6A DD3710D5 25E6F7CF
    07472F02 79DEFF08 ED215CA5 48FEB94B BC3646AB 83919A20 857F59F1 B281A98E
```

```
C : EC9906C8 F21851C0 C0A747BA 21DAD58B 35156FDF 088C98E7 26303F36 8A2D50AC
    FEEE45AE 8A1E966C 7B811214 CBC13AEB A351395D 1D2C372B 918C590C B58E1C9B

W : 00000000 00000000


block number = 0 : permutation (j = 1)

A : E11D69E6 69E4164B 0FC5373C 7E495272 A8B9B043 8CDC5330 90551180 5E5046D9
    9EB4F2BE E6580BF7 E5386C60 AD2654AD

B : 2CCEC394 1350B7FB 8F6F4688 88CED426 62696AA6 86FA62DD A0A9B234 191444CC
    106CC81D 65A617A4 2A787188 104BDF1B 2F2AC2EB 7400988E 65545D9C C4ACEA3B

C : EC9906C8 F21851C0 C0A747BA 21DAD58B 35156FDF 088C98E7 26303F36 8A2D50AC
    FEEE45AE 8A1E966C 7B811214 CBC13AEB A351395D 1D2C372B 918C590C B58E1C9B

W : 00000000 00000000


block number = 0 : permutation (j = 2)

A : 01F5E756 E4B92946 BCFDB7C5 EFD3BD7F 466630E8 D1059816 DBA401AC 7E64640B
    B75BD62C 2F4525B5 AE5698A7 C7AF460B

B : E4B573A1 4C1A0880 1E907C51 04807EFD 3AD8CDE5 16B21302 02512C53 2204CB18
    99405F2D E5B648A1 70AB1D43 A10C25C2 16F1AC05 38BBEB56 9B01DC60 B1096D83

C : EC9906C8 F21851C0 C0A747BA 21DAD58B 35156FDF 088C98E7 26303F36 8A2D50AC
    FEEE45AE 8A1E966C 7B811214 CBC13AEB A351395D 1D2C372B 918C590C B58E1C9B

W : 00000000 00000000


block number = 0 : permutation - add C to A

A : A5201467 A9B8D94A D4CED997 68379D7B A7FC73BA F1A2546B 606782BF E0BCFD0F
    2F25374E 069A149F 5E2DFF25 FAECF061

B : E4B573A1 4C1A0880 1E907C51 04807EFD 3AD8CDE5 16B21302 02512C53 2204CB18
    99405F2D E5B648A1 70AB1D43 A10C25C2 16F1AC05 38BBEB56 9B01DC60 B1096D83

C : EC9906C8 F21851C0 C0A747BA 21DAD58B 35156FDF 088C98E7 26303F36 8A2D50AC
    FEEE45AE 8A1E966C 7B811214 CBC13AEB A351395D 1D2C372B 918C590C B58E1C9B

W : 00000000 00000000


block number = 0 : subtract M from C

A : A5201467 A9B8D94A D4CED997 68379D7B A7FC73BA F1A2546B 606782BF E0BCFD0F
    2F25374E 069A149F 5E2DFF25 FAECF061

B : E4B573A1 4C1A0880 1E907C51 04807EFD 3AD8CDE5 16B21302 02512C53 2204CB18
    99405F2D E5B648A1 70AB1D43 A10C25C2 16F1AC05 38BBEB56 9B01DC60 B1096D83

C : EC9905D8 F21850CF C0A746C8 21DAD498 35156EEB 088C97F2 26303E40 8A2D4FB5
    FEEE44B6 8A1E9573 7B81111A CBC139F0 A3513861 1D2C362E 918C580E B58E1B9C

W : 00000000 00000000


block number = 0 : swap B with C

A : A5201467 A9B8D94A D4CED997 68379D7B A7FC73BA F1A2546B 606782BF E0BCFD0F
    2F25374E 069A149F 5E2DFF25 FAECF061
```

```
B : EC9905D8 F21850CF C0A746C8 21DAD498 35156EEB 088C97F2 26303E40 8A2D4FB5
    FEEE44B6 8A1E9573 7B81111A CBC139F0 A3513861 1D2C362E 918C580E B58E1B9C

C : E4B573A1 4C1A0880 1E907C51 04807EFD 3AD8CDE5 16B21302 02512C53 2204CB18
    99405F2D E5B648A1 70AB1D43 A10C25C2 16F1AC05 38BBEB56 9B01DC60 B1096D83

W : 00000000 00000000


block number = 1 : increment counter W

A : A5201467 A9B8D94A D4CED997 68379D7B A7FC73BA F1A2546B 606782BF E0BCFD0F
    2F25374E 069A149F 5E2DFF25 FAECF061

B : EC9905D8 F21850CF C0A746C8 21DAD498 35156EEB 088C97F2 26303E40 8A2D4FB5
    FEEE44B6 8A1E9573 7B81111A CBC139F0 A3513861 1D2C362E 918C580E B58E1B9C

C : E4B573A1 4C1A0880 1E907C51 04807EFD 3AD8CDE5 16B21302 02512C53 2204CB18
    99405F2D E5B648A1 70AB1D43 A10C25C2 16F1AC05 38BBEB56 9B01DC60 B1096D83

W : 00000001 00000000


block number = 1 : message block

M : 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
    00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000


block number = 1 : add M to B

A : A5201467 A9B8D94A D4CED997 68379D7B A7FC73BA F1A2546B 606782BF E0BCFD0F
    2F25374E 069A149F 5E2DFF25 FAECF061

B : EC9905D8 F21850CF C0A746C8 21DAD498 35156EEB 088C97F2 26303E40 8A2D4FB5
    FEEE44B6 8A1E9573 7B81111A CBC139F0 A3513861 1D2C362E 918C580E B58E1B9C

C : E4B573A1 4C1A0880 1E907C51 04807EFD 3AD8CDE5 16B21302 02512C53 2204CB18
    99405F2D E5B648A1 70AB1D43 A10C25C2 16F1AC05 38BBEB56 9B01DC60 B1096D83

W : 00000001 00000000


block number = 1 : xor counter W into A

A : A5201466 A9B8D94A D4CED997 68379D7B A7FC73BA F1A2546B 606782BF E0BCFD0F
    2F25374E 069A149F 5E2DFF25 FAECF061

B : EC9905D8 F21850CF C0A746C8 21DAD498 35156EEB 088C97F2 26303E40 8A2D4FB5
    FEEE44B6 8A1E9573 7B81111A CBC139F0 A3513861 1D2C362E 918C580E B58E1B9C

C : E4B573A1 4C1A0880 1E907C51 04807EFD 3AD8CDE5 16B21302 02512C53 2204CB18
    99405F2D E5B648A1 70AB1D43 A10C25C2 16F1AC05 38BBEB56 9B01DC60 B1096D83

W : 00000001 00000000


block number = 1 : permutation - rotate B

A : A5201466 A9B8D94A D4CED997 68379D7B A7FC73BA F1A2546B 606782BF E0BCFD0F
    2F25374E 069A149F 5E2DFF25 FAECF061

B : 0BB1D932 A19FE430 8D91814E A93043B5 DDD66A2A 2FE41119 7C804C60 9F6B145A
    896DFDDC 2AE7143D 2234F702 73E19782 70C346A2 6C5C3A58 B01D2318 37396B1C

C : E4B573A1 4C1A0880 1E907C51 04807EFD 3AD8CDE5 16B21302 02512C53 2204CB18
```

```
    99405F2D E5B648A1 70AB1D43 A10C25C2 16F1AC05 38BBEB56 9B01DC60 B1096D83

W : 00000001 00000000


block number = 1 : permutation (j = 0)

A : 1117E7A7 33F2EAC4 1C1B8823 477C7385 5A5AEA53 94ED38E9 CBE63E23 3AF61CE8
    B964BBFA EA3AEAAE 41D58BD0 13DE80E2

B : AB194FD1 06D8D84E 2227FA94 9ECE08DF 1E09C1F9 34DAE524 CD19591C FBDFCBA2
    5440BFBC 400B3D2B FA439A2B 0BE25019 0F6E951C 14B5618B 83DE31ED D6F15A42

C : E4B573A1 4C1A0880 1E907C51 04807EFD 3AD8CDE5 16B21302 02512C53 2204CB18
    99405F2D E5B648A1 70AB1D43 A10C25C2 16F1AC05 38BBEB56 9B01DC60 B1096D83

W : 00000001 00000000


block number = 1 : permutation (j = 1)

A : DF76C800 CA73BBD4 CF3287F9 0E9B0C4F 045BF121 196B8245 19357752 9C1517BB
    05DAAF5F C09AD7FF C350AF3F 79EDE108

B : 16FCB2AC DC80E49C 4A03A6CD DDEE277D C636D352 56D0E248 A69DE2F9 71AD89B2
    88084887 B59A3E7D C44A4C51 E6A05382 E57924E6 CFFEBEAC E176EB76 CE085CC1

C : E4B573A1 4C1A0880 1E907C51 04807EFD 3AD8CDE5 16B21302 02512C53 2204CB18
    99405F2D E5B648A1 70AB1D43 A10C25C2 16F1AC05 38BBEB56 9B01DC60 B1096D83

W : 00000001 00000000


block number = 1 : permutation (j = 2)

A : 0E6F0B4C 6559A807 5DF26C97 A40E5863 8B485A08 2337EE51 C933CD67 1FCBF8D8
    60869449 025AA346 F0381F84 180502C2

B : 7B95DACF 6919A8EF 9C66A3DD B6E3A04A 7DFD5216 37079368 EF36569B B8AAB4F8
    64A734F8 B7FC6D55 BE58AA3B 2D74A022 558B227B 625821E0 CD2A3696 7BEA44BE

C : E4B573A1 4C1A0880 1E907C51 04807EFD 3AD8CDE5 16B21302 02512C53 2204CB18
    99405F2D E5B648A1 70AB1D43 A10C25C2 16F1AC05 38BBEB56 9B01DC60 B1096D83

W : 00000001 00000000


block number = 1 : permutation - add C to A

A : 65051D8E 9BD99592 F97A736F 5FF1DD67 62D711A0 DC068F04 11B6318A B158BEBF
    28180420 ED657C5D 255C667D 260328B8

B : 7B95DACF 6919A8EF 9C66A3DD B6E3A04A 7DFD5216 37079368 EF36569B B8AAB4F8
    64A734F8 B7FC6D55 BE58AA3B 2D74A022 558B227B 625821E0 CD2A3696 7BEA44BE

C : E4B573A1 4C1A0880 1E907C51 04807EFD 3AD8CDE5 16B21302 02512C53 2204CB18
    99405F2D E5B648A1 70AB1D43 A10C25C2 16F1AC05 38BBEB56 9B01DC60 B1096D83

W : 00000001 00000000


block number = 1 : subtract M from C

A : 65051D8E 9BD99592 F97A736F 5FF1DD67 62D711A0 DC068F04 11B6318A B158BEBF
    28180420 ED657C5D 255C667D 260328B8

B : 7B95DACF 6919A8EF 9C66A3DD B6E3A04A 7DFD5216 37079368 EF36569B B8AAB4F8
```

```
    64A734F8 B7FC6D55 BE58AA3B 2D74A022 558B227B 625821E0 CD2A3696 7BEA44BE

C : E4B573A1 4C1A0880 1E907C51 04807EFD 3AD8CDE5 16B21302 02512C53 2204CB18
    99405F2D E5B648A1 70AB1D43 A10C25C2 16F1AC05 38BBEB56 9B01DC60 B1096D83

W : 00000001 00000000


block number = 1 : swap B with C

A : 65051D8E 9BD99592 F97A736F 5FF1DD67 62D711A0 DC068F04 11B6318A B158BEBF
    28180420 ED657C5D 255C667D 260328B8

B : E4B573A1 4C1A0880 1E907C51 04807EFD 3AD8CDE5 16B21302 02512C53 2204CB18
    99405F2D E5B648A1 70AB1D43 A10C25C2 16F1AC05 38BBEB56 9B01DC60 B1096D83

C : 7B95DACF 6919A8EF 9C66A3DD B6E3A04A 7DFD5216 37079368 EF36569B B8AAB4F8
    64A734F8 B7FC6D55 BE58AA3B 2D74A022 558B227B 625821E0 CD2A3696 7BEA44BE

W : 00000001 00000000


block number = 2 : increment counter W

A : 65051D8E 9BD99592 F97A736F 5FF1DD67 62D711A0 DC068F04 11B6318A B158BEBF
    28180420 ED657C5D 255C667D 260328B8

B : E4B573A1 4C1A0880 1E907C51 04807EFD 3AD8CDE5 16B21302 02512C53 2204CB18
    99405F2D E5B648A1 70AB1D43 A10C25C2 16F1AC05 38BBEB56 9B01DC60 B1096D83

C : 7B95DACF 6919A8EF 9C66A3DD B6E3A04A 7DFD5216 37079368 EF36569B B8AAB4F8
    64A734F8 B7FC6D55 BE58AA3B 2D74A022 558B227B 625821E0 CD2A3696 7BEA44BE

W : 00000002 00000000


block number = 2 : message block

M : 00000080 00000000 00000000 00000000 00000000 00000000 00000000 00000000
    00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000


block number = 2 : add M to B

A : 65051D8E 9BD99592 F97A736F 5FF1DD67 62D711A0 DC068F04 11B6318A B158BEBF
    28180420 ED657C5D 255C667D 260328B8

B : E4B57421 4C1A0880 1E907C51 04807EFD 3AD8CDE5 16B21302 02512C53 2204CB18
    99405F2D E5B648A1 70AB1D43 A10C25C2 16F1AC05 38BBEB56 9B01DC60 B1096D83

C : 7B95DACF 6919A8EF 9C66A3DD B6E3A04A 7DFD5216 37079368 EF36569B B8AAB4F8
    64A734F8 B7FC6D55 BE58AA3B 2D74A022 558B227B 625821E0 CD2A3696 7BEA44BE

W : 00000002 00000000


block number = 2 : xor counter W into A

A : 65051D8C 9BD99592 F97A736F 5FF1DD67 62D711A0 DC068F04 11B6318A B158BEBF
    28180420 ED657C5D 255C667D 260328B8

B : E4B57421 4C1A0880 1E907C51 04807EFD 3AD8CDE5 16B21302 02512C53 2204CB18
    99405F2D E5B648A1 70AB1D43 A10C25C2 16F1AC05 38BBEB56 9B01DC60 B1096D83

C : 7B95DACF 6919A8EF 9C66A3DD B6E3A04A 7DFD5216 37079368 EF36569B B8AAB4F8
    64A734F8 B7FC6D55 BE58AA3B 2D74A022 558B227B 625821E0 CD2A3696 7BEA44BE
```

```
W : 00000002 00000000


block number = 2 : permutation - rotate B

A : 65051D8C 9BD99592 F97A736F 5FF1DD67 62D711A0 DC068F04 11B6318A B158BEBF
    28180420 ED657C5D 255C667D 260328B8

B : E843C96A 11009834 F8A23D20 FDFA0900 9BCA75B1 26042D64 58A604A2 96304409
    BE5B3280 9143CB6C 3A86E156 4B854218 580A2DE3 D6AC7177 B8C13603 DB076212

C : 7B95DACF 6919A8EF 9C66A3DD B6E3A04A 7DFD5216 37079368 EF36569B B8AAB4F8
    64A734F8 B7FC6D55 BE58AA3B 2D74A022 558B227B 625821E0 CD2A3696 7BEA44BE

W : 00000002 00000000


block number = 2 : permutation (j = 0)

A : 4391A10E 3AE3E7C0 75D88812 B1835B7A 56932F34 B98C9226 7432E079 23E6A3AF
    2C75A3FD 909E90CD 773EF253 7BC6C86A

B : D5DEB4C2 46A6A31D DF7B434C 5526BAF2 9EF83BA8 0A7B3711 3A8116C2 F079D443
    AF3C3903 4DE6F9EB FDCCCF00 1333B3A5 0C7A0537 6844FAD0 FBA51BEA F87260A0

C : 7B95DACF 6919A8EF 9C66A3DD B6E3A04A 7DFD5216 37079368 EF36569B B8AAB4F8
    64A734F8 B7FC6D55 BE58AA3B 2D74A022 558B227B 625821E0 CD2A3696 7BEA44BE

W : 00000002 00000000


block number = 2 : permutation (j = 1)

A : ECC02E5D 5CD02843 23E21DC1 58EFEA08 76644896 4A4B0D3E DDB4188B 843C5EAC
    1DC12C83 938542E7 4E0625FA 29F64BBB

B : 7AF36518 276A5E1D 20B1BA18 DA49A6B6 DFCEA42D 788CD33A C4FBF781 36FA1CC3
    4D47A3A5 38E2246A 27847C3F 817772BD 916FBD07 653D0761 D501D0A1 8B276012

C : 7B95DACF 6919A8EF 9C66A3DD B6E3A04A 7DFD5216 37079368 EF36569B B8AAB4F8
    64A734F8 B7FC6D55 BE58AA3B 2D74A022 558B227B 625821E0 CD2A3696 7BEA44BE

W : 00000002 00000000


block number = 2 : permutation (j = 2)

A : 2822A50E 3FC020F8 775F71F0 FD8CE38C A40D9ABE 9B7E1E3A 5E22CF13 0A410C42
    219BFCC7 3AD12AC0 57D3B6A3 3E2D0EA5

B : 967FCA83 9A1FA1F8 087265EC B4E75594 684012AA 31267973 0157610C 6F8725F5
    C17D220B 1545A911 EED5C892 F75016C6 FCBB7937 0F54DBFD 022FE81F D79C317F

C : 7B95DACF 6919A8EF 9C66A3DD B6E3A04A 7DFD5216 37079368 EF36569B B8AAB4F8
    64A734F8 B7FC6D55 BE58AA3B 2D74A022 558B227B 625821E0 CD2A3696 7BEA44BE

W : 00000002 00000000


block number = 2 : permutation - add C to A

A : 88652A38 8EDE7058 79D8D027 5654149A 8F8634BE F9B88017 B64078BF 5436B0F5
    BE9EF22B 7300D449 A92FD940 B8E64611

B : 967FCA83 9A1FA1F8 087265EC B4E75594 684012AA 31267973 0157610C 6F8725F5
    C17D220B 1545A911 EED5C892 F75016C6 FCBB7937 0F54DBFD 022FE81F D79C317F
```

```
C : 7B95DACF 6919A8EF 9C66A3DD B6E3A04A 7DFD5216 37079368 EF36569B B8AAB4F8
    64A734F8 B7FC6D55 BE58AA3B 2D74A022 558B227B 625821E0 CD2A3696 7BEA44BE

W : 00000002 00000000


block number = 2 : subtract M from C

A : 88652A38 8EDE7058 79D8D027 5654149A 8F8634BE F9B88017 B64078BF 5436B0F5
    BE9EF22B 7300D449 A92FD940 B8E64611

B : 967FCA83 9A1FA1F8 087265EC B4E75594 684012AA 31267973 0157610C 6F8725F5
    C17D220B 1545A911 EED5C892 F75016C6 FCBB7937 0F54DBFD 022FE81F D79C317F

C : 7B95DA4F 6919A8EF 9C66A3DD B6E3A04A 7DFD5216 37079368 EF36569B B8AAB4F8
    64A734F8 B7FC6D55 BE58AA3B 2D74A022 558B227B 625821E0 CD2A3696 7BEA44BE

W : 00000002 00000000


block number = 2 : swap B with C

A : 88652A38 8EDE7058 79D8D027 5654149A 8F8634BE F9B88017 B64078BF 5436B0F5
    BE9EF22B 7300D449 A92FD940 B8E64611

B : 7B95DA4F 6919A8EF 9C66A3DD B6E3A04A 7DFD5216 37079368 EF36569B B8AAB4F8
    64A734F8 B7FC6D55 BE58AA3B 2D74A022 558B227B 625821E0 CD2A3696 7BEA44BE

C : 967FCA83 9A1FA1F8 087265EC B4E75594 684012AA 31267973 0157610C 6F8725F5
    C17D220B 1545A911 EED5C892 F75016C6 FCBB7937 0F54DBFD 022FE81F D79C317F

W : 00000002 00000000


n0_final = 0 : add M to B

A : 88652A38 8EDE7058 79D8D027 5654149A 8F8634BE F9B88017 B64078BF 5436B0F5
    BE9EF22B 7300D449 A92FD940 B8E64611

B : 7B95DACF 6919A8EF 9C66A3DD B6E3A04A 7DFD5216 37079368 EF36569B B8AAB4F8
    64A734F8 B7FC6D55 BE58AA3B 2D74A022 558B227B 625821E0 CD2A3696 7BEA44BE

C : 967FCA83 9A1FA1F8 087265EC B4E75594 684012AA 31267973 0157610C 6F8725F5
    C17D220B 1545A911 EED5C892 F75016C6 FCBB7937 0F54DBFD 022FE81F D79C317F

W : 00000002 00000000


n0_final = 0 : xor counter W into A

A : 88652A3A 8EDE7058 79D8D027 5654149A 8F8634BE F9B88017 B64078BF 5436B0F5
    BE9EF22B 7300D449 A92FD940 B8E64611

B : 7B95DACF 6919A8EF 9C66A3DD B6E3A04A 7DFD5216 37079368 EF36569B B8AAB4F8
    64A734F8 B7FC6D55 BE58AA3B 2D74A022 558B227B 625821E0 CD2A3696 7BEA44BE

C : 967FCA83 9A1FA1F8 087265EC B4E75594 684012AA 31267973 0157610C 6F8725F5
    C17D220B 1545A911 EED5C892 F75016C6 FCBB7937 0F54DBFD 022FE81F D79C317F

W : 00000002 00000000


n0_final = 0 : permutation - rotate B

A : 88652A3A 8EDE7058 79D8D027 5654149A 8F8634BE F9B88017 B64078BF 5436B0F5
    BE9EF22B 7300D449 A92FD940 B8E64611
```

```
B : B59EF72B 51DED233 47BB38CD 40956DC7 A42CFBFA 26D06E0F AD37DE6C 69F17155
    69F0C94E DAAB6FF8 54777CB1 40445AE9 44F6AB16 43C0C4B0 6D2D9A54 897CF7D4

C : 967FCA83 9A1FA1F8 087265EC B4E75594 684012AA 31267973 0157610C 6F8725F5
    C17D220B 1545A911 EED5C892 F75016C6 FCBB7937 0F54DBFD 022FE81F D79C317F

W : 00000002 00000000


n0_final = 0 : permutation (j = 0)

A : 31D8859A 81BBD69A D832456B 27B965C0 47A2C6A8 82833477 0E8FC174 CCB34CEA
    081D6329 C770D623 556C9A3E 167C914E

B : 37162622 67E2A9D0 965887A1 B0CD04F6 F004CEA2 30DC1796 AB1F8252 E0AE51BF
    24030E4A 8DD9F62D 027D9CA3 690BDB63 47CA2C49 F9C5A005 FD968E3C CABF7596

C : 967FCA83 9A1FA1F8 087265EC B4E75594 684012AA 31267973 0157610C 6F8725F5
    C17D220B 1545A911 EED5C892 F75016C6 FCBB7937 0F54DBFD 022FE81F D79C317F

W : 00000002 00000000


n0_final = 0 : permutation (j = 1)

A : 7BE20C18 AA2C47CF 982D6AA9 6AE528AE 647C8975 4D24674D 9D7FC322 D6A97646
    56D7EB06 47E799F5 EABC486C E353B646

B : 86E8F0DE 9F97ACDA FB3A9209 7B7A7CBD 492189BC D9A04926 437CB336 DDF0EAC6
    CC1BEF73 4E60546B 6329AC10 470D6197 14172E18 4150D8B9 99AD20A4 BC286294

C : 967FCA83 9A1FA1F8 087265EC B4E75594 684012AA 31267973 0157610C 6F8725F5
    C17D220B 1545A911 EED5C892 F75016C6 FCBB7937 0F54DBFD 022FE81F D79C317F

W : 00000002 00000000


n0_final = 0 : permutation (j = 2)

A : 29FA00C0 8D7A0BC5 115E952C E6562CB0 3CAB8D45 5AB7CC65 178DD218 E5232A74
    DADB9113 6BE977D8 3B017BC6 3045CAF4

B : 2F2DF6C9 BCC3BF71 4858641B 1EEE1974 4446EC47 C1C56677 68580CBF A24806C2
    5B63AC5D 39889B4C 2E2175C7 94C616A5 0D0A32DC 16B73955 F7A4C570 B7EAF022

C : 967FCA83 9A1FA1F8 087265EC B4E75594 684012AA 31267973 0157610C 6F8725F5
    C17D220B 1545A911 EED5C892 F75016C6 FCBB7937 0F54DBFD 022FE81F D79C317F

W : 00000002 00000000


n0_final = 0 : permutation - add C to A

A : ADCD9E99 88F56229 EBF98C94 F24FDBC7 38B63A4D 1AF4CB9D F8199694 DDC2B9FE
    F69A2362 926225C4 90C27A47 22A2DCB1

B : 2F2DF6C9 BCC3BF71 4858641B 1EEE1974 4446EC47 C1C56677 68580CBF A24806C2
    5B63AC5D 39889B4C 2E2175C7 94C616A5 0D0A32DC 16B73955 F7A4C570 B7EAF022

C : 967FCA83 9A1FA1F8 087265EC B4E75594 684012AA 31267973 0157610C 6F8725F5
    C17D220B 1545A911 EED5C892 F75016C6 FCBB7937 0F54DBFD 022FE81F D79C317F

W : 00000002 00000000


n0_final = 0 : subtract M from C
```

```
A : ADCD9E99 88F56229 EBF98C94 F24FDBC7 38B63A4D 1AF4CB9D F8199694 DDC2B9FE
    F69A2362 926225C4 90C27A47 22A2DCB1

B : 2F2DF6C9 BCC3BF71 4858641B 1EEE1974 4446EC47 C1C56677 68580CBF A24806C2
    5B63AC5D 39889B4C 2E2175C7 94C616A5 0D0A32DC 16B73955 F7A4C570 B7EAF022

C : 967FCA03 9A1FA1F8 087265EC B4E75594 684012AA 31267973 0157610C 6F8725F5
    C17D220B 1545A911 EED5C892 F75016C6 FCBB7937 0F54DBFD 022FE81F D79C317F

W : 00000002 00000000


n0_final = 0 : swap B with C

A : ADCD9E99 88F56229 EBF98C94 F24FDBC7 38B63A4D 1AF4CB9D F8199694 DDC2B9FE
    F69A2362 926225C4 90C27A47 22A2DCB1

B : 967FCA03 9A1FA1F8 087265EC B4E75594 684012AA 31267973 0157610C 6F8725F5
    C17D220B 1545A911 EED5C892 F75016C6 FCBB7937 0F54DBFD 022FE81F D79C317F

C : 2F2DF6C9 BCC3BF71 4858641B 1EEE1974 4446EC47 C1C56677 68580CBF A24806C2
    5B63AC5D 39889B4C 2E2175C7 94C616A5 0D0A32DC 16B73955 F7A4C570 B7EAF022

W : 00000002 00000000


n0_final = 1 : add M to B

A : ADCD9E99 88F56229 EBF98C94 F24FDBC7 38B63A4D 1AF4CB9D F8199694 DDC2B9FE
    F69A2362 926225C4 90C27A47 22A2DCB1

B : 967FCA83 9A1FA1F8 087265EC B4E75594 684012AA 31267973 0157610C 6F8725F5
    C17D220B 1545A911 EED5C892 F75016C6 FCBB7937 0F54DBFD 022FE81F D79C317F

C : 2F2DF6C9 BCC3BF71 4858641B 1EEE1974 4446EC47 C1C56677 68580CBF A24806C2
    5B63AC5D 39889B4C 2E2175C7 94C616A5 0D0A32DC 16B73955 F7A4C570 B7EAF022

W : 00000002 00000000


n0_final = 1 : xor counter W into A

A : ADCD9E9B 88F56229 EBF98C94 F24FDBC7 38B63A4D 1AF4CB9D F8199694 DDC2B9FE
    F69A2362 926225C4 90C27A47 22A2DCB1

B : 967FCA83 9A1FA1F8 087265EC B4E75594 684012AA 31267973 0157610C 6F8725F5
    C17D220B 1545A911 EED5C892 F75016C6 FCBB7937 0F54DBFD 022FE81F D79C317F

C : 2F2DF6C9 BCC3BF71 4858641B 1EEE1974 4446EC47 C1C56677 68580CBF A24806C2
    5B63AC5D 39889B4C 2E2175C7 94C616A5 0D0A32DC 16B73955 F7A4C570 B7EAF022

W : 00000002 00000000


n0_final = 1 : permutation - rotate B

A : ADCD9E9B 88F56229 EBF98C94 F24FDBC7 38B63A4D 1AF4CB9D F8199694 DDC2B9FE
    F69A2362 926225C4 90C27A47 22A2DCB1

B : 95072CFF 43F1343F CBD810E4 AB2969CE 2554D080 F2E6624C C21802AE 4BEADF0E
    441782FA 52222A8B 9125DDAB 2D8DEEA0 F26FF976 B7FA1EA9 D03E045F 62FFAF38

C : 2F2DF6C9 BCC3BF71 4858641B 1EEE1974 4446EC47 C1C56677 68580CBF A24806C2
    5B63AC5D 39889B4C 2E2175C7 94C616A5 0D0A32DC 16B73955 F7A4C570 B7EAF022

W : 00000002 00000000
```

```
n0_final = 1 : permutation (j = 0)

A : 9CAD29B1 A89358CB 9B5770C1 29C8BAF4 0B1899FD AE1ACC26 011F4255 70CB6B9E
    4F1BD2B6 EB3D9804 20497238 FE4AB14D

B : 01173CD1 C5334103 F08E1786 EBEC8137 BE4EC702 B429F740 7AD0B8F7 18E12A7D
    38CB28BD B08632ED FDFD3690 5AAE93F2 878D24A3 38989A67 C4D48781 13C81B7B

C : 2F2DF6C9 BCC3BF71 4858641B 1EEE1974 4446EC47 C1C56677 68580CBF A24806C2
    5B63AC5D 39889B4C 2E2175C7 94C616A5 0D0A32DC 16B73955 F7A4C570 B7EAF022

W : 00000002 00000000


n0_final = 1 : permutation (j = 1)

A : D550385F 9E18D944 38F76D2C 151A2FAC E7849148 A3F4532D 415AE1A2 A879EC8E
    E9B6AD1C B170E5DC C6B74FB6 BF92259F

B : CC0307CE BB664DC2 FA73BA78 D2D35DDF 6AD4DCE6 26DCF4A2 CCE9C1A7 71AF8E9A
    5B3996DA 00EB4360 3CF2FFF2 5FB8F7B7 176127F0 2D3A981C 370C115E 70162587

C : 2F2DF6C9 BCC3BF71 4858641B 1EEE1974 4446EC47 C1C56677 68580CBF A24806C2
    5B63AC5D 39889B4C 2E2175C7 94C616A5 0D0A32DC 16B73955 F7A4C570 B7EAF022

W : 00000002 00000000


n0_final = 1 : permutation (j = 2)

A : 5FFD77F6 7748D8F6 D1EEB1E2 2725E20E B541DD5C 09C7ADB7 798A6EF6 9B1BC0D0
    47610F1B 156A99CE 8AC10645 9BE82916

B : 33D587F9 1AD91014 D67F57EC FDADD8EC 75AB31C5 C50ECE4D B7C2CD52 3B8500C5
    FCCD0F17 F7EED488 FF906EED DB95D041 965CBF04 B0E05609 1B26DB06 843B9DE7

C : 2F2DF6C9 BCC3BF71 4858641B 1EEE1974 4446EC47 C1C56677 68580CBF A24806C2
    5B63AC5D 39889B4C 2E2175C7 94C616A5 0D0A32DC 16B73955 F7A4C570 B7EAF022

W : 00000002 00000000


n0_final = 1 : permutation - add C to A

A : CB9C9831 F7C7EEE2 672F111F CF7B1858 2E62EDB4 D8A03D24 319C302A 79EDA771
    9D5D45F6 C21F654E 9CC6415D 2A06710C

B : 33D587F9 1AD91014 D67F57EC FDADD8EC 75AB31C5 C50ECE4D B7C2CD52 3B8500C5
    FCCD0F17 F7EED488 FF906EED DB95D041 965CBF04 B0E05609 1B26DB06 843B9DE7

C : 2F2DF6C9 BCC3BF71 4858641B 1EEE1974 4446EC47 C1C56677 68580CBF A24806C2
    5B63AC5D 39889B4C 2E2175C7 94C616A5 0D0A32DC 16B73955 F7A4C570 B7EAF022

W : 00000002 00000000


n0_final = 1 : subtract M from C

A : CB9C9831 F7C7EEE2 672F111F CF7B1858 2E62EDB4 D8A03D24 319C302A 79EDA771
    9D5D45F6 C21F654E 9CC6415D 2A06710C

B : 33D587F9 1AD91014 D67F57EC FDADD8EC 75AB31C5 C50ECE4D B7C2CD52 3B8500C5
    FCCD0F17 F7EED488 FF906EED DB95D041 965CBF04 B0E05609 1B26DB06 843B9DE7

C : 2F2DF649 BCC3BF71 4858641B 1EEE1974 4446EC47 C1C56677 68580CBF A24806C2
    5B63AC5D 39889B4C 2E2175C7 94C616A5 0D0A32DC 16B73955 F7A4C570 B7EAF022
```

```
W : 00000002 00000000


n0_final = 1 : swap B with C

A : CB9C9831 F7C7EEE2 672F111F CF7B1858 2E62EDB4 D8A03D24 319C302A 79EDA771
    9D5D45F6 C21F654E 9CC6415D 2A06710C

B : 2F2DF649 BCC3BF71 4858641B 1EEE1974 4446EC47 C1C56677 68580CBF A24806C2
    5B63AC5D 39889B4C 2E2175C7 94C616A5 0D0A32DC 16B73955 F7A4C570 B7EAF022

C : 33D587F9 1AD91014 D67F57EC FDADD8EC 75AB31C5 C50ECE4D B7C2CD52 3B8500C5
    FCCD0F17 F7EED488 FF906EED DB95D041 965CBF04 B0E05609 1B26DB06 843B9DE7

W : 00000002 00000000


n0_final = 2 : add M to B

A : CB9C9831 F7C7EEE2 672F111F CF7B1858 2E62EDB4 D8A03D24 319C302A 79EDA771
    9D5D45F6 C21F654E 9CC6415D 2A06710C

B : 2F2DF6C9 BCC3BF71 4858641B 1EEE1974 4446EC47 C1C56677 68580CBF A24806C2
    5B63AC5D 39889B4C 2E2175C7 94C616A5 0D0A32DC 16B73955 F7A4C570 B7EAF022

C : 33D587F9 1AD91014 D67F57EC FDADD8EC 75AB31C5 C50ECE4D B7C2CD52 3B8500C5
    FCCD0F17 F7EED488 FF906EED DB95D041 965CBF04 B0E05609 1B26DB06 843B9DE7

W : 00000002 00000000


n0_final = 2 : xor counter W into A

A : CB9C9833 F7C7EEE2 672F111F CF7B1858 2E62EDB4 D8A03D24 319C302A 79EDA771
    9D5D45F6 C21F654E 9CC6415D 2A06710C

B : 2F2DF6C9 BCC3BF71 4858641B 1EEE1974 4446EC47 C1C56677 68580CBF A24806C2
    5B63AC5D 39889B4C 2E2175C7 94C616A5 0D0A32DC 16B73955 F7A4C570 B7EAF022

C : 33D587F9 1AD91014 D67F57EC FDADD8EC 75AB31C5 C50ECE4D B7C2CD52 3B8500C5
    FCCD0F17 F7EED488 FF906EED DB95D041 965CBF04 B0E05609 1B26DB06 843B9DE7

W : 00000002 00000000


n0_final = 2 : permutation - rotate B

A : CB9C9833 F7C7EEE2 672F111F CF7B1858 2E62EDB4 D8A03D24 319C302A 79EDA771
    9D5D45F6 C21F654E 9CC6415D 2A06710C

B : ED925E5B 7EE37987 C83690B0 32E83DDC D88E888D CCEF838A 197ED0B0 0D854490
    58BAB6C7 36987311 EB8E5C42 2D4B298C 65B81A14 72AA2D6E 8AE1EF49 E0456FD5

C : 33D587F9 1AD91014 D67F57EC FDADD8EC 75AB31C5 C50ECE4D B7C2CD52 3B8500C5
    FCCD0F17 F7EED488 FF906EED DB95D041 965CBF04 B0E05609 1B26DB06 843B9DE7

W : 00000002 00000000


n0_final = 2 : permutation (j = 0)

A : 8DFF438F E6CFE964 AC30082E D3C8B038 4C6300DE 0D9DB8BC 78FC2DFD BEF23703
    7AAD9D34 42832D7D 3F2A7DDE D6CE7FA3

B : F99EB726 E4D5AEDD 7A4E6F4A 0626997C 0281EE3A 6BBD4056 B5FE7362 5A0741DC
    34270F45 D04C34A0 17C93AA4 73A7D344 B9708858 FC644C47 460C2942 ECBD906C
```

```
C : 33D587F9 1AD91014 D67F57EC FDADD8EC 75AB31C5 C50ECE4D B7C2CD52 3B8500C5
    FCCD0F17 F7EED488 FF906EED DB95D041 965CBF04 B0E05609 1B26DB06 843B9DE7

W : 00000002 00000000


n0_final = 2 : permutation (j = 1)

A : C2D0320E D3313CC3 3B05577B FD933B4A EF7DDC9E F1F76F48 89B9ACCD B97CB043
    E64AFCF7 BC990ABD 1901F3DB 952A2D59

B : A8C685C2 7A93D67F ED5EFCC7 07229415 1CB6DF7C 941C75EE 8D02EAE1 DEDB511E
    5561D37B 8C56AA7D EB68DDCC E523623D 626333D0 F6C00838 FA5E01B6 9FF86F65

C : 33D587F9 1AD91014 D67F57EC FDADD8EC 75AB31C5 C50ECE4D B7C2CD52 3B8500C5
    FCCD0F17 F7EED488 FF906EED DB95D041 965CBF04 B0E05609 1B26DB06 843B9DE7

W : 00000002 00000000


n0_final = 2 : permutation (j = 2)

A : AB7D1FE8 8E5BB452 B1FA6ACB ACE0D826 1F67DA7C F3F4769D C1FC439F C2A14D05
    0956F16F 99B5418E 2EB779CD F12F9B77

B : BE6AC9DC 86AFB8BB 300C6C1B 237F0C8C 6DEF5EEF 599CA070 540040F7 EEA985E4
    4A5B8375 14A6DD99 E8D207F9 F7187681 326F6930 8BCAAE00 25F4855F 3120BA43

C : 33D587F9 1AD91014 D67F57EC FDADD8EC 75AB31C5 C50ECE4D B7C2CD52 3B8500C5
    FCCD0F17 F7EED488 FF906EED DB95D041 965CBF04 B0E05609 1B26DB06 843B9DE7

W : 00000002 00000000


n0_final = 2 : permutation - add C to A

A : 08FC66FC CE392D14 42C29F35 5649D86A DCD65214 9A423F72 99D2F688 5073E130
    1E1F9B61 A28A416E 9C9572AB C3A9B2BC

B : BE6AC9DC 86AFB8BB 300C6C1B 237F0C8C 6DEF5EEF 599CA070 540040F7 EEA985E4
    4A5B8375 14A6DD99 E8D207F9 F7187681 326F6930 8BCAAE00 25F4855F 3120BA43

C : 33D587F9 1AD91014 D67F57EC FDADD8EC 75AB31C5 C50ECE4D B7C2CD52 3B8500C5
    FCCD0F17 F7EED488 FF906EED DB95D041 965CBF04 B0E05609 1B26DB06 843B9DE7

W : 00000002 00000000


n0_final = 2 : subtract M from C

A : 08FC66FC CE392D14 42C29F35 5649D86A DCD65214 9A423F72 99D2F688 5073E130
    1E1F9B61 A28A416E 9C9572AB C3A9B2BC

B : BE6AC9DC 86AFB8BB 300C6C1B 237F0C8C 6DEF5EEF 599CA070 540040F7 EEA985E4
    4A5B8375 14A6DD99 E8D207F9 F7187681 326F6930 8BCAAE00 25F4855F 3120BA43

C : 33D58779 1AD91014 D67F57EC FDADD8EC 75AB31C5 C50ECE4D B7C2CD52 3B8500C5
    FCCD0F17 F7EED488 FF906EED DB95D041 965CBF04 B0E05609 1B26DB06 843B9DE7

W : 00000002 00000000


n0_final = 2 : swap B with C  (final state)

A : 08FC66FC CE392D14 42C29F35 5649D86A DCD65214 9A423F72 99D2F688 5073E130
    1E1F9B61 A28A416E 9C9572AB C3A9B2BC
```

```
B : 33D58779 1AD91014 D67F57EC FDADD8EC 75AB31C5 C50ECE4D B7C2CD52 3B8500C5
    FCCD0F17 F7EED488 FF906EED DB95D041 965CBF04 B0E05609 1B26DB06 843B9DE7

C : BE6AC9DC 86AFB8BB 300C6C1B 237F0C8C 6DEF5EEF 599CA070 540040F7 EEA985E4
    4A5B8375 14A6DD99 E8D207F9 F7187681 326F6930 8BCAAE00 25F4855F 3120BA43

W : 00000002 00000000


Hash value (word array):

H : 14A6DD99 E8D207F9 F7187681 326F6930 8BCAAE00 25F4855F 3120BA43


Hash value (byte array):

H : 99 DD A6 14 F9 07 D2 E8 81 76 18 F7 30 69 6F 32
    00 AE CA 8B 5F 85 F4 25 43 BA 20 31
```

B.4 Intermediate States for Shabal-224 (Message B)

```
init

A : 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
    00000000 00000000 00000000 00000000

B : 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
    00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000

C : 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
    00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000

W : FFFFFFFF FFFFFFFF


block number = -1 : message block

M : 000000E0 000000E1 000000E2 000000E3 000000E4 000000E5 000000E6 000000E7
    000000E8 000000E9 000000EA 000000EB 000000EC 000000ED 000000EE 000000EF


block number = -1 : add M to B

A : 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
    00000000 00000000 00000000 00000000

B : 000000E0 000000E1 000000E2 000000E3 000000E4 000000E5 000000E6 000000E7
    000000E8 000000E9 000000EA 000000EB 000000EC 000000ED 000000EE 000000EF

C : 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
    00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000

W : FFFFFFFF FFFFFFFF


block number = -1 : xor counter W into A

A : FFFFFFFF FFFFFFFF 00000000 00000000 00000000 00000000 00000000 00000000
    00000000 00000000 00000000 00000000

B : 000000E0 000000E1 000000E2 000000E3 000000E4 000000E5 000000E6 000000E7
    000000E8 000000E9 000000EA 000000EB 000000EC 000000ED 000000EE 000000EF
```

```
C : 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
    00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000

W : FFFFFFFF FFFFFFFF


block number = -1 : permutation - rotate B

A : FFFFFFFF FFFFFFFF 00000000 00000000 00000000 00000000 00000000 00000000
    00000000 00000000 00000000 00000000

B : 01C00000 01C20000 01C40000 01C60000 01C80000 01CA0000 01CC0000 01CE0000
    01D00000 01D20000 01D40000 01D60000 01D80000 01DA0000 01DC0000 01DE0000

C : 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
    00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000

W : FFFFFFFF FFFFFFFF


block number = -1 : permutation (j = 0)

A : E0F553CA 360FB9F0 13F563A2 1751BF18 97E7BD38 130C3A22 98E64161 83D3B87F
    F2FB6BE4 D594E9DE 8CA0C9A1 E61F0AC1

B : 024800E2 FB28F276 182C49FA 2B1FCF5D 6B8842C7 EF67C5DD 6481BE9E 7FB04780
    0EA4941B 29CF1621 70F7365E 1A4CF53E 1CBAAC35 CA44460F EFB29C5D EB1240E7

C : 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
    00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000

W : FFFFFFFF FFFFFFFF


block number = -1 : permutation (j = 1)

A : 257C1CF3 FAC54BC3 B544997B B9426917 594FC7A1 D13345E4 17044F51 13DC1230
    4D00378B E315C946 A6A4D823 AE0E1AAC

B : 67772EA5 A9D983AA 1B7E3DF8 B711623E 65EF4DFA C225BD02 90585AE0 AE916A53
    C7CACB3A 56A4987E AB550A38 72247C94 9FC56034 BA443604 379E8815 3A076C00

C : 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
    00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000

W : FFFFFFFF FFFFFFFF


block number = -1 : permutation (j = 2)

A : 01340BD4 73381D1D F97F7508 28F07BF4 8E842C24 D8A8596F D2D4F99A D0763C3C
    632406CB 965BA4DD 014EB6D9 3E7F3B64

B : EC9906C8 F21851C0 C0A747BA 21DAD58B 35156FDF 088C98E7 26303F36 8A2D50AC
    FEEE45AE 8A1E966C 7B811214 CBC13AEB A351395D 1D2C372B 918C590C B58E1C9B

C : 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
    00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000

W : FFFFFFFF FFFFFFFF


block number = -1 : permutation - add C to A

A : 01340BD4 73381D1D F97F7508 28F07BF4 8E842C24 D8A8596F D2D4F99A D0763C3C
    632406CB 965BA4DD 014EB6D9 3E7F3B64
```

```
B : EC9906C8 F21851C0 C0A747BA 21DAD58B 35156FDF 088C98E7 26303F36 8A2D50AC
    FEEE45AE 8A1E966C 7B811214 CBC13AEB A351395D 1D2C372B 918C590C B58E1C9B

C : 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
    00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000

W : FFFFFFFF FFFFFFFF


block number = -1 : subtract M from C

A : 01340BD4 73381D1D F97F7508 28F07BF4 8E842C24 D8A8596F D2D4F99A D0763C3C
    632406CB 965BA4DD 014EB6D9 3E7F3B64

B : EC9906C8 F21851C0 C0A747BA 21DAD58B 35156FDF 088C98E7 26303F36 8A2D50AC
    FEEE45AE 8A1E966C 7B811214 CBC13AEB A351395D 1D2C372B 918C590C B58E1C9B

C : FFFFFF20 FFFFFF1F FFFFFF1E FFFFFF1D FFFFFF1C FFFFFF1B FFFFFF1A FFFFFF19
    FFFFFF18 FFFFFF17 FFFFFF16 FFFFFF15 FFFFFF14 FFFFFF13 FFFFFF12 FFFFFF11

W : FFFFFFFF FFFFFFFF


block number = -1 : swap B with C

A : 01340BD4 73381D1D F97F7508 28F07BF4 8E842C24 D8A8596F D2D4F99A D0763C3C
    632406CB 965BA4DD 014EB6D9 3E7F3B64

B : FFFFFF20 FFFFFF1F FFFFFF1E FFFFFF1D FFFFFF1C FFFFFF1B FFFFFF1A FFFFFF19
    FFFFFF18 FFFFFF17 FFFFFF16 FFFFFF15 FFFFFF14 FFFFFF13 FFFFFF12 FFFFFF11

C : EC9906C8 F21851C0 C0A747BA 21DAD58B 35156FDF 088C98E7 26303F36 8A2D50AC
    FEEE45AE 8A1E966C 7B811214 CBC13AEB A351395D 1D2C372B 918C590C B58E1C9B

W : FFFFFFFF FFFFFFFF


block number = 0 : increment counter W

A : 01340BD4 73381D1D F97F7508 28F07BF4 8E842C24 D8A8596F D2D4F99A D0763C3C
    632406CB 965BA4DD 014EB6D9 3E7F3B64

B : FFFFFF20 FFFFFF1F FFFFFF1E FFFFFF1D FFFFFF1C FFFFFF1B FFFFFF1A FFFFFF19
    FFFFFF18 FFFFFF17 FFFFFF16 FFFFFF15 FFFFFF14 FFFFFF13 FFFFFF12 FFFFFF11

C : EC9906C8 F21851C0 C0A747BA 21DAD58B 35156FDF 088C98E7 26303F36 8A2D50AC
    FEEE45AE 8A1E966C 7B811214 CBC13AEB A351395D 1D2C372B 918C590C B58E1C9B

W : 00000000 00000000


block number = 0 : message block

M : 000000F0 000000F1 000000F2 000000F3 000000F4 000000F5 000000F6 000000F7
    000000F8 000000F9 000000FA 000000FB 000000FC 000000FD 000000FE 000000FF


block number = 0 : add M to B

A : 01340BD4 73381D1D F97F7508 28F07BF4 8E842C24 D8A8596F D2D4F99A D0763C3C
    632406CB 965BA4DD 014EB6D9 3E7F3B64

B : 00000010 00000010 00000010 00000010 00000010 00000010 00000010 00000010
    00000010 00000010 00000010 00000010 00000010 00000010 00000010 00000010

C : EC9906C8 F21851C0 C0A747BA 21DAD58B 35156FDF 088C98E7 26303F36 8A2D50AC
    FEEE45AE 8A1E966C 7B811214 CBC13AEB A351395D 1D2C372B 918C590C B58E1C9B
```

```
W : 00000000 00000000


block number = 0 : xor counter W into A

A : 01340BD4 73381D1D F97F7508 28F07BF4 8E842C24 D8A8596F D2D4F99A D0763C3C
    632406CB 965BA4DD 014EB6D9 3E7F3B64

B : 00000010 00000010 00000010 00000010 00000010 00000010 00000010 00000010
    00000010 00000010 00000010 00000010 00000010 00000010 00000010 00000010

C : EC9906C8 F21851C0 C0A747BA 21DAD58B 35156FDF 088C98E7 26303F36 8A2D50AC
    FEEE45AE 8A1E966C 7B811214 CBC13AEB A351395D 1D2C372B 918C590C B58E1C9B

W : 00000000 00000000


block number = 0 : permutation - rotate B

A : 01340BD4 73381D1D F97F7508 28F07BF4 8E842C24 D8A8596F D2D4F99A D0763C3C
    632406CB 965BA4DD 014EB6D9 3E7F3B64

B : 00200000 00200000 00200000 00200000 00200000 00200000 00200000 00200000
    00200000 00200000 00200000 00200000 00200000 00200000 00200000 00200000

C : EC9906C8 F21851C0 C0A747BA 21DAD58B 35156FDF 088C98E7 26303F36 8A2D50AC
    FEEE45AE 8A1E966C 7B811214 CBC13AEB A351395D 1D2C372B 918C590C B58E1C9B

W : 00000000 00000000


block number = 0 : permutation (j = 0)

A : 4389B954 7C2E65DF 7AC0A60E 4D3E5671 7E2ECC0C 30113495 2288EF2A DA590830
    F8F8D0FD 866100F7 129EA35A B74146B4

B : 3D7D89CC 64237EF2 265812A5 119D3DC4 819133F3 CFAECB6A DD3710D5 25E6F7CF
    07472F02 79DEFF08 ED215CA5 48FEB94B BC3646AB 83919A20 857F59F1 B281A98E

C : EC9906C8 F21851C0 C0A747BA 21DAD58B 35156FDF 088C98E7 26303F36 8A2D50AC
    FEEE45AE 8A1E966C 7B811214 CBC13AEB A351395D 1D2C372B 918C590C B58E1C9B

W : 00000000 00000000


block number = 0 : permutation (j = 1)

A : E11D69E6 69E4164B 0FC5373C 7E495272 A8B9B043 8CDC5330 90551180 5E5046D9
    9EB4F2BE E6580BF7 E5386C60 AD2654AD

B : 2CCEC394 1350B7FB 8F6F4688 88CED426 62696AA6 86FA62DD A0A9B234 191444CC
    106CC81D 65A617A4 2A787188 104BDF1B 2F2AC2EB 7400988E 65545D9C C4ACEA3B

C : EC9906C8 F21851C0 C0A747BA 21DAD58B 35156FDF 088C98E7 26303F36 8A2D50AC
    FEEE45AE 8A1E966C 7B811214 CBC13AEB A351395D 1D2C372B 918C590C B58E1C9B

W : 00000000 00000000


block number = 0 : permutation (j = 2)

A : 01F5E756 E4B92946 BCFDB7C5 EFD3BD7F 466630E8 D1059816 DBA401AC 7E64640B
    B75BD62C 2F4525B5 AE5698A7 C7AF460B

B : E4B573A1 4C1A0880 1E907C51 04807EFD 3AD8CDE5 16B21302 02512C53 2204CB18
    99405F2D E5B648A1 70AB1D43 A10C25C2 16F1AC05 38BBEB56 9B01DC60 B1096D83
```

```
C : EC9906C8 F21851C0 C0A747BA 21DAD58B 35156FDF 088C98E7 26303F36 8A2D50AC
    FEEE45AE 8A1E966C 7B811214 CBC13AEB A351395D 1D2C372B 918C590C B58E1C9B

W : 00000000 00000000


block number = 0 : permutation - add C to A

A : A5201467 A9B8D94A D4CED997 68379D7B A7FC73BA F1A2546B 606782BF E0BCFD0F
    2F25374E 069A149F 5E2DFF25 FAECF061

B : E4B573A1 4C1A0880 1E907C51 04807EFD 3AD8CDE5 16B21302 02512C53 2204CB18
    99405F2D E5B648A1 70AB1D43 A10C25C2 16F1AC05 38BBEB56 9B01DC60 B1096D83

C : EC9906C8 F21851C0 C0A747BA 21DAD58B 35156FDF 088C98E7 26303F36 8A2D50AC
    FEEE45AE 8A1E966C 7B811214 CBC13AEB A351395D 1D2C372B 918C590C B58E1C9B

W : 00000000 00000000


block number = 0 : subtract M from C

A : A5201467 A9B8D94A D4CED997 68379D7B A7FC73BA F1A2546B 606782BF E0BCFD0F
    2F25374E 069A149F 5E2DFF25 FAECF061

B : E4B573A1 4C1A0880 1E907C51 04807EFD 3AD8CDE5 16B21302 02512C53 2204CB18
    99405F2D E5B648A1 70AB1D43 A10C25C2 16F1AC05 38BBEB56 9B01DC60 B1096D83

C : EC9905D8 F21850CF C0A746C8 21DAD498 35156EEB 088C97F2 26303E40 8A2D4FB5
    FEEE44B6 8A1E9573 7B81111A CBC139F0 A3513861 1D2C362E 918C580E B58E1B9C

W : 00000000 00000000


block number = 0 : swap B with C

A : A5201467 A9B8D94A D4CED997 68379D7B A7FC73BA F1A2546B 606782BF E0BCFD0F
    2F25374E 069A149F 5E2DFF25 FAECF061

B : EC9905D8 F21850CF C0A746C8 21DAD498 35156EEB 088C97F2 26303E40 8A2D4FB5
    FEEE44B6 8A1E9573 7B81111A CBC139F0 A3513861 1D2C362E 918C580E B58E1B9C

C : E4B573A1 4C1A0880 1E907C51 04807EFD 3AD8CDE5 16B21302 02512C53 2204CB18
    99405F2D E5B648A1 70AB1D43 A10C25C2 16F1AC05 38BBEB56 9B01DC60 B1096D83

W : 00000000 00000000


block number = 1 : increment counter W

A : A5201467 A9B8D94A D4CED997 68379D7B A7FC73BA F1A2546B 606782BF E0BCFD0F
    2F25374E 069A149F 5E2DFF25 FAECF061

B : EC9905D8 F21850CF C0A746C8 21DAD498 35156EEB 088C97F2 26303E40 8A2D4FB5
    FEEE44B6 8A1E9573 7B81111A CBC139F0 A3513861 1D2C362E 918C580E B58E1B9C

C : E4B573A1 4C1A0880 1E907C51 04807EFD 3AD8CDE5 16B21302 02512C53 2204CB18
    99405F2D E5B648A1 70AB1D43 A10C25C2 16F1AC05 38BBEB56 9B01DC60 B1096D83

W : 00000001 00000000


block number = 1 : message block

M : 64636261 68676665 6C6B6A69 706F6E6D 74737271 78777675 302D7A79 34333231
    38373635 42412D39 46454443 4A494847 4E4D4C4B 5251504F 56555453 5A595857
```

```
block number = 1 : add M to B

A : A5201467 A9B8D94A D4CED997 68379D7B A7FC73BA F1A2546B 606782BF E0BCFD0F
    2F25374E 069A149F 5E2DFF25 FAECF061

B : 50FC6839 5A7FB734 2D12B131 924A4305 A988E15C 81040E67 565DB8B9 BE6081E6
    37257AEB CC5FC2AC C1C6555D 160A8237 F19E84AC 6F7D867D E7E1AC61 0FE773F3

C : E4B573A1 4C1A0880 1E907C51 04807EFD 3AD8CDE5 16B21302 02512C53 2204CB18
    99405F2D E5B648A1 70AB1D43 A10C25C2 16F1AC05 38BBEB56 9B01DC60 B1096D83

W : 00000001 00000000


block number = 1 : xor counter W into A

A : A5201466 A9B8D94A D4CED997 68379D7B A7FC73BA F1A2546B 606782BF E0BCFD0F
    2F25374E 069A149F 5E2DFF25 FAECF061

B : 50FC6839 5A7FB734 2D12B131 924A4305 A988E15C 81040E67 565DB8B9 BE6081E6
    37257AEB CC5FC2AC C1C6555D 160A8237 F19E84AC 6F7D867D E7E1AC61 0FE773F3

C : E4B573A1 4C1A0880 1E907C51 04807EFD 3AD8CDE5 16B21302 02512C53 2204CB18
    99405F2D E5B648A1 70AB1D43 A10C25C2 16F1AC05 38BBEB56 9B01DC60 B1096D83

W : 00000001 00000000


block number = 1 : permutation - rotate B

A : A5201466 A9B8D94A D4CED997 68379D7B A7FC73BA F1A2546B 606782BF E0BCFD0F
    2F25374E 069A149F 5E2DFF25 FAECF061

B : D072A1F8 6E68B4FF 62625A25 860B2494 C2B95311 1CCF0208 7172ACBB 03CD7CC1
    F5D66E4A 855998BF AABB838C 046E2C15 0959E33D 0CFADEFB 58C3CFC3 E7E61FCE

C : E4B573A1 4C1A0880 1E907C51 04807EFD 3AD8CDE5 16B21302 02512C53 2204CB18
    99405F2D E5B648A1 70AB1D43 A10C25C2 16F1AC05 38BBEB56 9B01DC60 B1096D83

W : 00000001 00000000


block number = 1 : permutation (j = 0)

A : 8C6560B6 46C63757 88C69A25 846FFE84 1D1CEFD6 5FBEA9B1 F659B29C 772D12C6
    30C10360 06A31BE8 EB368AA8 A21F7FDF

B : 9E34389F F9C9ED48 97683AF6 A4B76138 6791B60A 99DF525E EB431415 8F4814BB
    2492200A F3EFD568 41BE724E 553CD80A 61295933 A0CC755E C6BEFA5C B45C3EE6

C : E4B573A1 4C1A0880 1E907C51 04807EFD 3AD8CDE5 16B21302 02512C53 2204CB18
    99405F2D E5B648A1 70AB1D43 A10C25C2 16F1AC05 38BBEB56 9B01DC60 B1096D83

W : 00000001 00000000


block number = 1 : permutation (j = 1)

A : 23BBE3D7 CE453E25 7DE78C6F B5F31A7C A0A64F86 D1CB5DB0 2B5C3669 CEC91EF6
    714AD2A9 81CCD639 04DD0E5F C30C0B39

B : 7FD3AB2F 8844471A 5EB7789C 06DFC577 41964142 4D8D8D7B 2DA4D98B 2263DDB1
    95605C3C D6656B0B 0164970C E0755597 9D0B021F 6FAC48F2 59DE3D2F 598E9CC4

C : E4B573A1 4C1A0880 1E907C51 04807EFD 3AD8CDE5 16B21302 02512C53 2204CB18
```

```
    99405F2D E5B648A1 70AB1D43 A10C25C2 16F1AC05 38BBEB56 9B01DC60 B1096D83

W : 00000001 00000000


block number = 1 : permutation (j = 2)

A : 75457F4C CFC19617 D104D673 18ACCC64 0588C3A0 0AD46F0E 002D5D7B 45ACAD09
    33CE4C17 D042FCFB A1390D75 FDA502E4

B : A9FC3D62 AFA9B73D F0D3F99C 1D4856D0 09960237 AB25731E 75B29A9A A39488F9
    D0B78426 59E146E6 FD1B8C9C 7AB9F9D9 F627B7D7 F0E592E0 ED7A88D4 B147C493

C : E4B573A1 4C1A0880 1E907C51 04807EFD 3AD8CDE5 16B21302 02512C53 2204CB18
    99405F2D E5B648A1 70AB1D43 A10C25C2 16F1AC05 38BBEB56 9B01DC60 B1096D83

W : 00000001 00000000


block number = 1 : permutation - add C to A

A : CBDB918E 064183A2 6C8CDD4B D4905168 DD177B38 C3A30FC1 48AFC19E D73972F0
    FB5FBBEE BB4DD612 D65D546E 0BA328DA

B : A9FC3D62 AFA9B73D F0D3F99C 1D4856D0 09960237 AB25731E 75B29A9A A39488F9
    D0B78426 59E146E6 FD1B8C9C 7AB9F9D9 F627B7D7 F0E592E0 ED7A88D4 B147C493

C : E4B573A1 4C1A0880 1E907C51 04807EFD 3AD8CDE5 16B21302 02512C53 2204CB18
    99405F2D E5B648A1 70AB1D43 A10C25C2 16F1AC05 38BBEB56 9B01DC60 B1096D83

W : 00000001 00000000


block number = 1 : subtract M from C

A : CBDB918E 064183A2 6C8CDD4B D4905168 DD177B38 C3A30FC1 48AFC19E D73972F0
    FB5FBBEE BB4DD612 D65D546E 0BA328DA

B : A9FC3D62 AFA9B73D F0D3F99C 1D4856D0 09960237 AB25731E 75B29A9A A39488F9
    D0B78426 59E146E6 FD1B8C9C 7AB9F9D9 F627B7D7 F0E592E0 ED7A88D4 B147C493

C : 80521140 E3B2A21B B22511E8 94111090 C6655B74 9E3A9C8D D223B1DA EDD198E7
    610928F8 A3751B68 2A65D900 56C2DD7B C8A45FBA E66A9B07 44AC880D 56B0152C

W : 00000001 00000000


block number = 1 : swap B with C

A : CBDB918E 064183A2 6C8CDD4B D4905168 DD177B38 C3A30FC1 48AFC19E D73972F0
    FB5FBBEE BB4DD612 D65D546E 0BA328DA

B : 80521140 E3B2A21B B22511E8 94111090 C6655B74 9E3A9C8D D223B1DA EDD198E7
    610928F8 A3751B68 2A65D900 56C2DD7B C8A45FBA E66A9B07 44AC880D 56B0152C

C : A9FC3D62 AFA9B73D F0D3F99C 1D4856D0 09960237 AB25731E 75B29A9A A39488F9
    D0B78426 59E146E6 FD1B8C9C 7AB9F9D9 F627B7D7 F0E592E0 ED7A88D4 B147C493

W : 00000001 00000000


block number = 2 : increment counter W

A : CBDB918E 064183A2 6C8CDD4B D4905168 DD177B38 C3A30FC1 48AFC19E D73972F0
    FB5FBBEE BB4DD612 D65D546E 0BA328DA

B : 80521140 E3B2A21B B22511E8 94111090 C6655B74 9E3A9C8D D223B1DA EDD198E7
```

```
    610928F8 A3751B68 2A65D900 56C2DD7B C8A45FBA E66A9B07 44AC880D 56B0152C

C : A9FC3D62 AFA9B73D F0D3F99C 1D4856D0 09960237 AB25731E 75B29A9A A39488F9
    D0B78426 59E146E6 FD1B8C9C 7AB9F9D9 F627B7D7 F0E592E0 ED7A88D4 B147C493

W : 00000002 00000000


block number = 2 : message block

M : 3231302D 36353433 2D393837 64636261 68676665 6C6B6A69 706F6E6D 74737271
    78777675 00807A79 00000000 00000000 00000000 00000000 00000000 00000000


block number = 2 : add M to B

A : CBDB918E 064183A2 6C8CDD4B D4905168 DD177B38 C3A30FC1 48AFC19E D73972F0
    FB5FBBEE BB4DD612 D65D546E 0BA328DA

B : B283416D 19E7D64E DF5E4A1F F87472F1 2ECCC1D9 0AA606F6 42932047 62450B58
    D9809F6D A3F595E1 2A65D900 56C2DD7B C8A45FBA E66A9B07 44AC880D 56B0152C

C : A9FC3D62 AFA9B73D F0D3F99C 1D4856D0 09960237 AB25731E 75B29A9A A39488F9
    D0B78426 59E146E6 FD1B8C9C 7AB9F9D9 F627B7D7 F0E592E0 ED7A88D4 B147C493

W : 00000002 00000000


block number = 2 : xor counter W into A

A : CBDB918C 064183A2 6C8CDD4B D4905168 DD177B38 C3A30FC1 48AFC19E D73972F0
    FB5FBBEE BB4DD612 D65D546E 0BA328DA

B : B283416D 19E7D64E DF5E4A1F F87472F1 2ECCC1D9 0AA606F6 42932047 62450B58
    D9809F6D A3F595E1 2A65D900 56C2DD7B C8A45FBA E66A9B07 44AC880D 56B0152C

C : A9FC3D62 AFA9B73D F0D3F99C 1D4856D0 09960237 AB25731E 75B29A9A A39488F9
    D0B78426 59E146E6 FD1B8C9C 7AB9F9D9 F627B7D7 F0E592E0 ED7A88D4 B147C493

W : 00000002 00000000


block number = 2 : permutation - rotate B

A : CBDB918C 064183A2 6C8CDD4B D4905168 DD177B38 C3A30FC1 48AFC19E D73972F0
    FB5FBBEE BB4DD612 D65D546E 0BA328DA

B : 82DB6506 AC9C33CF 943FBEBC E5E3F0E8 83B25D99 0DEC154C 408E8526 16B0C48A
    3EDBB301 2BC347EB B20054CB BAF6AD85 BF759148 360FCCD5 101A8959 2A58AD60

C : A9FC3D62 AFA9B73D F0D3F99C 1D4856D0 09960237 AB25731E 75B29A9A A39488F9
    D0B78426 59E146E6 FD1B8C9C 7AB9F9D9 F627B7D7 F0E592E0 ED7A88D4 B147C493

W : 00000002 00000000


block number = 2 : permutation (j = 0)

A : DF1A97E9 D6B1A782 D96A6348 DA9DAC23 5E46C0AF 6CB06C45 1A6C68C6 AA007071
    9CFAB7D5 B45EA5D2 C44B5068 28B35FE0

B : 22D191FE 9F37773E BEC49E4E 4E7E0DA5 A6DD8463 8897B922 648E9D75 789E069A
    1EB22E28 1C27D5FB 5FB40600 A2A1FB14 5E0E4A87 4551C1D7 06A08E05 71D3091C

C : A9FC3D62 AFA9B73D F0D3F99C 1D4856D0 09960237 AB25731E 75B29A9A A39488F9
    D0B78426 59E146E6 FD1B8C9C 7AB9F9D9 F627B7D7 F0E592E0 ED7A88D4 B147C493
```

```
W : 00000002 00000000


block number = 2 : permutation (j = 1)

A : 3B44212B 38BAA6B1 DDAE09FA FA7EAC58 7DC6ACAB B9D314C2 E29E3349 BF392F25
    2684E514 E7DD225E 9F9F888E 276E49D7

B : 34E7DF5E 23C3AAF3 4F5BF5EE 6A40590D 94C0122C 090DAFE4 A97D4D9B 29ADBB1C
    F9DF8284 FF0AF2B8 9D39FA05 40C2A58E 3E25C65A CC8F6893 1020D0BC A360C2E2

C : A9FC3D62 AFA9B73D F0D3F99C 1D4856D0 09960237 AB25731E 75B29A9A A39488F9
    D0B78426 59E146E6 FD1B8C9C 7AB9F9D9 F627B7D7 F0E592E0 ED7A88D4 B147C493

W : 00000002 00000000


block number = 2 : permutation (j = 2)

A : 77A014BC EEA36A0D 3771AF76 CEBD1EF8 72060B53 F42159D8 2669B358 F1D02FEC
    912B7E1F F21D9DC4 1E589CD0 1B1EF3E6

B : 354B1667 52CB98D5 D099C207 2B43AF62 A1DFCF1A 0347CA3A 9A74CBBE 6219973F
    7E46F1A5 F5CB4356 E3E5B8AC 8FAA9B0F 129F0D54 94FCB31C C1E6C257 A22089DC

C : A9FC3D62 AFA9B73D F0D3F99C 1D4856D0 09960237 AB25731E 75B29A9A A39488F9
    D0B78426 59E146E6 FD1B8C9C 7AB9F9D9 F627B7D7 F0E592E0 ED7A88D4 B147C493

W : 00000002 00000000


block number = 2 : permutation - add C to A

A : C0EA29F8 985D617D 83266CB1 22BE3C02 E42AAFAF 786B1D97 DB1A2499 557250BE
    CCC257C1 C292DBF8 1444E9B4 7B67A3F0

B : 354B1667 52CB98D5 D099C207 2B43AF62 A1DFCF1A 0347CA3A 9A74CBBE 6219973F
    7E46F1A5 F5CB4356 E3E5B8AC 8FAA9B0F 129F0D54 94FCB31C C1E6C257 A22089DC

C : A9FC3D62 AFA9B73D F0D3F99C 1D4856D0 09960237 AB25731E 75B29A9A A39488F9
    D0B78426 59E146E6 FD1B8C9C 7AB9F9D9 F627B7D7 F0E592E0 ED7A88D4 B147C493

W : 00000002 00000000


block number = 2 : subtract M from C

A : C0EA29F8 985D617D 83266CB1 22BE3C02 E42AAFAF 786B1D97 DB1A2499 557250BE
    CCC257C1 C292DBF8 1444E9B4 7B67A3F0

B : 354B1667 52CB98D5 D099C207 2B43AF62 A1DFCF1A 0347CA3A 9A74CBBE 6219973F
    7E46F1A5 F5CB4356 E3E5B8AC 8FAA9B0F 129F0D54 94FCB31C C1E6C257 A22089DC

C : 77CB0D35 7974830A C39AC165 B8E4F46F A12E9BD2 3EBA08B5 05432C2D 2F211688
    58400DB1 5960CC6D FD1B8C9C 7AB9F9D9 F627B7D7 F0E592E0 ED7A88D4 B147C493

W : 00000002 00000000


block number = 2 : swap B with C

A : C0EA29F8 985D617D 83266CB1 22BE3C02 E42AAFAF 786B1D97 DB1A2499 557250BE
    CCC257C1 C292DBF8 1444E9B4 7B67A3F0

B : 77CB0D35 7974830A C39AC165 B8E4F46F A12E9BD2 3EBA08B5 05432C2D 2F211688
    58400DB1 5960CC6D FD1B8C9C 7AB9F9D9 F627B7D7 F0E592E0 ED7A88D4 B147C493
```

```
C : 354B1667 52CB98D5 D099C207 2B43AF62 A1DFCF1A 0347CA3A 9A74CBBE 6219973F
    7E46F1A5 F5CB4356 E3E5B8AC 8FAA9B0F 129F0D54 94FCB31C C1E6C257 A22089DC

W : 00000002 00000000


n0_final = 0 : add M to B

A : C0EA29F8 985D617D 83266CB1 22BE3C02 E42AAFAF 786B1D97 DB1A2499 557250BE
    CCC257C1 C292DBF8 1444E9B4 7B67A3F0

B : A9FC3D62 AFA9B73D F0D3F99C 1D4856D0 09960237 AB25731E 75B29A9A A39488F9
    D0B78426 59E146E6 FD1B8C9C 7AB9F9D9 F627B7D7 F0E592E0 ED7A88D4 B147C493

C : 354B1667 52CB98D5 D099C207 2B43AF62 A1DFCF1A 0347CA3A 9A74CBBE 6219973F
    7E46F1A5 F5CB4356 E3E5B8AC 8FAA9B0F 129F0D54 94FCB31C C1E6C257 A22089DC

W : 00000002 00000000


n0_final = 0 : xor counter W into A

A : C0EA29FA 985D617D 83266CB1 22BE3C02 E42AAFAF 786B1D97 DB1A2499 557250BE
    CCC257C1 C292DBF8 1444E9B4 7B67A3F0

B : A9FC3D62 AFA9B73D F0D3F99C 1D4856D0 09960237 AB25731E 75B29A9A A39488F9
    D0B78426 59E146E6 FD1B8C9C 7AB9F9D9 F627B7D7 F0E592E0 ED7A88D4 B147C493

C : 354B1667 52CB98D5 D099C207 2B43AF62 A1DFCF1A 0347CA3A 9A74CBBE 6219973F
    7E46F1A5 F5CB4356 E3E5B8AC 8FAA9B0F 129F0D54 94FCB31C C1E6C257 A22089DC

W : 00000002 00000000


n0_final = 0 : permutation - rotate B

A : C0EA29FA 985D617D 83266CB1 22BE3C02 E42AAFAF 786B1D97 DB1A2499 557250BE
    CCC257C1 C292DBF8 1444E9B4 7B67A3F0

B : 7AC553F8 6E7B5F53 F339E1A7 ADA03A90 046E132C E63D564A 3534EB65 11F34729
    084DA16F 8DCCB3C2 1939FA37 F3B2F573 6FAFEC4F 25C1E1CB 11A9DAF5 8927628F

C : 354B1667 52CB98D5 D099C207 2B43AF62 A1DFCF1A 0347CA3A 9A74CBBE 6219973F
    7E46F1A5 F5CB4356 E3E5B8AC 8FAA9B0F 129F0D54 94FCB31C C1E6C257 A22089DC

W : 00000002 00000000


n0_final = 0 : permutation (j = 0)

A : B06DABA1 C2FAAF6C D26BD04A C346339C 401C0339 4E95BA16 FF3D20E2 01CC9798
    E76A4A83 C859C1C9 D418DB48 FC04E5DE

B : 632C5D0B 68F2C4D3 338162E2 1F4E24CB B73FDA9E 7D10E97C 6AAB09D7 DDD5E635
    080EF7A2 2C3F59B3 1994D0D9 E49EF0C6 90CD8CC0 76869305 0EC79A5F 2EF7097C

C : 354B1667 52CB98D5 D099C207 2B43AF62 A1DFCF1A 0347CA3A 9A74CBBE 6219973F
    7E46F1A5 F5CB4356 E3E5B8AC 8FAA9B0F 129F0D54 94FCB31C C1E6C257 A22089DC

W : 00000002 00000000


n0_final = 0 : permutation (j = 1)

A : 90072ACB 3ECE3BEB D37570FA D6B90896 3A5BD5E8 AD157A12 62B0A0BE 533EBFC2
    740AC54E 5933F890 9A93B77C 3D826737
```

```
B : 4713E6F8 279E7D3C 14037EF6 79738F3B E58A8F8C 5CEDD597 B03A5B2D 79D654A3
    7FE53A70 994F7772 1FA32EB7 E07B16E4 E43F3396 BFE7A3E7 80C06BFF F12F52C5

C : 354B1667 52CB98D5 D099C207 2B43AF62 A1DFCF1A 0347CA3A 9A74CBBE 6219973F
    7E46F1A5 F5CB4356 E3E5B8AC 8FAA9B0F 129F0D54 94FCB31C C1E6C257 A22089DC

W : 00000002 00000000


n0_final = 0 : permutation (j = 2)

A : 08247C8E 390FEFE8 11B58153 9D1A521A 4DA938A6 85400B12 BF219BD8 3A0F40D3
    680284AD 94558929 1B7BE1CC 9F38E8DE

B : 8A362B47 0C433F2B 3D2F6EE3 0CA21A65 3CCE9C68 7F2BBB39 8E3EC8F7 914904A3
    4D9CB3B9 48211A08 7F983949 050692E5 5F831C7F 14653119 E504C9CC 8299B2AA

C : 354B1667 52CB98D5 D099C207 2B43AF62 A1DFCF1A 0347CA3A 9A74CBBE 6219973F
    7E46F1A5 F5CB4356 E3E5B8AC 8FAA9B0F 129F0D54 94FCB31C C1E6C257 A22089DC

W : 00000002 00000000


n0_final = 0 : permutation - add C to A

A : 653350DB 22D9E2BD FCC5977E CA0FA236 7D270923 DAB1E238 0B00423D 89038744
    850A665D C71B573C A98BA278 DF7A2F9F

B : 8A362B47 0C433F2B 3D2F6EE3 0CA21A65 3CCE9C68 7F2BBB39 8E3EC8F7 914904A3
    4D9CB3B9 48211A08 7F983949 050692E5 5F831C7F 14653119 E504C9CC 8299B2AA

C : 354B1667 52CB98D5 D099C207 2B43AF62 A1DFCF1A 0347CA3A 9A74CBBE 6219973F
    7E46F1A5 F5CB4356 E3E5B8AC 8FAA9B0F 129F0D54 94FCB31C C1E6C257 A22089DC

W : 00000002 00000000


n0_final = 0 : subtract M from C

A : 653350DB 22D9E2BD FCC5977E CA0FA236 7D270923 DAB1E238 0B00423D 89038744
    850A665D C71B573C A98BA278 DF7A2F9F

B : 8A362B47 0C433F2B 3D2F6EE3 0CA21A65 3CCE9C68 7F2BBB39 8E3EC8F7 914904A3
    4D9CB3B9 48211A08 7F983949 050692E5 5F831C7F 14653119 E504C9CC 8299B2AA

C : 0319E63A 1C9664A2 A36089D0 C6E04D01 397868B5 96DC5FD1 2A055D51 EDA624CE
    05CF7B30 F54AC8DD E3E5B8AC 8FAA9B0F 129F0D54 94FCB31C C1E6C257 A22089DC

W : 00000002 00000000


n0_final = 0 : swap B with C

A : 653350DB 22D9E2BD FCC5977E CA0FA236 7D270923 DAB1E238 0B00423D 89038744
    850A665D C71B573C A98BA278 DF7A2F9F

B : 0319E63A 1C9664A2 A36089D0 C6E04D01 397868B5 96DC5FD1 2A055D51 EDA624CE
    05CF7B30 F54AC8DD E3E5B8AC 8FAA9B0F 129F0D54 94FCB31C C1E6C257 A22089DC

C : 8A362B47 0C433F2B 3D2F6EE3 0CA21A65 3CCE9C68 7F2BBB39 8E3EC8F7 914904A3
    4D9CB3B9 48211A08 7F983949 050692E5 5F831C7F 14653119 E504C9CC 8299B2AA

W : 00000002 00000000


n0_final = 1 : add M to B
```

```
A : 653350DB 22D9E2BD FCC5977E CA0FA236 7D270923 DAB1E238 0B00423D 89038744
    850A665D C71B573C A98BA278 DF7A2F9F

B : 354B1667 52CB98D5 D099C207 2B43AF62 A1DFCF1A 0347CA3A 9A74CBBE 6219973F
    7E46F1A5 F5CB4356 E3E5B8AC 8FAA9B0F 129F0D54 94FCB31C C1E6C257 A22089DC

C : 8A362B47 0C433F2B 3D2F6EE3 0CA21A65 3CCE9C68 7F2BBB39 8E3EC8F7 914904A3
    4D9CB3B9 48211A08 7F983949 050692E5 5F831C7F 14653119 E504C9CC 8299B2AA

W : 00000002 00000000


n0_final = 1 : xor counter W into A

A : 653350D9 22D9E2BD FCC5977E CA0FA236 7D270923 DAB1E238 0B00423D 89038744
    850A665D C71B573C A98BA278 DF7A2F9F

B : 354B1667 52CB98D5 D099C207 2B43AF62 A1DFCF1A 0347CA3A 9A74CBBE 6219973F
    7E46F1A5 F5CB4356 E3E5B8AC 8FAA9B0F 129F0D54 94FCB31C C1E6C257 A22089DC

C : 8A362B47 0C433F2B 3D2F6EE3 0CA21A65 3CCE9C68 7F2BBB39 8E3EC8F7 914904A3
    4D9CB3B9 48211A08 7F983949 050692E5 5F831C7F 14653119 E504C9CC 8299B2AA

W : 00000002 00000000


n0_final = 1 : permutation - rotate B

A : 653350D9 22D9E2BD FCC5977E CA0FA236 7D270923 DAB1E238 0B00423D 89038744
    850A665D C71B573C A98BA278 DF7A2F9F

B : 2CCE6A96 31AAA597 840FA133 5EC45687 9E3543BF 9474068F 977D34E9 2E7EC433
    E34AFC8D 86ADEB96 7159C7CB 361F1F55 1AA8253E 663929F9 84AF83CD 13B94441

C : 8A362B47 0C433F2B 3D2F6EE3 0CA21A65 3CCE9C68 7F2BBB39 8E3EC8F7 914904A3
    4D9CB3B9 48211A08 7F983949 050692E5 5F831C7F 14653119 E504C9CC 8299B2AA

W : 00000002 00000000


n0_final = 1 : permutation (j = 0)

A : 60E8E576 316FE540 8A0B25AB 162A5B42 E6D7E5C1 6359F3DE 1F4EC558 E1651DB6
    9749184B 8137C54B 27C916A6 D0E26E39

B : EC0B1162 0B3294A9 B6F7B2E1 E8BE96F2 25429D41 B44E013E CE4B5374 42676A2F
    AE231EAF 7393ED99 3A8566CF 4323AF6C AA4750F5 02E2494D 7CABDDCF CEA72C3F

C : 8A362B47 0C433F2B 3D2F6EE3 0CA21A65 3CCE9C68 7F2BBB39 8E3EC8F7 914904A3
    4D9CB3B9 48211A08 7F983949 050692E5 5F831C7F 14653119 E504C9CC 8299B2AA

W : 00000002 00000000


n0_final = 1 : permutation (j = 1)

A : 86790693 8A20F14E 300DC8BB 62D4566C A37F1C26 180D8571 D72EB4D2 9E6B3933
    96B721ED BE48220A A21A0633 E6DF78B8

B : C8E12BE0 4A1ED32A E5676F5B 7920AF87 23CDE490 292BDF88 C1735F25 9DEE5319
    25C0C433 92F8D583 BAF8FADA 1B6CF74B 080E4232 E236E814 D186F0B3 FCDA9EB3

C : 8A362B47 0C433F2B 3D2F6EE3 0CA21A65 3CCE9C68 7F2BBB39 8E3EC8F7 914904A3
    4D9CB3B9 48211A08 7F983949 050692E5 5F831C7F 14653119 E504C9CC 8299B2AA

W : 00000002 00000000
```

```
n0_final = 1 : permutation (j = 2)

A : 7B677F6B 9264B604 1122E95A DA5BF2F1 A9924516 CC1A2335 2BDCF160 2A950BAF
    4B8F21E4 F6726F41 A14A2B78 0C3AABCB

B : D0CD2B92 9CEDC63A 96A6C121 CBE07950 C30349B4 3FCCF6EB 6C3BA8EE 1E78AB3D
    1DEC328F 161477CD A1D2FB2A E3B31AC6 A46C5A7F CDE04097 FDB835E0 0A706953

C : 8A362B47 0C433F2B 3D2F6EE3 0CA21A65 3CCE9C68 7F2BBB39 8E3EC8F7 914904A3
    4D9CB3B9 48211A08 7F983949 050692E5 5F831C7F 14653119 E504C9CC 8299B2AA

W : 00000002 00000000


n0_final = 1 : permutation - add C to A

A : 0FA9DF5F B8EC9A32 B0F714D7 8ACEF497 CA1716C8 E0BB9E9D FF6D05CC 759B7CD2
    EE80D3D1 E060DBE1 7CFC31D2 FF1677D7

B : D0CD2B92 9CEDC63A 96A6C121 CBE07950 C30349B4 3FCCF6EB 6C3BA8EE 1E78AB3D
    1DEC328F 161477CD A1D2FB2A E3B31AC6 A46C5A7F CDE04097 FDB835E0 0A706953

C : 8A362B47 0C433F2B 3D2F6EE3 0CA21A65 3CCE9C68 7F2BBB39 8E3EC8F7 914904A3
    4D9CB3B9 48211A08 7F983949 050692E5 5F831C7F 14653119 E504C9CC 8299B2AA

W : 00000002 00000000


n0_final = 1 : subtract M from C

A : 0FA9DF5F B8EC9A32 B0F714D7 8ACEF497 CA1716C8 E0BB9E9D FF6D05CC 759B7CD2
    EE80D3D1 E060DBE1 7CFC31D2 FF1677D7

B : D0CD2B92 9CEDC63A 96A6C121 CBE07950 C30349B4 3FCCF6EB 6C3BA8EE 1E78AB3D
    1DEC328F 161477CD A1D2FB2A E3B31AC6 A46C5A7F CDE04097 FDB835E0 0A706953

C : 5804FB1A D60E0AF8 0FF636AC A83EB804 D4673603 12C050D0 1DCF5A8A 1CD59232
    D5253D44 47A09F8F 7F983949 050692E5 5F831C7F 14653119 E504C9CC 8299B2AA

W : 00000002 00000000


n0_final = 1 : swap B with C

A : 0FA9DF5F B8EC9A32 B0F714D7 8ACEF497 CA1716C8 E0BB9E9D FF6D05CC 759B7CD2
    EE80D3D1 E060DBE1 7CFC31D2 FF1677D7

B : 5804FB1A D60E0AF8 0FF636AC A83EB804 D4673603 12C050D0 1DCF5A8A 1CD59232
    D5253D44 47A09F8F 7F983949 050692E5 5F831C7F 14653119 E504C9CC 8299B2AA

C : D0CD2B92 9CEDC63A 96A6C121 CBE07950 C30349B4 3FCCF6EB 6C3BA8EE 1E78AB3D
    1DEC328F 161477CD A1D2FB2A E3B31AC6 A46C5A7F CDE04097 FDB835E0 0A706953

W : 00000002 00000000


n0_final = 2 : add M to B

A : 0FA9DF5F B8EC9A32 B0F714D7 8ACEF497 CA1716C8 E0BB9E9D FF6D05CC 759B7CD2
    EE80D3D1 E060DBE1 7CFC31D2 FF1677D7

B : 8A362B47 0C433F2B 3D2F6EE3 0CA21A65 3CCE9C68 7F2BBB39 8E3EC8F7 914904A3
    4D9CB3B9 48211A08 7F983949 050692E5 5F831C7F 14653119 E504C9CC 8299B2AA

C : D0CD2B92 9CEDC63A 96A6C121 CBE07950 C30349B4 3FCCF6EB 6C3BA8EE 1E78AB3D
    1DEC328F 161477CD A1D2FB2A E3B31AC6 A46C5A7F CDE04097 FDB835E0 0A706953
```

```
W : 00000002 00000000


n0_final = 2 : xor counter W into A

A : 0FA9DF5D B8EC9A32 B0F714D7 8ACEF497 CA1716C8 E0BB9E9D FF6D05CC 759B7CD2
    EE80D3D1 E060DBE1 7CFC31D2 FF1677D7

B : 8A362B47 0C433F2B 3D2F6EE3 0CA21A65 3CCE9C68 7F2BBB39 8E3EC8F7 914904A3
    4D9CB3B9 48211A08 7F983949 050692E5 5F831C7F 14653119 E504C9CC 8299B2AA

C : D0CD2B92 9CEDC63A 96A6C121 CBE07950 C30349B4 3FCCF6EB 6C3BA8EE 1E78AB3D
    1DEC328F 161477CD A1D2FB2A E3B31AC6 A46C5A7F CDE04097 FDB835E0 0A706953

W : 00000002 00000000


n0_final = 2 : permutation - rotate B

A : 0FA9DF5D B8EC9A32 B0F714D7 8ACEF497 CA1716C8 E0BB9E9D FF6D05CC 759B7CD2
    EE80D3D1 E060DBE1 7CFC31D2 FF1677D7

B : 568F146C 7E561886 DDC67A5E 34CA1944 38D0799D 7672FE57 91EF1C7D 09472292
    67729B39 34109042 7292FF30 25CA0A0D 38FEBF06 623228CA 9399CA09 65550533

C : D0CD2B92 9CEDC63A 96A6C121 CBE07950 C30349B4 3FCCF6EB 6C3BA8EE 1E78AB3D
    1DEC328F 161477CD A1D2FB2A E3B31AC6 A46C5A7F CDE04097 FDB835E0 0A706953

W : 00000002 00000000


n0_final = 2 : permutation (j = 0)

A : 10DC5597 4EB0AB2A E9391D54 5A2C8304 10B0B326 F153EAD7 01430C09 05F436D0
    D7CD5D60 C56766D0 BC7937FC CB9FAB2A

B : 8B687EED 78A26178 B2A72BE1 DC5D3CB5 9EEFBFE3 E249E986 DD62CB0D E8858C0B
    E6D794ED 52B9B9AB A6A33663 7FF440CF 9EDED464 752B0541 31F576B8 6F79769D

C : D0CD2B92 9CEDC63A 96A6C121 CBE07950 C30349B4 3FCCF6EB 6C3BA8EE 1E78AB3D
    1DEC328F 161477CD A1D2FB2A E3B31AC6 A46C5A7F CDE04097 FDB835E0 0A706953

W : 00000002 00000000


n0_final = 2 : permutation (j = 1)

A : 5E74B6C5 5E7772C8 FAA3B326 84A173D3 024DBC7D 25724599 7EA4A347 8633B5DC
    70ECC490 DE3DC78E BFC0E264 E9260137

B : 4C2F25BF ABD255DA 039D8ABD A63E3971 B2CC44A8 E551EB7C FAFA8B80 C7D2E6DF
    6C2460E1 04FBFE61 481A201E 84B60DB2 C00FEB4B 30DBB0E4 E2B1B1C8 A73EA719

C : D0CD2B92 9CEDC63A 96A6C121 CBE07950 C30349B4 3FCCF6EB 6C3BA8EE 1E78AB3D
    1DEC328F 161477CD A1D2FB2A E3B31AC6 A46C5A7F CDE04097 FDB835E0 0A706953

W : 00000002 00000000


n0_final = 2 : permutation (j = 2)

A : CF452790 81296D85 8776EBA0 E26CDAAE 1DA6D373 7B25D5FA DBBF8BE9 EC424D6E
    3CDDF23A 51106F6C 6CC1D02B 44A7C0D1

B : E3BD6C0F E3B4A56E E9349EB2 29739374 5522513E B4754483 8D7C035E 9236E8EE
    3A11ED4E 8D2DD6C7 B474342A 1AD1A9F4 433DDB52 CF58F15B 565D4C45 F525711D
```

```
C : D0CD2B92 9CEDC63A 96A6C121 CBE07950 C30349B4 3FCCF6EB 6C3BA8EE 1E78AB3D
    1DEC328F 161477CD A1D2FB2A E3B31AC6 A46C5A7F CDE04097 FDB835E0 0A706953

W : 00000002 00000000


n0_final = 2 : permutation - add C to A

A : 894924F9 B9663D4A 3211E95C E3077A9D 12706153 2CE27DCF CE8EC0DB 90F7B2A7
    0AEA318D D66C462E 90837F7A 506E9AC9

B : E3BD6C0F E3B4A56E E9349EB2 29739374 5522513E B4754483 8D7C035E 9236E8EE
    3A11ED4E 8D2DD6C7 B474342A 1AD1A9F4 433DDB52 CF58F15B 565D4C45 F525711D

C : D0CD2B92 9CEDC63A 96A6C121 CBE07950 C30349B4 3FCCF6EB 6C3BA8EE 1E78AB3D
    1DEC328F 161477CD A1D2FB2A E3B31AC6 A46C5A7F CDE04097 FDB835E0 0A706953

W : 00000002 00000000


n0_final = 2 : subtract M from C

A : 894924F9 B9663D4A 3211E95C E3077A9D 12706153 2CE27DCF CE8EC0DB 90F7B2A7
    0AEA318D D66C462E 90837F7A 506E9AC9

B : E3BD6C0F E3B4A56E E9349EB2 29739374 5522513E B4754483 8D7C035E 9236E8EE
    3A11ED4E 8D2DD6C7 B474342A 1AD1A9F4 433DDB52 CF58F15B 565D4C45 F525711D

C : 9E9BFB65 66B89207 696D88EA 677D16EF 5A9BE34F D3618C82 FBCC3A81 AA0538CC
    A574BC1A 1593FD54 A1D2FB2A E3B31AC6 A46C5A7F CDE04097 FDB835E0 0A706953

W : 00000002 00000000


n0_final = 2 : swap B with C  (final state)

A : 894924F9 B9663D4A 3211E95C E3077A9D 12706153 2CE27DCF CE8EC0DB 90F7B2A7
    0AEA318D D66C462E 90837F7A 506E9AC9

B : 9E9BFB65 66B89207 696D88EA 677D16EF 5A9BE34F D3618C82 FBCC3A81 AA0538CC
    A574BC1A 1593FD54 A1D2FB2A E3B31AC6 A46C5A7F CDE04097 FDB835E0 0A706953

C : E3BD6C0F E3B4A56E E9349EB2 29739374 5522513E B4754483 8D7C035E 9236E8EE
    3A11ED4E 8D2DD6C7 B474342A 1AD1A9F4 433DDB52 CF58F15B 565D4C45 F525711D

W : 00000002 00000000


Hash value :

H : 8D2DD6C7 B474342A 1AD1A9F4 433DDB52 CF58F15B 565D4C45 F525711D


Hash value (byte array):

H : C7 D6 2D 8D 2A 34 74 B4 F4 A9 D1 1A 52 DB 3D 43
    5B F1 58 CF 45 4C 5D 56 1D 71 25 F5
```

B.5 Intermediate States for Shabal-256 (Message A)

```
init

A : 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
```

```
    00000000 00000000 00000000 00000000

B : 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
    00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000

C : 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
    00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000

W : FFFFFFFF FFFFFFFF


block number = -1 : message block

M : 00000100 00000101 00000102 00000103 00000104 00000105 00000106 00000107
    00000108 00000109 0000010A 0000010B 0000010C 0000010D 0000010E 0000010F


block number = -1 : add M to B

A : 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
    00000000 00000000 00000000 00000000

B : 00000100 00000101 00000102 00000103 00000104 00000105 00000106 00000107
    00000108 00000109 0000010A 0000010B 0000010C 0000010D 0000010E 0000010F

C : 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
    00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000

W : FFFFFFFF FFFFFFFF


block number = -1 : xor counter W into A

A : FFFFFFFF FFFFFFFF 00000000 00000000 00000000 00000000 00000000 00000000
    00000000 00000000 00000000 00000000

B : 00000100 00000101 00000102 00000103 00000104 00000105 00000106 00000107
    00000108 00000109 0000010A 0000010B 0000010C 0000010D 0000010E 0000010F

C : 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
    00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000

W : FFFFFFFF FFFFFFFF


block number = -1 : permutation - rotate B

A : FFFFFFFF FFFFFFFF 00000000 00000000 00000000 00000000 00000000 00000000
    00000000 00000000 00000000 00000000

B : 02000000 02020000 02040000 02060000 02080000 020A0000 020C0000 020E0000
    02100000 02120000 02140000 02160000 02180000 021A0000 021C0000 021E0000

C : 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
    00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000

W : FFFFFFFF FFFFFFFF


block number = -1 : permutation (j = 0)

A : AE3F47E2 C68A60D9 C63E700C 04FB82A0 8088FEAE 64D8954B CC08DF9F F2497B68
    7EC08316 F007C07B BBB8A77D 6C368247

B : 06080102 FE78F1B6 12CC57B2 93BBAACD 7B670151 9F336AB4 37EF2060 09AA8497
    851F7CE9 0BDC3F84 406F5882 97E57DB8 55F0B81D 3D419F26 3DF98FF3 FF387D5F
```

```
C : 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
    00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000

W : FFFFFFFF FFFFFFFF


block number = -1 : permutation (j = 1)

A : FF750C93 E5788A46 9BBD7565 CC341456 AD7FFC12 CE83FA1C 6511FD37 E0E9DCD1
    A72272C9 B2929B0B 78EE6F01 C3B82A6F

B : 14F4A4C4 CDE737FE 5675BD47 7A96E075 AE138F94 730BB19D E8CFD03E 2F12DCBE
    0AB40ABF 0D3F0AB1 E49C3B9E 1C0110D8 F96173D7 4BFF3BAF E11D1D2E E166D991

C : 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
    00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000

W : FFFFFFFF FFFFFFFF


block number = -1 : permutation (j = 2)

A : 79F023D8 708745B8 B2D69F0A AB24079A 4C77DDAC B4B92870 88E6ECE2 747657E4
    3395A2D7 5A9826A0 0D578B50 68F97918

B : B555C7FE 3E7106A7 A72A6641 93015272 DA28C30E 696FD97D 9CB6C088 0AFE4119
    A6E0372D 5138C2ED BE216420 B38B89AB 3EA8BA87 3299AE01 30924EF2 55CB35C4

C : 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
    00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000

W : FFFFFFFF FFFFFFFF


block number = -1 : permutation - add C to A

A : 79F023D8 708745B8 B2D69F0A AB24079A 4C77DDAC B4B92870 88E6ECE2 747657E4
    3395A2D7 5A9826A0 0D578B50 68F97918

B : B555C7FE 3E7106A7 A72A6641 93015272 DA28C30E 696FD97D 9CB6C088 0AFE4119
    A6E0372D 5138C2ED BE216420 B38B89AB 3EA8BA87 3299AE01 30924EF2 55CB35C4

C : 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
    00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000

W : FFFFFFFF FFFFFFFF


block number = -1 : subtract M from C

A : 79F023D8 708745B8 B2D69F0A AB24079A 4C77DDAC B4B92870 88E6ECE2 747657E4
    3395A2D7 5A9826A0 0D578B50 68F97918

B : B555C7FE 3E7106A7 A72A6641 93015272 DA28C30E 696FD97D 9CB6C088 0AFE4119
    A6E0372D 5138C2ED BE216420 B38B89AB 3EA8BA87 3299AE01 30924EF2 55CB35C4

C : FFFFFF00 FFFFFEFF FFFFFEFE FFFFFEFD FFFFFEFC FFFFFEFB FFFFFEFA FFFFFEF9
    FFFFFEF8 FFFFFEF7 FFFFFEF6 FFFFFEF5 FFFFFEF4 FFFFFEF3 FFFFFEF2 FFFFFEF1

W : FFFFFFFF FFFFFFFF


block number = -1 : swap B with C

A : 79F023D8 708745B8 B2D69F0A AB24079A 4C77DDAC B4B92870 88E6ECE2 747657E4
    3395A2D7 5A9826A0 0D578B50 68F97918
```

```
B : FFFFFF00 FFFFFEFF FFFFFEFE FFFFFEFD FFFFFEFC FFFFFEFB FFFFFEFA FFFFFEF9
    FFFFFEF8 FFFFFEF7 FFFFFEF6 FFFFFEF5 FFFFFEF4 FFFFFEF3 FFFFFEF2 FFFFFEF1

C : B555C7FE 3E7106A7 A72A6641 93015272 DA28C30E 696FD97D 9CB6C088 0AFE4119
    A6E0372D 5138C2ED BE216420 B38B89AB 3EA8BA87 3299AE01 30924EF2 55CB35C4

W : FFFFFFFF FFFFFFFF


block number = 0 : increment counter W

A : 79F023D8 708745B8 B2D69F0A AB24079A 4C77DDAC B4B92870 88E6ECE2 747657E4
    3395A2D7 5A9826A0 0D578B50 68F97918

B : FFFFFF00 FFFFFEFF FFFFFEFE FFFFFEFD FFFFFEFC FFFFFEFB FFFFFEFA FFFFFEF9
    FFFFFEF8 FFFFFEF7 FFFFFEF6 FFFFFEF5 FFFFFEF4 FFFFFEF3 FFFFFEF2 FFFFFEF1

C : B555C7FE 3E7106A7 A72A6641 93015272 DA28C30E 696FD97D 9CB6C088 0AFE4119
    A6E0372D 5138C2ED BE216420 B38B89AB 3EA8BA87 3299AE01 30924EF2 55CB35C4

W : 00000000 00000000


block number = 0 : message block

M : 00000110 00000111 00000112 00000113 00000114 00000115 00000116 00000117
    00000118 00000119 0000011A 0000011B 0000011C 0000011D 0000011E 0000011F


block number = 0 : add M to B

A : 79F023D8 708745B8 B2D69F0A AB24079A 4C77DDAC B4B92870 88E6ECE2 747657E4
    3395A2D7 5A9826A0 0D578B50 68F97918

B : 00000010 00000010 00000010 00000010 00000010 00000010 00000010 00000010
    00000010 00000010 00000010 00000010 00000010 00000010 00000010 00000010

C : B555C7FE 3E7106A7 A72A6641 93015272 DA28C30E 696FD97D 9CB6C088 0AFE4119
    A6E0372D 5138C2ED BE216420 B38B89AB 3EA8BA87 3299AE01 30924EF2 55CB35C4

W : 00000000 00000000


block number = 0 : xor counter W into A

A : 79F023D8 708745B8 B2D69F0A AB24079A 4C77DDAC B4B92870 88E6ECE2 747657E4
    3395A2D7 5A9826A0 0D578B50 68F97918

B : 00000010 00000010 00000010 00000010 00000010 00000010 00000010 00000010
    00000010 00000010 00000010 00000010 00000010 00000010 00000010 00000010

C : B555C7FE 3E7106A7 A72A6641 93015272 DA28C30E 696FD97D 9CB6C088 0AFE4119
    A6E0372D 5138C2ED BE216420 B38B89AB 3EA8BA87 3299AE01 30924EF2 55CB35C4

W : 00000000 00000000


block number = 0 : permutation - rotate B

A : 79F023D8 708745B8 B2D69F0A AB24079A 4C77DDAC B4B92870 88E6ECE2 747657E4
    3395A2D7 5A9826A0 0D578B50 68F97918

B : 00200000 00200000 00200000 00200000 00200000 00200000 00200000 00200000
    00200000 00200000 00200000 00200000 00200000 00200000 00200000 00200000

C : B555C7FE 3E7106A7 A72A6641 93015272 DA28C30E 696FD97D 9CB6C088 0AFE4119
    A6E0372D 5138C2ED BE216420 B38B89AB 3EA8BA87 3299AE01 30924EF2 55CB35C4
```

```
W : 00000000 00000000


block number = 0 : permutation (j = 0)

A : 7F0490C3 946910EC 827CA63E 409A1072 2B03CF2C FED97162 7F90F2B4 4069B7E1
    CCB74C96 09778D59 6B24AB90 80A11478

B : AA98C924 A5880F94 19DB44E2 61A3149F D4BC30D3 01668E9D 802F0D4B BFD6481E
    3308B369 F6C872A6 949B546F 7F1EEB87 80BB6F3C 6BD6EF13 7DC359C1 BF25EF8D

C : B555C7FE 3E7106A7 A72A6641 93015272 DA28C30E 696FD97D 9CB6C088 0AFE4119
    A6E0372D 5138C2ED BE216420 B38B89AB 3EA8BA87 3299AE01 30924EF2 55CB35C4

W : 00000000 00000000


block number = 0 : permutation (j = 1)

A : D455C979 9F6F4358 7F3D0246 E8A1F260 BE9C04EA 15A315D0 279AC5A8 1C34F0F3
    10A66C48 D8CCDB49 02B680C7 AB645FCF

B : 5923FD91 9DEBD978 66452E00 8510F1DD 4621F210 25FE398C FD1765AF 2B37300D
    4DBB5054 8D0059EA A9F45566 E963DA91 4015256C 3DF13409 23E389D5 9D80D017

C : B555C7FE 3E7106A7 A72A6641 93015272 DA28C30E 696FD97D 9CB6C088 0AFE4119
    A6E0372D 5138C2ED BE216420 B38B89AB 3EA8BA87 3299AE01 30924EF2 55CB35C4

W : 00000000 00000000


block number = 0 : permutation (j = 2)

A : B6A03371 17243406 531455C7 44F0DBD6 EC3CC237 851D8290 D99C1EB9 AED060A3
    C3433D0F 62FD9C1A 02BB7200 5FB743B0

B : B405F031 C4233EBA B3733979 C0DD9D55 C51C28AE A327B8E1 56C56167 ED614433
    88B59D60 60E2CEBA 758B4B8B 83E82A7F BC968828 E6E00BF7 BA839E55 9B491C60

C : B555C7FE 3E7106A7 A72A6641 93015272 DA28C30E 696FD97D 9CB6C088 0AFE4119
    A6E0372D 5138C2ED BE216420 B38B89AB 3EA8BA87 3299AE01 30924EF2 55CB35C4

W : 00000000 00000000


block number = 0 : permutation - add C to A

A : 52F84552 E54B7999 2D8EE3EC B9645191 E0078B86 BB7C44C9 D2B5C1CA B0D2EB8C
    14CE5A45 22AF50DC EFFDBC6B EB21B74A

B : B405F031 C4233EBA B3733979 C0DD9D55 C51C28AE A327B8E1 56C56167 ED614433
    88B59D60 60E2CEBA 758B4B8B 83E82A7F BC968828 E6E00BF7 BA839E55 9B491C60

C : B555C7FE 3E7106A7 A72A6641 93015272 DA28C30E 696FD97D 9CB6C088 0AFE4119
    A6E0372D 5138C2ED BE216420 B38B89AB 3EA8BA87 3299AE01 30924EF2 55CB35C4

W : 00000000 00000000


block number = 0 : subtract M from C

A : 52F84552 E54B7999 2D8EE3EC B9645191 E0078B86 BB7C44C9 D2B5C1CA B0D2EB8C
    14CE5A45 22AF50DC EFFDBC6B EB21B74A

B : B405F031 C4233EBA B3733979 C0DD9D55 C51C28AE A327B8E1 56C56167 ED614433
    88B59D60 60E2CEBA 758B4B8B 83E82A7F BC968828 E6E00BF7 BA839E55 9B491C60
```

```
C : B555C6EE 3E710596 A72A652F 9301515F DA28C1FA 696FD868 9CB6BF72 0AFE4002
    A6E03615 5138C1D4 BE216306 B38B8890 3EA8B96B 3299ACE4 30924DD4 55CB34A5

W : 00000000 00000000


block number = 0 : swap B with C

A : 52F84552 E54B7999 2D8EE3EC B9645191 E0078B86 BB7C44C9 D2B5C1CA B0D2EB8C
    14CE5A45 22AF50DC EFFDBC6B EB21B74A

B : B555C6EE 3E710596 A72A652F 9301515F DA28C1FA 696FD868 9CB6BF72 0AFE4002
    A6E03615 5138C1D4 BE216306 B38B8890 3EA8B96B 3299ACE4 30924DD4 55CB34A5

C : B405F031 C4233EBA B3733979 C0DD9D55 C51C28AE A327B8E1 56C56167 ED614433
    88B59D60 60E2CEBA 758B4B8B 83E82A7F BC968828 E6E00BF7 BA839E55 9B491C60

W : 00000000 00000000


block number = 1 : increment counter W

A : 52F84552 E54B7999 2D8EE3EC B9645191 E0078B86 BB7C44C9 D2B5C1CA B0D2EB8C
    14CE5A45 22AF50DC EFFDBC6B EB21B74A

B : B555C6EE 3E710596 A72A652F 9301515F DA28C1FA 696FD868 9CB6BF72 0AFE4002
    A6E03615 5138C1D4 BE216306 B38B8890 3EA8B96B 3299ACE4 30924DD4 55CB34A5

C : B405F031 C4233EBA B3733979 C0DD9D55 C51C28AE A327B8E1 56C56167 ED614433
    88B59D60 60E2CEBA 758B4B8B 83E82A7F BC968828 E6E00BF7 BA839E55 9B491C60

W : 00000001 00000000


block number = 1 : message block

M : 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
    00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000


block number = 1 : add M to B

A : 52F84552 E54B7999 2D8EE3EC B9645191 E0078B86 BB7C44C9 D2B5C1CA B0D2EB8C
    14CE5A45 22AF50DC EFFDBC6B EB21B74A

B : B555C6EE 3E710596 A72A652F 9301515F DA28C1FA 696FD868 9CB6BF72 0AFE4002
    A6E03615 5138C1D4 BE216306 B38B8890 3EA8B96B 3299ACE4 30924DD4 55CB34A5

C : B405F031 C4233EBA B3733979 C0DD9D55 C51C28AE A327B8E1 56C56167 ED614433
    88B59D60 60E2CEBA 758B4B8B 83E82A7F BC968828 E6E00BF7 BA839E55 9B491C60

W : 00000001 00000000


block number = 1 : xor counter W into A

A : 52F84553 E54B7999 2D8EE3EC B9645191 E0078B86 BB7C44C9 D2B5C1CA B0D2EB8C
    14CE5A45 22AF50DC EFFDBC6B EB21B74A

B : B555C6EE 3E710596 A72A652F 9301515F DA28C1FA 696FD868 9CB6BF72 0AFE4002
    A6E03615 5138C1D4 BE216306 B38B8890 3EA8B96B 3299ACE4 30924DD4 55CB34A5

C : B405F031 C4233EBA B3733979 C0DD9D55 C51C28AE A327B8E1 56C56167 ED614433
    88B59D60 60E2CEBA 758B4B8B 83E82A7F BC968828 E6E00BF7 BA839E55 9B491C60

W : 00000001 00000000
```

```
block number = 1 : permutation - rotate B

A : 52F84553 E54B7999 2D8EE3EC B9645191 E0078B86 BB7C44C9 D2B5C1CA B0D2EB8C
    14CE5A45 22AF50DC EFFDBC6B EB21B74A

B : 8DDD6AAB 0B2C7CE2 CA5F4E54 A2BF2602 83F5B451 B0D0D2DF 7EE5396D 800415FC
    6C2B4DC0 83A8A271 C60D7C42 11216717 72D67D51 59C86533 9BA86124 694AAB96

C : B405F031 C4233EBA B3733979 C0DD9D55 C51C28AE A327B8E1 56C56167 ED614433
    88B59D60 60E2CEBA 758B4B8B 83E82A7F BC968828 E6E00BF7 BA839E55 9B491C60

W : 00000001 00000000


block number = 1 : permutation (j = 0)

A : 6C7E9B27 87B2F2E5 31E5DEFF 6942E14C BEB25D8B 33D83C6D 90F3CDCD 21D06696
    2FE614D9 3D0961C9 CD249DAA 9B2F0544

B : 8DE67622 7A56AE20 0E25B32E 7D7CA4C6 46A6CAD7 AD86662D 92C640E8 DE27B290
    084F70A6 C5A7DAD5 BEC19AD0 46923495 762D9E7A CBDDC77C F94AE349 4428499F

C : B405F031 C4233EBA B3733979 C0DD9D55 C51C28AE A327B8E1 56C56167 ED614433
    88B59D60 60E2CEBA 758B4B8B 83E82A7F BC968828 E6E00BF7 BA839E55 9B491C60

W : 00000001 00000000


block number = 1 : permutation (j = 1)

A : D154BCD2 8029A7A8 BD8F33D3 8C8CB879 0EDFB793 F99B963E 9CA7B797 384AA841
    6E902BE0 DC3FF5DA 84C2EC43 0CEC8961

B : 656519B9 85CD79AD D4468C4E 4E1F2B1E 1C2241B1 78CCC67E 5EB1926D 4F5C13BF
    3E35A261 F499EDFC 3FF3F98D FE572EAC 1D7B7498 91DFE738 91CD8EFB 4FE5C480

C : B405F031 C4233EBA B3733979 C0DD9D55 C51C28AE A327B8E1 56C56167 ED614433
    88B59D60 60E2CEBA 758B4B8B 83E82A7F BC968828 E6E00BF7 BA839E55 9B491C60

W : 00000001 00000000


block number = 1 : permutation (j = 2)

A : 6F764332 149C2AD3 D6586B4D 242A0474 491260B1 8EF3ADBF 5C1C3969 11CF9D58
    4854D811 FAC675BA BA5C2B36 13DC2FD0

B : C376688E D70D68BB A00A2F8E A507C0C9 A8CD3FAF 1AFA59D0 94C4B068 456DDCF5
    CA86DB8C 983F89B9 DC04358C 129E3FFE 8D5DCEDE 26864434 6638C93E 73E8592F

C : B405F031 C4233EBA B3733979 C0DD9D55 C51C28AE A327B8E1 56C56167 ED614433
    88B59D60 60E2CEBA 758B4B8B 83E82A7F BC968828 E6E00BF7 BA839E55 9B491C60

W : 00000001 00000000


block number = 1 : permutation - add C to A

A : 4F852766 4A54CBDA 24836EDF E8E63DA9 929A5E99 90CB63FE 2449FFBE 919383C3
    7A7BE418 052EC3F0 A546BEC8 9AB07B17

B : C376688E D70D68BB A00A2F8E A507C0C9 A8CD3FAF 1AFA59D0 94C4B068 456DDCF5
    CA86DB8C 983F89B9 DC04358C 129E3FFE 8D5DCEDE 26864434 6638C93E 73E8592F

C : B405F031 C4233EBA B3733979 C0DD9D55 C51C28AE A327B8E1 56C56167 ED614433
```

```
    88B59D60 60E2CEBA 758B4B8B 83E82A7F BC968828 E6E00BF7 BA839E55 9B491C60

W : 00000001 00000000


block number = 1 : subtract M from C

A : 4F852766 4A54CBDA 24836EDF E8E63DA9 929A5E99 90CB63FE 2449FFBE 919383C3
    7A7BE418 052EC3F0 A546BEC8 9AB07B17

B : C376688E D70D68BB A00A2F8E A507C0C9 A8CD3FAF 1AFA59D0 94C4B068 456DDCF5
    CA86DB8C 983F89B9 DC04358C 129E3FFE 8D5DCEDE 26864434 6638C93E 73E8592F

C : B405F031 C4233EBA B3733979 C0DD9D55 C51C28AE A327B8E1 56C56167 ED614433
    88B59D60 60E2CEBA 758B4B8B 83E82A7F BC968828 E6E00BF7 BA839E55 9B491C60

W : 00000001 00000000


block number = 1 : swap B with C

A : 4F852766 4A54CBDA 24836EDF E8E63DA9 929A5E99 90CB63FE 2449FFBE 919383C3
    7A7BE418 052EC3F0 A546BEC8 9AB07B17

B : B405F031 C4233EBA B3733979 C0DD9D55 C51C28AE A327B8E1 56C56167 ED614433
    88B59D60 60E2CEBA 758B4B8B 83E82A7F BC968828 E6E00BF7 BA839E55 9B491C60

C : C376688E D70D68BB A00A2F8E A507C0C9 A8CD3FAF 1AFA59D0 94C4B068 456DDCF5
    CA86DB8C 983F89B9 DC04358C 129E3FFE 8D5DCEDE 26864434 6638C93E 73E8592F

W : 00000001 00000000


block number = 2 : increment counter W

A : 4F852766 4A54CBDA 24836EDF E8E63DA9 929A5E99 90CB63FE 2449FFBE 919383C3
    7A7BE418 052EC3F0 A546BEC8 9AB07B17

B : B405F031 C4233EBA B3733979 C0DD9D55 C51C28AE A327B8E1 56C56167 ED614433
    88B59D60 60E2CEBA 758B4B8B 83E82A7F BC968828 E6E00BF7 BA839E55 9B491C60

C : C376688E D70D68BB A00A2F8E A507C0C9 A8CD3FAF 1AFA59D0 94C4B068 456DDCF5
    CA86DB8C 983F89B9 DC04358C 129E3FFE 8D5DCEDE 26864434 6638C93E 73E8592F

W : 00000002 00000000


block number = 2 : message block

M : 00000080 00000000 00000000 00000000 00000000 00000000 00000000 00000000
    00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000


block number = 2 : add M to B

A : 4F852766 4A54CBDA 24836EDF E8E63DA9 929A5E99 90CB63FE 2449FFBE 919383C3
    7A7BE418 052EC3F0 A546BEC8 9AB07B17

B : B405F0B1 C4233EBA B3733979 C0DD9D55 C51C28AE A327B8E1 56C56167 ED614433
    88B59D60 60E2CEBA 758B4B8B 83E82A7F BC968828 E6E00BF7 BA839E55 9B491C60

C : C376688E D70D68BB A00A2F8E A507C0C9 A8CD3FAF 1AFA59D0 94C4B068 456DDCF5
    CA86DB8C 983F89B9 DC04358C 129E3FFE 8D5DCEDE 26864434 6638C93E 73E8592F

W : 00000002 00000000
```

```
block number = 2 : xor counter W into A

A : 4F852764 4A54CBDA 24836EDF E8E63DA9 929A5E99 90CB63FE 2449FFBE 919383C3
    7A7BE418 052EC3F0 A546BEC8 9AB07B17

B : B405F0B1 C4233EBA B3733979 C0DD9D55 C51C28AE A327B8E1 56C56167 ED614433
    88B59D60 60E2CEBA 758B4B8B 83E82A7F BC968828 E6E00BF7 BA839E55 9B491C60

C : C376688E D70D68BB A00A2F8E A507C0C9 A8CD3FAF 1AFA59D0 94C4B068 456DDCF5
    CA86DB8C 983F89B9 DC04358C 129E3FFE 8D5DCEDE 26864434 6638C93E 73E8592F

W : 00000002 00000000


block number = 2 : permutation - rotate B

A : 4F852764 4A54CBDA 24836EDF E8E63DA9 929A5E99 90CB63FE 2449FFBE 919383C3
    7A7BE418 052EC3F0 A546BEC8 9AB07B17

B : E163680B 7D758846 72F366E6 3AAB81BB 515D8A38 71C3464F C2CEAD8A 8867DAC2
    3AC1116B 9D74C1C5 9716EB16 54FF07D0 1051792D 17EFCDC0 3CAB7507 38C13692

C : C376688E D70D68BB A00A2F8E A507C0C9 A8CD3FAF 1AFA59D0 94C4B068 456DDCF5
    CA86DB8C 983F89B9 DC04358C 129E3FFE 8D5DCEDE 26864434 6638C93E 73E8592F

W : 00000002 00000000


block number = 2 : permutation (j = 0)

A : 47280149 E0472C08 4C728C37 96F8D4EA 12DDB762 FCA64123 B97C739F 792DA29A
    2E65DDAA D6DBED1F 4D3A2665 00F33A7D

B : 13CC581D C0361D87 AB8A9517 E2320200 4F995CED E0DF3242 C31ED775 961DE8E0
    A4180083 13CD916B 9CE80FB7 56F2CA22 98750CEC 30674877 CADB99C6 18854631

C : C376688E D70D68BB A00A2F8E A507C0C9 A8CD3FAF 1AFA59D0 94C4B068 456DDCF5
    CA86DB8C 983F89B9 DC04358C 129E3FFE 8D5DCEDE 26864434 6638C93E 73E8592F

W : 00000002 00000000


block number = 2 : permutation (j = 1)

A : 774A249D EBA2B79B E65FFD6E DD6651EA 7CD9B57D 0F492665 368BCCC9 36F9F21E
    9BABB0B8 69550AAF ED7CDA53 95787B45

B : 9CF247BE 3B5779CD 3F68E196 8B6F3469 FB66F69D 571491D5 94BE8B47 46BC557B
    C085DA65 33C66AB2 20701DFE 8F7C3A51 B3CC535B 90784974 5CC300BB F80C8183

C : C376688E D70D68BB A00A2F8E A507C0C9 A8CD3FAF 1AFA59D0 94C4B068 456DDCF5
    CA86DB8C 983F89B9 DC04358C 129E3FFE 8D5DCEDE 26864434 6638C93E 73E8592F

W : 00000002 00000000


block number = 2 : permutation (j = 2)

A : 05F99A5A 361D65F4 FFFBD130 CD7FEDF4 232C8A7D 59E6C9C8 48671625 AC36413E
    0DF498ED 2BD35EEB CF993C21 E244DA6C

B : D017FECE 1B46A796 8B530864 1DDD5142 0CCB889E 67CBB9A1 29793840 BFF8B8FD
    5DD8C149 C195E353 F778D226 4D31CA62 9593C1A5 F4DC33FD 89E0C2A8 EDA22694

C : C376688E D70D68BB A00A2F8E A507C0C9 A8CD3FAF 1AFA59D0 94C4B068 456DDCF5
    CA86DB8C 983F89B9 DC04358C 129E3FFE 8D5DCEDE 26864434 6638C93E 73E8592F
```

```
W : 00000002 00000000


block number = 2 : permutation - add C to A

A : 3187F450 2FBEDD0F 1889D7EF 68879728 818A816A 90B14D91 D2AE6269 BD0956C0
    0B0876A9 2C854904 A95963DE B946899E

B : D017FECE 1B46A796 8B530864 1DDD5142 0CCB889E 67CBB9A1 29793840 BFF8B8FD
    5DD8C149 C195E353 F778D226 4D31CA62 9593C1A5 F4DC33FD 89E0C2A8 EDA22694

C : C376688E D70D68BB A00A2F8E A507C0C9 A8CD3FAF 1AFA59D0 94C4B068 456DDCF5
    CA86DB8C 983F89B9 DC04358C 129E3FFE 8D5DCEDE 26864434 6638C93E 73E8592F

W : 00000002 00000000


block number = 2 : subtract M from C

A : 3187F450 2FBEDD0F 1889D7EF 68879728 818A816A 90B14D91 D2AE6269 BD0956C0
    0B0876A9 2C854904 A95963DE B946899E

B : D017FECE 1B46A796 8B530864 1DDD5142 0CCB889E 67CBB9A1 29793840 BFF8B8FD
    5DD8C149 C195E353 F778D226 4D31CA62 9593C1A5 F4DC33FD 89E0C2A8 EDA22694

C : C376680E D70D68BB A00A2F8E A507C0C9 A8CD3FAF 1AFA59D0 94C4B068 456DDCF5
    CA86DB8C 983F89B9 DC04358C 129E3FFE 8D5DCEDE 26864434 6638C93E 73E8592F

W : 00000002 00000000


block number = 2 : swap B with C

A : 3187F450 2FBEDD0F 1889D7EF 68879728 818A816A 90B14D91 D2AE6269 BD0956C0
    0B0876A9 2C854904 A95963DE B946899E

B : C376680E D70D68BB A00A2F8E A507C0C9 A8CD3FAF 1AFA59D0 94C4B068 456DDCF5
    CA86DB8C 983F89B9 DC04358C 129E3FFE 8D5DCEDE 26864434 6638C93E 73E8592F

C : D017FECE 1B46A796 8B530864 1DDD5142 0CCB889E 67CBB9A1 29793840 BFF8B8FD
    5DD8C149 C195E353 F778D226 4D31CA62 9593C1A5 F4DC33FD 89E0C2A8 EDA22694

W : 00000002 00000000


n0_final = 0 : add M to B

A : 3187F450 2FBEDD0F 1889D7EF 68879728 818A816A 90B14D91 D2AE6269 BD0956C0
    0B0876A9 2C854904 A95963DE B946899E

B : C376688E D70D68BB A00A2F8E A507C0C9 A8CD3FAF 1AFA59D0 94C4B068 456DDCF5
    CA86DB8C 983F89B9 DC04358C 129E3FFE 8D5DCEDE 26864434 6638C93E 73E8592F

C : D017FECE 1B46A796 8B530864 1DDD5142 0CCB889E 67CBB9A1 29793840 BFF8B8FD
    5DD8C149 C195E353 F778D226 4D31CA62 9593C1A5 F4DC33FD 89E0C2A8 EDA22694

W : 00000002 00000000


n0_final = 0 : xor counter W into A

A : 3187F452 2FBEDD0F 1889D7EF 68879728 818A816A 90B14D91 D2AE6269 BD0956C0
    0B0876A9 2C854904 A95963DE B946899E

B : C376688E D70D68BB A00A2F8E A507C0C9 A8CD3FAF 1AFA59D0 94C4B068 456DDCF5
    CA86DB8C 983F89B9 DC04358C 129E3FFE 8D5DCEDE 26864434 6638C93E 73E8592F
```

```
C : D017FECE 1B46A796 8B530864 1DDD5142 0CCB889E 67CBB9A1 29793840 BFF8B8FD
    5DD8C149 C195E353 F778D226 4D31CA62 9593C1A5 F4DC33FD 89E0C2A8 EDA22694

W : 00000002 00000000


n0_final = 0 : permutation - rotate B

A : 3187F452 2FBEDD0F 1889D7EF 68879728 818A816A 90B14D91 D2AE6269 BD0956C0
    0B0876A9 2C854904 A95963DE B946899E

B : D11D86EC D177AE1A 5F1D4014 81934A0F 7F5F519A B3A035F4 60D12989 B9EA8ADB
    B719950D 1373307F 6B19B808 7FFC253C 9DBD1ABB 88684D0C 927CCC71 B25EE7D0

C : D017FECE 1B46A796 8B530864 1DDD5142 0CCB889E 67CBB9A1 29793840 BFF8B8FD
    5DD8C149 C195E353 F778D226 4D31CA62 9593C1A5 F4DC33FD 89E0C2A8 EDA22694

W : 00000002 00000000


n0_final = 0 : permutation (j = 0)

A : 64D53798 30B6E019 E5F13356 422A0182 D89FD7C5 11EF0E93 B713DA4F 9D51A1BF
    89102A8C CA86DD00 2C5A70C9 E59682E4

B : 5A754140 FDD13BE5 D3E842DA 6DE96B95 D9DE8B0E 89509A85 894E76A2 117B4BF7
    18DCFF68 139F4201 0596FF26 E5913763 A050FD10 DF9985FF 3EF7544A D96831DC

C : D017FECE 1B46A796 8B530864 1DDD5142 0CCB889E 67CBB9A1 29793840 BFF8B8FD
    5DD8C149 C195E353 F778D226 4D31CA62 9593C1A5 F4DC33FD 89E0C2A8 EDA22694

W : 00000002 00000000


n0_final = 0 : permutation (j = 1)

A : 47DE9969 50A707E4 872B7E5F AE533A8E 84218E27 8A923B0E D74F3A07 F20E2576
    09DB05EC 6D7E3C4B 78EF080E 0541AF6D

B : 1595E3AE 5778E6F4 7BD8C2EA B4E6B380 4599EC0E 8020F6BF 958C1AB4 D848C77C
    89989846 88667C19 73F97FEC 9A8EABB6 3B7F8BF9 CA5ECF0E 555E6D6C BF21B930

C : D017FECE 1B46A796 8B530864 1DDD5142 0CCB889E 67CBB9A1 29793840 BFF8B8FD
    5DD8C149 C195E353 F778D226 4D31CA62 9593C1A5 F4DC33FD 89E0C2A8 EDA22694

W : 00000002 00000000


n0_final = 0 : permutation (j = 2)

A : 9136A866 BA3C98DF 001403F9 D30AC84B 2E00660D 5CC346E5 03D364DB 9390F4E8
    3F31482A FD4A8250 B78A73F2 C4476199

B : 0A175FB6 DE711F02 46026474 D31144FE E5FA8F85 45828A5F D4F3C96F 9C64B94D
    C2CEA97F B3F04129 1BDE64FC 59725C7A B631A027 9608E3B2 E2C956D5 45FBEC07

C : D017FECE 1B46A796 8B530864 1DDD5142 0CCB889E 67CBB9A1 29793840 BFF8B8FD
    5DD8C149 C195E353 F778D226 4D31CA62 9593C1A5 F4DC33FD 89E0C2A8 EDA22694

W : 00000002 00000000


n0_final = 0 : permutation - add C to A

A : E9E7EA9E 2CB3E1F0 7802992D 11B7CB97 F97896E0 977F8F9A 487BA965 3FD607B2
    6A391CCB FD828DDC D5C844E3 6F1A2EA7
```

```
B : 0A175FB6 DE711F02 46026474 D31144FE E5FA8F85 45828A5F D4F3C96F 9C64B94D
    C2CEA97F B3F04129 1BDE64FC 59725C7A B631A027 9608E3B2 E2C956D5 45FBEC07

C : D017FECE 1B46A796 8B530864 1DDD5142 0CCB889E 67CBB9A1 29793840 BFF8B8FD
    5DD8C149 C195E353 F778D226 4D31CA62 9593C1A5 F4DC33FD 89E0C2A8 EDA22694

W : 00000002 00000000


n0_final = 0 : subtract M from C

A : E9E7EA9E 2CB3E1F0 7802992D 11B7CB97 F97896E0 977F8F9A 487BA965 3FD607B2
    6A391CCB FD828DDC D5C844E3 6F1A2EA7

B : 0A175FB6 DE711F02 46026474 D31144FE E5FA8F85 45828A5F D4F3C96F 9C64B94D
    C2CEA97F B3F04129 1BDE64FC 59725C7A B631A027 9608E3B2 E2C956D5 45FBEC07

C : D017FE4E 1B46A796 8B530864 1DDD5142 0CCB889E 67CBB9A1 29793840 BFF8B8FD
    5DD8C149 C195E353 F778D226 4D31CA62 9593C1A5 F4DC33FD 89E0C2A8 EDA22694

W : 00000002 00000000


n0_final = 0 : swap B with C

A : E9E7EA9E 2CB3E1F0 7802992D 11B7CB97 F97896E0 977F8F9A 487BA965 3FD607B2
    6A391CCB FD828DDC D5C844E3 6F1A2EA7

B : D017FE4E 1B46A796 8B530864 1DDD5142 0CCB889E 67CBB9A1 29793840 BFF8B8FD
    5DD8C149 C195E353 F778D226 4D31CA62 9593C1A5 F4DC33FD 89E0C2A8 EDA22694

C : 0A175FB6 DE711F02 46026474 D31144FE E5FA8F85 45828A5F D4F3C96F 9C64B94D
    C2CEA97F B3F04129 1BDE64FC 59725C7A B631A027 9608E3B2 E2C956D5 45FBEC07

W : 00000002 00000000


n0_final = 1 : add M to B

A : E9E7EA9E 2CB3E1F0 7802992D 11B7CB97 F97896E0 977F8F9A 487BA965 3FD607B2
    6A391CCB FD828DDC D5C844E3 6F1A2EA7

B : D017FECE 1B46A796 8B530864 1DDD5142 0CCB889E 67CBB9A1 29793840 BFF8B8FD
    5DD8C149 C195E353 F778D226 4D31CA62 9593C1A5 F4DC33FD 89E0C2A8 EDA22694

C : 0A175FB6 DE711F02 46026474 D31144FE E5FA8F85 45828A5F D4F3C96F 9C64B94D
    C2CEA97F B3F04129 1BDE64FC 59725C7A B631A027 9608E3B2 E2C956D5 45FBEC07

W : 00000002 00000000


n0_final = 1 : xor counter W into A

A : E9E7EA9C 2CB3E1F0 7802992D 11B7CB97 F97896E0 977F8F9A 487BA965 3FD607B2
    6A391CCB FD828DDC D5C844E3 6F1A2EA7

B : D017FECE 1B46A796 8B530864 1DDD5142 0CCB889E 67CBB9A1 29793840 BFF8B8FD
    5DD8C149 C195E353 F778D226 4D31CA62 9593C1A5 F4DC33FD 89E0C2A8 EDA22694

C : 0A175FB6 DE711F02 46026474 D31144FE E5FA8F85 45828A5F D4F3C96F 9C64B94D
    C2CEA97F B3F04129 1BDE64FC 59725C7A B631A027 9608E3B2 E2C956D5 45FBEC07

W : 00000002 00000000


n0_final = 1 : permutation - rotate B
```

```
A : E9E7EA9C 2CB3E1F0 7802992D 11B7CB97 F97896E0 977F8F9A 487BA965 3FD607B2
    6A391CCB FD828DDC D5C844E3 6F1A2EA7

B : FD9DA02F 4F2C368D 10C916A6 A2843BBA 113C1997 7342CF97 708052F2 71FB7FF1
    8292BBB1 C6A7832B A44DEEF1 94C49A63 834B2B27 67FBE9B8 855113C1 4D29DB44

C : 0A175FB6 DE711F02 46026474 D31144FE E5FA8F85 45828A5F D4F3C96F 9C64B94D
    C2CEA97F B3F04129 1BDE64FC 59725C7A B631A027 9608E3B2 E2C956D5 45FBEC07

W : 00000002 00000000


n0_final = 1 : permutation (j = 0)

A : 90CAB5E7 8F875CB3 7A5A2121 F5BF07BB 45942608 DD5BB4F9 405C941D BBD9579C
    7FDBF0BD F84551D3 D522845D 0EE87DA8

B : FBBB55F7 0BD7F336 8E2586BB D260690D 9813EAD9 C421D428 5EA3CE06 A7D05781
    85017821 8AF5A87B 6246A641 D89EB690 69A31C57 BF8F703C 8F07F95D 90134ECC

C : 0A175FB6 DE711F02 46026474 D31144FE E5FA8F85 45828A5F D4F3C96F 9C64B94D
    C2CEA97F B3F04129 1BDE64FC 59725C7A B631A027 9608E3B2 E2C956D5 45FBEC07

W : 00000002 00000000


n0_final = 1 : permutation (j = 1)

A : D48B50A9 54EA4D22 29DBD192 1EEFCACC D3335894 F7A1F666 EEF7923E 89CDAB95
    49461B6D 77D67895 B2A65404 8156AAFF

B : 0B7C3F51 4009CD8C 3F9857DE 1849F5B6 869E3121 006A2F3B F01E37F7 3109FA03
    21765F15 BEFEE22A 12A962EF 502D5812 FF8A9FC5 7740E9E0 0F079F7A 5614C9F3

C : 0A175FB6 DE711F02 46026474 D31144FE E5FA8F85 45828A5F D4F3C96F 9C64B94D
    C2CEA97F B3F04129 1BDE64FC 59725C7A B631A027 9608E3B2 E2C956D5 45FBEC07

W : 00000002 00000000


n0_final = 1 : permutation (j = 2)

A : 0A2A24C8 F8586095 94E8187F 0096CBFB 89255366 02C67A5E 2CDBC68C 62CF0AE1
    6C5402F0 3A43CB77 B61ED7D0 D30D9311

B : 66DF9771 E4D309BD 6377D5E2 48F253E3 F8E9B974 0773C11C 8B2B886F 9D7AC002
    343612B3 80C441F4 F676FCAD 3D6A453A 6CBEC284 2B3DE748 57EE16DB 80DBFF08

C : 0A175FB6 DE711F02 46026474 D31144FE E5FA8F85 45828A5F D4F3C96F 9C64B94D
    C2CEA97F B3F04129 1BDE64FC 59725C7A B631A027 9608E3B2 E2C956D5 45FBEC07

W : 00000002 00000000


n0_final = 1 : permutation - add C to A

A : 7CA9B247 9E9BEFF7 4EE4A592 FE5650B3 3E973DB8 B5A71318 04BFB116 99A39DC0
    353C5DB5 993EA4A2 459A870A A6A91851

B : 66DF9771 E4D309BD 6377D5E2 48F253E3 F8E9B974 0773C11C 8B2B886F 9D7AC002
    343612B3 80C441F4 F676FCAD 3D6A453A 6CBEC284 2B3DE748 57EE16DB 80DBFF08

C : 0A175FB6 DE711F02 46026474 D31144FE E5FA8F85 45828A5F D4F3C96F 9C64B94D
    C2CEA97F B3F04129 1BDE64FC 59725C7A B631A027 9608E3B2 E2C956D5 45FBEC07

W : 00000002 00000000
```

```
n0_final = 1 : subtract M from C

A : 7CA9B247 9E9BEFF7 4EE4A592 FE5650B3 3E973DB8 B5A71318 04BFB116 99A39DC0
    353C5DB5 993EA4A2 459A870A A6A91851

B : 66DF9771 E4D309BD 6377D5E2 48F253E3 F8E9B974 0773C11C 8B2B886F 9D7AC002
    343612B3 80C441F4 F676FCAD 3D6A453A 6CBEC284 2B3DE748 57EE16DB 80DBFF08

C : 0A175F36 DE711F02 46026474 D31144FE E5FA8F85 45828A5F D4F3C96F 9C64B94D
    C2CEA97F B3F04129 1BDE64FC 59725C7A B631A027 9608E3B2 E2C956D5 45FBEC07

W : 00000002 00000000


n0_final = 1 : swap B with C

A : 7CA9B247 9E9BEFF7 4EE4A592 FE5650B3 3E973DB8 B5A71318 04BFB116 99A39DC0
    353C5DB5 993EA4A2 459A870A A6A91851

B : 0A175F36 DE711F02 46026474 D31144FE E5FA8F85 45828A5F D4F3C96F 9C64B94D
    C2CEA97F B3F04129 1BDE64FC 59725C7A B631A027 9608E3B2 E2C956D5 45FBEC07

C : 66DF9771 E4D309BD 6377D5E2 48F253E3 F8E9B974 0773C11C 8B2B886F 9D7AC002
    343612B3 80C441F4 F676FCAD 3D6A453A 6CBEC284 2B3DE748 57EE16DB 80DBFF08

W : 00000002 00000000


n0_final = 2 : add M to B

A : 7CA9B247 9E9BEFF7 4EE4A592 FE5650B3 3E973DB8 B5A71318 04BFB116 99A39DC0
    353C5DB5 993EA4A2 459A870A A6A91851

B : 0A175FB6 DE711F02 46026474 D31144FE E5FA8F85 45828A5F D4F3C96F 9C64B94D
    C2CEA97F B3F04129 1BDE64FC 59725C7A B631A027 9608E3B2 E2C956D5 45FBEC07

C : 66DF9771 E4D309BD 6377D5E2 48F253E3 F8E9B974 0773C11C 8B2B886F 9D7AC002
    343612B3 80C441F4 F676FCAD 3D6A453A 6CBEC284 2B3DE748 57EE16DB 80DBFF08

W : 00000002 00000000


n0_final = 2 : xor counter W into A

A : 7CA9B245 9E9BEFF7 4EE4A592 FE5650B3 3E973DB8 B5A71318 04BFB116 99A39DC0
    353C5DB5 993EA4A2 459A870A A6A91851

B : 0A175FB6 DE711F02 46026474 D31144FE E5FA8F85 45828A5F D4F3C96F 9C64B94D
    C2CEA97F B3F04129 1BDE64FC 59725C7A B631A027 9608E3B2 E2C956D5 45FBEC07

C : 66DF9771 E4D309BD 6377D5E2 48F253E3 F8E9B974 0773C11C 8B2B886F 9D7AC002
    343612B3 80C441F4 F676FCAD 3D6A453A 6CBEC284 2B3DE748 57EE16DB 80DBFF08

W : 00000002 00000000


n0_final = 2 : permutation - rotate B

A : 7CA9B245 9E9BEFF7 4EE4A592 FE5650B3 3E973DB8 B5A71318 04BFB116 99A39DC0
    353C5DB5 993EA4A2 459A870A A6A91851

B : BF6C142E 3E05BCE2 C8E88C04 89FDA622 1F0BCBF5 14BE8B05 92DFA9E7 729B38C9
    52FF859D 825367E0 C9F837BC B8F4B2E4 404F6C63 C7652C11 ADABC592 D80E8BF7

C : 66DF9771 E4D309BD 6377D5E2 48F253E3 F8E9B974 0773C11C 8B2B886F 9D7AC002
    343612B3 80C441F4 F676FCAD 3D6A453A 6CBEC284 2B3DE748 57EE16DB 80DBFF08
```

```
W : 00000002 00000000


n0_final = 2 : permutation (j = 0)

A : 2EB1121F 2F42E458 B41B281A 218E48E7 952E465D 23650000 751EC9DA DD262DAE
    C2774FC1 B28D8D45 5DB96F4E CFB0AEF3

B : 9AB83DC5 FA9D8467 DE58590A 5A46FFE4 54C62E48 F5E7E9F5 AF5E65EA C7EFA3C3
    9877BB04 49D4BD7B 31B6FFC8 41A634C5 51D03526 5E774384 10B35CC0 6E6CA0F7

C : 66DF9771 E4D309BD 6377D5E2 48F253E3 F8E9B974 0773C11C 8B2B886F 9D7AC002
    343612B3 80C441F4 F676FCAD 3D6A453A 6CBEC284 2B3DE748 57EE16DB 80DBFF08

W : 00000002 00000000


n0_final = 2 : permutation (j = 1)

A : 4EFB4B09 A0E2C3B8 B4E8EB92 7E77C334 DA205714 956A50C4 403A77F8 4A3A786D
    AAB74DE4 86726A02 2D88B2F1 C0686507

B : 9274CA46 D241ACA1 BC2E7F21 A4FEF233 FCC4EE8B 92424616 8CCB86DB B048DD7F
    81EBC2FF CCB446B1 287AEBFD 02C45541 867FC2A7 D67B2833 9EA33187 691CC67C

C : 66DF9771 E4D309BD 6377D5E2 48F253E3 F8E9B974 0773C11C 8B2B886F 9D7AC002
    343612B3 80C441F4 F676FCAD 3D6A453A 6CBEC284 2B3DE748 57EE16DB 80DBFF08

W : 00000002 00000000


n0_final = 2 : permutation (j = 2)

A : 36818005 044E6515 C35442A8 8574DDFC 3C20F5DA FC2D15B6 D6DA9553 FDD91DF3
    FA5E5BA4 558CF4A3 DACE3FB9 8F924C9E

B : 57E837B3 3B2C6ACA E0358DC2 2BD758E9 30F7A2ED DF3516C7 253CB0E0 1A1A98FC
    C0088FDA 9ABA672A 79D0BD56 07AE488E 095E2114 06855B3B 1877A349 A2543F99

C : 66DF9771 E4D309BD 6377D5E2 48F253E3 F8E9B974 0773C11C 8B2B886F 9D7AC002
    343612B3 80C441F4 F676FCAD 3D6A453A 6CBEC284 2B3DE748 57EE16DB 80DBFF08

W : 00000002 00000000


n0_final = 2 : permutation - add C to A

A : 3DBA182A D0D6787E DAD8F4C9 CC065328 A36A08C7 902C794E 43E5A220 E2F378F1
    1E35B4C3 EF6B834E 8E442A11 6922E895

B : 57E837B3 3B2C6ACA E0358DC2 2BD758E9 30F7A2ED DF3516C7 253CB0E0 1A1A98FC
    C0088FDA 9ABA672A 79D0BD56 07AE488E 095E2114 06855B3B 1877A349 A2543F99

C : 66DF9771 E4D309BD 6377D5E2 48F253E3 F8E9B974 0773C11C 8B2B886F 9D7AC002
    343612B3 80C441F4 F676FCAD 3D6A453A 6CBEC284 2B3DE748 57EE16DB 80DBFF08

W : 00000002 00000000


n0_final = 2 : subtract M from C

A : 3DBA182A D0D6787E DAD8F4C9 CC065328 A36A08C7 902C794E 43E5A220 E2F378F1
    1E35B4C3 EF6B834E 8E442A11 6922E895

B : 57E837B3 3B2C6ACA E0358DC2 2BD758E9 30F7A2ED DF3516C7 253CB0E0 1A1A98FC
    C0088FDA 9ABA672A 79D0BD56 07AE488E 095E2114 06855B3B 1877A349 A2543F99
```

```
C : 66DF96F1 E4D309BD 6377D5E2 48F253E3 F8E9B974 0773C11C 8B2B886F 9D7AC002
    343612B3 80C441F4 F676FCAD 3D6A453A 6CBEC284 2B3DE748 57EE16DB 80DBFF08

W : 00000002 00000000


n0_final = 2 : swap B with C  (final state)

A : 3DBA182A D0D6787E DAD8F4C9 CC065328 A36A08C7 902C794E 43E5A220 E2F378F1
    1E35B4C3 EF6B834E 8E442A11 6922E895

B : 66DF96F1 E4D309BD 6377D5E2 48F253E3 F8E9B974 0773C11C 8B2B886F 9D7AC002
    343612B3 80C441F4 F676FCAD 3D6A453A 6CBEC284 2B3DE748 57EE16DB 80DBFF08

C : 57E837B3 3B2C6ACA E0358DC2 2BD758E9 30F7A2ED DF3516C7 253CB0E0 1A1A98FC
    C0088FDA 9ABA672A 79D0BD56 07AE488E 095E2114 06855B3B 1877A349 A2543F99

W : 00000002 00000000


Hash value (word array):

H : C0088FDA 9ABA672A 79D0BD56 07AE488E 095E2114 06855B3B 1877A349 A2543F99


Hash value (byte array):

H : DA 8F 08 C0 2A 67 BA 9A 56 BD D0 79 8E 48 AE 07
    14 21 5E 09 3B 5B 85 06 49 A3 77 18 99 3F 54 A2
```

B.6 Intermediate States for Shabal-256 (Message B)

```
init

A : 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
    00000000 00000000 00000000 00000000

B : 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
    00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000

C : 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
    00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000

W : FFFFFFFF FFFFFFFF


block number = -1 : message block

M : 00000100 00000101 00000102 00000103 00000104 00000105 00000106 00000107
    00000108 00000109 0000010A 0000010B 0000010C 0000010D 0000010E 0000010F


block number = -1 : add M to B

A : 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
    00000000 00000000 00000000 00000000

B : 00000100 00000101 00000102 00000103 00000104 00000105 00000106 00000107
    00000108 00000109 0000010A 0000010B 0000010C 0000010D 0000010E 0000010F

C : 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
    00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
```

```
W : FFFFFFFF FFFFFFFF


block number = -1 : xor counter W into A

A : FFFFFFFF FFFFFFFF 00000000 00000000 00000000 00000000 00000000 00000000
    00000000 00000000 00000000 00000000

B : 00000100 00000101 00000102 00000103 00000104 00000105 00000106 00000107
    00000108 00000109 0000010A 0000010B 0000010C 0000010D 0000010E 0000010F

C : 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
    00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000

W : FFFFFFFF FFFFFFFF


block number = -1 : permutation - rotate B

A : FFFFFFFF FFFFFFFF 00000000 00000000 00000000 00000000 00000000 00000000
    00000000 00000000 00000000 00000000

B : 02000000 02020000 02040000 02060000 02080000 020A0000 020C0000 020E0000
    02100000 02120000 02140000 02160000 02180000 021A0000 021C0000 021E0000

C : 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
    00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000

W : FFFFFFFF FFFFFFFF


block number = -1 : permutation (j = 0)

A : AE3F47E2 C68A60D9 C63E700C 04FB82A0 8088FEAE 64D8954B CC08DF9F F2497B68
    7EC08316 F007C07B BBB8A77D 6C368247

B : 06080102 FE78F1B6 12CC57B2 93BBAACD 7B670151 9F336AB4 37EF2060 09AA8497
    851F7CE9 0BDC3F84 406F5882 97E57DB8 55F0B81D 3D419F26 3DF98FF3 FF387D5F

C : 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
    00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000

W : FFFFFFFF FFFFFFFF


block number = -1 : permutation (j = 1)

A : FF750C93 E5788A46 9BBD7565 CC341456 AD7FFC12 CE83FA1C 6511FD37 E0E9DCD1
    A72272C9 B2929B0B 78EE6F01 C3B82A6F

B : 14F4A4C4 CDE737FE 5675BD47 7A96E075 AE138F94 730BB19D E8CFD03E 2F12DCBE
    0AB40ABF 0D3F0AB1 E49C3B9E 1C0110D8 F96173D7 4BFF3BAF E11D1D2E E166D991

C : 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
    00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000

W : FFFFFFFF FFFFFFFF


block number = -1 : permutation (j = 2)

A : 79F023D8 708745B8 B2D69F0A AB24079A 4C77DDAC B4B92870 88E6ECE2 747657E4
    3395A2D7 5A9826A0 0D578B50 68F97918

B : B555C7FE 3E7106A7 A72A6641 93015272 DA28C30E 696FD97D 9CB6C088 0AFE4119
    A6E0372D 5138C2ED BE216420 B38B89AB 3EA8BA87 3299AE01 30924EF2 55CB35C4
```

```
C : 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
    00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000

W : FFFFFFFF FFFFFFFF


block number = -1 : permutation - add C to A

A : 79F023D8 708745B8 B2D69F0A AB24079A 4C77DDAC B4B92870 88E6ECE2 747657E4
    3395A2D7 5A9826A0 0D578B50 68F97918

B : B555C7FE 3E7106A7 A72A6641 93015272 DA28C30E 696FD97D 9CB6C088 0AFE4119
    A6E0372D 5138C2ED BE216420 B38B89AB 3EA8BA87 3299AE01 30924EF2 55CB35C4

C : 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
    00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000

W : FFFFFFFF FFFFFFFF


block number = -1 : subtract M from C

A : 79F023D8 708745B8 B2D69F0A AB24079A 4C77DDAC B4B92870 88E6ECE2 747657E4
    3395A2D7 5A9826A0 0D578B50 68F97918

B : B555C7FE 3E7106A7 A72A6641 93015272 DA28C30E 696FD97D 9CB6C088 0AFE4119
    A6E0372D 5138C2ED BE216420 B38B89AB 3EA8BA87 3299AE01 30924EF2 55CB35C4

C : FFFFFF00 FFFFFEFF FFFFFEFE FFFFFEFD FFFFFEFC FFFFFEFB FFFFFEFA FFFFFEF9
    FFFFFEF8 FFFFFEF7 FFFFFEF6 FFFFFEF5 FFFFFEF4 FFFFFEF3 FFFFFEF2 FFFFFEF1

W : FFFFFFFF FFFFFFFF


block number = -1 : swap B with C

A : 79F023D8 708745B8 B2D69F0A AB24079A 4C77DDAC B4B92870 88E6ECE2 747657E4
    3395A2D7 5A9826A0 0D578B50 68F97918

B : FFFFFF00 FFFFFEFF FFFFFEFE FFFFFEFD FFFFFEFC FFFFFEFB FFFFFEFA FFFFFEF9
    FFFFFEF8 FFFFFEF7 FFFFFEF6 FFFFFEF5 FFFFFEF4 FFFFFEF3 FFFFFEF2 FFFFFEF1

C : B555C7FE 3E7106A7 A72A6641 93015272 DA28C30E 696FD97D 9CB6C088 0AFE4119
    A6E0372D 5138C2ED BE216420 B38B89AB 3EA8BA87 3299AE01 30924EF2 55CB35C4

W : FFFFFFFF FFFFFFFF


block number = 0 : increment counter W

A : 79F023D8 708745B8 B2D69F0A AB24079A 4C77DDAC B4B92870 88E6ECE2 747657E4
    3395A2D7 5A9826A0 0D578B50 68F97918

B : FFFFFF00 FFFFFEFF FFFFFEFE FFFFFEFD FFFFFEFC FFFFFEFB FFFFFEFA FFFFFEF9
    FFFFFEF8 FFFFFEF7 FFFFFEF6 FFFFFEF5 FFFFFEF4 FFFFFEF3 FFFFFEF2 FFFFFEF1

C : B555C7FE 3E7106A7 A72A6641 93015272 DA28C30E 696FD97D 9CB6C088 0AFE4119
    A6E0372D 5138C2ED BE216420 B38B89AB 3EA8BA87 3299AE01 30924EF2 55CB35C4

W : 00000000 00000000


block number = 0 : message block

M : 00000110 00000111 00000112 00000113 00000114 00000115 00000116 00000117
    00000118 00000119 0000011A 0000011B 0000011C 0000011D 0000011E 0000011F
```

```
block number = 0 : add M to B

A : 79F023D8 708745B8 B2D69F0A AB24079A 4C77DDAC B4B92870 88E6ECE2 747657E4
    3395A2D7 5A9826A0 0D578B50 68F97918

B : 00000010 00000010 00000010 00000010 00000010 00000010 00000010 00000010
    00000010 00000010 00000010 00000010 00000010 00000010 00000010 00000010

C : B555C7FE 3E7106A7 A72A6641 93015272 DA28C30E 696FD97D 9CB6C088 0AFE4119
    A6E0372D 5138C2ED BE216420 B38B89AB 3EA8BA87 3299AE01 30924EF2 55CB35C4

W : 00000000 00000000


block number = 0 : xor counter W into A

A : 79F023D8 708745B8 B2D69F0A AB24079A 4C77DDAC B4B92870 88E6ECE2 747657E4
    3395A2D7 5A9826A0 0D578B50 68F97918

B : 00000010 00000010 00000010 00000010 00000010 00000010 00000010 00000010
    00000010 00000010 00000010 00000010 00000010 00000010 00000010 00000010

C : B555C7FE 3E7106A7 A72A6641 93015272 DA28C30E 696FD97D 9CB6C088 0AFE4119
    A6E0372D 5138C2ED BE216420 B38B89AB 3EA8BA87 3299AE01 30924EF2 55CB35C4

W : 00000000 00000000


block number = 0 : permutation - rotate B

A : 79F023D8 708745B8 B2D69F0A AB24079A 4C77DDAC B4B92870 88E6ECE2 747657E4
    3395A2D7 5A9826A0 0D578B50 68F97918

B : 00200000 00200000 00200000 00200000 00200000 00200000 00200000 00200000
    00200000 00200000 00200000 00200000 00200000 00200000 00200000 00200000

C : B555C7FE 3E7106A7 A72A6641 93015272 DA28C30E 696FD97D 9CB6C088 0AFE4119
    A6E0372D 5138C2ED BE216420 B38B89AB 3EA8BA87 3299AE01 30924EF2 55CB35C4

W : 00000000 00000000


block number = 0 : permutation (j = 0)

A : 7F0490C3 946910EC 827CA63E 409A1072 2B03CF2C FED97162 7F90F2B4 4069B7E1
    CCB74C96 09778D59 6B24AB90 80A11478

B : AA98C924 A5880F94 19DB44E2 61A3149F D4BC30D3 01668E9D 802F0D4B BFD6481E
    3308B369 F6C872A6 949B546F 7F1EEB87 80BB6F3C 6BD6EF13 7DC359C1 BF25EF8D

C : B555C7FE 3E7106A7 A72A6641 93015272 DA28C30E 696FD97D 9CB6C088 0AFE4119
    A6E0372D 5138C2ED BE216420 B38B89AB 3EA8BA87 3299AE01 30924EF2 55CB35C4

W : 00000000 00000000


block number = 0 : permutation (j = 1)

A : D455C979 9F6F4358 7F3D0246 E8A1F260 BE9C04EA 15A315D0 279AC5A8 1C34F0F3
    10A66C48 D8CCDB49 02B680C7 AB645FCF

B : 5923FD91 9DEBD978 66452E00 8510F1DD 4621F210 25FE398C FD1765AF 2B37300D
    4DBB5054 8D0059EA A9F45566 E963DA91 4015256C 3DF13409 23E389D5 9D80D017

C : B555C7FE 3E7106A7 A72A6641 93015272 DA28C30E 696FD97D 9CB6C088 0AFE4119
    A6E0372D 5138C2ED BE216420 B38B89AB 3EA8BA87 3299AE01 30924EF2 55CB35C4
```

```
W : 00000000 00000000


block number = 0 : permutation (j = 2)

A : B6A03371 17243406 531455C7 44F0DBD6 EC3CC237 851D8290 D99C1EB9 AED060A3
    C3433D0F 62FD9C1A 02BB7200 5FB743B0

B : B405F031 C4233EBA B3733979 C0DD9D55 C51C28AE A327B8E1 56C56167 ED614433
    88B59D60 60E2CEBA 758B4B8B 83E82A7F BC968828 E6E00BF7 BA839E55 9B491C60

C : B555C7FE 3E7106A7 A72A6641 93015272 DA28C30E 696FD97D 9CB6C088 0AFE4119
    A6E0372D 5138C2ED BE216420 B38B89AB 3EA8BA87 3299AE01 30924EF2 55CB35C4

W : 00000000 00000000


block number = 0 : permutation - add C to A

A : 52F84552 E54B7999 2D8EE3EC B9645191 E0078B86 BB7C44C9 D2B5C1CA B0D2EB8C
    14CE5A45 22AF50DC EFFDBC6B EB21B74A

B : B405F031 C4233EBA B3733979 C0DD9D55 C51C28AE A327B8E1 56C56167 ED614433
    88B59D60 60E2CEBA 758B4B8B 83E82A7F BC968828 E6E00BF7 BA839E55 9B491C60

C : B555C7FE 3E7106A7 A72A6641 93015272 DA28C30E 696FD97D 9CB6C088 0AFE4119
    A6E0372D 5138C2ED BE216420 B38B89AB 3EA8BA87 3299AE01 30924EF2 55CB35C4

W : 00000000 00000000


block number = 0 : subtract M from C

A : 52F84552 E54B7999 2D8EE3EC B9645191 E0078B86 BB7C44C9 D2B5C1CA B0D2EB8C
    14CE5A45 22AF50DC EFFDBC6B EB21B74A

B : B405F031 C4233EBA B3733979 C0DD9D55 C51C28AE A327B8E1 56C56167 ED614433
    88B59D60 60E2CEBA 758B4B8B 83E82A7F BC968828 E6E00BF7 BA839E55 9B491C60

C : B555C6EE 3E710596 A72A652F 9301515F DA28C1FA 696FD868 9CB6BF72 0AFE4002
    A6E03615 5138C1D4 BE216306 B38B8890 3EA8B96B 3299ACE4 30924DD4 55CB34A5

W : 00000000 00000000


block number = 0 : swap B with C

A : 52F84552 E54B7999 2D8EE3EC B9645191 E0078B86 BB7C44C9 D2B5C1CA B0D2EB8C
    14CE5A45 22AF50DC EFFDBC6B EB21B74A

B : B555C6EE 3E710596 A72A652F 9301515F DA28C1FA 696FD868 9CB6BF72 0AFE4002
    A6E03615 5138C1D4 BE216306 B38B8890 3EA8B96B 3299ACE4 30924DD4 55CB34A5

C : B405F031 C4233EBA B3733979 C0DD9D55 C51C28AE A327B8E1 56C56167 ED614433
    88B59D60 60E2CEBA 758B4B8B 83E82A7F BC968828 E6E00BF7 BA839E55 9B491C60

W : 00000000 00000000


block number = 1 : increment counter W

A : 52F84552 E54B7999 2D8EE3EC B9645191 E0078B86 BB7C44C9 D2B5C1CA B0D2EB8C
    14CE5A45 22AF50DC EFFDBC6B EB21B74A

B : B555C6EE 3E710596 A72A652F 9301515F DA28C1FA 696FD868 9CB6BF72 0AFE4002
    A6E03615 5138C1D4 BE216306 B38B8890 3EA8B96B 3299ACE4 30924DD4 55CB34A5
```

```
C : B405F031 C4233EBA B3733979 C0DD9D55 C51C28AE A327B8E1 56C56167 ED614433
    88B59D60 60E2CEBA 758B4B8B 83E82A7F BC968828 E6E00BF7 BA839E55 9B491C60

W : 00000001 00000000


block number = 1 : message block

M : 64636261 68676665 6C6B6A69 706F6E6D 74737271 78777675 302D7A79 34333231
    38373635 42412D39 46454443 4A494847 4E4D4C4B 5251504F 56555453 5A595857


block number = 1 : add M to B

A : 52F84552 E54B7999 2D8EE3EC B9645191 E0078B86 BB7C44C9 D2B5C1CA B0D2EB8C
    14CE5A45 22AF50DC EFFDBC6B EB21B74A

B : 19B9294F A6D86BFB 1395CF98 0370BFCC 4E9C346B E1E74EDD CCE439EB 3F317233
    DF176C4A 9379EF0D 0466A749 FDD4D0D7 8CF605B6 84EAFD33 86E7A227 B0248CFC

C : B405F031 C4233EBA B3733979 C0DD9D55 C51C28AE A327B8E1 56C56167 ED614433
    88B59D60 60E2CEBA 758B4B8B 83E82A7F BC968828 E6E00BF7 BA839E55 9B491C60

W : 00000001 00000000


block number = 1 : xor counter W into A

A : 52F84553 E54B7999 2D8EE3EC B9645191 E0078B86 BB7C44C9 D2B5C1CA B0D2EB8C
    14CE5A45 22AF50DC EFFDBC6B EB21B74A

B : 19B9294F A6D86BFB 1395CF98 0370BFCC 4E9C346B E1E74EDD CCE439EB 3F317233
    DF176C4A 9379EF0D 0466A749 FDD4D0D7 8CF605B6 84EAFD33 86E7A227 B0248CFC

C : B405F031 C4233EBA B3733979 C0DD9D55 C51C28AE A327B8E1 56C56167 ED614433
    88B59D60 60E2CEBA 758B4B8B 83E82A7F BC968828 E6E00BF7 BA839E55 9B491C60

W : 00000001 00000000


block number = 1 : permutation - rotate B

A : 52F84553 E54B7999 2D8EE3EC B9645191 E0078B86 BB7C44C9 D2B5C1CA B0D2EB8C
    14CE5A45 22AF50DC EFFDBC6B EB21B74A

B : 529E3372 D7F74DB0 9F30272B 7F9806E1 68D69D38 9DBBC3CE 73D799C8 E4667E62
    D895BE2E DE1B26F3 4E9208CD A1AFFBA9 0B6D19EC FA6709D5 444F0DCF 19F96049

C : B405F031 C4233EBA B3733979 C0DD9D55 C51C28AE A327B8E1 56C56167 ED614433
    88B59D60 60E2CEBA 758B4B8B 83E82A7F BC968828 E6E00BF7 BA839E55 9B491C60

W : 00000001 00000000


block number = 1 : permutation (j = 0)

A : F6D73330 89E4C140 3E1E3376 8A749508 A272998B B4E5B3E3 4C0D8E29 7F5FC96D
    76E421F9 550550FF 576B6AC2 E314C484

B : F9AC6F35 B8B2E012 85A4AF53 0DCFA1CA 8C205C04 706DCB81 545D4246 486CCA57
    3830A25B 16CCE2E7 35B084A7 5FB4CC28 1FF2FF17 82D52D14 497FD717 4679AA65

C : B405F031 C4233EBA B3733979 C0DD9D55 C51C28AE A327B8E1 56C56167 ED614433
    88B59D60 60E2CEBA 758B4B8B 83E82A7F BC968828 E6E00BF7 BA839E55 9B491C60

W : 00000001 00000000
```

```
block number = 1 : permutation (j = 1)

A : C07120A5 04EE925E 300C7989 C5E1B76B 01DA3B70 7E8672A6 ACC3FAD9 B8572BD6
    4814BD95 BCDD8C44 7871F536 AD059F2B

B : F266D21B B63BAB98 206805EE B84F2DF8 AFABFA63 A3F9E4B9 2F348E45 C223F47A
    4FEF9BEC D688A86F A4928F38 8577D0C4 C1C03AA1 84D3D770 C1C3AB08 CB5B80E3

C : B405F031 C4233EBA B3733979 C0DD9D55 C51C28AE A327B8E1 56C56167 ED614433
    88B59D60 60E2CEBA 758B4B8B 83E82A7F BC968828 E6E00BF7 BA839E55 9B491C60

W : 00000001 00000000


block number = 1 : permutation (j = 2)

A : 8B5E482B EFEC43B6 A1FE969E 5C0B2ED9 A3A90B23 3C6E5B7C F85A8647 9B0EEF10
    6AD2B985 A869C2D6 BF63A720 4D5430F0

B : ACA70A95 4E3FE92C 860464EB E56A291F 2BF64313 57E0753A 006875EB 27B339D3
    C389C304 6E80F45C 4E8067C9 6E1EB166 16AD3339 5E3193C8 C31B0ECE 241CCEC8

C : B405F031 C4233EBA B3733979 C0DD9D55 C51C28AE A327B8E1 56C56167 ED614433
    88B59D60 60E2CEBA 758B4B8B 83E82A7F BC968828 E6E00BF7 BA839E55 9B491C60

W : 00000001 00000000


block number = 1 : permutation - add C to A

A : 6B6D2C5F 25A4E4BD F0299A30 20C7680E ED31090B 3E4611BB C0884C9C 1AD2D57B
    9CF9C58C B2D2110C AA4E3AB2 D4287C37

B : ACA70A95 4E3FE92C 860464EB E56A291F 2BF64313 57E0753A 006875EB 27B339D3
    C389C304 6E80F45C 4E8067C9 6E1EB166 16AD3339 5E3193C8 C31B0ECE 241CCEC8

C : B405F031 C4233EBA B3733979 C0DD9D55 C51C28AE A327B8E1 56C56167 ED614433
    88B59D60 60E2CEBA 758B4B8B 83E82A7F BC968828 E6E00BF7 BA839E55 9B491C60

W : 00000001 00000000


block number = 1 : subtract M from C

A : 6B6D2C5F 25A4E4BD F0299A30 20C7680E ED31090B 3E4611BB C0884C9C 1AD2D57B
    9CF9C58C B2D2110C AA4E3AB2 D4287C37

B : ACA70A95 4E3FE92C 860464EB E56A291F 2BF64313 57E0753A 006875EB 27B339D3
    C389C304 6E80F45C 4E8067C9 6E1EB166 16AD3339 5E3193C8 C31B0ECE 241CCEC8

C : 4FA28DD0 5BBBD855 4707CF10 506E2EE8 50A8B63D 2AB0426C 2697E6EE B92E1202
    507E672B 1EA1A181 2F460748 399EE238 6E493BDD 948EBBA8 642E4A02 40EFC409

W : 00000001 00000000


block number = 1 : swap B with C

A : 6B6D2C5F 25A4E4BD F0299A30 20C7680E ED31090B 3E4611BB C0884C9C 1AD2D57B
    9CF9C58C B2D2110C AA4E3AB2 D4287C37

B : 4FA28DD0 5BBBD855 4707CF10 506E2EE8 50A8B63D 2AB0426C 2697E6EE B92E1202
    507E672B 1EA1A181 2F460748 399EE238 6E493BDD 948EBBA8 642E4A02 40EFC409

C : ACA70A95 4E3FE92C 860464EB E56A291F 2BF64313 57E0753A 006875EB 27B339D3
```

```
    C389C304 6E80F45C 4E8067C9 6E1EB166 16AD3339 5E3193C8 C31B0ECE 241CCEC8

W : 00000001 00000000


block number = 2 : increment counter W

A : 6B6D2C5F 25A4E4BD F0299A30 20C7680E ED31090B 3E4611BB C0884C9C 1AD2D57B
    9CF9C58C B2D2110C AA4E3AB2 D4287C37

B : 4FA28DD0 5BBBD855 4707CF10 506E2EE8 50A8B63D 2AB0426C 2697E6EE B92E1202
    507E672B 1EA1A181 2F460748 399EE238 6E493BDD 948EBBA8 642E4A02 40EFC409

C : ACA70A95 4E3FE92C 860464EB E56A291F 2BF64313 57E0753A 006875EB 27B339D3
    C389C304 6E80F45C 4E8067C9 6E1EB166 16AD3339 5E3193C8 C31B0ECE 241CCEC8

W : 00000002 00000000


block number = 2 : message block

M : 3231302D 36353433 2D393837 64636261 68676665 6C6B6A69 706F6E6D 74737271
    78777675 00807A79 00000000 00000000 00000000 00000000 00000000 00000000


block number = 2 : add M to B

A : 6B6D2C5F 25A4E4BD F0299A30 20C7680E ED31090B 3E4611BB C0884C9C 1AD2D57B
    9CF9C58C B2D2110C AA4E3AB2 D4287C37

B : 81D3BDFD 91F10C88 74410747 B4D19149 B9101CA2 971BACD5 9707555B 2DA18473
    C8F5DDA0 1F221BFA 2F460748 399EE238 6E493BDD 948EBBA8 642E4A02 40EFC409

C : ACA70A95 4E3FE92C 860464EB E56A291F 2BF64313 57E0753A 006875EB 27B339D3
    C389C304 6E80F45C 4E8067C9 6E1EB166 16AD3339 5E3193C8 C31B0ECE 241CCEC8

W : 00000002 00000000


block number = 2 : xor counter W into A

A : 6B6D2C5D 25A4E4BD F0299A30 20C7680E ED31090B 3E4611BB C0884C9C 1AD2D57B
    9CF9C58C B2D2110C AA4E3AB2 D4287C37

B : 81D3BDFD 91F10C88 74410747 B4D19149 B9101CA2 971BACD5 9707555B 2DA18473
    C8F5DDA0 1F221BFA 2F460748 399EE238 6E493BDD 948EBBA8 642E4A02 40EFC409

C : ACA70A95 4E3FE92C 860464EB E56A291F 2BF64313 57E0753A 006875EB 27B339D3
    C389C304 6E80F45C 4E8067C9 6E1EB166 16AD3339 5E3193C8 C31B0ECE 241CCEC8

W : 00000002 00000000


block number = 2 : permutation - rotate B

A : 6B6D2C5D 25A4E4BD F0299A30 20C7680E ED31090B 3E4611BB C0884C9C 1AD2D57B
    9CF9C58C B2D2110C AA4E3AB2 D4287C37

B : 7BFB03A7 191123E2 0E8EE882 229369A3 39457220 59AB2E37 AAB72E0E 08E65B43
    BB4191EB 37F43E44 0E905E8C C470733D 77BADC92 7751291D 9404C85C 881281DF

C : ACA70A95 4E3FE92C 860464EB E56A291F 2BF64313 57E0753A 006875EB 27B339D3
    C389C304 6E80F45C 4E8067C9 6E1EB166 16AD3339 5E3193C8 C31B0ECE 241CCEC8

W : 00000002 00000000
```

```
block number = 2 : permutation (j = 0)

A : 27DDABAE 3485C441 EF71221A A12B9143 A9CF4744 5E5BDF73 9120D522 105DAB7D
    7D653849 E57D34A3 EE830FDA 5D81CF9F

B : 83678676 D553452F 99228D2F D1CB70C0 24BA5CFB 12F27CE2 3BB176C0 FE6EE204
    F419E461 756AB7D4 0C5C4D3D 2A9ED61B 3757ED75 25D86984 38874D5C 4EF16D03

C : ACA70A95 4E3FE92C 860464EB E56A291F 2BF64313 57E0753A 006875EB 27B339D3
    C389C304 6E80F45C 4E8067C9 6E1EB166 16AD3339 5E3193C8 C31B0ECE 241CCEC8

W : 00000002 00000000


block number = 2 : permutation (j = 1)

A : FDEC5D48 7F32E3BF 3312EE26 73F81D45 E771CB44 2A8DED27 D7AE1BDD A87D95D5
    09F3FF4A CF3EAA2E 1B2F4B4C 4F8F64A2

B : 2EDDE854 6276F9BC 49DED920 6637625D BF78B943 1525AC15 93B25933 4CAD5F54
    EA206A74 6A1873E8 D4558BA3 D93A4E8C 7621EE51 9EC2C1D0 595F7E9A CA60B02C

C : ACA70A95 4E3FE92C 860464EB E56A291F 2BF64313 57E0753A 006875EB 27B339D3
    C389C304 6E80F45C 4E8067C9 6E1EB166 16AD3339 5E3193C8 C31B0ECE 241CCEC8

W : 00000002 00000000


block number = 2 : permutation (j = 2)

A : D22C0DEA A7764ACD C57D3D58 0AA68F74 A26B8BA2 47726CDB 66B4B2AC 505FB429
    085C7159 85E454FC 0CD7F1DE 568BADDB

B : D07BB00A 7084EDBF E2E347DC 176FBA27 53228092 72C2ED18 1DE670C0 6C03CE23
    89D4A0B4 6CBD74F4 31E05A14 1DD4D6CF 1BE05204 479E28A2 4196F315 3DB5327D

C : ACA70A95 4E3FE92C 860464EB E56A291F 2BF64313 57E0753A 006875EB 27B339D3
    C389C304 6E80F45C 4E8067C9 6E1EB166 16AD3339 5E3193C8 C31B0ECE 241CCEC8

W : 00000002 00000000


block number = 2 : permutation - add C to A

A : 49D1B737 96C0CBAE C9CF2F86 542E7918 D3A5BD5C E3997D87 7B56056E 254CF6C8
    839885B1 8C118E4C 316AEF3C 688F9A5D

B : D07BB00A 7084EDBF E2E347DC 176FBA27 53228092 72C2ED18 1DE670C0 6C03CE23
    89D4A0B4 6CBD74F4 31E05A14 1DD4D6CF 1BE05204 479E28A2 4196F315 3DB5327D

C : ACA70A95 4E3FE92C 860464EB E56A291F 2BF64313 57E0753A 006875EB 27B339D3
    C389C304 6E80F45C 4E8067C9 6E1EB166 16AD3339 5E3193C8 C31B0ECE 241CCEC8

W : 00000002 00000000


block number = 2 : subtract M from C

A : 49D1B737 96C0CBAE C9CF2F86 542E7918 D3A5BD5C E3997D87 7B56056E 254CF6C8
    839885B1 8C118E4C 316AEF3C 688F9A5D

B : D07BB00A 7084EDBF E2E347DC 176FBA27 53228092 72C2ED18 1DE670C0 6C03CE23
    89D4A0B4 6CBD74F4 31E05A14 1DD4D6CF 1BE05204 479E28A2 4196F315 3DB5327D

C : 7A75DA68 180AB4F9 58CB2CB4 8106C6BE C38EDCAE EB750AD1 8FF9077E B33FC762
    4B124C8F 6E0079E3 4E8067C9 6E1EB166 16AD3339 5E3193C8 C31B0ECE 241CCEC8
```

```
W : 00000002 00000000


block number = 2 : swap B with C

A : 49D1B737 96C0CBAE C9CF2F86 542E7918 D3A5BD5C E3997D87 7B56056E 254CF6C8
    839885B1 8C118E4C 316AEF3C 688F9A5D

B : 7A75DA68 180AB4F9 58CB2CB4 8106C6BE C38EDCAE EB750AD1 8FF9077E B33FC762
    4B124C8F 6E0079E3 4E8067C9 6E1EB166 16AD3339 5E3193C8 C31B0ECE 241CCEC8

C : D07BB00A 7084EDBF E2E347DC 176FBA27 53228092 72C2ED18 1DE670C0 6C03CE23
    89D4A0B4 6CBD74F4 31E05A14 1DD4D6CF 1BE05204 479E28A2 4196F315 3DB5327D

W : 00000002 00000000


n0_final = 0 : add M to B

A : 49D1B737 96C0CBAE C9CF2F86 542E7918 D3A5BD5C E3997D87 7B56056E 254CF6C8
    839885B1 8C118E4C 316AEF3C 688F9A5D

B : ACA70A95 4E3FE92C 860464EB E56A291F 2BF64313 57E0753A 006875EB 27B339D3
    C389C304 6E80F45C 4E8067C9 6E1EB166 16AD3339 5E3193C8 C31B0ECE 241CCEC8

C : D07BB00A 7084EDBF E2E347DC 176FBA27 53228092 72C2ED18 1DE670C0 6C03CE23
    89D4A0B4 6CBD74F4 31E05A14 1DD4D6CF 1BE05204 479E28A2 4196F315 3DB5327D

W : 00000002 00000000


n0_final = 0 : xor counter W into A

A : 49D1B735 96C0CBAE C9CF2F86 542E7918 D3A5BD5C E3997D87 7B56056E 254CF6C8
    839885B1 8C118E4C 316AEF3C 688F9A5D

B : ACA70A95 4E3FE92C 860464EB E56A291F 2BF64313 57E0753A 006875EB 27B339D3
    C389C304 6E80F45C 4E8067C9 6E1EB166 16AD3339 5E3193C8 C31B0ECE 241CCEC8

C : D07BB00A 7084EDBF E2E347DC 176FBA27 53228092 72C2ED18 1DE670C0 6C03CE23
    89D4A0B4 6CBD74F4 31E05A14 1DD4D6CF 1BE05204 479E28A2 4196F315 3DB5327D

W : 00000002 00000000


n0_final = 0 : permutation - rotate B

A : 49D1B735 96C0CBAE C9CF2F86 542E7918 D3A5BD5C E3997D87 7B56056E 254CF6C8
    839885B1 8C118E4C 316AEF3C 688F9A5D

B : 152B594E D2589C7F C9D70C08 523FCAD4 862657EC EA74AFC0 EBD600D0 73A64F66
    86098713 E8B8DD01 CF929D00 62CCDC3D 66722D5A 2790BC63 1D9D8636 9D904839

C : D07BB00A 7084EDBF E2E347DC 176FBA27 53228092 72C2ED18 1DE670C0 6C03CE23
    89D4A0B4 6CBD74F4 31E05A14 1DD4D6CF 1BE05204 479E28A2 4196F315 3DB5327D

W : 00000002 00000000


n0_final = 0 : permutation (j = 0)

A : 67D32091 5303AF58 5A861FAD 0C4A05CE 7A6147AA 345CEC5B 7C1199C6 7CECED68
    826D1307 0C41559C 00B9D8B4 68FAEE46

B : 85E5A48A EAFA029C 12B066C0 100D7776 89D2178C 1F4A4C25 54426798 645F8C5B
    7181E2DF 22CF1060 60631D4A 529CA9C3 54C885DA E3DD2861 9E42EC3E C8956A42
```

```
C : D07BB00A 7084EDBF E2E347DC 176FBA27 53228092 72C2ED18 1DE670C0 6C03CE23
    89D4A0B4 6CBD74F4 31E05A14 1DD4D6CF 1BE05204 479E28A2 4196F315 3DB5327D

W : 00000002 00000000


n0_final = 0 : permutation (j = 1)

A : F1D330FB 55A76650 3645D1D3 79E5AA6A A4480A6E C8A221B8 2D30A72C B7B74535
    235DF568 17EB4789 BF57EC17 29851CE7

B : F0F15233 2E98CF7D 0BC24E4B 36898599 CF06258E D680203C E82CDCD8 1EC5FBAE
    ED2F0ABA EFC6B96F 097C14B8 23230613 F226FE25 F0E78E84 EE4A80AE D9626E4F

C : D07BB00A 7084EDBF E2E347DC 176FBA27 53228092 72C2ED18 1DE670C0 6C03CE23
    89D4A0B4 6CBD74F4 31E05A14 1DD4D6CF 1BE05204 479E28A2 4196F315 3DB5327D

W : 00000002 00000000


n0_final = 0 : permutation (j = 2)

A : 553BD29F C35AB8A9 A7ECA963 47F2291E D1AF3761 7068CAA6 D7DFE602 2DC59B1A
    22D5EA34 88C9C00A 2FF881C6 D21852D1

B : 4DA49FC7 489D5B3A 07AACE5E 95EF3C0E 34C8667D 91A5072F 884AEF2D 858621BD
    F40EDDEB 501A4786 3AD8308D 947C68C3 3967E980 96F922FC 0C927F64 9F2371B1

C : D07BB00A 7084EDBF E2E347DC 176FBA27 53228092 72C2ED18 1DE670C0 6C03CE23
    89D4A0B4 6CBD74F4 31E05A14 1DD4D6CF 1BE05204 479E28A2 4196F315 3DB5327D

W : 00000002 00000000


n0_final = 0 : permutation - add C to A

A : C8359612 02D93B49 D2D2ACDC 8A52D4CF 92D7F228 1DDB9BF6 27E535CD 606FADCA
    C41E494D 81A13354 57170C74 637610BA

B : 4DA49FC7 489D5B3A 07AACE5E 95EF3C0E 34C8667D 91A5072F 884AEF2D 858621BD
    F40EDDEB 501A4786 3AD8308D 947C68C3 3967E980 96F922FC 0C927F64 9F2371B1

C : D07BB00A 7084EDBF E2E347DC 176FBA27 53228092 72C2ED18 1DE670C0 6C03CE23
    89D4A0B4 6CBD74F4 31E05A14 1DD4D6CF 1BE05204 479E28A2 4196F315 3DB5327D

W : 00000002 00000000


n0_final = 0 : subtract M from C

A : C8359612 02D93B49 D2D2ACDC 8A52D4CF 92D7F228 1DDB9BF6 27E535CD 606FADCA
    C41E494D 81A13354 57170C74 637610BA

B : 4DA49FC7 489D5B3A 07AACE5E 95EF3C0E 34C8667D 91A5072F 884AEF2D 858621BD
    F40EDDEB 501A4786 3AD8308D 947C68C3 3967E980 96F922FC 0C927F64 9F2371B1

C : 9E4A7FDD 3A4FB98C B5AA0FA5 B30C57C6 EABB1A2D 065782AF AD770253 F7905BB2
    115D2A3F 6C3CFA7B 31E05A14 1DD4D6CF 1BE05204 479E28A2 4196F315 3DB5327D

W : 00000002 00000000


n0_final = 0 : swap B with C

A : C8359612 02D93B49 D2D2ACDC 8A52D4CF 92D7F228 1DDB9BF6 27E535CD 606FADCA
    C41E494D 81A13354 57170C74 637610BA
```

```
B : 9E4A7FDD 3A4FB98C B5AA0FA5 B30C57C6 EABB1A2D 065782AF AD770253 F7905BB2
    115D2A3F 6C3CFA7B 31E05A14 1DD4D6CF 1BE05204 479E28A2 4196F315 3DB5327D

C : 4DA49FC7 489D5B3A 07AACE5E 95EF3C0E 34C8667D 91A5072F 884AEF2D 858621BD
    F40EDDEB 501A4786 3AD8308D 947C68C3 3967E980 96F922FC 0C927F64 9F2371B1

W : 00000002 00000000


n0_final = 1 : add M to B

A : C8359612 02D93B49 D2D2ACDC 8A52D4CF 92D7F228 1DDB9BF6 27E535CD 606FADCA
    C41E494D 81A13354 57170C74 637610BA

B : D07BB00A 7084EDBF E2E347DC 176FBA27 53228092 72C2ED18 1DE670C0 6C03CE23
    89D4A0B4 6CBD74F4 31E05A14 1DD4D6CF 1BE05204 479E28A2 4196F315 3DB5327D

C : 4DA49FC7 489D5B3A 07AACE5E 95EF3C0E 34C8667D 91A5072F 884AEF2D 858621BD
    F40EDDEB 501A4786 3AD8308D 947C68C3 3967E980 96F922FC 0C927F64 9F2371B1

W : 00000002 00000000


n0_final = 1 : xor counter W into A

A : C8359610 02D93B49 D2D2ACDC 8A52D4CF 92D7F228 1DDB9BF6 27E535CD 606FADCA
    C41E494D 81A13354 57170C74 637610BA

B : D07BB00A 7084EDBF E2E347DC 176FBA27 53228092 72C2ED18 1DE670C0 6C03CE23
    89D4A0B4 6CBD74F4 31E05A14 1DD4D6CF 1BE05204 479E28A2 4196F315 3DB5327D

C : 4DA49FC7 489D5B3A 07AACE5E 95EF3C0E 34C8667D 91A5072F 884AEF2D 858621BD
    F40EDDEB 501A4786 3AD8308D 947C68C3 3967E980 96F922FC 0C927F64 9F2371B1

W : 00000002 00000000


n0_final = 1 : permutation - rotate B

A : C8359610 02D93B49 D2D2ACDC 8A52D4CF 92D7F228 1DDB9BF6 27E535CD 606FADCA
    C41E494D 81A13354 57170C74 637610BA

B : 6015A0F7 DB7EE109 8FB9C5C6 744E2EDF 0124A645 DA30E585 E1803BCC 9C46D807
    416913A9 E9E8D97A B42863C0 AD9E3BA9 A40837C0 51448F3C E62A832D 64FA7B6A

C : 4DA49FC7 489D5B3A 07AACE5E 95EF3C0E 34C8667D 91A5072F 884AEF2D 858621BD
    F40EDDEB 501A4786 3AD8308D 947C68C3 3967E980 96F922FC 0C927F64 9F2371B1

W : 00000002 00000000


n0_final = 1 : permutation (j = 0)

A : BD28A5CF BA0BBFB0 00095A8C C1B09C32 D38E01FD F1CC7A37 CED0E9B6 7C714B13
    C42B6040 A1C0E60C 5CD392A3 C0483484

B : 1509DB26 56317C44 79ED350B CB01B836 2E38B288 BA524EC3 F22F61D0 BB0304E3
    B906B8ED 8DEEAB06 CB7CAADD 648BBC28 0AC735B1 E77D5E37 33A3A328 F7BB9519

C : 4DA49FC7 489D5B3A 07AACE5E 95EF3C0E 34C8667D 91A5072F 884AEF2D 858621BD
    F40EDDEB 501A4786 3AD8308D 947C68C3 3967E980 96F922FC 0C927F64 9F2371B1

W : 00000002 00000000


n0_final = 1 : permutation (j = 1)
```

```
A : 71C9F985 C800394A 8FBD7622 AC657B69 2FB2355C D16C61B4 0D746211 A309E118
    17C31A6B D8076499 B38EA0F6 9885FDE2

B : E89A0525 77CE7CF7 452129C5 54340C45 B44D8084 535C06E1 A82F9CA8 117C0BDA
    FC3B77A1 2C2290B8 E6BBDC66 9A8DFCC6 C5C3A1C1 E0692224 95CCDBBE B38134D4

C : 4DA49FC7 489D5B3A 07AACE5E 95EF3C0E 34C8667D 91A5072F 884AEF2D 858621BD
    F40EDDEB 501A4786 3AD8308D 947C68C3 3967E980 96F922FC 0C927F64 9F2371B1

W : 00000002 00000000


n0_final = 1 : permutation (j = 2)

A : 1ADC4E68 7A0843ED C62C5677 43303AC9 E39D6B8C 6D6358A3 6A5C56A4 AC6F15A0
    E3AF684E 1E4B0104 4DABDD43 BCD9E6D6

B : 0CC31791 6F818D28 DEE2F99A E1113036 8DB8B09E 234FB1D0 698C90D9 9E37D282
    E4147B30 CAD9862C 58D41196 668B13D2 97D7D432 2166BAB2 99CD95C1 24247080

C : 4DA49FC7 489D5B3A 07AACE5E 95EF3C0E 34C8667D 91A5072F 884AEF2D 858621BD
    F40EDDEB 501A4786 3AD8308D 947C68C3 3967E980 96F922FC 0C927F64 9F2371B1

W : 00000002 00000000


n0_final = 1 : permutation - add C to A

A : E46B64EA 35DD33B1 3767DBDC DFB877B8 9E363B08 E3DF3CD2 94B90093 773D03B8
    93A12EDC 808A2EEC C6644EF4 8C8F85F4

B : 0CC31791 6F818D28 DEE2F99A E1113036 8DB8B09E 234FB1D0 698C90D9 9E37D282
    E4147B30 CAD9862C 58D41196 668B13D2 97D7D432 2166BAB2 99CD95C1 24247080

C : 4DA49FC7 489D5B3A 07AACE5E 95EF3C0E 34C8667D 91A5072F 884AEF2D 858621BD
    F40EDDEB 501A4786 3AD8308D 947C68C3 3967E980 96F922FC 0C927F64 9F2371B1

W : 00000002 00000000


n0_final = 1 : subtract M from C

A : E46B64EA 35DD33B1 3767DBDC DFB877B8 9E363B08 E3DF3CD2 94B90093 773D03B8
    93A12EDC 808A2EEC C6644EF4 8C8F85F4

B : 0CC31791 6F818D28 DEE2F99A E1113036 8DB8B09E 234FB1D0 698C90D9 9E37D282
    E4147B30 CAD9862C 58D41196 668B13D2 97D7D432 2166BAB2 99CD95C1 24247080

C : 1B736F9A 12682707 DA719627 318BD9AD CC610018 25399CC6 17DB80C0 1112AF4C
    7B976776 4F99CD0D 3AD8308D 947C68C3 3967E980 96F922FC 0C927F64 9F2371B1

W : 00000002 00000000


n0_final = 1 : swap B with C

A : E46B64EA 35DD33B1 3767DBDC DFB877B8 9E363B08 E3DF3CD2 94B90093 773D03B8
    93A12EDC 808A2EEC C6644EF4 8C8F85F4

B : 1B736F9A 12682707 DA719627 318BD9AD CC610018 25399CC6 17DB80C0 1112AF4C
    7B976776 4F99CD0D 3AD8308D 947C68C3 3967E980 96F922FC 0C927F64 9F2371B1

C : 0CC31791 6F818D28 DEE2F99A E1113036 8DB8B09E 234FB1D0 698C90D9 9E37D282
    E4147B30 CAD9862C 58D41196 668B13D2 97D7D432 2166BAB2 99CD95C1 24247080

W : 00000002 00000000
```

```
n0_final = 2 : add M to B

A : E46B64EA 35DD33B1 3767DBDC DFB877B8 9E363B08 E3DF3CD2 94B90093 773D03B8
    93A12EDC 808A2EEC C6644EF4 8C8F85F4

B : 4DA49FC7 489D5B3A 07AACE5E 95EF3C0E 34C8667D 91A5072F 884AEF2D 858621BD
    F40EDDEB 501A4786 3AD8308D 947C68C3 3967E980 96F922FC 0C927F64 9F2371B1

C : 0CC31791 6F818D28 DEE2F99A E1113036 8DB8B09E 234FB1D0 698C90D9 9E37D282
    E4147B30 CAD9862C 58D41196 668B13D2 97D7D432 2166BAB2 99CD95C1 24247080

W : 00000002 00000000


n0_final = 2 : xor counter W into A

A : E46B64E8 35DD33B1 3767DBDC DFB877B8 9E363B08 E3DF3CD2 94B90093 773D03B8
    93A12EDC 808A2EEC C6644EF4 8C8F85F4

B : 4DA49FC7 489D5B3A 07AACE5E 95EF3C0E 34C8667D 91A5072F 884AEF2D 858621BD
    F40EDDEB 501A4786 3AD8308D 947C68C3 3967E980 96F922FC 0C927F64 9F2371B1

C : 0CC31791 6F818D28 DEE2F99A E1113036 8DB8B09E 234FB1D0 698C90D9 9E37D282
    E4147B30 CAD9862C 58D41196 668B13D2 97D7D432 2166BAB2 99CD95C1 24247080

W : 00000002 00000000


n0_final = 2 : permutation - rotate B

A : E46B64E8 35DD33B1 3767DBDC DFB877B8 9E363B08 E3DF3CD2 94B90093 773D03B8
    93A12EDC 808A2EEC C6644EF4 8C8F85F4

B : 3F8E9B49 B674913A 9CBC0F55 781D2BDE CCFA6990 0E5F234A DE5B1095 437B0B0C
    BBD7E81D 8F0CA034 611A75B0 D18728F8 D30072CF 45F92DF2 FEC81924 E3633E46

C : 0CC31791 6F818D28 DEE2F99A E1113036 8DB8B09E 234FB1D0 698C90D9 9E37D282
    E4147B30 CAD9862C 58D41196 668B13D2 97D7D432 2166BAB2 99CD95C1 24247080

W : 00000002 00000000


n0_final = 2 : permutation (j = 0)

A : A72DE636 5297305E F2707159 668235DB BD196957 5D16B98F 97EDABFF A8A4C5E3
    93A64094 C93B80A3 493E6060 15BAF34E

B : 9DFAB6A3 C6D77880 8F8A3FEA 56D97CA8 DB124589 BE5700E4 D4A4752B D1AD2C04
    1BF66F50 28DD3F35 74F574FF 494B5D40 FED2FC56 269A9445 F01FBCEF 5FBBB6A9

C : 0CC31791 6F818D28 DEE2F99A E1113036 8DB8B09E 234FB1D0 698C90D9 9E37D282
    E4147B30 CAD9862C 58D41196 668B13D2 97D7D432 2166BAB2 99CD95C1 24247080

W : 00000002 00000000


n0_final = 2 : permutation (j = 1)

A : C9D3F423 B47F8AE4 359A3049 BCC98A64 DDBC4864 E9BD6633 87ABF033 CED5B802
    2D289F8C 80B46CEB FCF20C9A 61AC8102

B : 66048AA2 6E002C53 A815E77E 1B0568F6 64F3EB60 03E592DD AA451932 3D0926F4
    01C0D57C 1A3A0B71 238F2648 D1A0CF1B DFE64F36 5B77B146 986B7613 8E5D2AAF

C : 0CC31791 6F818D28 DEE2F99A E1113036 8DB8B09E 234FB1D0 698C90D9 9E37D282
    E4147B30 CAD9862C 58D41196 668B13D2 97D7D432 2166BAB2 99CD95C1 24247080
```

```
W : 00000002 00000000


n0_final = 2 : permutation (j = 2)

A : 9CEAF17C 69EC9966 D73B3761 D13C3EBC 434ACAB3 FBC76F4C D4258F3C 8295350A
    905C9850 D343C11C 97847D27 99775D23

B : C539CC9B CA52634F 214754DE 19A73AD2 AAF2D843 91D84323 7C4EFAFB 54D18CAB
    BF349FB4 304C8651 6CC43C53 DE2B54C2 D06FF9C2 9A535C6F 58AD6EFF 7A32F783

C : 0CC31791 6F818D28 DEE2F99A E1113036 8DB8B09E 234FB1D0 698C90D9 9E37D282
    E4147B30 CAD9862C 58D41196 668B13D2 97D7D432 2166BAB2 99CD95C1 24247080

W : 00000002 00000000


n0_final = 2 : permutation - add C to A

A : 08ABA604 9C4035C7 8B73310B B3795EF0 E6B83DEB 7A57B2AB 31D05460 23D8D113
    7630AEDA DCE8C11C A7146FD5 F5A59553

B : C539CC9B CA52634F 214754DE 19A73AD2 AAF2D843 91D84323 7C4EFAFB 54D18CAB
    BF349FB4 304C8651 6CC43C53 DE2B54C2 D06FF9C2 9A535C6F 58AD6EFF 7A32F783

C : 0CC31791 6F818D28 DEE2F99A E1113036 8DB8B09E 234FB1D0 698C90D9 9E37D282
    E4147B30 CAD9862C 58D41196 668B13D2 97D7D432 2166BAB2 99CD95C1 24247080

W : 00000002 00000000


n0_final = 2 : subtract M from C

A : 08ABA604 9C4035C7 8B73310B B3795EF0 E6B83DEB 7A57B2AB 31D05460 23D8D113
    7630AEDA DCE8C11C A7146FD5 F5A59553

B : C539CC9B CA52634F 214754DE 19A73AD2 AAF2D843 91D84323 7C4EFAFB 54D18CAB
    BF349FB4 304C8651 6CC43C53 DE2B54C2 D06FF9C2 9A535C6F 58AD6EFF 7A32F783

C : DA91E764 394C58F5 B1A9C163 7CADCDD5 25514A39 B6E44767 F91D226C 29C46011
    6B9D04BB CA590BB3 58D41196 668B13D2 97D7D432 2166BAB2 99CD95C1 24247080

W : 00000002 00000000


n0_final = 2 : swap B with C  (final state)

A : 08ABA604 9C4035C7 8B73310B B3795EF0 E6B83DEB 7A57B2AB 31D05460 23D8D113
    7630AEDA DCE8C11C A7146FD5 F5A59553

B : DA91E764 394C58F5 B1A9C163 7CADCDD5 25514A39 B6E44767 F91D226C 29C46011
    6B9D04BB CA590BB3 58D41196 668B13D2 97D7D432 2166BAB2 99CD95C1 24247080

C : C539CC9B CA52634F 214754DE 19A73AD2 AAF2D843 91D84323 7C4EFAFB 54D18CAB
    BF349FB4 304C8651 6CC43C53 DE2B54C2 D06FF9C2 9A535C6F 58AD6EFF 7A32F783

W : 00000002 00000000


Hash value :

H : BF349FB4 304C8651 6CC43C53 DE2B54C2 D06FF9C2 9A535C6F 58AD6EFF 7A32F783


Hash value (byte array):
```

```
H : B4 9F 34 BF 51 86 4C 30 53 3C C4 6C C2 54 2B DE
    C2 F9 6F D0 6F 5C 53 9A FF 6E AD 58 83 F7 32 7A
```

B.7 Intermediate States for Shabal-384 (Message A)

```
init

A : 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
    00000000 00000000 00000000 00000000

B : 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
    00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000

C : 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
    00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000

W : FFFFFFFF FFFFFFFF


block number = -1 : message block

M : 00000180 00000181 00000182 00000183 00000184 00000185 00000186 00000187
    00000188 00000189 0000018A 0000018B 0000018C 0000018D 0000018E 0000018F


block number = -1 : add M to B

A : 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
    00000000 00000000 00000000 00000000

B : 00000180 00000181 00000182 00000183 00000184 00000185 00000186 00000187
    00000188 00000189 0000018A 0000018B 0000018C 0000018D 0000018E 0000018F

C : 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
    00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000

W : FFFFFFFF FFFFFFFF


block number = -1 : xor counter W into A

A : FFFFFFFF FFFFFFFF 00000000 00000000 00000000 00000000 00000000 00000000
    00000000 00000000 00000000 00000000

B : 00000180 00000181 00000182 00000183 00000184 00000185 00000186 00000187
    00000188 00000189 0000018A 0000018B 0000018C 0000018D 0000018E 0000018F

C : 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
    00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000

W : FFFFFFFF FFFFFFFF


block number = -1 : permutation - rotate B

A : FFFFFFFF FFFFFFFF 00000000 00000000 00000000 00000000 00000000 00000000
    00000000 00000000 00000000 00000000

B : 03000000 03020000 03040000 03060000 03080000 030A0000 030C0000 030E0000
    03100000 03120000 03140000 03160000 03180000 031A0000 031C0000 031E0000

C : 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
    00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
```

```
W : FFFFFFFF FFFFFFFF


block number = -1 : permutation (j = 0)

A : 73B2093A 2A799470 05BF52F2 607A3AE1 BDA5E845 9A4F4C59 45672DC2 977B1FC2
    FD372A8B D7D97B33 8A5D1B6E 4D17CE45

B : 05080182 F1B8E8B6 D44C4392 F9C1290D 444A17BA 63A4B3A6 BC80D23D 6E98E03D
    04E8D574 2E0284CC 738AE491 B4C431BA 8A7DF6C5 D3B26B8F FC78AD0D 99B9C51E

C : 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
    00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000

W : FFFFFFFF FFFFFFFF


block number = -1 : permutation (j = 1)

A : 34DCDD7F 3399F8DC A47ABF91 51043F32 FDF93568 64558055 3E48B569 0D011A26
    DB56076A C383CE67 FA750B12 041BF733

B : AF9C5E68 55490C52 C44C5DCB C0EE508F AC3DD7E1 FB3556D4 7C8B5096 26D5C8B6
    C2F28868 90630EBB BC90894C C773A3B8 16FD271C 3CCEA8B5 3946108D C18D6FE4

C : 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
    00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000

W : FFFFFFFF FFFFFFFF


block number = -1 : permutation (j = 2)

A : BB22A196 A1C1B9EC 8B50D287 4092494B 4A0B904D 1065A62A 15AFF6D9 88EECBDD
    62189375 B55DE3B3 3FF60DE6 FA67BCE3

B : 5272938C 2A1646F0 78E62977 944F1832 1CA6F1AA A854EBBA 8DB98C54 F2C627D8
    30117F63 CF5C44A2 93711BBF F9F67353 B01D22B2 333F4D26 B285D303 86829CD5

C : 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
    00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000

W : FFFFFFFF FFFFFFFF


block number = -1 : permutation - add C to A

A : BB22A196 A1C1B9EC 8B50D287 4092494B 4A0B904D 1065A62A 15AFF6D9 88EECBDD
    62189375 B55DE3B3 3FF60DE6 FA67BCE3

B : 5272938C 2A1646F0 78E62977 944F1832 1CA6F1AA A854EBBA 8DB98C54 F2C627D8
    30117F63 CF5C44A2 93711BBF F9F67353 B01D22B2 333F4D26 B285D303 86829CD5

C : 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
    00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000

W : FFFFFFFF FFFFFFFF


block number = -1 : subtract M from C

A : BB22A196 A1C1B9EC 8B50D287 4092494B 4A0B904D 1065A62A 15AFF6D9 88EECBDD
    62189375 B55DE3B3 3FF60DE6 FA67BCE3

B : 5272938C 2A1646F0 78E62977 944F1832 1CA6F1AA A854EBBA 8DB98C54 F2C627D8
    30117F63 CF5C44A2 93711BBF F9F67353 B01D22B2 333F4D26 B285D303 86829CD5
```

```
C : FFFFFE80 FFFFFE7F FFFFFE7E FFFFFE7D FFFFFE7C FFFFFE7B FFFFFE7A FFFFFE79
    FFFFFE78 FFFFFE77 FFFFFE76 FFFFFE75 FFFFFE74 FFFFFE73 FFFFFE72 FFFFFE71

W : FFFFFFFF FFFFFFFF


block number = -1 : swap B with C

A : BB22A196 A1C1B9EC 8B50D287 4092494B 4A0B904D 1065A62A 15AFF6D9 88EECBDD
    62189375 B55DE3B3 3FF60DE6 FA67BCE3

B : FFFFFE80 FFFFFE7F FFFFFE7E FFFFFE7D FFFFFE7C FFFFFE7B FFFFFE7A FFFFFE79
    FFFFFE78 FFFFFE77 FFFFFE76 FFFFFE75 FFFFFE74 FFFFFE73 FFFFFE72 FFFFFE71

C : 5272938C 2A1646F0 78E62977 944F1832 1CA6F1AA A854EBBA 8DB98C54 F2C627D8
    30117F63 CF5C44A2 93711BBF F9F67353 B01D22B2 333F4D26 B285D303 86829CD5

W : FFFFFFFF FFFFFFFF


block number = 0 : increment counter W

A : BB22A196 A1C1B9EC 8B50D287 4092494B 4A0B904D 1065A62A 15AFF6D9 88EECBDD
    62189375 B55DE3B3 3FF60DE6 FA67BCE3

B : FFFFFE80 FFFFFE7F FFFFFE7E FFFFFE7D FFFFFE7C FFFFFE7B FFFFFE7A FFFFFE79
    FFFFFE78 FFFFFE77 FFFFFE76 FFFFFE75 FFFFFE74 FFFFFE73 FFFFFE72 FFFFFE71

C : 5272938C 2A1646F0 78E62977 944F1832 1CA6F1AA A854EBBA 8DB98C54 F2C627D8
    30117F63 CF5C44A2 93711BBF F9F67353 B01D22B2 333F4D26 B285D303 86829CD5

W : 00000000 00000000


block number = 0 : message block

M : 00000190 00000191 00000192 00000193 00000194 00000195 00000196 00000197
    00000198 00000199 0000019A 0000019B 0000019C 0000019D 0000019E 0000019F


block number = 0 : add M to B

A : BB22A196 A1C1B9EC 8B50D287 4092494B 4A0B904D 1065A62A 15AFF6D9 88EECBDD
    62189375 B55DE3B3 3FF60DE6 FA67BCE3

B : 00000010 00000010 00000010 00000010 00000010 00000010 00000010 00000010
    00000010 00000010 00000010 00000010 00000010 00000010 00000010 00000010

C : 5272938C 2A1646F0 78E62977 944F1832 1CA6F1AA A854EBBA 8DB98C54 F2C627D8
    30117F63 CF5C44A2 93711BBF F9F67353 B01D22B2 333F4D26 B285D303 86829CD5

W : 00000000 00000000


block number = 0 : xor counter W into A

A : BB22A196 A1C1B9EC 8B50D287 4092494B 4A0B904D 1065A62A 15AFF6D9 88EECBDD
    62189375 B55DE3B3 3FF60DE6 FA67BCE3

B : 00000010 00000010 00000010 00000010 00000010 00000010 00000010 00000010
    00000010 00000010 00000010 00000010 00000010 00000010 00000010 00000010

C : 5272938C 2A1646F0 78E62977 944F1832 1CA6F1AA A854EBBA 8DB98C54 F2C627D8
    30117F63 CF5C44A2 93711BBF F9F67353 B01D22B2 333F4D26 B285D303 86829CD5

W : 00000000 00000000
```

```
block number = 0 : permutation - rotate B

A : BB22A196 A1C1B9EC 8B50D287 4092494B 4A0B904D 1065A62A 15AFF6D9 88EECBDD
    62189375 B55DE3B3 3FF60DE6 FA67BCE3

B : 00200000 00200000 00200000 00200000 00200000 00200000 00200000 00200000
    00200000 00200000 00200000 00200000 00200000 00200000 00200000 00200000

C : 5272938C 2A1646F0 78E62977 944F1832 1CA6F1AA A854EBBA 8DB98C54 F2C627D8
    30117F63 CF5C44A2 93711BBF F9F67353 B01D22B2 333F4D26 B285D303 86829CD5

W : 00000000 00000000


block number = 0 : permutation (j = 0)

A : CCCBA1B5 9E2FD849 89AD738F EA03554C 4C6371DE B511FBE1 C32FB2C4 E167BC5D
    A3B8F2F7 861F866F 0FA17732 5CBE8AEC

B : 86817371 DAB4EDE3 FC92F8B3 09D76240 B3DC8E21 4AAE041E 3C904D3B 1ED843A2
    5C070D08 79A07990 F01E88CD A3017513 33745E4A 619027B6 76128C70 15BCAAB3

C : 5272938C 2A1646F0 78E62977 944F1832 1CA6F1AA A854EBBA 8DB98C54 F2C627D8
    30117F63 CF5C44A2 93711BBF F9F67353 B01D22B2 333F4D26 B285D303 86829CD5

W : 00000000 00000000


block number = 0 : permutation (j = 1)

A : B98F348F 138FE48E 0084D32A B0E0FA9B 421966F9 31F37984 0D872018 F8EA2855
    42A78E98 D97ED055 AC4EF3A7 14091C2E

B : A9DBD892 301E9E26 6FE8F3E6 D673177A DAE16D24 B3DD2796 2A91962E D6466495
    FE7ED160 1F30E851 1F463D4E 091DEF43 DB0E2592 0D2CC917 1E5DC707 2C6C82CC

C : 5272938C 2A1646F0 78E62977 944F1832 1CA6F1AA A854EBBA 8DB98C54 F2C627D8
    30117F63 CF5C44A2 93711BBF F9F67353 B01D22B2 333F4D26 B285D303 86829CD5

W : 00000000 00000000


block number = 0 : permutation (j = 2)

A : B4347AD7 C625A866 FA943F56 024604B5 6D6C6BE2 A28D22F0 68B51EB5 F354CB75
    35D6B3E1 CD7F33BC 69707FFC 6C1B1417

B : F764B11A 76172146 CEF6934D C6D28399 FE095F61 5E6018B4 5048ECF5 51353261
    6E6E36DC 63130DAD A9C69BD6 1E90EA0C 7C35073B 28D95E6D AA340E0D CB3DEE70

C : 5272938C 2A1646F0 78E62977 944F1832 1CA6F1AA A854EBBA 8DB98C54 F2C627D8
    30117F63 CF5C44A2 93711BBF F9F67353 B01D22B2 333F4D26 B285D303 86829CD5

W : 00000000 00000000


block number = 0 : permutation - add C to A

A : C8FCA331 E55C504E 003EBF26 BB6B8D83 7B0448C1 41B82789 0A7C9601 8D659CFF
    B6E2673E CA54C77B 1460FD7E 3FCB8F2D

B : F764B11A 76172146 CEF6934D C6D28399 FE095F61 5E6018B4 5048ECF5 51353261
    6E6E36DC 63130DAD A9C69BD6 1E90EA0C 7C35073B 28D95E6D AA340E0D CB3DEE70

C : 5272938C 2A1646F0 78E62977 944F1832 1CA6F1AA A854EBBA 8DB98C54 F2C627D8
    30117F63 CF5C44A2 93711BBF F9F67353 B01D22B2 333F4D26 B285D303 86829CD5
```

```
W : 00000000 00000000


block number = 0 : subtract M from C

A : C8FCA331 E55C504E 003EBF26 BB6B8D83 7B0448C1 41B82789 0A7C9601 8D659CFF
    B6E2673E CA54C77B 1460FD7E 3FCB8F2D

B : F764B11A 76172146 CEF6934D C6D28399 FE095F61 5E6018B4 5048ECF5 51353261
    6E6E36DC 63130DAD A9C69BD6 1E90EA0C 7C35073B 28D95E6D AA340E0D CB3DEE70

C : 527291FC 2A16455F 78E627E5 944F169F 1CA6F016 A854EA25 8DB98ABE F2C62641
    30117DCB CF5C4309 93711A25 F9F671B8 B01D2116 333F4B89 B285D165 86829B36

W : 00000000 00000000


block number = 0 : swap B with C

A : C8FCA331 E55C504E 003EBF26 BB6B8D83 7B0448C1 41B82789 0A7C9601 8D659CFF
    B6E2673E CA54C77B 1460FD7E 3FCB8F2D

B : 527291FC 2A16455F 78E627E5 944F169F 1CA6F016 A854EA25 8DB98ABE F2C62641
    30117DCB CF5C4309 93711A25 F9F671B8 B01D2116 333F4B89 B285D165 86829B36

C : F764B11A 76172146 CEF6934D C6D28399 FE095F61 5E6018B4 5048ECF5 51353261
    6E6E36DC 63130DAD A9C69BD6 1E90EA0C 7C35073B 28D95E6D AA340E0D CB3DEE70

W : 00000000 00000000


block number = 1 : increment counter W

A : C8FCA331 E55C504E 003EBF26 BB6B8D83 7B0448C1 41B82789 0A7C9601 8D659CFF
    B6E2673E CA54C77B 1460FD7E 3FCB8F2D

B : 527291FC 2A16455F 78E627E5 944F169F 1CA6F016 A854EA25 8DB98ABE F2C62641
    30117DCB CF5C4309 93711A25 F9F671B8 B01D2116 333F4B89 B285D165 86829B36

C : F764B11A 76172146 CEF6934D C6D28399 FE095F61 5E6018B4 5048ECF5 51353261
    6E6E36DC 63130DAD A9C69BD6 1E90EA0C 7C35073B 28D95E6D AA340E0D CB3DEE70

W : 00000001 00000000


block number = 1 : message block

M : 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
    00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000


block number = 1 : add M to B

A : C8FCA331 E55C504E 003EBF26 BB6B8D83 7B0448C1 41B82789 0A7C9601 8D659CFF
    B6E2673E CA54C77B 1460FD7E 3FCB8F2D

B : 527291FC 2A16455F 78E627E5 944F169F 1CA6F016 A854EA25 8DB98ABE F2C62641
    30117DCB CF5C4309 93711A25 F9F671B8 B01D2116 333F4B89 B285D165 86829B36

C : F764B11A 76172146 CEF6934D C6D28399 FE095F61 5E6018B4 5048ECF5 51353261
    6E6E36DC 63130DAD A9C69BD6 1E90EA0C 7C35073B 28D95E6D AA340E0D CB3DEE70

W : 00000001 00000000


block number = 1 : xor counter W into A
```

```
A : C8FCA330 E55C504E 003EBF26 BB6B8D83 7B0448C1 41B82789 0A7C9601 8D659CFF
    B6E2673E CA54C77B 1460FD7E 3FCB8F2D

B : 527291FC 2A16455F 78E627E5 944F169F 1CA6F016 A854EA25 8DB98ABE F2C62641
    30117DCB CF5C4309 93711A25 F9F671B8 B01D2116 333F4B89 B285D165 86829B36

C : F764B11A 76172146 CEF6934D C6D28399 FE095F61 5E6018B4 5048ECF5 51353261
    6E6E36DC 63130DAD A9C69BD6 1E90EA0C 7C35073B 28D95E6D AA340E0D CB3DEE70

W : 00000001 00000000


block number = 1 : permutation - rotate B

A : C8FCA330 E55C504E 003EBF26 BB6B8D83 7B0448C1 41B82789 0A7C9601 8D659CFF
    B6E2673E CA54C77B 1460FD7E 3FCB8F2D

B : 23F8A4E5 8ABE542C 4FCAF1CC 2D3F289E E02C394D D44B50A9 157D1B73 4C83E58C
    FB966022 86139EB8 344B26E2 E371F3EC 422D603A 9712667E A2CB650B 366D0D05

C : F764B11A 76172146 CEF6934D C6D28399 FE095F61 5E6018B4 5048ECF5 51353261
    6E6E36DC 63130DAD A9C69BD6 1E90EA0C 7C35073B 28D95E6D AA340E0D CB3DEE70

W : 00000001 00000000


block number = 1 : permutation (j = 0)

A : 29DABD96 F74E1392 FCCD6D6D 6016B327 193F7D56 E1E754D0 C1A21C25 713AF436
    82967D97 F48C9BBA A69674D7 3350318A

B : 6734CB7C DFF55475 5BD581F2 446284F9 2698F032 B68E0A7C 14A7D53C 17C2C0D1
    8A45422D 07545934 31FFC6EC 0A4C29AC 527F821D 26952090 46A45885 F33356D2

C : F764B11A 76172146 CEF6934D C6D28399 FE095F61 5E6018B4 5048ECF5 51353261
    6E6E36DC 63130DAD A9C69BD6 1E90EA0C 7C35073B 28D95E6D AA340E0D CB3DEE70

W : 00000001 00000000


block number = 1 : permutation (j = 1)

A : EE45873E 91F25A63 FC45FB83 01A2AA3A CF67FE6E 033DA5F2 D00AF246 1D17F764
    821A18CB C44843D5 800E8C1C 299E27BD

B : 4E4AF2A0 875E82F8 AE008BF2 64ED7F54 30D40750 56ABA8D3 56BED99B F9E459E0
    0530FC9A 60A517F4 604589A4 EAC5069D 946705AB B1E81B2D A2BDBCB3 048EA53E

C : F764B11A 76172146 CEF6934D C6D28399 FE095F61 5E6018B4 5048ECF5 51353261
    6E6E36DC 63130DAD A9C69BD6 1E90EA0C 7C35073B 28D95E6D AA340E0D CB3DEE70

W : 00000001 00000000


block number = 1 : permutation (j = 2)

A : DBE86E2D 3B7FD0AC 33B7A95F F72D25D1 8E3BCD73 4EE405AF A5468DBC 7CFF44FA
    029982C2 652E9095 527DEF92 42BF4BEA

B : E0A83416 F66EECCA F7B01C34 0266FC14 45BF9F72 69D77EF5 6135E596 FB1A69EF
    7BA5CBB8 7051D5B8 9A32610B 568AB63E D5A8766A F9015931 E8F9690A B45DFE69

C : F764B11A 76172146 CEF6934D C6D28399 FE095F61 5E6018B4 5048ECF5 51353261
    6E6E36DC 63130DAD A9C69BD6 1E90EA0C 7C35073B 28D95E6D AA340E0D CB3DEE70

W : 00000001 00000000
```

```
block number = 1 : permutation - add C to A

A : 8C89CA42 AD22E862 310841C6 C0A0B420 718171DD B2C04D06 DCD0D563 46056112
    393222C8 4DDB2E0D 3CCA7460 E702E2C2

B : E0A83416 F66EECCA F7B01C34 0266FC14 45BF9F72 69D77EF5 6135E596 FB1A69EF
    7BA5CBB8 7051D5B8 9A32610B 568AB63E D5A8766A F9015931 E8F9690A B45DFE69

C : F764B11A 76172146 CEF6934D C6D28399 FE095F61 5E6018B4 5048ECF5 51353261
    6E6E36DC 63130DAD A9C69BD6 1E90EA0C 7C35073B 28D95E6D AA340E0D CB3DEE70

W : 00000001 00000000


block number = 1 : subtract M from C

A : 8C89CA42 AD22E862 310841C6 C0A0B420 718171DD B2C04D06 DCD0D563 46056112
    393222C8 4DDB2E0D 3CCA7460 E702E2C2

B : E0A83416 F66EECCA F7B01C34 0266FC14 45BF9F72 69D77EF5 6135E596 FB1A69EF
    7BA5CBB8 7051D5B8 9A32610B 568AB63E D5A8766A F9015931 E8F9690A B45DFE69

C : F764B11A 76172146 CEF6934D C6D28399 FE095F61 5E6018B4 5048ECF5 51353261
    6E6E36DC 63130DAD A9C69BD6 1E90EA0C 7C35073B 28D95E6D AA340E0D CB3DEE70

W : 00000001 00000000


block number = 1 : swap B with C

A : 8C89CA42 AD22E862 310841C6 C0A0B420 718171DD B2C04D06 DCD0D563 46056112
    393222C8 4DDB2E0D 3CCA7460 E702E2C2

B : F764B11A 76172146 CEF6934D C6D28399 FE095F61 5E6018B4 5048ECF5 51353261
    6E6E36DC 63130DAD A9C69BD6 1E90EA0C 7C35073B 28D95E6D AA340E0D CB3DEE70

C : E0A83416 F66EECCA F7B01C34 0266FC14 45BF9F72 69D77EF5 6135E596 FB1A69EF
    7BA5CBB8 7051D5B8 9A32610B 568AB63E D5A8766A F9015931 E8F9690A B45DFE69

W : 00000001 00000000


block number = 2 : increment counter W

A : 8C89CA42 AD22E862 310841C6 C0A0B420 718171DD B2C04D06 DCD0D563 46056112
    393222C8 4DDB2E0D 3CCA7460 E702E2C2

B : F764B11A 76172146 CEF6934D C6D28399 FE095F61 5E6018B4 5048ECF5 51353261
    6E6E36DC 63130DAD A9C69BD6 1E90EA0C 7C35073B 28D95E6D AA340E0D CB3DEE70

C : E0A83416 F66EECCA F7B01C34 0266FC14 45BF9F72 69D77EF5 6135E596 FB1A69EF
    7BA5CBB8 7051D5B8 9A32610B 568AB63E D5A8766A F9015931 E8F9690A B45DFE69

W : 00000002 00000000


block number = 2 : message block

M : 00000080 00000000 00000000 00000000 00000000 00000000 00000000 00000000
    00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000


block number = 2 : add M to B

A : 8C89CA42 AD22E862 310841C6 C0A0B420 718171DD B2C04D06 DCD0D563 46056112
```

```
    393222C8 4DDB2E0D 3CCA7460 E702E2C2

B : F764B19A 76172146 CEF6934D C6D28399 FE095F61 5E6018B4 5048ECF5 51353261
    6E6E36DC 63130DAD A9C69BD6 1E90EA0C 7C35073B 28D95E6D AA340E0D CB3DEE70

C : E0A83416 F66EECCA F7B01C34 0266FC14 45BF9F72 69D77EF5 6135E596 FB1A69EF
    7BA5CBB8 7051D5B8 9A32610B 568AB63E D5A8766A F9015931 E8F9690A B45DFE69

W : 00000002 00000000


block number = 2 : xor counter W into A

A : 8C89CA40 AD22E862 310841C6 C0A0B420 718171DD B2C04D06 DCD0D563 46056112
    393222C8 4DDB2E0D 3CCA7460 E702E2C2

B : F764B19A 76172146 CEF6934D C6D28399 FE095F61 5E6018B4 5048ECF5 51353261
    6E6E36DC 63130DAD A9C69BD6 1E90EA0C 7C35073B 28D95E6D AA340E0D CB3DEE70

C : E0A83416 F66EECCA F7B01C34 0266FC14 45BF9F72 69D77EF5 6135E596 FB1A69EF
    7BA5CBB8 7051D5B8 9A32610B 568AB63E D5A8766A F9015931 E8F9690A B45DFE69

W : 00000002 00000000


block number = 2 : permutation - rotate B

A : 8C89CA40 AD22E862 310841C6 C0A0B420 718171DD B2C04D06 DCD0D563 46056112
    393222C8 4DDB2E0D 3CCA7460 E702E2C2

B : 6335EEC9 428CEC2E 269B9DED 07338DA5 BEC3FC12 3168BCC0 D9EAA091 64C2A26A
    6DB8DCDC 1B5AC626 37AD538D D4183D21 0E76F86A BCDA51B2 1C1B5468 DCE1967B

C : E0A83416 F66EECCA F7B01C34 0266FC14 45BF9F72 69D77EF5 6135E596 FB1A69EF
    7BA5CBB8 7051D5B8 9A32610B 568AB63E D5A8766A F9015931 E8F9690A B45DFE69

W : 00000002 00000000


block number = 2 : permutation (j = 0)

A : BCAC6965 BC456130 D2C53A25 9C87DCC6 E79EEF6B E013BF9B 8BB56810 BB0B3D73
    1D08DDB3 BEA6B59C BFEFC1E6 0CAE3785

B : C23FF40E 30947FA2 5A92A998 0604026F 65E6E8B1 7D3D39E4 C79FD6CC 8D718658
    39869BF4 77ECC62F 2F4A9903 5B61B239 5FBE664E 3A0E3DAA 150C6D0A DABB0FCE

C : E0A83416 F66EECCA F7B01C34 0266FC14 45BF9F72 69D77EF5 6135E596 FB1A69EF
    7BA5CBB8 7051D5B8 9A32610B 568AB63E D5A8766A F9015931 E8F9690A B45DFE69

W : 00000002 00000000


block number = 2 : permutation (j = 1)

A : 5FC1D6C9 C32754E8 211541BE 13AADBA3 031836B2 2A4EA1D9 852D1D23 E29834B8
    9109742B 28D6A906 216CD635 A27780E0

B : C28CC03F A83BBE0C 70DF0D50 6D1FECFA A53B5AB6 2D532531 51AC8453 476B73AE
    D3331EDE D3012749 807F8C47 5A96402E 439B05D1 A1AD2572 50CA38C8 A811D4DA

C : E0A83416 F66EECCA F7B01C34 0266FC14 45BF9F72 69D77EF5 6135E596 FB1A69EF
    7BA5CBB8 7051D5B8 9A32610B 568AB63E D5A8766A F9015931 E8F9690A B45DFE69

W : 00000002 00000000
```

```
block number = 2 : permutation (j = 2)

A : 4108B13A 325F6DBC 5DC0FA9B 75002C36 98BFAF7A AC636087 4BAF2F01 0FE87D16
    A1201E01 6A947CD4 E4844B86 EEC546A5

B : 656DE4B7 F8191DFF 5CE74A44 3B194BB2 F481FBA8 9706D821 01660DC2 04293495
    C1266D38 F59ED1EB B4AFC871 453B02B5 D9E9EA5C D631C9CE BAEFC5E9 411910EF

C : E0A83416 F66EECCA F7B01C34 0266FC14 45BF9F72 69D77EF5 6135E596 FB1A69EF
    7BA5CBB8 7051D5B8 9A32610B 568AB63E D5A8766A F9015931 E8F9690A B45DFE69

W : 00000002 00000000


block number = 2 : permutation - add C to A

A : 4E5861F5 2E6FB7AE B708BF8B B6DF970A 4A9F13E6 4E70FFC7 1C477078 0300DFEB
    F52C3A42 01A25E68 B7AEF964 D326F650

B : 656DE4B7 F8191DFF 5CE74A44 3B194BB2 F481FBA8 9706D821 01660DC2 04293495
    C1266D38 F59ED1EB B4AFC871 453B02B5 D9E9EA5C D631C9CE BAEFC5E9 411910EF

C : E0A83416 F66EECCA F7B01C34 0266FC14 45BF9F72 69D77EF5 6135E596 FB1A69EF
    7BA5CBB8 7051D5B8 9A32610B 568AB63E D5A8766A F9015931 E8F9690A B45DFE69

W : 00000002 00000000


block number = 2 : subtract M from C

A : 4E5861F5 2E6FB7AE B708BF8B B6DF970A 4A9F13E6 4E70FFC7 1C477078 0300DFEB
    F52C3A42 01A25E68 B7AEF964 D326F650

B : 656DE4B7 F8191DFF 5CE74A44 3B194BB2 F481FBA8 9706D821 01660DC2 04293495
    C1266D38 F59ED1EB B4AFC871 453B02B5 D9E9EA5C D631C9CE BAEFC5E9 411910EF

C : E0A83396 F66EECCA F7B01C34 0266FC14 45BF9F72 69D77EF5 6135E596 FB1A69EF
    7BA5CBB8 7051D5B8 9A32610B 568AB63E D5A8766A F9015931 E8F9690A B45DFE69

W : 00000002 00000000


block number = 2 : swap B with C

A : 4E5861F5 2E6FB7AE B708BF8B B6DF970A 4A9F13E6 4E70FFC7 1C477078 0300DFEB
    F52C3A42 01A25E68 B7AEF964 D326F650

B : E0A83396 F66EECCA F7B01C34 0266FC14 45BF9F72 69D77EF5 6135E596 FB1A69EF
    7BA5CBB8 7051D5B8 9A32610B 568AB63E D5A8766A F9015931 E8F9690A B45DFE69

C : 656DE4B7 F8191DFF 5CE74A44 3B194BB2 F481FBA8 9706D821 01660DC2 04293495
    C1266D38 F59ED1EB B4AFC871 453B02B5 D9E9EA5C D631C9CE BAEFC5E9 411910EF

W : 00000002 00000000


n0_final = 0 : add M to B

A : 4E5861F5 2E6FB7AE B708BF8B B6DF970A 4A9F13E6 4E70FFC7 1C477078 0300DFEB
    F52C3A42 01A25E68 B7AEF964 D326F650

B : E0A83416 F66EECCA F7B01C34 0266FC14 45BF9F72 69D77EF5 6135E596 FB1A69EF
    7BA5CBB8 7051D5B8 9A32610B 568AB63E D5A8766A F9015931 E8F9690A B45DFE69

C : 656DE4B7 F8191DFF 5CE74A44 3B194BB2 F481FBA8 9706D821 01660DC2 04293495
    C1266D38 F59ED1EB B4AFC871 453B02B5 D9E9EA5C D631C9CE BAEFC5E9 411910EF
```

```
W : 00000002 00000000


n0_final = 0 : xor counter W into A

A : 4E5861F7 2E6FB7AE B708BF8B B6DF970A 4A9F13E6 4E70FFC7 1C477078 0300DFEB
    F52C3A42 01A25E68 B7AEF964 D326F650

B : E0A83416 F66EECCA F7B01C34 0266FC14 45BF9F72 69D77EF5 6135E596 FB1A69EF
    7BA5CBB8 7051D5B8 9A32610B 568AB63E D5A8766A F9015931 E8F9690A B45DFE69

C : 656DE4B7 F8191DFF 5CE74A44 3B194BB2 F481FBA8 9706D821 01660DC2 04293495
    C1266D38 F59ED1EB B4AFC871 453B02B5 D9E9EA5C D631C9CE BAEFC5E9 411910EF

W : 00000002 00000000


n0_final = 0 : permutation - rotate B

A : 4E5861F7 2E6FB7AE B708BF8B B6DF970A 4A9F13E6 4E70FFC7 1C477078 0300DFEB
    F52C3A42 01A25E68 B7AEF964 D326F650

B : 682DC150 D995ECDD 3869EF60 F82804CD 3EE48B7F FDEAD3AE CB2CC26B D3DFF634
    9770F74B AB70E0A3 C2173464 6C7CAD15 ECD5AB50 B263F202 D215D1F2 FCD368BB

C : 656DE4B7 F8191DFF 5CE74A44 3B194BB2 F481FBA8 9706D821 01660DC2 04293495
    C1266D38 F59ED1EB B4AFC871 453B02B5 D9E9EA5C D631C9CE BAEFC5E9 411910EF

W : 00000002 00000000


n0_final = 0 : permutation (j = 0)

A : 5AC032DB 7835FAF9 9D3ACA4F 52A4439B 34003EF8 D9CBD4EF 371F3370 413D3009
    E50777E8 9FC51900 5A4F213A 02096846

B : 078BA66D 03F2422E F3C52752 59FBBD14 B636D7F9 DDE18C4D 5EB94858 197D239F
    34196680 36DB27B8 219EB60C 250FCD93 7C949B85 E30DE103 C6EE9655 54FD6D13

C : 656DE4B7 F8191DFF 5CE74A44 3B194BB2 F481FBA8 9706D821 01660DC2 04293495
    C1266D38 F59ED1EB B4AFC871 453B02B5 D9E9EA5C D631C9CE BAEFC5E9 411910EF

W : 00000002 00000000


n0_final = 0 : permutation (j = 1)

A : 0F05B0E0 32FAB2B8 97D516BC 4E391F09 FEBAEDD0 6C75B39A A57AD791 8CF52FFE
    E0AD084F F9602E76 D2232347 1194BE8F

B : 27C12D08 9DC6152C ABD39566 85C94B7F 733F5843 BD5CC912 90AE4C08 DC91064E
    98C8821F A0B30237 2B17855B FBD97BD0 F86C2525 55918E62 D75804C5 DAF00A27

C : 656DE4B7 F8191DFF 5CE74A44 3B194BB2 F481FBA8 9706D821 01660DC2 04293495
    C1266D38 F59ED1EB B4AFC871 453B02B5 D9E9EA5C D631C9CE BAEFC5E9 411910EF

W : 00000002 00000000


n0_final = 0 : permutation (j = 2)

A : 75DB00C6 E780CF9A 4B5D5FFE E3C1540A 1700396D 5C013C57 97B4F350 36965EFD
    A5BC40C4 09DFFE2D 6B698281 F379046C

B : 46982605 ADAA0BF9 7BBB549E 2D5E0F54 6C5A4FBF 62C6A240 95FE3810 A51CA768
    D96EC2AD E298C7C7 3E640619 3EDB56A3 AA9BF570 5D031D16 3A2674F5 B966EFDC
```

```
C : 656DE4B7 F8191DFF 5CE74A44 3B194BB2 F481FBA8 9706D821 01660DC2 04293495
    C1266D38 F59ED1EB B4AFC871 453B02B5 D9E9EA5C D631C9CE BAEFC5E9 411910EF

W : 00000002 00000000


n0_final = 0 : permutation - add C to A

A : 3748601C 1B5A9A55 B0AF1FEC FCFE71F9 975BCAA3 771789EE 1C73BB5B 49937F74
    2A39C3C0 99725169 CE40F65B 647EA088

B : 46982605 ADAA0BF9 7BBB549E 2D5E0F54 6C5A4FBF 62C6A240 95FE3810 A51CA768
    D96EC2AD E298C7C7 3E640619 3EDB56A3 AA9BF570 5D031D16 3A2674F5 B966EFDC

C : 656DE4B7 F8191DFF 5CE74A44 3B194BB2 F481FBA8 9706D821 01660DC2 04293495
    C1266D38 F59ED1EB B4AFC871 453B02B5 D9E9EA5C D631C9CE BAEFC5E9 411910EF

W : 00000002 00000000


n0_final = 0 : subtract M from C

A : 3748601C 1B5A9A55 B0AF1FEC FCFE71F9 975BCAA3 771789EE 1C73BB5B 49937F74
    2A39C3C0 99725169 CE40F65B 647EA088

B : 46982605 ADAA0BF9 7BBB549E 2D5E0F54 6C5A4FBF 62C6A240 95FE3810 A51CA768
    D96EC2AD E298C7C7 3E640619 3EDB56A3 AA9BF570 5D031D16 3A2674F5 B966EFDC

C : 656DE437 F8191DFF 5CE74A44 3B194BB2 F481FBA8 9706D821 01660DC2 04293495
    C1266D38 F59ED1EB B4AFC871 453B02B5 D9E9EA5C D631C9CE BAEFC5E9 411910EF

W : 00000002 00000000


n0_final = 0 : swap B with C

A : 3748601C 1B5A9A55 B0AF1FEC FCFE71F9 975BCAA3 771789EE 1C73BB5B 49937F74
    2A39C3C0 99725169 CE40F65B 647EA088

B : 656DE437 F8191DFF 5CE74A44 3B194BB2 F481FBA8 9706D821 01660DC2 04293495
    C1266D38 F59ED1EB B4AFC871 453B02B5 D9E9EA5C D631C9CE BAEFC5E9 411910EF

C : 46982605 ADAA0BF9 7BBB549E 2D5E0F54 6C5A4FBF 62C6A240 95FE3810 A51CA768
    D96EC2AD E298C7C7 3E640619 3EDB56A3 AA9BF570 5D031D16 3A2674F5 B966EFDC

W : 00000002 00000000


n0_final = 1 : add M to B

A : 3748601C 1B5A9A55 B0AF1FEC FCFE71F9 975BCAA3 771789EE 1C73BB5B 49937F74
    2A39C3C0 99725169 CE40F65B 647EA088

B : 656DE4B7 F8191DFF 5CE74A44 3B194BB2 F481FBA8 9706D821 01660DC2 04293495
    C1266D38 F59ED1EB B4AFC871 453B02B5 D9E9EA5C D631C9CE BAEFC5E9 411910EF

C : 46982605 ADAA0BF9 7BBB549E 2D5E0F54 6C5A4FBF 62C6A240 95FE3810 A51CA768
    D96EC2AD E298C7C7 3E640619 3EDB56A3 AA9BF570 5D031D16 3A2674F5 B966EFDC

W : 00000002 00000000


n0_final = 1 : xor counter W into A

A : 3748601E 1B5A9A55 B0AF1FEC FCFE71F9 975BCAA3 771789EE 1C73BB5B 49937F74
    2A39C3C0 99725169 CE40F65B 647EA088
```

```
B : 656DE4B7 F8191DFF 5CE74A44 3B194BB2 F481FBA8 9706D821 01660DC2 04293495
    C1266D38 F59ED1EB B4AFC871 453B02B5 D9E9EA5C D631C9CE BAEFC5E9 411910EF

C : 46982605 ADAA0BF9 7BBB549E 2D5E0F54 6C5A4FBF 62C6A240 95FE3810 A51CA768
    D96EC2AD E298C7C7 3E640619 3EDB56A3 AA9BF570 5D031D16 3A2674F5 B966EFDC

W : 00000002 00000000


n0_final = 1 : permutation - rotate B

A : 3748601E 1B5A9A55 B0AF1FEC FCFE71F9 975BCAA3 771789EE 1C73BB5B 49937F74
    2A39C3C0 99725169 CE40F65B 647EA088

B : C96ECADB 3BFFF032 9488B9CE 97647632 F751E903 B0432E0D 1B8402CC 692A0852
    DA71824C A3D7EB3D 90E3695F 056A8A76 D4B9B3D3 939DAC63 8BD375DF 21DE8232

C : 46982605 ADAA0BF9 7BBB549E 2D5E0F54 6C5A4FBF 62C6A240 95FE3810 A51CA768
    D96EC2AD E298C7C7 3E640619 3EDB56A3 AA9BF570 5D031D16 3A2674F5 B966EFDC

W : 00000002 00000000


n0_final = 1 : permutation (j = 0)

A : 1A4EA719 F0698671 F1A059F1 78AC9659 E7FC4989 09A85C59 C0005790 6AF3DAD6
    FDD0BC75 A7610544 6EBB4706 5C84BC8F

B : 20BB2302 0F6DFD0A 4B8AE634 62053013 F6A06471 96D1FFBD 08F7ADF7 4758358D
    B6CC4713 1F312CC0 B0826A46 A9AE579C 4CC23F41 28AD2149 19F94DB1 C4EE6DC2

C : 46982605 ADAA0BF9 7BBB549E 2D5E0F54 6C5A4FBF 62C6A240 95FE3810 A51CA768
    D96EC2AD E298C7C7 3E640619 3EDB56A3 AA9BF570 5D031D16 3A2674F5 B966EFDC

W : 00000002 00000000


n0_final = 1 : permutation (j = 1)

A : 817E579C E55778E3 48717CB5 A5F41442 11A17941 C8FDF843 AF1660C4 4A030F74
    3F12381F 9E73AA5A A44E22BC 88044CB8

B : 5AC3EBAC 6C12C680 9D71E54C 92C5BCB1 2DAD0F03 4C2FAADE 4A5E86AD F94BD85D
    13192644 24CADE9C D68A57C7 09574484 77DAF83C 6658452E 631B0459 3C202B0E

C : 46982605 ADAA0BF9 7BBB549E 2D5E0F54 6C5A4FBF 62C6A240 95FE3810 A51CA768
    D96EC2AD E298C7C7 3E640619 3EDB56A3 AA9BF570 5D031D16 3A2674F5 B966EFDC

W : 00000002 00000000


n0_final = 1 : permutation (j = 2)

A : C02FFE30 A65235AD A6DDBBDD F5741E0E 9D2636FC 65514408 C077EC60 8D365A4C
    30F9C34D D85DAB73 65047BDA A3B324FF

B : CA3AFE3C 298219E1 5E0BF2B8 8F151F6D 648A1FC9 C1F29FEE CD9F4978 F81C514A
    44EB858B D33B06CF 929CBC10 60672CBB 20B3CCCA EB12DED0 5CCD8C97 240C8D1C

C : 46982605 ADAA0BF9 7BBB549E 2D5E0F54 6C5A4FBF 62C6A240 95FE3810 A51CA768
    D96EC2AD E298C7C7 3E640619 3EDB56A3 AA9BF570 5D031D16 3A2674F5 B966EFDC

W : 00000002 00000000


n0_final = 1 : permutation - add C to A
```

```
A : E5D05403 03E0A0E1 1451872C 41541FB1 2907DD94 F1B27C79 B3816260 DD53ED13
    424FD0AC C8C2B34F 076702F7 B23BD81D

B : CA3AFE3C 298219E1 5E0BF2B8 8F151F6D 648A1FC9 C1F29FEE CD9F4978 F81C514A
    44EB858B D33B06CF 929CBC10 60672CBB 20B3CCCA EB12DED0 5CCD8C97 240C8D1C

C : 46982605 ADAA0BF9 7BBB549E 2D5E0F54 6C5A4FBF 62C6A240 95FE3810 A51CA768
    D96EC2AD E298C7C7 3E640619 3EDB56A3 AA9BF570 5D031D16 3A2674F5 B966EFDC

W : 00000002 00000000


n0_final = 1 : subtract M from C

A : E5D05403 03E0A0E1 1451872C 41541FB1 2907DD94 F1B27C79 B3816260 DD53ED13
    424FD0AC C8C2B34F 076702F7 B23BD81D

B : CA3AFE3C 298219E1 5E0BF2B8 8F151F6D 648A1FC9 C1F29FEE CD9F4978 F81C514A
    44EB858B D33B06CF 929CBC10 60672CBB 20B3CCCA EB12DED0 5CCD8C97 240C8D1C

C : 46982585 ADAA0BF9 7BBB549E 2D5E0F54 6C5A4FBF 62C6A240 95FE3810 A51CA768
    D96EC2AD E298C7C7 3E640619 3EDB56A3 AA9BF570 5D031D16 3A2674F5 B966EFDC

W : 00000002 00000000


n0_final = 1 : swap B with C

A : E5D05403 03E0A0E1 1451872C 41541FB1 2907DD94 F1B27C79 B3816260 DD53ED13
    424FD0AC C8C2B34F 076702F7 B23BD81D

B : 46982585 ADAA0BF9 7BBB549E 2D5E0F54 6C5A4FBF 62C6A240 95FE3810 A51CA768
    D96EC2AD E298C7C7 3E640619 3EDB56A3 AA9BF570 5D031D16 3A2674F5 B966EFDC

C : CA3AFE3C 298219E1 5E0BF2B8 8F151F6D 648A1FC9 C1F29FEE CD9F4978 F81C514A
    44EB858B D33B06CF 929CBC10 60672CBB 20B3CCCA EB12DED0 5CCD8C97 240C8D1C

W : 00000002 00000000


n0_final = 2 : add M to B

A : E5D05403 03E0A0E1 1451872C 41541FB1 2907DD94 F1B27C79 B3816260 DD53ED13
    424FD0AC C8C2B34F 076702F7 B23BD81D

B : 46982605 ADAA0BF9 7BBB549E 2D5E0F54 6C5A4FBF 62C6A240 95FE3810 A51CA768
    D96EC2AD E298C7C7 3E640619 3EDB56A3 AA9BF570 5D031D16 3A2674F5 B966EFDC

C : CA3AFE3C 298219E1 5E0BF2B8 8F151F6D 648A1FC9 C1F29FEE CD9F4978 F81C514A
    44EB858B D33B06CF 929CBC10 60672CBB 20B3CCCA EB12DED0 5CCD8C97 240C8D1C

W : 00000002 00000000


n0_final = 2 : xor counter W into A

A : E5D05401 03E0A0E1 1451872C 41541FB1 2907DD94 F1B27C79 B3816260 DD53ED13
    424FD0AC C8C2B34F 076702F7 B23BD81D

B : 46982605 ADAA0BF9 7BBB549E 2D5E0F54 6C5A4FBF 62C6A240 95FE3810 A51CA768
    D96EC2AD E298C7C7 3E640619 3EDB56A3 AA9BF570 5D031D16 3A2674F5 B966EFDC

C : CA3AFE3C 298219E1 5E0BF2B8 8F151F6D 648A1FC9 C1F29FEE CD9F4978 F81C514A
    44EB858B D33B06CF 929CBC10 60672CBB 20B3CCCA EB12DED0 5CCD8C97 240C8D1C

W : 00000002 00000000
```

```
n0_final = 2 : permutation - rotate B

A : E5D05401 03E0A0E1 1451872C 41541FB1 2907DD94 F1B27C79 B3816260 DD53ED13
    424FD0AC C8C2B34F 076702F7 B23BD81D

B : 4C0A8D30 17F35B54 A93CF776 1EA85ABC 9F7ED8B4 4480C58D 70212BFC 4ED14A39
    855BB2DD 8F8FC531 0C327CC8 AD467DB6 EAE15537 3A2CBA06 E9EA744C DFB972CD

C : CA3AFE3C 298219E1 5E0BF2B8 8F151F6D 648A1FC9 C1F29FEE CD9F4978 F81C514A
    44EB858B D33B06CF 929CBC10 60672CBB 20B3CCCA EB12DED0 5CCD8C97 240C8D1C

W : 00000002 00000000


n0_final = 2 : permutation (j = 0)

A : 3476CE6E C2B87F0B 22FFE820 9EB55800 EE3D7C7F C4B99A54 901CB997 DEAF21BC
    A4D0E8EB 706AFF93 CCFCAC28 9858645E

B : 6A1D5F49 C312703F D8DB8B9A 4008EA63 2F3F32E9 B247EEB1 8FA11190 BCF24A31
    519872AF 908A8A0F 2B67AA47 3D2B60CC 1E4B9BFE 491EF4F8 0ED4FF46 DE384264

C : CA3AFE3C 298219E1 5E0BF2B8 8F151F6D 648A1FC9 C1F29FEE CD9F4978 F81C514A
    44EB858B D33B06CF 929CBC10 60672CBB 20B3CCCA EB12DED0 5CCD8C97 240C8D1C

W : 00000002 00000000


n0_final = 2 : permutation (j = 1)

A : 31F27E78 3D42EF58 A5D409BE 17C42E99 4F341777 40F90AE5 7F89E078 3F3E37F8
    D619A8FA EFF73328 723396D0 3BA31091

B : 965760B8 4C90F859 66054B6C F530CAB5 779832D7 748711B4 928E4A0E BDB87B0D
    6D3D64D9 E3A804B8 0CE4A2CF 926D10FE 8C5CDF74 2D3B1CEA 9DDFE10B 7CB14CCE

C : CA3AFE3C 298219E1 5E0BF2B8 8F151F6D 648A1FC9 C1F29FEE CD9F4978 F81C514A
    44EB858B D33B06CF 929CBC10 60672CBB 20B3CCCA EB12DED0 5CCD8C97 240C8D1C

W : 00000002 00000000


n0_final = 2 : permutation (j = 2)

A : 23DD44CC CC74D106 D99B37D8 429EB8F6 789EFCB1 3B98A35C DDB85A3F CE00C04C
    842C3359 552123AF 8EEBD747 14B2A6A7

B : D4163C6A 49313E63 0D1ACCBE 7AD73B3E 3312DE9D DA850D91 03785C3A C611B112
    5D1BCAFC 033755D2 3B8EE05E 15251E4E 636A724F F0A8E584 4AABEAAF 122FC0C4

C : CA3AFE3C 298219E1 5E0BF2B8 8F151F6D 648A1FC9 C1F29FEE CD9F4978 F81C514A
    44EB858B D33B06CF 929CBC10 60672CBB 20B3CCCA EB12DED0 5CCD8C97 240C8D1C

W : 00000002 00000000


n0_final = 2 : permutation - add C to A

A : 37661E10 1BEDBBD5 B022D077 CB1781BD 23DCFA84 AF4946EC 9C681ADD 8C48B88C
    6BC4D0CB 1F4A95CD 0F2C5CD4 D1BC38C6

B : D4163C6A 49313E63 0D1ACCBE 7AD73B3E 3312DE9D DA850D91 03785C3A C611B112
    5D1BCAFC 033755D2 3B8EE05E 15251E4E 636A724F F0A8E584 4AABEAAF 122FC0C4

C : CA3AFE3C 298219E1 5E0BF2B8 8F151F6D 648A1FC9 C1F29FEE CD9F4978 F81C514A
    44EB858B D33B06CF 929CBC10 60672CBB 20B3CCCA EB12DED0 5CCD8C97 240C8D1C
```

```
W : 00000002 00000000


n0_final = 2 : subtract M from C

A : 37661E10 1BEDBBD5 B022D077 CB1781BD 23DCFA84 AF4946EC 9C681ADD 8C48B88C
    6BC4D0CB 1F4A95CD 0F2C5CD4 D1BC38C6

B : D4163C6A 49313E63 0D1ACCBE 7AD73B3E 3312DE9D DA850D91 03785C3A C611B112
    5D1BCAFC 033755D2 3B8EE05E 15251E4E 636A724F F0A8E584 4AABEAAF 122FC0C4

C : CA3AFDBC 298219E1 5E0BF2B8 8F151F6D 648A1FC9 C1F29FEE CD9F4978 F81C514A
    44EB858B D33B06CF 929CBC10 60672CBB 20B3CCCA EB12DED0 5CCD8C97 240C8D1C

W : 00000002 00000000


n0_final = 2 : swap B with C  (final state)

A : 37661E10 1BEDBBD5 B022D077 CB1781BD 23DCFA84 AF4946EC 9C681ADD 8C48B88C
    6BC4D0CB 1F4A95CD 0F2C5CD4 D1BC38C6

B : CA3AFDBC 298219E1 5E0BF2B8 8F151F6D 648A1FC9 C1F29FEE CD9F4978 F81C514A
    44EB858B D33B06CF 929CBC10 60672CBB 20B3CCCA EB12DED0 5CCD8C97 240C8D1C

C : D4163C6A 49313E63 0D1ACCBE 7AD73B3E 3312DE9D DA850D91 03785C3A C611B112
    5D1BCAFC 033755D2 3B8EE05E 15251E4E 636A724F F0A8E584 4AABEAAF 122FC0C4

W : 00000002 00000000


Hash value (word array):

H : 3312DE9D DA850D91 03785C3A C611B112 5D1BCAFC 033755D2 3B8EE05E 15251E4E
    636A724F F0A8E584 4AABEAAF 122FC0C4


Hash value (byte array):

H : 9D DE 12 33 91 0D 85 DA 3A 5C 78 03 12 B1 11 C6
    FC CA 1B 5D D2 55 37 03 5E E0 8E 3B 4E 1E 25 15
    4F 72 6A 63 84 E5 A8 F0 AF EA AB 4A C4 C0 2F 12
```

B.8 Intermediate States for Shabal-384 (Message B)

```
init

A : 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
    00000000 00000000 00000000 00000000

B : 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
    00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000

C : 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
    00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000

W : FFFFFFFF FFFFFFFF


block number = -1 : message block

M : 00000180 00000181 00000182 00000183 00000184 00000185 00000186 00000187
    00000188 00000189 0000018A 0000018B 0000018C 0000018D 0000018E 0000018F
```

```
block number = -1 : add M to B

A : 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
    00000000 00000000 00000000 00000000

B : 00000180 00000181 00000182 00000183 00000184 00000185 00000186 00000187
    00000188 00000189 0000018A 0000018B 0000018C 0000018D 0000018E 0000018F

C : 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
    00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000

W : FFFFFFFF FFFFFFFF


block number = -1 : xor counter W into A

A : FFFFFFFF FFFFFFFF 00000000 00000000 00000000 00000000 00000000 00000000
    00000000 00000000 00000000 00000000

B : 00000180 00000181 00000182 00000183 00000184 00000185 00000186 00000187
    00000188 00000189 0000018A 0000018B 0000018C 0000018D 0000018E 0000018F

C : 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
    00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000

W : FFFFFFFF FFFFFFFF


block number = -1 : permutation - rotate B

A : FFFFFFFF FFFFFFFF 00000000 00000000 00000000 00000000 00000000 00000000
    00000000 00000000 00000000 00000000

B : 03000000 03020000 03040000 03060000 03080000 030A0000 030C0000 030E0000
    03100000 03120000 03140000 03160000 03180000 031A0000 031C0000 031E0000

C : 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
    00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000

W : FFFFFFFF FFFFFFFF


block number = -1 : permutation (j = 0)

A : 73B2093A 2A799470 05BF52F2 607A3AE1 BDA5E845 9A4F4C59 45672DC2 977B1FC2
    FD372A8B D7D97B33 8A5D1B6E 4D17CE45

B : 05080182 F1B8E8B6 D44C4392 F9C1290D 444A17BA 63A4B3A6 BC80D23D 6E98E03D
    04E8D574 2E0284CC 738AE491 B4C431BA 8A7DF6C5 D3B26B8F FC78AD0D 99B9C51E

C : 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
    00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000

W : FFFFFFFF FFFFFFFF


block number = -1 : permutation (j = 1)

A : 34DCDD7F 3399F8DC A47ABF91 51043F32 FDF93568 64558055 3E48B569 0D011A26
    DB56076A C383CE67 FA750B12 041BF733

B : AF9C5E68 55490C52 C44C5DCB C0EE508F AC3DD7E1 FB3556D4 7C8B5096 26D5C8B6
    C2F28868 90630EBB BC90894C C773A3B8 16FD271C 3CCEA8B5 3946108D C18D6FE4

C : 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
```

```
    00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000

W : FFFFFFFF FFFFFFFF


block number = -1 : permutation (j = 2)

A : BB22A196 A1C1B9EC 8B50D287 4092494B 4A0B904D 1065A62A 15AFF6D9 88EECBDD
    62189375 B55DE3B3 3FF60DE6 FA67BCE3

B : 5272938C 2A1646F0 78E62977 944F1832 1CA6F1AA A854EBBA 8DB98C54 F2C627D8
    30117F63 CF5C44A2 93711BBF F9F67353 B01D22B2 333F4D26 B285D303 86829CD5

C : 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
    00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000

W : FFFFFFFF FFFFFFFF


block number = -1 : permutation - add C to A

A : BB22A196 A1C1B9EC 8B50D287 4092494B 4A0B904D 1065A62A 15AFF6D9 88EECBDD
    62189375 B55DE3B3 3FF60DE6 FA67BCE3

B : 5272938C 2A1646F0 78E62977 944F1832 1CA6F1AA A854EBBA 8DB98C54 F2C627D8
    30117F63 CF5C44A2 93711BBF F9F67353 B01D22B2 333F4D26 B285D303 86829CD5

C : 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
    00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000

W : FFFFFFFF FFFFFFFF


block number = -1 : subtract M from C

A : BB22A196 A1C1B9EC 8B50D287 4092494B 4A0B904D 1065A62A 15AFF6D9 88EECBDD
    62189375 B55DE3B3 3FF60DE6 FA67BCE3

B : 5272938C 2A1646F0 78E62977 944F1832 1CA6F1AA A854EBBA 8DB98C54 F2C627D8
    30117F63 CF5C44A2 93711BBF F9F67353 B01D22B2 333F4D26 B285D303 86829CD5

C : FFFFFE80 FFFFFE7F FFFFFE7E FFFFFE7D FFFFFE7C FFFFFE7B FFFFFE7A FFFFFE79
    FFFFFE78 FFFFFE77 FFFFFE76 FFFFFE75 FFFFFE74 FFFFFE73 FFFFFE72 FFFFFE71

W : FFFFFFFF FFFFFFFF


block number = -1 : swap B with C

A : BB22A196 A1C1B9EC 8B50D287 4092494B 4A0B904D 1065A62A 15AFF6D9 88EECBDD
    62189375 B55DE3B3 3FF60DE6 FA67BCE3

B : FFFFFE80 FFFFFE7F FFFFFE7E FFFFFE7D FFFFFE7C FFFFFE7B FFFFFE7A FFFFFE79
    FFFFFE78 FFFFFE77 FFFFFE76 FFFFFE75 FFFFFE74 FFFFFE73 FFFFFE72 FFFFFE71

C : 5272938C 2A1646F0 78E62977 944F1832 1CA6F1AA A854EBBA 8DB98C54 F2C627D8
    30117F63 CF5C44A2 93711BBF F9F67353 B01D22B2 333F4D26 B285D303 86829CD5

W : FFFFFFFF FFFFFFFF


block number = 0 : increment counter W

A : BB22A196 A1C1B9EC 8B50D287 4092494B 4A0B904D 1065A62A 15AFF6D9 88EECBDD
    62189375 B55DE3B3 3FF60DE6 FA67BCE3

B : FFFFFE80 FFFFFE7F FFFFFE7E FFFFFE7D FFFFFE7C FFFFFE7B FFFFFE7A FFFFFE79
```

```
    FFFFFE78 FFFFFE77 FFFFFE76 FFFFFE75 FFFFFE74 FFFFFE73 FFFFFE72 FFFFFE71

C : 5272938C 2A1646F0 78E62977 944F1832 1CA6F1AA A854EBBA 8DB98C54 F2C627D8
    30117F63 CF5C44A2 93711BBF F9F67353 B01D22B2 333F4D26 B285D303 86829CD5

W : 00000000 00000000


block number = 0 : message block

M : 00000190 00000191 00000192 00000193 00000194 00000195 00000196 00000197
    00000198 00000199 0000019A 0000019B 0000019C 0000019D 0000019E 0000019F


block number = 0 : add M to B

A : BB22A196 A1C1B9EC 8B50D287 4092494B 4A0B904D 1065A62A 15AFF6D9 88EECBDD
    62189375 B55DE3B3 3FF60DE6 FA67BCE3

B : 00000010 00000010 00000010 00000010 00000010 00000010 00000010 00000010
    00000010 00000010 00000010 00000010 00000010 00000010 00000010 00000010

C : 5272938C 2A1646F0 78E62977 944F1832 1CA6F1AA A854EBBA 8DB98C54 F2C627D8
    30117F63 CF5C44A2 93711BBF F9F67353 B01D22B2 333F4D26 B285D303 86829CD5

W : 00000000 00000000


block number = 0 : xor counter W into A

A : BB22A196 A1C1B9EC 8B50D287 4092494B 4A0B904D 1065A62A 15AFF6D9 88EECBDD
    62189375 B55DE3B3 3FF60DE6 FA67BCE3

B : 00000010 00000010 00000010 00000010 00000010 00000010 00000010 00000010
    00000010 00000010 00000010 00000010 00000010 00000010 00000010 00000010

C : 5272938C 2A1646F0 78E62977 944F1832 1CA6F1AA A854EBBA 8DB98C54 F2C627D8
    30117F63 CF5C44A2 93711BBF F9F67353 B01D22B2 333F4D26 B285D303 86829CD5

W : 00000000 00000000


block number = 0 : permutation - rotate B

A : BB22A196 A1C1B9EC 8B50D287 4092494B 4A0B904D 1065A62A 15AFF6D9 88EECBDD
    62189375 B55DE3B3 3FF60DE6 FA67BCE3

B : 00200000 00200000 00200000 00200000 00200000 00200000 00200000 00200000
    00200000 00200000 00200000 00200000 00200000 00200000 00200000 00200000

C : 5272938C 2A1646F0 78E62977 944F1832 1CA6F1AA A854EBBA 8DB98C54 F2C627D8
    30117F63 CF5C44A2 93711BBF F9F67353 B01D22B2 333F4D26 B285D303 86829CD5

W : 00000000 00000000


block number = 0 : permutation (j = 0)

A : CCCBA1B5 9E2FD849 89AD738F EA03554C 4C6371DE B511FBE1 C32FB2C4 E167BC5D
    A3B8F2F7 861F866F 0FA17732 5CBE8AEC

B : 86817371 DAB4EDE3 FC92F8B3 09D76240 B3DC8E21 4AAE041E 3C904D3B 1ED843A2
    5C070D08 79A07990 F01E88CD A3017513 33745E4A 619027B6 76128C70 15BCAAB3

C : 5272938C 2A1646F0 78E62977 944F1832 1CA6F1AA A854EBBA 8DB98C54 F2C627D8
    30117F63 CF5C44A2 93711BBF F9F67353 B01D22B2 333F4D26 B285D303 86829CD5
```

```
W : 00000000 00000000


block number = 0 : permutation (j = 1)

A : B98F348F 138FE48E 0084D32A B0E0FA9B 421966F9 31F37984 0D872018 F8EA2855
    42A78E98 D97ED055 AC4EF3A7 14091C2E

B : A9DBD892 301E9E26 6FE8F3E6 D673177A DAE16D24 B3DD2796 2A91962E D6466495
    FE7ED160 1F30E851 1F463D4E 091DEF43 DB0E2592 0D2CC917 1E5DC707 2C6C82CC

C : 5272938C 2A1646F0 78E62977 944F1832 1CA6F1AA A854EBBA 8DB98C54 F2C627D8
    30117F63 CF5C44A2 93711BBF F9F67353 B01D22B2 333F4D26 B285D303 86829CD5

W : 00000000 00000000


block number = 0 : permutation (j = 2)

A : B4347AD7 C625A866 FA943F56 024604B5 6D6C6BE2 A28D22F0 68B51EB5 F354CB75
    35D6B3E1 CD7F33BC 69707FFC 6C1B1417

B : F764B11A 76172146 CEF6934D C6D28399 FE095F61 5E6018B4 5048ECF5 51353261
    6E6E36DC 63130DAD A9C69BD6 1E90EA0C 7C35073B 28D95E6D AA340E0D CB3DEE70

C : 5272938C 2A1646F0 78E62977 944F1832 1CA6F1AA A854EBBA 8DB98C54 F2C627D8
    30117F63 CF5C44A2 93711BBF F9F67353 B01D22B2 333F4D26 B285D303 86829CD5

W : 00000000 00000000


block number = 0 : permutation - add C to A

A : C8FCA331 E55C504E 003EBF26 BB6B8D83 7B0448C1 41B82789 0A7C9601 8D659CFF
    B6E2673E CA54C77B 1460FD7E 3FCB8F2D

B : F764B11A 76172146 CEF6934D C6D28399 FE095F61 5E6018B4 5048ECF5 51353261
    6E6E36DC 63130DAD A9C69BD6 1E90EA0C 7C35073B 28D95E6D AA340E0D CB3DEE70

C : 5272938C 2A1646F0 78E62977 944F1832 1CA6F1AA A854EBBA 8DB98C54 F2C627D8
    30117F63 CF5C44A2 93711BBF F9F67353 B01D22B2 333F4D26 B285D303 86829CD5

W : 00000000 00000000


block number = 0 : subtract M from C

A : C8FCA331 E55C504E 003EBF26 BB6B8D83 7B0448C1 41B82789 0A7C9601 8D659CFF
    B6E2673E CA54C77B 1460FD7E 3FCB8F2D

B : F764B11A 76172146 CEF6934D C6D28399 FE095F61 5E6018B4 5048ECF5 51353261
    6E6E36DC 63130DAD A9C69BD6 1E90EA0C 7C35073B 28D95E6D AA340E0D CB3DEE70

C : 527291FC 2A16455F 78E627E5 944F169F 1CA6F016 A854EA25 8DB98ABE F2C62641
    30117DCB CF5C4309 93711A25 F9F671B8 B01D2116 333F4B89 B285D165 86829B36

W : 00000000 00000000


block number = 0 : swap B with C

A : C8FCA331 E55C504E 003EBF26 BB6B8D83 7B0448C1 41B82789 0A7C9601 8D659CFF
    B6E2673E CA54C77B 1460FD7E 3FCB8F2D

B : 527291FC 2A16455F 78E627E5 944F169F 1CA6F016 A854EA25 8DB98ABE F2C62641
    30117DCB CF5C4309 93711A25 F9F671B8 B01D2116 333F4B89 B285D165 86829B36
```

```
C : F764B11A 76172146 CEF6934D C6D28399 FE095F61 5E6018B4 5048ECF5 51353261
    6E6E36DC 63130DAD A9C69BD6 1E90EA0C 7C35073B 28D95E6D AA340E0D CB3DEE70

W : 00000000 00000000


block number = 1 : increment counter W

A : C8FCA331 E55C504E 003EBF26 BB6B8D83 7B0448C1 41B82789 0A7C9601 8D659CFF
    B6E2673E CA54C77B 1460FD7E 3FCB8F2D

B : 527291FC 2A16455F 78E627E5 944F169F 1CA6F016 A854EA25 8DB98ABE F2C62641
    30117DCB CF5C4309 93711A25 F9F671B8 B01D2116 333F4B89 B285D165 86829B36

C : F764B11A 76172146 CEF6934D C6D28399 FE095F61 5E6018B4 5048ECF5 51353261
    6E6E36DC 63130DAD A9C69BD6 1E90EA0C 7C35073B 28D95E6D AA340E0D CB3DEE70

W : 00000001 00000000


block number = 1 : message block

M : 64636261 68676665 6C6B6A69 706F6E6D 74737271 78777675 302D7A79 34333231
    38373635 42412D39 46454443 4A494847 4E4D4C4B 5251504F 56555453 5A595857


block number = 1 : add M to B

A : C8FCA331 E55C504E 003EBF26 BB6B8D83 7B0448C1 41B82789 0A7C9601 8D659CFF
    B6E2673E CA54C77B 1460FD7E 3FCB8F2D

B : B6D5F45D 927DABC4 E551924E 04BE850C 911A6287 20CC609A BDE70537 26F95872
    6848B400 119D7042 D9B65E68 443FB9FF FE6A6D61 85909BD8 08DB25B8 E0DBF38D

C : F764B11A 76172146 CEF6934D C6D28399 FE095F61 5E6018B4 5048ECF5 51353261
    6E6E36DC 63130DAD A9C69BD6 1E90EA0C 7C35073B 28D95E6D AA340E0D CB3DEE70

W : 00000001 00000000


block number = 1 : xor counter W into A

A : C8FCA330 E55C504E 003EBF26 BB6B8D83 7B0448C1 41B82789 0A7C9601 8D659CFF
    B6E2673E CA54C77B 1460FD7E 3FCB8F2D

B : B6D5F45D 927DABC4 E551924E 04BE850C 911A6287 20CC609A BDE70537 26F95872
    6848B400 119D7042 D9B65E68 443FB9FF FE6A6D61 85909BD8 08DB25B8 E0DBF38D

C : F764B11A 76172146 CEF6934D C6D28399 FE095F61 5E6018B4 5048ECF5 51353261
    6E6E36DC 63130DAD A9C69BD6 1E90EA0C 7C35073B 28D95E6D AA340E0D CB3DEE70

W : 00000001 00000000


block number = 1 : permutation - rotate B

A : C8FCA330 E55C504E 003EBF26 BB6B8D83 7B0448C1 41B82789 0A7C9601 8D659CFF
    B6E2673E CA54C77B 1460FD7E 3FCB8F2D

B : E8BB6DAB 578924FB 249DCAA3 0A18097D C50F2234 C1344198 0A6F7BCE B0E44DF2
    6800D091 E084233A BCD1B36C 73FE887F DAC3FCD4 37B10B21 4B7011B6 E71BC1B7

C : F764B11A 76172146 CEF6934D C6D28399 FE095F61 5E6018B4 5048ECF5 51353261
    6E6E36DC 63130DAD A9C69BD6 1E90EA0C 7C35073B 28D95E6D AA340E0D CB3DEE70

W : 00000001 00000000
```

```
block number = 1 : permutation (j = 0)

A : 7A6E1776 DDF02970 D65D1D53 BAE24561 94D00BB5 B33B9301 201C72B1 241B1BA9
    DFFFF69C 3C1F4733 F8959DC0 E20B74B0

B : 57F1D267 E7A41B5F 0DC104BC B403D9FC E131B023 CEACEFCF CB3D7AD2 BA2C7FB3
    F001A841 02E8FEB9 7EC904E6 FA099BB1 30161120 4D6DC0CD BF42C1C0 8B2A39F1

C : F764B11A 76172146 CEF6934D C6D28399 FE095F61 5E6018B4 5048ECF5 51353261
    6E6E36DC 63130DAD A9C69BD6 1E90EA0C 7C35073B 28D95E6D AA340E0D CB3DEE70

W : 00000001 00000000


block number = 1 : permutation (j = 1)

A : 62CDE717 7C878A2A F18DF07D D07D4C1E 62FC6E38 D68DA6EA D27ACA81 18B395DF
    D32F10CD 8C02A241 D174FEBD 52541198

B : 7E8D36A0 F8FF91B9 035A767F 8133DD8D EEB38F75 EEA48221 B8F1F4E7 D9F31100
    7D31486B 86A988A7 F3E0064E DB918482 FD2FB387 B3A9D88F 5300B6FF F11819C3

C : F764B11A 76172146 CEF6934D C6D28399 FE095F61 5E6018B4 5048ECF5 51353261
    6E6E36DC 63130DAD A9C69BD6 1E90EA0C 7C35073B 28D95E6D AA340E0D CB3DEE70

W : 00000001 00000000


block number = 1 : permutation (j = 2)

A : 0D3444D1 A2C4A76B 6F438852 4C30C6D3 3D4487EC 40C7C67B D901E7E7 AD4F7F14
    26F1A6B1 623E0EEF 76A36B21 FAD1A8BD

B : 30E2F355 EA9BCDD5 99AD1970 869D9BE5 2FACA5C5 80725CD7 E15F9E62 00291B2D
    38D9E8C5 B26B28CB C13E1485 E59389EE 23513E41 FA92400F 2F5DF920 E71E64C5

C : F764B11A 76172146 CEF6934D C6D28399 FE095F61 5E6018B4 5048ECF5 51353261
    6E6E36DC 63130DAD A9C69BD6 1E90EA0C 7C35073B 28D95E6D AA340E0D CB3DEE70

W : 00000001 00000000


block number = 1 : permutation - add C to A

A : BDD5A0E6 1467BF21 6C9420B9 15A45522 208A2C56 A4A40DD2 108C2F8E 76559B2C
    5D8A46B7 4AEAAC67 60EFEFEF 9F153F95

B : 30E2F355 EA9BCDD5 99AD1970 869D9BE5 2FACA5C5 80725CD7 E15F9E62 00291B2D
    38D9E8C5 B26B28CB C13E1485 E59389EE 23513E41 FA92400F 2F5DF920 E71E64C5

C : F764B11A 76172146 CEF6934D C6D28399 FE095F61 5E6018B4 5048ECF5 51353261
    6E6E36DC 63130DAD A9C69BD6 1E90EA0C 7C35073B 28D95E6D AA340E0D CB3DEE70

W : 00000001 00000000


block number = 1 : subtract M from C

A : BDD5A0E6 1467BF21 6C9420B9 15A45522 208A2C56 A4A40DD2 108C2F8E 76559B2C
    5D8A46B7 4AEAAC67 60EFEFEF 9F153F95

B : 30E2F355 EA9BCDD5 99AD1970 869D9BE5 2FACA5C5 80725CD7 E15F9E62 00291B2D
    38D9E8C5 B26B28CB C13E1485 E59389EE 23513E41 FA92400F 2F5DF920 E71E64C5

C : 93014EB9 0DAFBAE1 628B28E4 5663152C 8995ECF0 E5E8A23F 201B727C 1D020030
    363700A7 20D1E074 63815793 D447A1C5 2DE7BAF0 D6880E1E 53DEB9BA 70E49619
```

```
W : 00000001 00000000


block number = 1 : swap B with C

A : BDD5A0E6 1467BF21 6C9420B9 15A45522 208A2C56 A4A40DD2 108C2F8E 76559B2C
    5D8A46B7 4AEAAC67 60EFEFEF 9F153F95

B : 93014EB9 0DAFBAE1 628B28E4 5663152C 8995ECF0 E5E8A23F 201B727C 1D020030
    363700A7 20D1E074 63815793 D447A1C5 2DE7BAF0 D6880E1E 53DEB9BA 70E49619

C : 30E2F355 EA9BCDD5 99AD1970 869D9BE5 2FACA5C5 80725CD7 E15F9E62 00291B2D
    38D9E8C5 B26B28CB C13E1485 E59389EE 23513E41 FA92400F 2F5DF920 E71E64C5

W : 00000001 00000000


block number = 2 : increment counter W

A : BDD5A0E6 1467BF21 6C9420B9 15A45522 208A2C56 A4A40DD2 108C2F8E 76559B2C
    5D8A46B7 4AEAAC67 60EFEFEF 9F153F95

B : 93014EB9 0DAFBAE1 628B28E4 5663152C 8995ECF0 E5E8A23F 201B727C 1D020030
    363700A7 20D1E074 63815793 D447A1C5 2DE7BAF0 D6880E1E 53DEB9BA 70E49619

C : 30E2F355 EA9BCDD5 99AD1970 869D9BE5 2FACA5C5 80725CD7 E15F9E62 00291B2D
    38D9E8C5 B26B28CB C13E1485 E59389EE 23513E41 FA92400F 2F5DF920 E71E64C5

W : 00000002 00000000


block number = 2 : message block

M : 3231302D 36353433 2D393837 64636261 68676665 6C6B6A69 706F6E6D 74737271
    78777675 00807A79 00000000 00000000 00000000 00000000 00000000 00000000


block number = 2 : add M to B

A : BDD5A0E6 1467BF21 6C9420B9 15A45522 208A2C56 A4A40DD2 108C2F8E 76559B2C
    5D8A46B7 4AEAAC67 60EFEFEF 9F153F95

B : C5327EE6 43E4EF14 8FC4611B BAC6778D F1FD5355 52540CA8 908AE0E9 917572A1
    AEAE771C 21525AED 63815793 D447A1C5 2DE7BAF0 D6880E1E 53DEB9BA 70E49619

C : 30E2F355 EA9BCDD5 99AD1970 869D9BE5 2FACA5C5 80725CD7 E15F9E62 00291B2D
    38D9E8C5 B26B28CB C13E1485 E59389EE 23513E41 FA92400F 2F5DF920 E71E64C5

W : 00000002 00000000


block number = 2 : xor counter W into A

A : BDD5A0E4 1467BF21 6C9420B9 15A45522 208A2C56 A4A40DD2 108C2F8E 76559B2C
    5D8A46B7 4AEAAC67 60EFEFEF 9F153F95

B : C5327EE6 43E4EF14 8FC4611B BAC6778D F1FD5355 52540CA8 908AE0E9 917572A1
    AEAE771C 21525AED 63815793 D447A1C5 2DE7BAF0 D6880E1E 53DEB9BA 70E49619

C : 30E2F355 EA9BCDD5 99AD1970 869D9BE5 2FACA5C5 80725CD7 E15F9E62 00291B2D
    38D9E8C5 B26B28CB C13E1485 E59389EE 23513E41 FA92400F 2F5DF920 E71E64C5

W : 00000002 00000000


block number = 2 : permutation - rotate B
```

```
A : BDD5A0E4 1467BF21 6C9420B9 15A45522 208A2C56 A4A40DD2 108C2F8E 76559B2C
    5D8A46B7 4AEAAC67 60EFEFEF 9F153F95

B : FDCD8A64 DE2887C9 C2371F88 EF1B758C A6ABE3FA 1950A4A8 C1D32115 E54322EA
    EE395D5C B5DA42A4 AF26C702 438BA88F 75E05BCF 1C3DAD10 7374A7BD 2C32E1C9

C : 30E2F355 EA9BCDD5 99AD1970 869D9BE5 2FACA5C5 80725CD7 E15F9E62 00291B2D
    38D9E8C5 B26B28CB C13E1485 E59389EE 23513E41 FA92400F 2F5DF920 E71E64C5

W : 00000002 00000000


block number = 2 : permutation (j = 0)

A : C1621A94 41DAD100 646F34DE 55E98D21 962B3AB6 B15F2393 B1EE66F1 49C50EB1
    D1075BD1 C7DA6DCB 55B384F0 2B94E072

B : CD8EE412 980BB0AD 7F95EAD0 3CE38023 248302BC 7C01953C CDB7DB25 7CBCB49B
    F28A1E97 5391177D F401F50A 537C4E93 D55D52F5 865E74DF 7D79845B F273B14C

C : 30E2F355 EA9BCDD5 99AD1970 869D9BE5 2FACA5C5 80725CD7 E15F9E62 00291B2D
    38D9E8C5 B26B28CB C13E1485 E59389EE 23513E41 FA92400F 2F5DF920 E71E64C5

W : 00000002 00000000


block number = 2 : permutation (j = 1)

A : 96BCB891 93A5C50B 1ECEEEDC 23677846 8295EE09 2DD7F4FD 9F40DA5D 12625134
    74516C2E 41B17558 B8B8517C BE50CC99

B : 28E9F955 9B957C39 6C40E1CB D1333331 C2A896A9 464DA0DF DC2818C8 B8D65A50
    8C577A41 CB78140E 0932FB36 7A601A9F D7D0B41D DE94E2BD 9A4C2D14 097ACC52

C : 30E2F355 EA9BCDD5 99AD1970 869D9BE5 2FACA5C5 80725CD7 E15F9E62 00291B2D
    38D9E8C5 B26B28CB C13E1485 E59389EE 23513E41 FA92400F 2F5DF920 E71E64C5

W : 00000002 00000000


block number = 2 : permutation (j = 2)

A : C98DF9C8 1B9A2E9B 3A634908 382A3427 E4878759 64B26D5A 5A8F2219 BBBAD729
    E62E6EB4 846E49F4 A0F9A526 8ACE8904

B : D7462099 FE397A79 58159A62 9986F720 B3232B64 68FE90DA 7DCC8766 B6797F79
    03D68C25 0DBDBAB8 B7152B8A B0851DE8 B670F970 C6B87370 6B9E00F0 67C4EE5F

C : 30E2F355 EA9BCDD5 99AD1970 869D9BE5 2FACA5C5 80725CD7 E15F9E62 00291B2D
    38D9E8C5 B26B28CB C13E1485 E59389EE 23513E41 FA92400F 2F5DF920 E71E64C5

W : 00000002 00000000


block number = 2 : permutation - add C to A

A : 1CDD8460 9F7B05F6 A003B3C3 E294E519 526CA330 FE1BEF39 78087590 F805A380
    5288AFB4 104616BF CE696AD7 5CCA350B

B : D7462099 FE397A79 58159A62 9986F720 B3232B64 68FE90DA 7DCC8766 B6797F79
    03D68C25 0DBDBAB8 B7152B8A B0851DE8 B670F970 C6B87370 6B9E00F0 67C4EE5F

C : 30E2F355 EA9BCDD5 99AD1970 869D9BE5 2FACA5C5 80725CD7 E15F9E62 00291B2D
    38D9E8C5 B26B28CB C13E1485 E59389EE 23513E41 FA92400F 2F5DF920 E71E64C5

W : 00000002 00000000
```

```
block number = 2 : subtract M from C

A : 1CDD8460 9F7B05F6 A003B3C3 E294E519 526CA330 FE1BEF39 78087590 F805A380
    5288AFB4 104616BF CE696AD7 5CCA350B

B : D7462099 FE397A79 58159A62 9986F720 B3232B64 68FE90DA 7DCC8766 B6797F79
    03D68C25 0DBDBAB8 B7152B8A B0851DE8 B670F970 C6B87370 6B9E00F0 67C4EE5F

C : FEB1C328 B46699A2 6C73E139 223A3984 C7453F60 1406F26E 70F02FF5 8BB5A8BC
    C0627250 B1EAAE52 C13E1485 E59389EE 23513E41 FA92400F 2F5DF920 E71E64C5

W : 00000002 00000000


block number = 2 : swap B with C

A : 1CDD8460 9F7B05F6 A003B3C3 E294E519 526CA330 FE1BEF39 78087590 F805A380
    5288AFB4 104616BF CE696AD7 5CCA350B

B : FEB1C328 B46699A2 6C73E139 223A3984 C7453F60 1406F26E 70F02FF5 8BB5A8BC
    C0627250 B1EAAE52 C13E1485 E59389EE 23513E41 FA92400F 2F5DF920 E71E64C5

C : D7462099 FE397A79 58159A62 9986F720 B3232B64 68FE90DA 7DCC8766 B6797F79
    03D68C25 0DBDBAB8 B7152B8A B0851DE8 B670F970 C6B87370 6B9E00F0 67C4EE5F

W : 00000002 00000000


n0_final = 0 : add M to B

A : 1CDD8460 9F7B05F6 A003B3C3 E294E519 526CA330 FE1BEF39 78087590 F805A380
    5288AFB4 104616BF CE696AD7 5CCA350B

B : 30E2F355 EA9BCDD5 99AD1970 869D9BE5 2FACA5C5 80725CD7 E15F9E62 00291B2D
    38D9E8C5 B26B28CB C13E1485 E59389EE 23513E41 FA92400F 2F5DF920 E71E64C5

C : D7462099 FE397A79 58159A62 9986F720 B3232B64 68FE90DA 7DCC8766 B6797F79
    03D68C25 0DBDBAB8 B7152B8A B0851DE8 B670F970 C6B87370 6B9E00F0 67C4EE5F

W : 00000002 00000000


n0_final = 0 : xor counter W into A

A : 1CDD8462 9F7B05F6 A003B3C3 E294E519 526CA330 FE1BEF39 78087590 F805A380
    5288AFB4 104616BF CE696AD7 5CCA350B

B : 30E2F355 EA9BCDD5 99AD1970 869D9BE5 2FACA5C5 80725CD7 E15F9E62 00291B2D
    38D9E8C5 B26B28CB C13E1485 E59389EE 23513E41 FA92400F 2F5DF920 E71E64C5

C : D7462099 FE397A79 58159A62 9986F720 B3232B64 68FE90DA 7DCC8766 B6797F79
    03D68C25 0DBDBAB8 B7152B8A B0851DE8 B670F970 C6B87370 6B9E00F0 67C4EE5F

W : 00000002 00000000


n0_final = 0 : permutation - rotate B

A : 1CDD8462 9F7B05F6 A003B3C3 E294E519 526CA330 FE1BEF39 78087590 F805A380
    5288AFB4 104616BF CE696AD7 5CCA350B

B : E6AA61C5 9BABD537 32E1335A 37CB0D3B 4B8A5F59 B9AF00E4 3CC5C2BF 365A0052
    D18A71B3 519764D6 290B827C 13DDCB27 7C8246A2 801FF524 F2405EBB C98BCE3C

C : D7462099 FE397A79 58159A62 9986F720 B3232B64 68FE90DA 7DCC8766 B6797F79
```

```
    03D68C25 0DBDBAB8 B7152B8A B0851DE8 B670F970 C6B87370 6B9E00F0 67C4EE5F

W : 00000002 00000000


n0_final = 0 : permutation (j = 0)

A : 37F8FEB6 65C6EBD9 91C48E11 EB2AD4AF 161441CB 9628F755 AF1E2252 E21B3462
    F7D6F016 B6AC98CE 622A2142 DB40F538

B : 13519207 D285B64E 51F78425 80A08DE1 7EFF0086 1A890963 296A58D3 7150CB39
    AB3DEC8E EA7DAE9D CFC2DA45 03049C89 31038C0D 9A06FE6F 8ABBCC99 87C2B729

C : D7462099 FE397A79 58159A62 9986F720 B3232B64 68FE90DA 7DCC8766 B6797F79
    03D68C25 0DBDBAB8 B7152B8A B0851DE8 B670F970 C6B87370 6B9E00F0 67C4EE5F

W : 00000002 00000000


n0_final = 0 : permutation (j = 1)

A : 56A48A27 12C42D25 9731E707 3B293489 DD497502 163AACCF AC4E260B 1CA47C57
    534979CA 434BA096 D56E0E11 17E5A4BE

B : A021A1AA DDE70D42 7A60B561 A2196A17 51488739 89A64DAF 78454048 0ABBCD33
    FF20ACC5 39C08FE1 F74BAC73 C2DFF264 40B192E7 DDC8AFEF 46C640C7 ECDEEDFB

C : D7462099 FE397A79 58159A62 9986F720 B3232B64 68FE90DA 7DCC8766 B6797F79
    03D68C25 0DBDBAB8 B7152B8A B0851DE8 B670F970 C6B87370 6B9E00F0 67C4EE5F

W : 00000002 00000000


n0_final = 0 : permutation (j = 2)

A : 405CB774 99277B8C 828C69C1 230CBE92 3EC5764E 1FE1F7BD 405620E2 AE36D90B
    8E5DE0C1 026056E6 E11FD4A7 E3073C49

B : 63173B96 B402591C 34ED61CF 2D5C619A 1D3246F9 75941F2C 8DF916AE C984DB0B
    3F7BD03A 939F1780 513E87FA D476C23D F0C13AF0 460EF6C6 936CAAD6 C5451841

C : D7462099 FE397A79 58159A62 9986F720 B3232B64 68FE90DA 7DCC8766 B6797F79
    03D68C25 0DBDBAB8 B7152B8A B0851DE8 B670F970 C6B87370 6B9E00F0 67C4EE5F

W : 00000002 00000000


n0_final = 0 : permutation - add C to A

A : F22DBADB DA01C0F9 B07CE884 648CE14A F68ADB46 AE21CFDF B54BE6ED 3B2E265D
    8EE37542 6FCB07DF 1E9493A9 8386F029

B : 63173B96 B402591C 34ED61CF 2D5C619A 1D3246F9 75941F2C 8DF916AE C984DB0B
    3F7BD03A 939F1780 513E87FA D476C23D F0C13AF0 460EF6C6 936CAAD6 C5451841

C : D7462099 FE397A79 58159A62 9986F720 B3232B64 68FE90DA 7DCC8766 B6797F79
    03D68C25 0DBDBAB8 B7152B8A B0851DE8 B670F970 C6B87370 6B9E00F0 67C4EE5F

W : 00000002 00000000


n0_final = 0 : subtract M from C

A : F22DBADB DA01C0F9 B07CE884 648CE14A F68ADB46 AE21CFDF B54BE6ED 3B2E265D
    8EE37542 6FCB07DF 1E9493A9 8386F029

B : 63173B96 B402591C 34ED61CF 2D5C619A 1D3246F9 75941F2C 8DF916AE C984DB0B
```

```
    3F7BD03A 939F1780 513E87FA D476C23D F0C13AF0 460EF6C6 936CAAD6 C5451841

C : A514F06C C8044646 2ADC622B 352394BF 4ABBC4FF FC932671 0D5D18F9 42060D08
    8B5F15B0 0D3D403F B7152B8A B0851DE8 B670F970 C6B87370 6B9E00F0 67C4EE5F

W : 00000002 00000000


n0_final = 0 : swap B with C

A : F22DBADB DA01C0F9 B07CE884 648CE14A F68ADB46 AE21CFDF B54BE6ED 3B2E265D
    8EE37542 6FCB07DF 1E9493A9 8386F029

B : A514F06C C8044646 2ADC622B 352394BF 4ABBC4FF FC932671 0D5D18F9 42060D08
    8B5F15B0 0D3D403F B7152B8A B0851DE8 B670F970 C6B87370 6B9E00F0 67C4EE5F

C : 63173B96 B402591C 34ED61CF 2D5C619A 1D3246F9 75941F2C 8DF916AE C984DB0B
    3F7BD03A 939F1780 513E87FA D476C23D F0C13AF0 460EF6C6 936CAAD6 C5451841

W : 00000002 00000000


n0_final = 1 : add M to B

A : F22DBADB DA01C0F9 B07CE884 648CE14A F68ADB46 AE21CFDF B54BE6ED 3B2E265D
    8EE37542 6FCB07DF 1E9493A9 8386F029

B : D7462099 FE397A79 58159A62 9986F720 B3232B64 68FE90DA 7DCC8766 B6797F79
    03D68C25 0DBDBAB8 B7152B8A B0851DE8 B670F970 C6B87370 6B9E00F0 67C4EE5F

C : 63173B96 B402591C 34ED61CF 2D5C619A 1D3246F9 75941F2C 8DF916AE C984DB0B
    3F7BD03A 939F1780 513E87FA D476C23D F0C13AF0 460EF6C6 936CAAD6 C5451841

W : 00000002 00000000


n0_final = 1 : xor counter W into A

A : F22DBAD9 DA01C0F9 B07CE884 648CE14A F68ADB46 AE21CFDF B54BE6ED 3B2E265D
    8EE37542 6FCB07DF 1E9493A9 8386F029

B : D7462099 FE397A79 58159A62 9986F720 B3232B64 68FE90DA 7DCC8766 B6797F79
    03D68C25 0DBDBAB8 B7152B8A B0851DE8 B670F970 C6B87370 6B9E00F0 67C4EE5F

C : 63173B96 B402591C 34ED61CF 2D5C619A 1D3246F9 75941F2C 8DF916AE C984DB0B
    3F7BD03A 939F1780 513E87FA D476C23D F0C13AF0 460EF6C6 936CAAD6 C5451841

W : 00000002 00000000


n0_final = 1 : permutation - rotate B

A : F22DBAD9 DA01C0F9 B07CE884 648CE14A F68ADB46 AE21CFDF B54BE6ED 3B2E265D
    8EE37542 6FCB07DF 1E9493A9 8386F029

B : 4133AE8C F4F3FC72 34C4B02B EE41330D 56C96646 21B4D1FD 0ECCFB99 FEF36CF2
    184A07AD 75701B7B 57156E2A 3BD1610A F2E16CE1 E6E18D70 01E0D73C DCBECF89

C : 63173B96 B402591C 34ED61CF 2D5C619A 1D3246F9 75941F2C 8DF916AE C984DB0B
    3F7BD03A 939F1780 513E87FA D476C23D F0C13AF0 460EF6C6 936CAAD6 C5451841

W : 00000002 00000000


n0_final = 1 : permutation (j = 0)

A : 7A7FAB0C 81526FA0 B31A1667 0391CAE7 6D7CC15C 10709079 28FB14F2 F83DBBC2
```

```
    3925F48E 1794C5FE 5B3F955E 5FF53CF5

B : 67ECF95C C44D6B6D 5C6B53FD ED127339 3F11F22F ACE6CC7C CA9D1C3F FA249DD8
    F64E042B 028B0CF7 0AEAB6F5 D7A8011E 60428D30 B36E8ABE 4F2447E0 4513AA0B

C : 63173B96 B402591C 34ED61CF 2D5C619A 1D3246F9 75941F2C 8DF916AE C984DB0B
    3F7BD03A 939F1780 513E87FA D476C23D F0C13AF0 460EF6C6 936CAAD6 C5451841

W : 00000002 00000000


n0_final = 1 : permutation (j = 1)

A : D377F9BD 6DD422F4 697DBDEF 9D8B94C4 FBD3908D 3CDD3EAE FF47969E 3EA721F7
    C18EA1D5 0614C29B 331C3A5B 129C7788

B : CAA87BBE DF3053E1 6967B887 DD9604D4 4052BA74 A026A59D 59D9FDDB 192AB3C6
    C0140E15 973DC4E5 83572FFA CD246906 C4A97512 A5FFD42C 9EF0E6A1 4B7F8A1E

C : 63173B96 B402591C 34ED61CF 2D5C619A 1D3246F9 75941F2C 8DF916AE C984DB0B
    3F7BD03A 939F1780 513E87FA D476C23D F0C13AF0 460EF6C6 936CAAD6 C5451841

W : 00000002 00000000


n0_final = 1 : permutation (j = 2)

A : 11334407 E269A64C B805785A 8A5B98C4 4B2CB453 E5112548 54620017 7B0103DB
    2C263CF3 90FEE027 D17F1169 11B5B480

B : 051BA4F7 492F2AE0 A31FF63D 05CE2257 6E69CF10 5DDB1288 F4497C13 47F100B7
    34FB5787 3495537C AD33A01D 1EB62E29 5A8B2929 24FEB781 136123D5 78B55F43

C : 63173B96 B402591C 34ED61CF 2D5C619A 1D3246F9 75941F2C 8DF916AE C984DB0B
    3F7BD03A 939F1780 513E87FA D476C23D F0C13AF0 460EF6C6 936CAAD6 C5451841

W : 00000002 00000000


n0_final = 1 : permutation - add C to A

A : D84B801F 537463CB 27AAE768 E0AEBC17 07530939 A4D67811 11978FDF 8F260452
    F77E3BD5 DE6E324A 20C13EDB 8459FDFE

B : 051BA4F7 492F2AE0 A31FF63D 05CE2257 6E69CF10 5DDB1288 F4497C13 47F100B7
    34FB5787 3495537C AD33A01D 1EB62E29 5A8B2929 24FEB781 136123D5 78B55F43

C : 63173B96 B402591C 34ED61CF 2D5C619A 1D3246F9 75941F2C 8DF916AE C984DB0B
    3F7BD03A 939F1780 513E87FA D476C23D F0C13AF0 460EF6C6 936CAAD6 C5451841

W : 00000002 00000000


n0_final = 1 : subtract M from C

A : D84B801F 537463CB 27AAE768 E0AEBC17 07530939 A4D67811 11978FDF 8F260452
    F77E3BD5 DE6E324A 20C13EDB 8459FDFE

B : 051BA4F7 492F2AE0 A31FF63D 05CE2257 6E69CF10 5DDB1288 F4497C13 47F100B7
    34FB5787 3495537C AD33A01D 1EB62E29 5A8B2929 24FEB781 136123D5 78B55F43

C : 30E60B69 7DCD24E9 07B42998 C8F8FF39 B4CAE094 0928B4C3 1D89A841 5511689A
    C70459C5 931E9D07 513E87FA D476C23D F0C13AF0 460EF6C6 936CAAD6 C5451841

W : 00000002 00000000
```

```
n0_final = 1 : swap B with C

A : D84B801F 537463CB 27AAE768 E0AEBC17 07530939 A4D67811 11978FDF 8F260452
    F77E3BD5 DE6E324A 20C13EDB 8459FDFE

B : 30E60B69 7DCD24E9 07B42998 C8F8FF39 B4CAE094 0928B4C3 1D89A841 5511689A
    C70459C5 931E9D07 513E87FA D476C23D F0C13AF0 460EF6C6 936CAAD6 C5451841

C : 051BA4F7 492F2AE0 A31FF63D 05CE2257 6E69CF10 5DDB1288 F4497C13 47F100B7
    34FB5787 3495537C AD33A01D 1EB62E29 5A8B2929 24FEB781 136123D5 78B55F43

W : 00000002 00000000


n0_final = 2 : add M to B

A : D84B801F 537463CB 27AAE768 E0AEBC17 07530939 A4D67811 11978FDF 8F260452
    F77E3BD5 DE6E324A 20C13EDB 8459FDFE

B : 63173B96 B402591C 34ED61CF 2D5C619A 1D3246F9 75941F2C 8DF916AE C984DB0B
    3F7BD03A 939F1780 513E87FA D476C23D F0C13AF0 460EF6C6 936CAAD6 C5451841

C : 051BA4F7 492F2AE0 A31FF63D 05CE2257 6E69CF10 5DDB1288 F4497C13 47F100B7
    34FB5787 3495537C AD33A01D 1EB62E29 5A8B2929 24FEB781 136123D5 78B55F43

W : 00000002 00000000


n0_final = 2 : xor counter W into A

A : D84B801D 537463CB 27AAE768 E0AEBC17 07530939 A4D67811 11978FDF 8F260452
    F77E3BD5 DE6E324A 20C13EDB 8459FDFE

B : 63173B96 B402591C 34ED61CF 2D5C619A 1D3246F9 75941F2C 8DF916AE C984DB0B
    3F7BD03A 939F1780 513E87FA D476C23D F0C13AF0 460EF6C6 936CAAD6 C5451841

C : 051BA4F7 492F2AE0 A31FF63D 05CE2257 6E69CF10 5DDB1288 F4497C13 47F100B7
    34FB5787 3495537C AD33A01D 1EB62E29 5A8B2929 24FEB781 136123D5 78B55F43

W : 00000002 00000000


n0_final = 2 : permutation - rotate B

A : D84B801D 537463CB 27AAE768 E0AEBC17 07530939 A4D67811 11978FDF 8F260452
    F77E3BD5 DE6E324A 20C13EDB 8459FDFE

B : 772CC62E B2396804 C39E69DA C3345AB8 8DF23A64 3E58EB28 2D5D1BF2 B6179309
    A0747EF7 2F01273E 0FF4A27D 847BA8ED 75E1E182 ED8C8C1D 55AD26D9 30838A8A

C : 051BA4F7 492F2AE0 A31FF63D 05CE2257 6E69CF10 5DDB1288 F4497C13 47F100B7
    34FB5787 3495537C AD33A01D 1EB62E29 5A8B2929 24FEB781 136123D5 78B55F43

W : 00000002 00000000


n0_final = 2 : permutation (j = 0)

A : 96B22A2E 7E596DA0 39851350 5B993B8B D05FD685 6FD3086E 159504C0 ABFE4E42
    89A3173F 2839BDE6 1271F888 EB2A3DA9

B : 8EFE3C4D 8800DA08 FE381C10 132D2BE8 34445DB3 EC9D21C1 B0D0CCDB 382E97AE
    36B4152F 89C40C65 F267438D 1C22938D 828E16D5 5ABF8A64 6D20A11D C561D160

C : 051BA4F7 492F2AE0 A31FF63D 05CE2257 6E69CF10 5DDB1288 F4497C13 47F100B7
    34FB5787 3495537C AD33A01D 1EB62E29 5A8B2929 24FEB781 136123D5 78B55F43
```

```
W : 00000002 00000000


n0_final = 2 : permutation (j = 1)

A : 2E5F9370 47C6AA9B 52F67D34 7E6B4C97 0404918E 1E580D65 3A596788 5E0BADA7
    CEFB86A2 F6147458 385A5BB3 F870E621

B : 52E38CF3 E821067A 035EEE61 B645C86B 598CC23B D0D1C824 A6043DFB 77D23682
    BCC846D1 ABB14DAF 49C705D0 B9D19472 FEE743DA 54D8E652 1FE7DA4D 2B37F099

C : 051BA4F7 492F2AE0 A31FF63D 05CE2257 6E69CF10 5DDB1288 F4497C13 47F100B7
    34FB5787 3495537C AD33A01D 1EB62E29 5A8B2929 24FEB781 136123D5 78B55F43

W : 00000002 00000000


n0_final = 2 : permutation (j = 2)

A : 42CA7AB9 551AB388 71D002DF C21F9B38 1C09FBFD 3FE7495A 1B5E1B4C 28289356
    CCFA8970 72FAC99E 304343E9 097BE5F5

B : 6D085ED6 71C06B61 6D009973 9317C3EB 0E2C0130 0B46DC3E C22786D7 D24409C3
    9A6689A1 977A2DFA 772FEF13 A474444C CECBF13A 24B4FAC5 F073088C A0EBFB38

C : 051BA4F7 492F2AE0 A31FF63D 05CE2257 6E69CF10 5DDB1288 F4497C13 47F100B7
    34FB5787 3495537C AD33A01D 1EB62E29 5A8B2929 24FEB781 136123D5 78B55F43

W : 00000002 00000000


n0_final = 2 : permutation - add C to A

A : E0042A7C 232B50B8 3DD8F7C8 6CEA315D E27E7E4E E86814E8 F6FDAC30 6CC5A5C3
    396FDAA7 70EB195E E7B2616E BE5A25FA

B : 6D085ED6 71C06B61 6D009973 9317C3EB 0E2C0130 0B46DC3E C22786D7 D24409C3
    9A6689A1 977A2DFA 772FEF13 A474444C CECBF13A 24B4FAC5 F073088C A0EBFB38

C : 051BA4F7 492F2AE0 A31FF63D 05CE2257 6E69CF10 5DDB1288 F4497C13 47F100B7
    34FB5787 3495537C AD33A01D 1EB62E29 5A8B2929 24FEB781 136123D5 78B55F43

W : 00000002 00000000


n0_final = 2 : subtract M from C

A : E0042A7C 232B50B8 3DD8F7C8 6CEA315D E27E7E4E E86814E8 F6FDAC30 6CC5A5C3
    396FDAA7 70EB195E E7B2616E BE5A25FA

B : 6D085ED6 71C06B61 6D009973 9317C3EB 0E2C0130 0B46DC3E C22786D7 D24409C3
    9A6689A1 977A2DFA 772FEF13 A474444C CECBF13A 24B4FAC5 F073088C A0EBFB38

C : D2EA74CA 12F9F6AD 75E6BE06 A16ABFF6 060268AB F16FA81F 83DA0DA6 D37D8E46
    BC83E112 3414D903 AD33A01D 1EB62E29 5A8B2929 24FEB781 136123D5 78B55F43

W : 00000002 00000000


n0_final = 2 : swap B with C  (final state)

A : E0042A7C 232B50B8 3DD8F7C8 6CEA315D E27E7E4E E86814E8 F6FDAC30 6CC5A5C3
    396FDAA7 70EB195E E7B2616E BE5A25FA

B : D2EA74CA 12F9F6AD 75E6BE06 A16ABFF6 060268AB F16FA81F 83DA0DA6 D37D8E46
    BC83E112 3414D903 AD33A01D 1EB62E29 5A8B2929 24FEB781 136123D5 78B55F43
```

```
C : 6D085ED6 71C06B61 6D009973 9317C3EB 0E2C0130 0B46DC3E C22786D7 D24409C3
    9A6689A1 977A2DFA 772FEF13 A474444C CECBF13A 24B4FAC5 F073088C A0EBFB38

W : 00000002 00000000


Hash value :

H : 0E2C0130 0B46DC3E C22786D7 D24409C3 9A6689A1 977A2DFA 772FEF13 A474444C
    CECBF13A 24B4FAC5 F073088C A0EBFB38


Hash value (byte array):

H : 30 01 2C 0E 3E DC 46 0B D7 86 27 C2 C3 09 44 D2
    A1 89 66 9A FA 2D 7A 97 13 EF 2F 77 4C 44 74 A4
    3A F1 CB CE C5 FA B4 24 8C 08 73 F0 38 FB EB A0
```

B.9 Intermediate States for Shabal-512 (Message A)

```
init

A : 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
    00000000 00000000 00000000 00000000

B : 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
    00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000

C : 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
    00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000

W : FFFFFFFF FFFFFFFF


block number = -1 : message block

M : 00000200 00000201 00000202 00000203 00000204 00000205 00000206 00000207
    00000208 00000209 0000020A 0000020B 0000020C 0000020D 0000020E 0000020F


block number = -1 : add M to B

A : 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
    00000000 00000000 00000000 00000000

B : 00000200 00000201 00000202 00000203 00000204 00000205 00000206 00000207
    00000208 00000209 0000020A 0000020B 0000020C 0000020D 0000020E 0000020F

C : 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
    00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000

W : FFFFFFFF FFFFFFFF


block number = -1 : xor counter W into A

A : FFFFFFFF FFFFFFFF 00000000 00000000 00000000 00000000 00000000 00000000
    00000000 00000000 00000000 00000000

B : 00000200 00000201 00000202 00000203 00000204 00000205 00000206 00000207
    00000208 00000209 0000020A 0000020B 0000020C 0000020D 0000020E 0000020F

C : 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
    00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
```

```
W : FFFFFFFF FFFFFFFF


block number = -1 : permutation - rotate B

A : FFFFFFFF FFFFFFFF 00000000 00000000 00000000 00000000 00000000 00000000
    00000000 00000000 00000000 00000000

B : 04000000 04020000 04040000 04060000 04080000 040A0000 040C0000 040E0000
    04100000 04120000 04140000 04160000 04180000 041A0000 041C0000 041E0000

C : 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
    00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000

W : FFFFFFFF FFFFFFFF


block number = -1 : permutation (j = 0)

A : E2AA101E 3A5C2CEA E01C099D 8D1BE979 1C49D94D 7A06796B E9703766 7C25F07F
    FEB14424 C24293CA BC669928 76D9392D

B : 0C080202 FCF8E3B6 A7CC2F72 67D0264D EBA626B2 8DED8694 1E97C899 8BC60F80
    096EBBDB 35996C35 4BB166D7 810AC6D2 1565EFE1 CD97D315 17DBF662 7AD81686

C : 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
    00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000

W : FFFFFFFF FFFFFFFF


block number = -1 : permutation (j = 1)

A : E524968F 5F83748F 98E7810D 23285DCF 2195FE4E 7188ACDA 7AF7FC83 31F5BA1C
    CC8D5743 4E46BE5D DAE5E19B BCEB09F1

B : 98C2A26E 203FE129 91D449C4 86D27C0E E43EE5D9 AA624C8B 18358F56 5498E90F
    08061EC6 CB4E531A F07AB35C DEC22F95 F4A1DE73 1558F50E AABFEFB8 3BBA68EF

C : 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
    00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000

W : FFFFFFFF FFFFFFFF


block number = -1 : permutation (j = 2)

A : DCED6C97 B5937A56 BC2D3479 4B2672FE A762CE01 E0587D3E DAAA7818 F9BF94BD
    646EB744 A0DA0A55 C70BC02E 292C7C6B

B : C1099EC7 07B38804 E7442E38 CC8AD853 EB6F58DB 1EA81CBE 73B9D52A 1DE85F1F
    48910C72 893B24F4 C5A0E15E BBC43469 72D2F45C 75941FB6 6D8BE0A0 A1A7524A

C : 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
    00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000

W : FFFFFFFF FFFFFFFF


block number = -1 : permutation - add C to A

A : DCED6C97 B5937A56 BC2D3479 4B2672FE A762CE01 E0587D3E DAAA7818 F9BF94BD
    646EB744 A0DA0A55 C70BC02E 292C7C6B

B : C1099EC7 07B38804 E7442E38 CC8AD853 EB6F58DB 1EA81CBE 73B9D52A 1DE85F1F
    48910C72 893B24F4 C5A0E15E BBC43469 72D2F45C 75941FB6 6D8BE0A0 A1A7524A
```

```
C : 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
    00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000

W : FFFFFFFF FFFFFFFF


block number = -1 : subtract M from C

A : DCED6C97 B5937A56 BC2D3479 4B2672FE A762CE01 E0587D3E DAAA7818 F9BF94BD
    646EB744 A0DA0A55 C70BC02E 292C7C6B

B : C1099EC7 07B38804 E7442E38 CC8AD853 EB6F58DB 1EA81CBE 73B9D52A 1DE85F1F
    48910C72 893B24F4 C5A0E15E BBC43469 72D2F45C 75941FB6 6D8BE0A0 A1A7524A

C : FFFFFE00 FFFFFDFF FFFFFDFE FFFFFDFD FFFFFDFC FFFFFDFB FFFFFDFA FFFFFDF9
    FFFFFDF8 FFFFFDF7 FFFFFDF6 FFFFFDF5 FFFFFDF4 FFFFFDF3 FFFFFDF2 FFFFFDF1

W : FFFFFFFF FFFFFFFF


block number = -1 : swap B with C

A : DCED6C97 B5937A56 BC2D3479 4B2672FE A762CE01 E0587D3E DAAA7818 F9BF94BD
    646EB744 A0DA0A55 C70BC02E 292C7C6B

B : FFFFFE00 FFFFFDFF FFFFFDFE FFFFFDFD FFFFFDFC FFFFFDFB FFFFFDFA FFFFFDF9
    FFFFFDF8 FFFFFDF7 FFFFFDF6 FFFFFDF5 FFFFFDF4 FFFFFDF3 FFFFFDF2 FFFFFDF1

C : C1099EC7 07B38804 E7442E38 CC8AD853 EB6F58DB 1EA81CBE 73B9D52A 1DE85F1F
    48910C72 893B24F4 C5A0E15E BBC43469 72D2F45C 75941FB6 6D8BE0A0 A1A7524A

W : FFFFFFFF FFFFFFFF


block number = 0 : increment counter W

A : DCED6C97 B5937A56 BC2D3479 4B2672FE A762CE01 E0587D3E DAAA7818 F9BF94BD
    646EB744 A0DA0A55 C70BC02E 292C7C6B

B : FFFFFE00 FFFFFDFF FFFFFDFE FFFFFDFD FFFFFDFC FFFFFDFB FFFFFDFA FFFFFDF9
    FFFFFDF8 FFFFFDF7 FFFFFDF6 FFFFFDF5 FFFFFDF4 FFFFFDF3 FFFFFDF2 FFFFFDF1

C : C1099EC7 07B38804 E7442E38 CC8AD853 EB6F58DB 1EA81CBE 73B9D52A 1DE85F1F
    48910C72 893B24F4 C5A0E15E BBC43469 72D2F45C 75941FB6 6D8BE0A0 A1A7524A

W : 00000000 00000000


block number = 0 : message block

M : 00000210 00000211 00000212 00000213 00000214 00000215 00000216 00000217
    00000218 00000219 0000021A 0000021B 0000021C 0000021D 0000021E 0000021F


block number = 0 : add M to B

A : DCED6C97 B5937A56 BC2D3479 4B2672FE A762CE01 E0587D3E DAAA7818 F9BF94BD
    646EB744 A0DA0A55 C70BC02E 292C7C6B

B : 00000010 00000010 00000010 00000010 00000010 00000010 00000010 00000010
    00000010 00000010 00000010 00000010 00000010 00000010 00000010 00000010

C : C1099EC7 07B38804 E7442E38 CC8AD853 EB6F58DB 1EA81CBE 73B9D52A 1DE85F1F
    48910C72 893B24F4 C5A0E15E BBC43469 72D2F45C 75941FB6 6D8BE0A0 A1A7524A

W : 00000000 00000000
```

```
block number = 0 : xor counter W into A

A : DCED6C97 B5937A56 BC2D3479 4B2672FE A762CE01 E0587D3E DAAA7818 F9BF94BD
    646EB744 A0DA0A55 C70BC02E 292C7C6B

B : 00000010 00000010 00000010 00000010 00000010 00000010 00000010 00000010
    00000010 00000010 00000010 00000010 00000010 00000010 00000010 00000010

C : C1099EC7 07B38804 E7442E38 CC8AD853 EB6F58DB 1EA81CBE 73B9D52A 1DE85F1F
    48910C72 893B24F4 C5A0E15E BBC43469 72D2F45C 75941FB6 6D8BE0A0 A1A7524A

W : 00000000 00000000


block number = 0 : permutation - rotate B

A : DCED6C97 B5937A56 BC2D3479 4B2672FE A762CE01 E0587D3E DAAA7818 F9BF94BD
    646EB744 A0DA0A55 C70BC02E 292C7C6B

B : 00200000 00200000 00200000 00200000 00200000 00200000 00200000 00200000
    00200000 00200000 00200000 00200000 00200000 00200000 00200000 00200000

C : C1099EC7 07B38804 E7442E38 CC8AD853 EB6F58DB 1EA81CBE 73B9D52A 1DE85F1F
    48910C72 893B24F4 C5A0E15E BBC43469 72D2F45C 75941FB6 6D8BE0A0 A1A7524A

W : 00000000 00000000


block number = 0 : permutation (j = 0)

A : E3C2DE4A 15013A27 9779A969 2E533DBB F9542F32 1830E439 49354FD1 3FA73E25
    62A98156 E3930697 3AEC9AE8 02836D8A

B : 15F96FCE BF883B5C 13B9B891 5F17F2CF 06EBD0CD E78F1BC6 B68AB02E C018C1DA
    9D167EA9 1C2CF968 C5536517 FD3C9275 1C7D21B5 EABEC5D8 68C65696 D1ECC244

C : C1099EC7 07B38804 E7442E38 CC8AD853 EB6F58DB 1EA81CBE 73B9D52A 1DE85F1F
    48910C72 893B24F4 C5A0E15E BBC43469 72D2F45C 75941FB6 6D8BE0A0 A1A7524A

W : 00000000 00000000


block number = 0 : permutation (j = 1)

A : F762247A 9C5C1A1B 82879280 482A0C43 F996F842 F06FAEB4 6FBAE548 DD572216
    AC03820D 16A91946 E8198C7A C460216D

B : 3453DF4C 85E7DAA3 A8D88D92 0E0FE970 5E2BDC68 2648D134 7AF313D8 BBAE5D27
    32B126D6 5BFA1734 F7DEA750 4DACD757 3E9344D7 DAEDDAFA 41C9B79B 81715960

C : C1099EC7 07B38804 E7442E38 CC8AD853 EB6F58DB 1EA81CBE 73B9D52A 1DE85F1F
    48910C72 893B24F4 C5A0E15E BBC43469 72D2F45C 75941FB6 6D8BE0A0 A1A7524A

W : 00000000 00000000


block number = 0 : permutation (j = 2)

A : F67C2EF7 2774D155 4B92F221 8CA66230 E59A65D4 199FE418 2C22673B DA3186CB
    17D54265 E4FDEA67 595B4C44 81C61457

B : D9BF68D1 58BAD750 56028CB2 8134F359 B5D469D8 941A8CC2 418B2A6E 04052780
    7F07D787 5194358F 3C60D665 BE97D79A 950C3434 AED9A06D 2537DC8D 7CDB5969

C : C1099EC7 07B38804 E7442E38 CC8AD853 EB6F58DB 1EA81CBE 73B9D52A 1DE85F1F
```

```
    48910C72 893B24F4 C5A0E15E BBC43469 72D2F45C 75941FB6 6D8BE0A0 A1A7524A

W : 00000000 00000000


block number = 0 : permutation - add C to A

A : 20728DFD 46C0BD53 E782B699 55304632 71B4EF90 0EA9E82C DBB930F1 FAD06B8B
    BE0CAE40 8BD14410 76D2ADAC 28ACAB7F

B : D9BF68D1 58BAD750 56028CB2 8134F359 B5D469D8 941A8CC2 418B2A6E 04052780
    7F07D787 5194358F 3C60D665 BE97D79A 950C3434 AED9A06D 2537DC8D 7CDB5969

C : C1099EC7 07B38804 E7442E38 CC8AD853 EB6F58DB 1EA81CBE 73B9D52A 1DE85F1F
    48910C72 893B24F4 C5A0E15E BBC43469 72D2F45C 75941FB6 6D8BE0A0 A1A7524A

W : 00000000 00000000


block number = 0 : subtract M from C

A : 20728DFD 46C0BD53 E782B699 55304632 71B4EF90 0EA9E82C DBB930F1 FAD06B8B
    BE0CAE40 8BD14410 76D2ADAC 28ACAB7F

B : D9BF68D1 58BAD750 56028CB2 8134F359 B5D469D8 941A8CC2 418B2A6E 04052780
    7F07D787 5194358F 3C60D665 BE97D79A 950C3434 AED9A06D 2537DC8D 7CDB5969

C : C1099CB7 07B385F3 E7442C26 CC8AD640 EB6F56C7 1EA81AA9 73B9D314 1DE85D08
    48910A5A 893B22DB C5A0DF44 BBC4324E 72D2F240 75941D99 6D8BDE82 A1A7502B

W : 00000000 00000000


block number = 0 : swap B with C

A : 20728DFD 46C0BD53 E782B699 55304632 71B4EF90 0EA9E82C DBB930F1 FAD06B8B
    BE0CAE40 8BD14410 76D2ADAC 28ACAB7F

B : C1099CB7 07B385F3 E7442C26 CC8AD640 EB6F56C7 1EA81AA9 73B9D314 1DE85D08
    48910A5A 893B22DB C5A0DF44 BBC4324E 72D2F240 75941D99 6D8BDE82 A1A7502B

C : D9BF68D1 58BAD750 56028CB2 8134F359 B5D469D8 941A8CC2 418B2A6E 04052780
    7F07D787 5194358F 3C60D665 BE97D79A 950C3434 AED9A06D 2537DC8D 7CDB5969

W : 00000000 00000000


block number = 1 : increment counter W

A : 20728DFD 46C0BD53 E782B699 55304632 71B4EF90 0EA9E82C DBB930F1 FAD06B8B
    BE0CAE40 8BD14410 76D2ADAC 28ACAB7F

B : C1099CB7 07B385F3 E7442C26 CC8AD640 EB6F56C7 1EA81AA9 73B9D314 1DE85D08
    48910A5A 893B22DB C5A0DF44 BBC4324E 72D2F240 75941D99 6D8BDE82 A1A7502B

C : D9BF68D1 58BAD750 56028CB2 8134F359 B5D469D8 941A8CC2 418B2A6E 04052780
    7F07D787 5194358F 3C60D665 BE97D79A 950C3434 AED9A06D 2537DC8D 7CDB5969

W : 00000001 00000000


block number = 1 : message block

M : 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
    00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
```

```
block number = 1 : add M to B

A : 20728DFD 46C0BD53 E782B699 55304632 71B4EF90 0EA9E82C DBB930F1 FAD06B8B
    BE0CAE40 8BD14410 76D2ADAC 28ACAB7F

B : C1099CB7 07B385F3 E7442C26 CC8AD640 EB6F56C7 1EA81AA9 73B9D314 1DE85D08
    48910A5A 893B22DB C5A0DF44 BBC4324E 72D2F240 75941D99 6D8BDE82 A1A7502B

C : D9BF68D1 58BAD750 56028CB2 8134F359 B5D469D8 941A8CC2 418B2A6E 04052780
    7F07D787 5194358F 3C60D665 BE97D79A 950C3434 AED9A06D 2537DC8D 7CDB5969

W : 00000001 00000000


block number = 1 : xor counter W into A

A : 20728DFC 46C0BD53 E782B699 55304632 71B4EF90 0EA9E82C DBB930F1 FAD06B8B
    BE0CAE40 8BD14410 76D2ADAC 28ACAB7F

B : C1099CB7 07B385F3 E7442C26 CC8AD640 EB6F56C7 1EA81AA9 73B9D314 1DE85D08
    48910A5A 893B22DB C5A0DF44 BBC4324E 72D2F240 75941D99 6D8BDE82 A1A7502B

C : D9BF68D1 58BAD750 56028CB2 8134F359 B5D469D8 941A8CC2 418B2A6E 04052780
    7F07D787 5194358F 3C60D665 BE97D79A 950C3434 AED9A06D 2537DC8D 7CDB5969

W : 00000001 00000000


block number = 1 : permutation - rotate B

A : 20728DFC 46C0BD53 E782B699 55304632 71B4EF90 0EA9E82C DBB930F1 FAD06B8B
    BE0CAE40 8BD14410 76D2ADAC 28ACAB7F

B : 396F8213 0BE60F67 584DCE88 AC819915 AD8FD6DE 35523D50 A628E773 BA103BD0
    14B49122 45B71276 BE898B41 649D7788 E480E5A5 3B32EB28 BD04DB17 A057434E

C : D9BF68D1 58BAD750 56028CB2 8134F359 B5D469D8 941A8CC2 418B2A6E 04052780
    7F07D787 5194358F 3C60D665 BE97D79A 950C3434 AED9A06D 2537DC8D 7CDB5969

W : 00000001 00000000


block number = 1 : permutation (j = 0)

A : E09EDBEE C276F931 B0D8D4CE C640E1B5 58895064 B8D5F730 14A7145B C93F3B68
    74AFE6B9 57F8351D D7447331 BACD8A8A

B : 2CDF3F8A 2983105F 77E22D44 383740E2 FC690226 2D8E726F A7092543 42E0B336
    A2393B02 2369EE0E 55A89A4D 8C089A65 D660EF5A 4BECD09E 352E9D1E 791198D7

C : D9BF68D1 58BAD750 56028CB2 8134F359 B5D469D8 941A8CC2 418B2A6E 04052780
    7F07D787 5194358F 3C60D665 BE97D79A 950C3434 AED9A06D 2537DC8D 7CDB5969

W : 00000001 00000000


block number = 1 : permutation (j = 1)

A : B44A2D19 F0309531 7F1C7793 C8EE5AA1 58680662 EB3F6CD2 7CF1E331 DE187243
    C37AE049 287F262C A892AFC2 AF71D3FE

B : D69722B0 F74395E3 AA072FE9 9BFA6F6F C4571BFB 8C9C3D0D 197F1ABA D54F4A6D
    0FC7A4E3 491CB6D2 2BB2BCF6 2F009195 0B562728 83193211 E95326F2 D3C4BC12

C : D9BF68D1 58BAD750 56028CB2 8134F359 B5D469D8 941A8CC2 418B2A6E 04052780
    7F07D787 5194358F 3C60D665 BE97D79A 950C3434 AED9A06D 2537DC8D 7CDB5969
```

```
W : 00000001 00000000


block number = 1 : permutation (j = 2)

A : 648A9B4A BC9B82CF BA639897 20F00F46 B9BC68E2 E22AA0F7 9693C179 6145A212
    2530A066 B7509372 69C4375A D41A85F9

B : 74B5FA19 7518DD5A A370733E A0AE0565 13DB5342 5A5C072B 7762521C 75916462
    59CCDEDB 8FEC32AC 3E09476A C0BB7EC7 CC6311C9 4E9D08AE 449D8540 8C6C0223

C : D9BF68D1 58BAD750 56028CB2 8134F359 B5D469D8 941A8CC2 418B2A6E 04052780
    7F07D787 5194358F 3C60D665 BE97D79A 950C3434 AED9A06D 2537DC8D 7CDB5969

W : 00000001 00000000


block number = 1 : permutation - add C to A

A : 2132BFA6 E13B89AC 56129D16 DDB5A2F3 BBD1DD24 F0C64B27 D4FD5B1A 35342F97
    690292D9 81390905 FE4C9A18 773E6359

B : 74B5FA19 7518DD5A A370733E A0AE0565 13DB5342 5A5C072B 7762521C 75916462
    59CCDEDB 8FEC32AC 3E09476A C0BB7EC7 CC6311C9 4E9D08AE 449D8540 8C6C0223

C : D9BF68D1 58BAD750 56028CB2 8134F359 B5D469D8 941A8CC2 418B2A6E 04052780
    7F07D787 5194358F 3C60D665 BE97D79A 950C3434 AED9A06D 2537DC8D 7CDB5969

W : 00000001 00000000


block number = 1 : subtract M from C

A : 2132BFA6 E13B89AC 56129D16 DDB5A2F3 BBD1DD24 F0C64B27 D4FD5B1A 35342F97
    690292D9 81390905 FE4C9A18 773E6359

B : 74B5FA19 7518DD5A A370733E A0AE0565 13DB5342 5A5C072B 7762521C 75916462
    59CCDEDB 8FEC32AC 3E09476A C0BB7EC7 CC6311C9 4E9D08AE 449D8540 8C6C0223

C : D9BF68D1 58BAD750 56028CB2 8134F359 B5D469D8 941A8CC2 418B2A6E 04052780
    7F07D787 5194358F 3C60D665 BE97D79A 950C3434 AED9A06D 2537DC8D 7CDB5969

W : 00000001 00000000


block number = 1 : swap B with C

A : 2132BFA6 E13B89AC 56129D16 DDB5A2F3 BBD1DD24 F0C64B27 D4FD5B1A 35342F97
    690292D9 81390905 FE4C9A18 773E6359

B : D9BF68D1 58BAD750 56028CB2 8134F359 B5D469D8 941A8CC2 418B2A6E 04052780
    7F07D787 5194358F 3C60D665 BE97D79A 950C3434 AED9A06D 2537DC8D 7CDB5969

C : 74B5FA19 7518DD5A A370733E A0AE0565 13DB5342 5A5C072B 7762521C 75916462
    59CCDEDB 8FEC32AC 3E09476A C0BB7EC7 CC6311C9 4E9D08AE 449D8540 8C6C0223

W : 00000001 00000000


block number = 2 : increment counter W

A : 2132BFA6 E13B89AC 56129D16 DDB5A2F3 BBD1DD24 F0C64B27 D4FD5B1A 35342F97
    690292D9 81390905 FE4C9A18 773E6359

B : D9BF68D1 58BAD750 56028CB2 8134F359 B5D469D8 941A8CC2 418B2A6E 04052780
    7F07D787 5194358F 3C60D665 BE97D79A 950C3434 AED9A06D 2537DC8D 7CDB5969
```

```
C : 74B5FA19 7518DD5A A370733E A0AE0565 13DB5342 5A5C072B 7762521C 75916462
    59CCDEDB 8FEC32AC 3E09476A C0BB7EC7 CC6311C9 4E9D08AE 449D8540 8C6C0223

W : 00000002 00000000


block number = 2 : message block

M : 00000080 00000000 00000000 00000000 00000000 00000000 00000000 00000000
    00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000


block number = 2 : add M to B

A : 2132BFA6 E13B89AC 56129D16 DDB5A2F3 BBD1DD24 F0C64B27 D4FD5B1A 35342F97
    690292D9 81390905 FE4C9A18 773E6359

B : D9BF6951 58BAD750 56028CB2 8134F359 B5D469D8 941A8CC2 418B2A6E 04052780
    7F07D787 5194358F 3C60D665 BE97D79A 950C3434 AED9A06D 2537DC8D 7CDB5969

C : 74B5FA19 7518DD5A A370733E A0AE0565 13DB5342 5A5C072B 7762521C 75916462
    59CCDEDB 8FEC32AC 3E09476A C0BB7EC7 CC6311C9 4E9D08AE 449D8540 8C6C0223

W : 00000002 00000000


block number = 2 : xor counter W into A

A : 2132BFA4 E13B89AC 56129D16 DDB5A2F3 BBD1DD24 F0C64B27 D4FD5B1A 35342F97
    690292D9 81390905 FE4C9A18 773E6359

B : D9BF6951 58BAD750 56028CB2 8134F359 B5D469D8 941A8CC2 418B2A6E 04052780
    7F07D787 5194358F 3C60D665 BE97D79A 950C3434 AED9A06D 2537DC8D 7CDB5969

C : 74B5FA19 7518DD5A A370733E A0AE0565 13DB5342 5A5C072B 7762521C 75916462
    59CCDEDB 8FEC32AC 3E09476A C0BB7EC7 CC6311C9 4E9D08AE 449D8540 8C6C0223

W : 00000002 00000000


block number = 2 : permutation - rotate B

A : 2132BFA4 E13B89AC 56129D16 DDB5A2F3 BBD1DD24 F0C64B27 D4FD5B1A 35342F97
    690292D9 81390905 FE4C9A18 773E6359

B : D2A3B37E AEA0B175 1964AC05 E6B30269 D3B16BA8 19852835 54DC8316 4F00080A
    AF0EFE0F 6B1EA328 ACCA78C1 AF357D2F 68692A18 40DB5DB3 B91A4A6F B2D2F9B6

C : 74B5FA19 7518DD5A A370733E A0AE0565 13DB5342 5A5C072B 7762521C 75916462
    59CCDEDB 8FEC32AC 3E09476A C0BB7EC7 CC6311C9 4E9D08AE 449D8540 8C6C0223

W : 00000002 00000000


block number = 2 : permutation (j = 0)

A : F7A45AB6 1B7B9FDD D4AE951B 98E4D1C7 4052E830 3D1FBFEB 49353DB2 A6F893F7
    B07B4DEA F089D424 BDC0D6F0 7B2B5AC3

B : B0848635 348DCAB2 11277606 A3647ED9 18CFC09E F1EA107E 1F73C461 C7077C1C
    11994E0A D94B6D8B 1BABD88C DABE5F63 D889F179 6532DB44 5965FE3B 02BEDD55

C : 74B5FA19 7518DD5A A370733E A0AE0565 13DB5342 5A5C072B 7762521C 75916462
    59CCDEDB 8FEC32AC 3E09476A C0BB7EC7 CC6311C9 4E9D08AE 449D8540 8C6C0223

W : 00000002 00000000
```

```
block number = 2 : permutation (j = 1)

A : 9E9110D9 C861EBC1 4A80AA8B B28E5A69 BE4F19B6 B920408A CBB77E5D A598F1B6
    49D3DDF2 78B226D9 37DF0A9E 2393870E

B : 7A0A5BA9 131423DD 891CBBA5 4AE41FC3 87B3A331 6499F9DB F6C77DA3 526280C8
    425C7332 8508CF29 8228E46C F80D1B51 F0A304BA 8CBA09FD 86837DD4 5F1AB4E3

C : 74B5FA19 7518DD5A A370733E A0AE0565 13DB5342 5A5C072B 7762521C 75916462
    59CCDEDB 8FEC32AC 3E09476A C0BB7EC7 CC6311C9 4E9D08AE 449D8540 8C6C0223

W : 00000002 00000000


block number = 2 : permutation (j = 2)

A : 80F57FDC 5D4889F6 E62F1768 641AF882 BDCDCA7F 9AE71255 CFF6CE83 5B672F19
    B9351F52 535483EF 8C3EB8EC 56356B34

B : 635F1FA7 A17777DE DDF0EB20 FA7236AA 706DC640 6B8485BF F45E13D0 3F2006ED
    C68AD3E4 6F0973F9 3458F9A5 5482E645 A78CE9D8 B5DF6FEB 7EC7BCBA 17FFFD0D

C : 74B5FA19 7518DD5A A370733E A0AE0565 13DB5342 5A5C072B 7762521C 75916462
    59CCDEDB 8FEC32AC 3E09476A C0BB7EC7 CC6311C9 4E9D08AE 449D8540 8C6C0223

W : 00000002 00000000


block number = 2 : permutation - add C to A

A : 6ECB062B B23CE91A 0441049B C38B431C 60793669 7D453E8B 2F57E5B4 B4433BDD
    903007E0 8D5FC7D5 C523FB71 503E89FA

B : 635F1FA7 A17777DE DDF0EB20 FA7236AA 706DC640 6B8485BF F45E13D0 3F2006ED
    C68AD3E4 6F0973F9 3458F9A5 5482E645 A78CE9D8 B5DF6FEB 7EC7BCBA 17FFFD0D

C : 74B5FA19 7518DD5A A370733E A0AE0565 13DB5342 5A5C072B 7762521C 75916462
    59CCDEDB 8FEC32AC 3E09476A C0BB7EC7 CC6311C9 4E9D08AE 449D8540 8C6C0223

W : 00000002 00000000


block number = 2 : subtract M from C

A : 6ECB062B B23CE91A 0441049B C38B431C 60793669 7D453E8B 2F57E5B4 B4433BDD
    903007E0 8D5FC7D5 C523FB71 503E89FA

B : 635F1FA7 A17777DE DDF0EB20 FA7236AA 706DC640 6B8485BF F45E13D0 3F2006ED
    C68AD3E4 6F0973F9 3458F9A5 5482E645 A78CE9D8 B5DF6FEB 7EC7BCBA 17FFFD0D

C : 74B5F999 7518DD5A A370733E A0AE0565 13DB5342 5A5C072B 7762521C 75916462
    59CCDEDB 8FEC32AC 3E09476A C0BB7EC7 CC6311C9 4E9D08AE 449D8540 8C6C0223

W : 00000002 00000000


block number = 2 : swap B with C

A : 6ECB062B B23CE91A 0441049B C38B431C 60793669 7D453E8B 2F57E5B4 B4433BDD
    903007E0 8D5FC7D5 C523FB71 503E89FA

B : 74B5F999 7518DD5A A370733E A0AE0565 13DB5342 5A5C072B 7762521C 75916462
    59CCDEDB 8FEC32AC 3E09476A C0BB7EC7 CC6311C9 4E9D08AE 449D8540 8C6C0223

C : 635F1FA7 A17777DE DDF0EB20 FA7236AA 706DC640 6B8485BF F45E13D0 3F2006ED
    C68AD3E4 6F0973F9 3458F9A5 5482E645 A78CE9D8 B5DF6FEB 7EC7BCBA 17FFFD0D
```

```
W : 00000002 00000000


n0_final = 0 : add M to B

A : 6ECB062B B23CE91A 0441049B C38B431C 60793669 7D453E8B 2F57E5B4 B4433BDD
    903007E0 8D5FC7D5 C523FB71 503E89FA

B : 74B5FA19 7518DD5A A370733E A0AE0565 13DB5342 5A5C072B 7762521C 75916462
    59CCDEDB 8FEC32AC 3E09476A C0BB7EC7 CC6311C9 4E9D08AE 449D8540 8C6C0223

C : 635F1FA7 A17777DE DDF0EB20 FA7236AA 706DC640 6B8485BF F45E13D0 3F2006ED
    C68AD3E4 6F0973F9 3458F9A5 5482E645 A78CE9D8 B5DF6FEB 7EC7BCBA 17FFFD0D

W : 00000002 00000000


n0_final = 0 : xor counter W into A

A : 6ECB0629 B23CE91A 0441049B C38B431C 60793669 7D453E8B 2F57E5B4 B4433BDD
    903007E0 8D5FC7D5 C523FB71 503E89FA

B : 74B5FA19 7518DD5A A370733E A0AE0565 13DB5342 5A5C072B 7762521C 75916462
    59CCDEDB 8FEC32AC 3E09476A C0BB7EC7 CC6311C9 4E9D08AE 449D8540 8C6C0223

C : 635F1FA7 A17777DE DDF0EB20 FA7236AA 706DC640 6B8485BF F45E13D0 3F2006ED
    C68AD3E4 6F0973F9 3458F9A5 5482E645 A78CE9D8 B5DF6FEB 7EC7BCBA 17FFFD0D

W : 00000002 00000000


n0_final = 0 : permutation - rotate B

A : 6ECB0629 B23CE91A 0441049B C38B431C 60793669 7D453E8B 2F57E5B4 B4433BDD
    903007E0 8D5FC7D5 C523FB71 503E89FA

B : F432E96B BAB4EA31 E67D46E0 0ACB415C A68427B6 0E56B4B8 A438EEC4 C8C4EB22
    BDB6B399 65591FD8 8ED47C12 FD8F8176 239398C6 115C9D3A 0A80893B 044718D8

C : 635F1FA7 A17777DE DDF0EB20 FA7236AA 706DC640 6B8485BF F45E13D0 3F2006ED
    C68AD3E4 6F0973F9 3458F9A5 5482E645 A78CE9D8 B5DF6FEB 7EC7BCBA 17FFFD0D

W : 00000002 00000000


n0_final = 0 : permutation (j = 0)

A : 149D7202 183AC7E6 54A20EE4 71EFEB74 4B531C29 5B5C9818 91630F92 66A9BA59
    8530F23A E9D24A81 265E3865 1AF5CFAB

B : 9597F988 DDAFB60C 9AAC0D55 3EF4CF5C F9A4ACBB B80E0E97 26ED2DE4 08DF93E3
    01A26AF6 DC9F8ACE C4093FBF 1E1532B9 AC45BC71 C57C026D BE5CE36D 869E253B

C : 635F1FA7 A17777DE DDF0EB20 FA7236AA 706DC640 6B8485BF F45E13D0 3F2006ED
    C68AD3E4 6F0973F9 3458F9A5 5482E645 A78CE9D8 B5DF6FEB 7EC7BCBA 17FFFD0D

W : 00000002 00000000


n0_final = 0 : permutation (j = 1)

A : DC5BAB2B 8C22F3E3 E64FD398 3FCC92AC F04712D0 395B6353 2C7F8D1F A8E0C0F6
    EDF6D320 81F104D5 221FCB0E A9A74700

B : 0DCEF553 028E5C2D 74EF49F5 47B48CD3 E14075A8 0E12E605 903A6F39 47E79F39
    20E08138 CAE21981 91A25318 FC190821 573395CC 4C5C9877 AF39B43B 5A23757E
```

```
C : 635F1FA7 A17777DE DDF0EB20 FA7236AA 706DC640 6B8485BF F45E13D0 3F2006ED
    C68AD3E4 6F0973F9 3458F9A5 5482E645 A78CE9D8 B5DF6FEB 7EC7BCBA 17FFFD0D

W : 00000002 00000000


n0_final = 0 : permutation (j = 2)

A : CCE62684 2BDFF7E8 53FEF52D AEFA4E31 69AB67CB A5759544 2A7D89AD B8A165CD
    C145171C 18B81F66 A5213421 22837339

B : B9C369F2 D9A28BE6 6AA91390 2A8F6A26 F199322A C805C41D 8C75D4A1 DECA8FBC
    D7959A44 CF4E59B8 F6C6D063 BF6C8A71 90DDC37B 7FFED077 04ADA3A9 693A663A

C : 635F1FA7 A17777DE DDF0EB20 FA7236AA 706DC640 6B8485BF F45E13D0 3F2006ED
    C68AD3E4 6F0973F9 3458F9A5 5482E645 A78CE9D8 B5DF6FEB 7EC7BCBA 17FFFD0D

W : 00000002 00000000


n0_final = 0 : permutation - add C to A

A : 33DB4080 A739C7A7 16DA62B5 001109DB BB3DA26F 3FCD4F0F A682FB43 BF495E62
    4F5A3AF8 F73DA362 358E9DC4 CA023D68

B : B9C369F2 D9A28BE6 6AA91390 2A8F6A26 F199322A C805C41D 8C75D4A1 DECA8FBC
    D7959A44 CF4E59B8 F6C6D063 BF6C8A71 90DDC37B 7FFED077 04ADA3A9 693A663A

C : 635F1FA7 A17777DE DDF0EB20 FA7236AA 706DC640 6B8485BF F45E13D0 3F2006ED
    C68AD3E4 6F0973F9 3458F9A5 5482E645 A78CE9D8 B5DF6FEB 7EC7BCBA 17FFFD0D

W : 00000002 00000000


n0_final = 0 : subtract M from C

A : 33DB4080 A739C7A7 16DA62B5 001109DB BB3DA26F 3FCD4F0F A682FB43 BF495E62
    4F5A3AF8 F73DA362 358E9DC4 CA023D68

B : B9C369F2 D9A28BE6 6AA91390 2A8F6A26 F199322A C805C41D 8C75D4A1 DECA8FBC
    D7959A44 CF4E59B8 F6C6D063 BF6C8A71 90DDC37B 7FFED077 04ADA3A9 693A663A

C : 635F1F27 A17777DE DDF0EB20 FA7236AA 706DC640 6B8485BF F45E13D0 3F2006ED
    C68AD3E4 6F0973F9 3458F9A5 5482E645 A78CE9D8 B5DF6FEB 7EC7BCBA 17FFFD0D

W : 00000002 00000000


n0_final = 0 : swap B with C

A : 33DB4080 A739C7A7 16DA62B5 001109DB BB3DA26F 3FCD4F0F A682FB43 BF495E62
    4F5A3AF8 F73DA362 358E9DC4 CA023D68

B : 635F1F27 A17777DE DDF0EB20 FA7236AA 706DC640 6B8485BF F45E13D0 3F2006ED
    C68AD3E4 6F0973F9 3458F9A5 5482E645 A78CE9D8 B5DF6FEB 7EC7BCBA 17FFFD0D

C : B9C369F2 D9A28BE6 6AA91390 2A8F6A26 F199322A C805C41D 8C75D4A1 DECA8FBC
    D7959A44 CF4E59B8 F6C6D063 BF6C8A71 90DDC37B 7FFED077 04ADA3A9 693A663A

W : 00000002 00000000


n0_final = 1 : add M to B

A : 33DB4080 A739C7A7 16DA62B5 001109DB BB3DA26F 3FCD4F0F A682FB43 BF495E62
    4F5A3AF8 F73DA362 358E9DC4 CA023D68
```

```
B : 635F1FA7 A17777DE DDF0EB20 FA7236AA 706DC640 6B8485BF F45E13D0 3F2006ED
    C68AD3E4 6F0973F9 3458F9A5 5482E645 A78CE9D8 B5DF6FEB 7EC7BCBA 17FFFD0D

C : B9C369F2 D9A28BE6 6AA91390 2A8F6A26 F199322A C805C41D 8C75D4A1 DECA8FBC
    D7959A44 CF4E59B8 F6C6D063 BF6C8A71 90DDC37B 7FFED077 04ADA3A9 693A663A

W : 00000002 00000000


n0_final = 1 : xor counter W into A

A : 33DB4082 A739C7A7 16DA62B5 001109DB BB3DA26F 3FCD4F0F A682FB43 BF495E62
    4F5A3AF8 F73DA362 358E9DC4 CA023D68

B : 635F1FA7 A17777DE DDF0EB20 FA7236AA 706DC640 6B8485BF F45E13D0 3F2006ED
    C68AD3E4 6F0973F9 3458F9A5 5482E645 A78CE9D8 B5DF6FEB 7EC7BCBA 17FFFD0D

C : B9C369F2 D9A28BE6 6AA91390 2A8F6A26 F199322A C805C41D 8C75D4A1 DECA8FBC
    D7959A44 CF4E59B8 F6C6D063 BF6C8A71 90DDC37B 7FFED077 04ADA3A9 693A663A

W : 00000002 00000000


n0_final = 1 : permutation - rotate B

A : 33DB4082 A739C7A7 16DA62B5 001109DB BB3DA26F 3FCD4F0F A682FB43 BF495E62
    4F5A3AF8 F73DA362 358E9DC4 CA023D68

B : 3F4EC6BE EFBD42EE D641BBE1 6D55F4E4 8C80E0DB 0B7ED709 27A1E8BC 0DDA7E40
    A7C98D15 E7F2DE12 F34A68B1 CC8AA905 D3B14F19 DFD76BBE 7974FD8F FA1A2FFF

C : B9C369F2 D9A28BE6 6AA91390 2A8F6A26 F199322A C805C41D 8C75D4A1 DECA8FBC
    D7959A44 CF4E59B8 F6C6D063 BF6C8A71 90DDC37B 7FFED077 04ADA3A9 693A663A

W : 00000002 00000000


n0_final = 1 : permutation (j = 0)

A : 91502CEF FA5B99ED AE56D69D 589B022C 7BDA9079 AA8CBB35 2FA4D3A1 DACFCDAB
    1B913AB3 3D90A836 F648A85B 09B6DD66

B : E78664F6 6E75DC7F EEB05E27 4F27B4E0 9D24AE31 438EEAD8 9F18FD26 3E84CED4
    ABFDDF67 0D8AEBEC EF2386C7 6F5C7092 C9CD4D23 BA0AB16F A340D27C 5350A22C

C : B9C369F2 D9A28BE6 6AA91390 2A8F6A26 F199322A C805C41D 8C75D4A1 DECA8FBC
    D7959A44 CF4E59B8 F6C6D063 BF6C8A71 90DDC37B 7FFED077 04ADA3A9 693A663A

W : 00000002 00000000


n0_final = 1 : permutation (j = 1)

A : F0F3C949 1A14D83E 7FABC2E3 DCE75A6F 67C8EA56 A9ABF3BE 4C8A9A07 51256499
    A2AE3DDD FC862E24 F276FC93 D03C05B4

B : F11E0201 221265AB D5A7D1A1 2F5F3B13 67189E41 8464046B 33B8F921 52CA67E3
    58F78879 FEFEF019 5E133093 FDA044B4 0BAD8FEE 22416E9E F5F4C101 087BDF3E

C : B9C369F2 D9A28BE6 6AA91390 2A8F6A26 F199322A C805C41D 8C75D4A1 DECA8FBC
    D7959A44 CF4E59B8 F6C6D063 BF6C8A71 90DDC37B 7FFED077 04ADA3A9 693A663A

W : 00000002 00000000


n0_final = 1 : permutation (j = 2)
```

```
A : 0C9FC9C6 7AC06949 2349DF2D 1E4407AD E84F7CB7 7DC90184 235A999E B72D5614
    7B08EA46 44C9F247 6D16C208 6A402409

B : D90A5233 625DBAFC A04AE0F1 15BBA584 3D510ABB 8DF79E61 BBC7D290 442F3794
    A65F93BA 7FCB1E48 60830747 B3922082 93AC0A65 FFB4D084 7900BFF4 8548658A

C : B9C369F2 D9A28BE6 6AA91390 2A8F6A26 F199322A C805C41D 8C75D4A1 DECA8FBC
    D7959A44 CF4E59B8 F6C6D063 BF6C8A71 90DDC37B 7FFED077 04ADA3A9 693A663A

W : 00000002 00000000


n0_final = 1 : permutation - add C to A

A : 5FD62497 B6FAC8E0 44F0FFA7 1A109387 5AE3DCD3 00BB37E4 94514359 A5130EA8
    43CF6E99 9ED68230 8469B054 F22A6CB6

B : D90A5233 625DBAFC A04AE0F1 15BBA584 3D510ABB 8DF79E61 BBC7D290 442F3794
    A65F93BA 7FCB1E48 60830747 B3922082 93AC0A65 FFB4D084 7900BFF4 8548658A

C : B9C369F2 D9A28BE6 6AA91390 2A8F6A26 F199322A C805C41D 8C75D4A1 DECA8FBC
    D7959A44 CF4E59B8 F6C6D063 BF6C8A71 90DDC37B 7FFED077 04ADA3A9 693A663A

W : 00000002 00000000


n0_final = 1 : subtract M from C

A : 5FD62497 B6FAC8E0 44F0FFA7 1A109387 5AE3DCD3 00BB37E4 94514359 A5130EA8
    43CF6E99 9ED68230 8469B054 F22A6CB6

B : D90A5233 625DBAFC A04AE0F1 15BBA584 3D510ABB 8DF79E61 BBC7D290 442F3794
    A65F93BA 7FCB1E48 60830747 B3922082 93AC0A65 FFB4D084 7900BFF4 8548658A

C : B9C36972 D9A28BE6 6AA91390 2A8F6A26 F199322A C805C41D 8C75D4A1 DECA8FBC
    D7959A44 CF4E59B8 F6C6D063 BF6C8A71 90DDC37B 7FFED077 04ADA3A9 693A663A

W : 00000002 00000000


n0_final = 1 : swap B with C

A : 5FD62497 B6FAC8E0 44F0FFA7 1A109387 5AE3DCD3 00BB37E4 94514359 A5130EA8
    43CF6E99 9ED68230 8469B054 F22A6CB6

B : B9C36972 D9A28BE6 6AA91390 2A8F6A26 F199322A C805C41D 8C75D4A1 DECA8FBC
    D7959A44 CF4E59B8 F6C6D063 BF6C8A71 90DDC37B 7FFED077 04ADA3A9 693A663A

C : D90A5233 625DBAFC A04AE0F1 15BBA584 3D510ABB 8DF79E61 BBC7D290 442F3794
    A65F93BA 7FCB1E48 60830747 B3922082 93AC0A65 FFB4D084 7900BFF4 8548658A

W : 00000002 00000000


n0_final = 2 : add M to B

A : 5FD62497 B6FAC8E0 44F0FFA7 1A109387 5AE3DCD3 00BB37E4 94514359 A5130EA8
    43CF6E99 9ED68230 8469B054 F22A6CB6

B : B9C369F2 D9A28BE6 6AA91390 2A8F6A26 F199322A C805C41D 8C75D4A1 DECA8FBC
    D7959A44 CF4E59B8 F6C6D063 BF6C8A71 90DDC37B 7FFED077 04ADA3A9 693A663A

C : D90A5233 625DBAFC A04AE0F1 15BBA584 3D510ABB 8DF79E61 BBC7D290 442F3794
    A65F93BA 7FCB1E48 60830747 B3922082 93AC0A65 FFB4D084 7900BFF4 8548658A

W : 00000002 00000000
```

```
n0_final = 2 : xor counter W into A

A : 5FD62495 B6FAC8E0 44F0FFA7 1A109387 5AE3DCD3 00BB37E4 94514359 A5130EA8
    43CF6E99 9ED68230 8469B054 F22A6CB6

B : B9C369F2 D9A28BE6 6AA91390 2A8F6A26 F199322A C805C41D 8C75D4A1 DECA8FBC
    D7959A44 CF4E59B8 F6C6D063 BF6C8A71 90DDC37B 7FFED077 04ADA3A9 693A663A

C : D90A5233 625DBAFC A04AE0F1 15BBA584 3D510ABB 8DF79E61 BBC7D290 442F3794
    A65F93BA 7FCB1E48 60830747 B3922082 93AC0A65 FFB4D084 7900BFF4 8548658A

W : 00000002 00000000


n0_final = 2 : permutation - rotate B

A : 5FD62495 B6FAC8E0 44F0FFA7 1A109387 5AE3DCD3 00BB37E4 94514359 A5130EA8
    43CF6E99 9ED68230 8469B054 F22A6CB6

B : D3E57386 17CDB345 2720D552 D44C551E 6455E332 883B900B A94318EB 1F79BD95
    3489AF2B B3719E9C A0C7ED8D 14E37ED9 86F721BB A0EEFFFD 4752095B CC74D274

C : D90A5233 625DBAFC A04AE0F1 15BBA584 3D510ABB 8DF79E61 BBC7D290 442F3794
    A65F93BA 7FCB1E48 60830747 B3922082 93AC0A65 FFB4D084 7900BFF4 8548658A

W : 00000002 00000000


n0_final = 2 : permutation (j = 0)

A : B50F955E 14BC3748 C5FA1146 194908C5 7004768B D42009BE 05496831 86CDE330
    8C38660A 26BE8839 7BE28900 4FCA5136

B : 0829DF49 3335F5B0 2F241782 C99DAF9D 47504F10 3BA8D656 A830A619 47C167E5
    1AD4C7A3 BFA24AFF C592ADE4 99F3537B 471E29D6 AA9E374C B4A1FC0F 7E5F53D3

C : D90A5233 625DBAFC A04AE0F1 15BBA584 3D510ABB 8DF79E61 BBC7D290 442F3794
    A65F93BA 7FCB1E48 60830747 B3922082 93AC0A65 FFB4D084 7900BFF4 8548658A

W : 00000002 00000000


n0_final = 2 : permutation (j = 1)

A : EEEC612F 0C2858E0 110AB337 D2E00B4E F5654CBC 0C7EE713 8760F284 AB6C79AE
    F7C8228E 1E53B0CE 8A9CF588 4F5F50B5

B : 99963048 0D50ADA1 0A73C891 17DCC13D 86974351 96FDE39D 25024644 3F226080
    24BA1196 8C9332E0 65D01701 1EF95246 84A6E0EF A6BD7675 11DCF564 A82D21F7

C : D90A5233 625DBAFC A04AE0F1 15BBA584 3D510ABB 8DF79E61 BBC7D290 442F3794
    A65F93BA 7FCB1E48 60830747 B3922082 93AC0A65 FFB4D084 7900BFF4 8548658A

W : 00000002 00000000


n0_final = 2 : permutation (j = 2)

A : D13EEC24 6EE69F0F 10237A16 6772D8CD 218A7975 16EE22A1 149915B3 1E746A94
    DD0795A2 0128DE26 40A44561 8D95E288

B : C6168015 0A3F1FC8 688DD952 8E9E2FED 23EF9578 BCE2A7CB A5D80961 E6C9E632
    9701A5A6 F037B89F 20C6C44E DC7931E7 2BB5AB82 B3ADCD32 9CE25056 22305E98

C : D90A5233 625DBAFC A04AE0F1 15BBA584 3D510ABB 8DF79E61 BBC7D290 442F3794
```

```
    A65F93BA 7FCB1E48 60830747 B3922082 93AC0A65 FFB4D084 7900BFF4 8548658A

W : 00000002 00000000


n0_final = 2 : permutation - add C to A

A : 1FD517B4 18EE0662 002DA3F7 3C864C42 00BDBC17 D3A91349 84B98D58 DB0A255C
    EA84933C 78858700 4E1BD28E 22E17C53

B : C6168015 0A3F1FC8 688DD952 8E9E2FED 23EF9578 BCE2A7CB A5D80961 E6C9E632
    9701A5A6 F037B89F 20C6C44E DC7931E7 2BB5AB82 B3ADCD32 9CE25056 22305E98

C : D90A5233 625DBAFC A04AE0F1 15BBA584 3D510ABB 8DF79E61 BBC7D290 442F3794
    A65F93BA 7FCB1E48 60830747 B3922082 93AC0A65 FFB4D084 7900BFF4 8548658A

W : 00000002 00000000


n0_final = 2 : subtract M from C

A : 1FD517B4 18EE0662 002DA3F7 3C864C42 00BDBC17 D3A91349 84B98D58 DB0A255C
    EA84933C 78858700 4E1BD28E 22E17C53

B : C6168015 0A3F1FC8 688DD952 8E9E2FED 23EF9578 BCE2A7CB A5D80961 E6C9E632
    9701A5A6 F037B89F 20C6C44E DC7931E7 2BB5AB82 B3ADCD32 9CE25056 22305E98

C : D90A51B3 625DBAFC A04AE0F1 15BBA584 3D510ABB 8DF79E61 BBC7D290 442F3794
    A65F93BA 7FCB1E48 60830747 B3922082 93AC0A65 FFB4D084 7900BFF4 8548658A

W : 00000002 00000000


n0_final = 2 : swap B with C  (final state)

A : 1FD517B4 18EE0662 002DA3F7 3C864C42 00BDBC17 D3A91349 84B98D58 DB0A255C
    EA84933C 78858700 4E1BD28E 22E17C53

B : D90A51B3 625DBAFC A04AE0F1 15BBA584 3D510ABB 8DF79E61 BBC7D290 442F3794
    A65F93BA 7FCB1E48 60830747 B3922082 93AC0A65 FFB4D084 7900BFF4 8548658A

C : C6168015 0A3F1FC8 688DD952 8E9E2FED 23EF9578 BCE2A7CB A5D80961 E6C9E632
    9701A5A6 F037B89F 20C6C44E DC7931E7 2BB5AB82 B3ADCD32 9CE25056 22305E98

W : 00000002 00000000


Hash value (word array):

H : C6168015 0A3F1FC8 688DD952 8E9E2FED 23EF9578 BCE2A7CB A5D80961 E6C9E632
    9701A5A6 F037B89F 20C6C44E DC7931E7 2BB5AB82 B3ADCD32 9CE25056 22305E98


Hash value (byte array):

H : 15 80 16 C6 C8 1F 3F 0A 52 D9 8D 68 ED 2F 9E 8E
    78 95 EF 23 CB A7 E2 BC 61 09 D8 A5 32 E6 C9 E6
    A6 A5 01 97 9F B8 37 F0 4E C4 C6 20 E7 31 79 DC
    82 AB B5 2B 32 CD AD B3 56 50 E2 9C 98 5E 30 22
```

B.10 Intermediate States for Shabal-512 (Message B)

```
init
```

```
A : 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
    00000000 00000000 00000000 00000000

B : 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
    00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000

C : 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
    00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000

W : FFFFFFFF FFFFFFFF


block number = -1 : message block

M : 00000200 00000201 00000202 00000203 00000204 00000205 00000206 00000207
    00000208 00000209 0000020A 0000020B 0000020C 0000020D 0000020E 0000020F


block number = -1 : add M to B

A : 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
    00000000 00000000 00000000 00000000

B : 00000200 00000201 00000202 00000203 00000204 00000205 00000206 00000207
    00000208 00000209 0000020A 0000020B 0000020C 0000020D 0000020E 0000020F

C : 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
    00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000

W : FFFFFFFF FFFFFFFF


block number = -1 : xor counter W into A

A : FFFFFFFF FFFFFFFF 00000000 00000000 00000000 00000000 00000000 00000000
    00000000 00000000 00000000 00000000

B : 00000200 00000201 00000202 00000203 00000204 00000205 00000206 00000207
    00000208 00000209 0000020A 0000020B 0000020C 0000020D 0000020E 0000020F

C : 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
    00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000

W : FFFFFFFF FFFFFFFF


block number = -1 : permutation - rotate B

A : FFFFFFFF FFFFFFFF 00000000 00000000 00000000 00000000 00000000 00000000
    00000000 00000000 00000000 00000000

B : 04000000 04020000 04040000 04060000 04080000 040A0000 040C0000 040E0000
    04100000 04120000 04140000 04160000 04180000 041A0000 041C0000 041E0000

C : 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
    00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000

W : FFFFFFFF FFFFFFFF


block number = -1 : permutation (j = 0)

A : E2AA101E 3A5C2CEA E01C099D 8D1BE979 1C49D94D 7A06796B E9703766 7C25F07F
    FEB14424 C24293CA BC669928 76D9392D

B : 0C080202 FCF8E3B6 A7CC2F72 67D0264D EBA626B2 8DED8694 1E97C899 8BC60F80
    096EBBDB 35996C35 4BB166D7 810AC6D2 1565EFE1 CD97D315 17DBF662 7AD81686
```

```
C : 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
    00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000

W : FFFFFFFF FFFFFFFF


block number = -1 : permutation (j = 1)

A : E524968F 5F83748F 98E7810D 23285DCF 2195FE4E 7188ACDA 7AF7FC83 31F5BA1C
    CC8D5743 4E46BE5D DAE5E19B BCEB09F1

B : 98C2A26E 203FE129 91D449C4 86D27C0E E43EE5D9 AA624C8B 18358F56 5498E90F
    08061EC6 CB4E531A F07AB35C DEC22F95 F4A1DE73 1558F50E AABFEFB8 3BBA68EF

C : 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
    00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000

W : FFFFFFFF FFFFFFFF


block number = -1 : permutation (j = 2)

A : DCED6C97 B5937A56 BC2D3479 4B2672FE A762CE01 E0587D3E DAAA7818 F9BF94BD
    646EB744 A0DA0A55 C70BC02E 292C7C6B

B : C1099EC7 07B38804 E7442E38 CC8AD853 EB6F58DB 1EA81CBE 73B9D52A 1DE85F1F
    48910C72 893B24F4 C5A0E15E BBC43469 72D2F45C 75941FB6 6D8BE0A0 A1A7524A

C : 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
    00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000

W : FFFFFFFF FFFFFFFF


block number = -1 : permutation - add C to A

A : DCED6C97 B5937A56 BC2D3479 4B2672FE A762CE01 E0587D3E DAAA7818 F9BF94BD
    646EB744 A0DA0A55 C70BC02E 292C7C6B

B : C1099EC7 07B38804 E7442E38 CC8AD853 EB6F58DB 1EA81CBE 73B9D52A 1DE85F1F
    48910C72 893B24F4 C5A0E15E BBC43469 72D2F45C 75941FB6 6D8BE0A0 A1A7524A

C : 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
    00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000

W : FFFFFFFF FFFFFFFF


block number = -1 : subtract M from C

A : DCED6C97 B5937A56 BC2D3479 4B2672FE A762CE01 E0587D3E DAAA7818 F9BF94BD
    646EB744 A0DA0A55 C70BC02E 292C7C6B

B : C1099EC7 07B38804 E7442E38 CC8AD853 EB6F58DB 1EA81CBE 73B9D52A 1DE85F1F
    48910C72 893B24F4 C5A0E15E BBC43469 72D2F45C 75941FB6 6D8BE0A0 A1A7524A

C : FFFFFE00 FFFFFDFF FFFFFDFE FFFFFDFD FFFFFDFC FFFFFDFB FFFFFDFA FFFFFDF9
    FFFFFDF8 FFFFFDF7 FFFFFDF6 FFFFFDF5 FFFFFDF4 FFFFFDF3 FFFFFDF2 FFFFFDF1

W : FFFFFFFF FFFFFFFF


block number = -1 : swap B with C

A : DCED6C97 B5937A56 BC2D3479 4B2672FE A762CE01 E0587D3E DAAA7818 F9BF94BD
    646EB744 A0DA0A55 C70BC02E 292C7C6B
```

```
B : FFFFFE00 FFFFFDFF FFFFFDFE FFFFFDFD FFFFFDFC FFFFFDFB FFFFFDFA FFFFFDF9
    FFFFFDF8 FFFFFDF7 FFFFFDF6 FFFFFDF5 FFFFFDF4 FFFFFDF3 FFFFFDF2 FFFFFDF1

C : C1099EC7 07B38804 E7442E38 CC8AD853 EB6F58DB 1EA81CBE 73B9D52A 1DE85F1F
    48910C72 893B24F4 C5A0E15E BBC43469 72D2F45C 75941FB6 6D8BE0A0 A1A7524A

W : FFFFFFFF FFFFFFFF


block number = 0 : increment counter W

A : DCED6C97 B5937A56 BC2D3479 4B2672FE A762CE01 E0587D3E DAAA7818 F9BF94BD
    646EB744 A0DA0A55 C70BC02E 292C7C6B

B : FFFFFE00 FFFFFDFF FFFFFDFE FFFFFDFD FFFFFDFC FFFFFDFB FFFFFDFA FFFFFDF9
    FFFFFDF8 FFFFFDF7 FFFFFDF6 FFFFFDF5 FFFFFDF4 FFFFFDF3 FFFFFDF2 FFFFFDF1

C : C1099EC7 07B38804 E7442E38 CC8AD853 EB6F58DB 1EA81CBE 73B9D52A 1DE85F1F
    48910C72 893B24F4 C5A0E15E BBC43469 72D2F45C 75941FB6 6D8BE0A0 A1A7524A

W : 00000000 00000000


block number = 0 : message block

M : 00000210 00000211 00000212 00000213 00000214 00000215 00000216 00000217
    00000218 00000219 0000021A 0000021B 0000021C 0000021D 0000021E 0000021F


block number = 0 : add M to B

A : DCED6C97 B5937A56 BC2D3479 4B2672FE A762CE01 E0587D3E DAAA7818 F9BF94BD
    646EB744 A0DA0A55 C70BC02E 292C7C6B

B : 00000010 00000010 00000010 00000010 00000010 00000010 00000010 00000010
    00000010 00000010 00000010 00000010 00000010 00000010 00000010 00000010

C : C1099EC7 07B38804 E7442E38 CC8AD853 EB6F58DB 1EA81CBE 73B9D52A 1DE85F1F
    48910C72 893B24F4 C5A0E15E BBC43469 72D2F45C 75941FB6 6D8BE0A0 A1A7524A

W : 00000000 00000000


block number = 0 : xor counter W into A

A : DCED6C97 B5937A56 BC2D3479 4B2672FE A762CE01 E0587D3E DAAA7818 F9BF94BD
    646EB744 A0DA0A55 C70BC02E 292C7C6B

B : 00000010 00000010 00000010 00000010 00000010 00000010 00000010 00000010
    00000010 00000010 00000010 00000010 00000010 00000010 00000010 00000010

C : C1099EC7 07B38804 E7442E38 CC8AD853 EB6F58DB 1EA81CBE 73B9D52A 1DE85F1F
    48910C72 893B24F4 C5A0E15E BBC43469 72D2F45C 75941FB6 6D8BE0A0 A1A7524A

W : 00000000 00000000


block number = 0 : permutation - rotate B

A : DCED6C97 B5937A56 BC2D3479 4B2672FE A762CE01 E0587D3E DAAA7818 F9BF94BD
    646EB744 A0DA0A55 C70BC02E 292C7C6B

B : 00200000 00200000 00200000 00200000 00200000 00200000 00200000 00200000
    00200000 00200000 00200000 00200000 00200000 00200000 00200000 00200000

C : C1099EC7 07B38804 E7442E38 CC8AD853 EB6F58DB 1EA81CBE 73B9D52A 1DE85F1F
```

```
    48910C72 893B24F4 C5A0E15E BBC43469 72D2F45C 75941FB6 6D8BE0A0 A1A7524A

W : 00000000 00000000


block number = 0 : permutation (j = 0)

A : E3C2DE4A 15013A27 9779A969 2E533DBB F9542F32 1830E439 49354FD1 3FA73E25
    62A98156 E3930697 3AEC9AE8 02836D8A

B : 15F96FCE BF883B5C 13B9B891 5F17F2CF 06EBD0CD E78F1BC6 B68AB02E C018C1DA
    9D167EA9 1C2CF968 C5536517 FD3C9275 1C7D21B5 EABEC5D8 68C65696 D1ECC244

C : C1099EC7 07B38804 E7442E38 CC8AD853 EB6F58DB 1EA81CBE 73B9D52A 1DE85F1F
    48910C72 893B24F4 C5A0E15E BBC43469 72D2F45C 75941FB6 6D8BE0A0 A1A7524A

W : 00000000 00000000


block number = 0 : permutation (j = 1)

A : F762247A 9C5C1A1B 82879280 482A0C43 F996F842 F06FAEB4 6FBAE548 DD572216
    AC03820D 16A91946 E8198C7A C460216D

B : 3453DF4C 85E7DAA3 A8D88D92 0E0FE970 5E2BDC68 2648D134 7AF313D8 BBAE5D27
    32B126D6 5BFA1734 F7DEA750 4DACD757 3E9344D7 DAEDDAFA 41C9B79B 81715960

C : C1099EC7 07B38804 E7442E38 CC8AD853 EB6F58DB 1EA81CBE 73B9D52A 1DE85F1F
    48910C72 893B24F4 C5A0E15E BBC43469 72D2F45C 75941FB6 6D8BE0A0 A1A7524A

W : 00000000 00000000


block number = 0 : permutation (j = 2)

A : F67C2EF7 2774D155 4B92F221 8CA66230 E59A65D4 199FE418 2C22673B DA3186CB
    17D54265 E4FDEA67 595B4C44 81C61457

B : D9BF68D1 58BAD750 56028CB2 8134F359 B5D469D8 941A8CC2 418B2A6E 04052780
    7F07D787 5194358F 3C60D665 BE97D79A 950C3434 AED9A06D 2537DC8D 7CDB5969

C : C1099EC7 07B38804 E7442E38 CC8AD853 EB6F58DB 1EA81CBE 73B9D52A 1DE85F1F
    48910C72 893B24F4 C5A0E15E BBC43469 72D2F45C 75941FB6 6D8BE0A0 A1A7524A

W : 00000000 00000000


block number = 0 : permutation - add C to A

A : 20728DFD 46C0BD53 E782B699 55304632 71B4EF90 0EA9E82C DBB930F1 FAD06B8B
    BE0CAE40 8BD14410 76D2ADAC 28ACAB7F

B : D9BF68D1 58BAD750 56028CB2 8134F359 B5D469D8 941A8CC2 418B2A6E 04052780
    7F07D787 5194358F 3C60D665 BE97D79A 950C3434 AED9A06D 2537DC8D 7CDB5969

C : C1099EC7 07B38804 E7442E38 CC8AD853 EB6F58DB 1EA81CBE 73B9D52A 1DE85F1F
    48910C72 893B24F4 C5A0E15E BBC43469 72D2F45C 75941FB6 6D8BE0A0 A1A7524A

W : 00000000 00000000


block number = 0 : subtract M from C

A : 20728DFD 46C0BD53 E782B699 55304632 71B4EF90 0EA9E82C DBB930F1 FAD06B8B
    BE0CAE40 8BD14410 76D2ADAC 28ACAB7F

B : D9BF68D1 58BAD750 56028CB2 8134F359 B5D469D8 941A8CC2 418B2A6E 04052780
```

```
    7F07D787 5194358F 3C60D665 BE97D79A 950C3434 AED9A06D 2537DC8D 7CDB5969

C : C1099CB7 07B385F3 E7442C26 CC8AD640 EB6F56C7 1EA81AA9 73B9D314 1DE85D08
    48910A5A 893B22DB C5A0DF44 BBC4324E 72D2F240 75941D99 6D8BDE82 A1A7502B

W : 00000000 00000000


block number = 0 : swap B with C

A : 20728DFD 46C0BD53 E782B699 55304632 71B4EF90 0EA9E82C DBB930F1 FAD06B8B
    BE0CAE40 8BD14410 76D2ADAC 28ACAB7F

B : C1099CB7 07B385F3 E7442C26 CC8AD640 EB6F56C7 1EA81AA9 73B9D314 1DE85D08
    48910A5A 893B22DB C5A0DF44 BBC4324E 72D2F240 75941D99 6D8BDE82 A1A7502B

C : D9BF68D1 58BAD750 56028CB2 8134F359 B5D469D8 941A8CC2 418B2A6E 04052780
    7F07D787 5194358F 3C60D665 BE97D79A 950C3434 AED9A06D 2537DC8D 7CDB5969

W : 00000000 00000000


block number = 1 : increment counter W

A : 20728DFD 46C0BD53 E782B699 55304632 71B4EF90 0EA9E82C DBB930F1 FAD06B8B
    BE0CAE40 8BD14410 76D2ADAC 28ACAB7F

B : C1099CB7 07B385F3 E7442C26 CC8AD640 EB6F56C7 1EA81AA9 73B9D314 1DE85D08
    48910A5A 893B22DB C5A0DF44 BBC4324E 72D2F240 75941D99 6D8BDE82 A1A7502B

C : D9BF68D1 58BAD750 56028CB2 8134F359 B5D469D8 941A8CC2 418B2A6E 04052780
    7F07D787 5194358F 3C60D665 BE97D79A 950C3434 AED9A06D 2537DC8D 7CDB5969

W : 00000001 00000000


block number = 1 : message block

M : 64636261 68676665 6C6B6A69 706F6E6D 74737271 78777675 302D7A79 34333231
    38373635 42412D39 46454443 4A494847 4E4D4C4B 5251504F 56555453 5A595857


block number = 1 : add M to B

A : 20728DFD 46C0BD53 E782B699 55304632 71B4EF90 0EA9E82C DBB930F1 FAD06B8B
    BE0CAE40 8BD14410 76D2ADAC 28ACAB7F

B : 256CFF18 701AEC58 53AF968F 3CFA44AD 5FE2C938 971F911E A3E74D8D 521B8F39
    80C8408F CB7C5014 0BE62387 060D7A95 C1203E8B C7E56DE8 C3E132D5 FC00A882

C : D9BF68D1 58BAD750 56028CB2 8134F359 B5D469D8 941A8CC2 418B2A6E 04052780
    7F07D787 5194358F 3C60D665 BE97D79A 950C3434 AED9A06D 2537DC8D 7CDB5969

W : 00000001 00000000


block number = 1 : xor counter W into A

A : 20728DFC 46C0BD53 E782B699 55304632 71B4EF90 0EA9E82C DBB930F1 FAD06B8B
    BE0CAE40 8BD14410 76D2ADAC 28ACAB7F

B : 256CFF18 701AEC58 53AF968F 3CFA44AD 5FE2C938 971F911E A3E74D8D 521B8F39
    80C8408F CB7C5014 0BE62387 060D7A95 C1203E8B C7E56DE8 C3E132D5 FC00A882

C : D9BF68D1 58BAD750 56028CB2 8134F359 B5D469D8 941A8CC2 418B2A6E 04052780
    7F07D787 5194358F 3C60D665 BE97D79A 950C3434 AED9A06D 2537DC8D 7CDB5969
```

```
W : 00000001 00000000


block number = 1 : permutation - rotate B

A : 20728DFC 46C0BD53 E782B699 55304632 71B4EF90 0EA9E82C DBB930F1 FAD06B8B
    BE0CAE40 8BD14410 76D2ADAC 28ACAB7F

B : FE304AD9 D8B0E035 2D1EA75F 895A79F4 9270BFC5 223D2E3F 9B1B47CE 1E72A437
    811F0190 A02996F8 470E17CC F52A0C1A 7D178240 DBD18FCA 65AB87C2 5105F801

C : D9BF68D1 58BAD750 56028CB2 8134F359 B5D469D8 941A8CC2 418B2A6E 04052780
    7F07D787 5194358F 3C60D665 BE97D79A 950C3434 AED9A06D 2537DC8D 7CDB5969

W : 00000001 00000000


block number = 1 : permutation (j = 0)

A : 521A8160 F2C7A2D8 19943F75 98C8339B 765B34C2 4F6BDE89 F053F1FC E6C80AE5
    2B7E001E 1CF7FF61 CBF8D1B7 0E09704D

B : 475728A8 B3A35E8E 92EC9E80 A5B803DA AD45B4B6 F4EE7D08 399A819E 25D2BD74
    D6BFFCC0 A35B2D6F BA1B01D0 1BA29787 57CA7A1F BA9B42B2 2D3CCF0E C53C3C66

C : D9BF68D1 58BAD750 56028CB2 8134F359 B5D469D8 941A8CC2 418B2A6E 04052780
    7F07D787 5194358F 3C60D665 BE97D79A 950C3434 AED9A06D 2537DC8D 7CDB5969

W : 00000001 00000000


block number = 1 : permutation (j = 1)

A : 07350D5E 86098A1D D0B29046 34CE1DA3 37BD68F0 C7F82F4B A6EDC376 F2200D32
    D0A7263D 5DF27B32 C4642766 D919D727

B : B4E0FE0E 4F042E9A C695D5BA 7E6A7EF7 75D3B0AF 4BD17EDC 48AEDBA5 6D435230
    55B50B20 3F402F3D 5B7B6C18 FC74CD52 67D66331 4D3155D1 036BA295 87A78A00

C : D9BF68D1 58BAD750 56028CB2 8134F359 B5D469D8 941A8CC2 418B2A6E 04052780
    7F07D787 5194358F 3C60D665 BE97D79A 950C3434 AED9A06D 2537DC8D 7CDB5969

W : 00000001 00000000


block number = 1 : permutation (j = 2)

A : ADA9B505 9CF7B61A 3ED16596 9B4A1C40 E9734A02 BC4E95ED EE66F036 DA2C1B9A
    8352CCD2 E5870916 50B25176 1624D3E7

B : 263316F4 DF2CE437 2C9200CA CB0A2A56 B9F12BA4 F4AAB45D 50732D23 BE3347DF
    BDE6A3BD 3D313468 A76FD7F9 DD3A7EC0 B301F54F 801A5D4B A99AEBA3 E6943819

C : D9BF68D1 58BAD750 56028CB2 8134F359 B5D469D8 941A8CC2 418B2A6E 04052780
    7F07D787 5194358F 3C60D665 BE97D79A 950C3434 AED9A06D 2537DC8D 7CDB5969

W : 00000001 00000000


block number = 1 : permutation - add C to A

A : 6A51D961 C197BCF7 DA806A15 580FAFED EB88BE44 CAEA401D 2CD089D7 AE1AA91F
    C724BF45 AF6F7EA9 E53AB434 B948B147

B : 263316F4 DF2CE437 2C9200CA CB0A2A56 B9F12BA4 F4AAB45D 50732D23 BE3347DF
    BDE6A3BD 3D313468 A76FD7F9 DD3A7EC0 B301F54F 801A5D4B A99AEBA3 E6943819
```

```
C : D9BF68D1 58BAD750 56028CB2 8134F359 B5D469D8 941A8CC2 418B2A6E 04052780
    7F07D787 5194358F 3C60D665 BE97D79A 950C3434 AED9A06D 2537DC8D 7CDB5969

W : 00000001 00000000


block number = 1 : subtract M from C

A : 6A51D961 C197BCF7 DA806A15 580FAFED EB88BE44 CAEA401D 2CD089D7 AE1AA91F
    C724BF45 AF6F7EA9 E53AB434 B948B147

B : 263316F4 DF2CE437 2C9200CA CB0A2A56 B9F12BA4 F4AAB45D 50732D23 BE3347DF
    BDE6A3BD 3D313468 A76FD7F9 DD3A7EC0 B301F54F 801A5D4B A99AEBA3 E6943819

C : 755C0670 F05370EB E9972249 10C584EC 4160F767 1BA3164D 115DAFF5 CFD1F54F
    46D0A152 0F530856 F61B9222 744E8F53 46BEE7E9 5C88501E CEE2883A 22820112

W : 00000001 00000000


block number = 1 : swap B with C

A : 6A51D961 C197BCF7 DA806A15 580FAFED EB88BE44 CAEA401D 2CD089D7 AE1AA91F
    C724BF45 AF6F7EA9 E53AB434 B948B147

B : 755C0670 F05370EB E9972249 10C584EC 4160F767 1BA3164D 115DAFF5 CFD1F54F
    46D0A152 0F530856 F61B9222 744E8F53 46BEE7E9 5C88501E CEE2883A 22820112

C : 263316F4 DF2CE437 2C9200CA CB0A2A56 B9F12BA4 F4AAB45D 50732D23 BE3347DF
    BDE6A3BD 3D313468 A76FD7F9 DD3A7EC0 B301F54F 801A5D4B A99AEBA3 E6943819

W : 00000001 00000000


block number = 2 : increment counter W

A : 6A51D961 C197BCF7 DA806A15 580FAFED EB88BE44 CAEA401D 2CD089D7 AE1AA91F
    C724BF45 AF6F7EA9 E53AB434 B948B147

B : 755C0670 F05370EB E9972249 10C584EC 4160F767 1BA3164D 115DAFF5 CFD1F54F
    46D0A152 0F530856 F61B9222 744E8F53 46BEE7E9 5C88501E CEE2883A 22820112

C : 263316F4 DF2CE437 2C9200CA CB0A2A56 B9F12BA4 F4AAB45D 50732D23 BE3347DF
    BDE6A3BD 3D313468 A76FD7F9 DD3A7EC0 B301F54F 801A5D4B A99AEBA3 E6943819

W : 00000002 00000000


block number = 2 : message block

M : 3231302D 36353433 2D393837 64636261 68676665 6C6B6A69 706F6E6D 74737271
    78777675 00807A79 00000000 00000000 00000000 00000000 00000000 00000000


block number = 2 : add M to B

A : 6A51D961 C197BCF7 DA806A15 580FAFED EB88BE44 CAEA401D 2CD089D7 AE1AA91F
    C724BF45 AF6F7EA9 E53AB434 B948B147

B : A78D369D 2688A51E 16D05A80 7528E74D A9C85DCC 880E80B6 81CD1E62 444567C0
    BF4817C7 0FD382CF F61B9222 744E8F53 46BEE7E9 5C88501E CEE2883A 22820112

C : 263316F4 DF2CE437 2C9200CA CB0A2A56 B9F12BA4 F4AAB45D 50732D23 BE3347DF
    BDE6A3BD 3D313468 A76FD7F9 DD3A7EC0 B301F54F 801A5D4B A99AEBA3 E6943819

W : 00000002 00000000
```

```
block number = 2 : xor counter W into A

A : 6A51D963 C197BCF7 DA806A15 580FAFED EB88BE44 CAEA401D 2CD089D7 AE1AA91F
    C724BF45 AF6F7EA9 E53AB434 B948B147

B : A78D369D 2688A51E 16D05A80 7528E74D A9C85DCC 880E80B6 81CD1E62 444567C0
    BF4817C7 0FD382CF F61B9222 744E8F53 46BEE7E9 5C88501E CEE2883A 22820112

C : 263316F4 DF2CE437 2C9200CA CB0A2A56 B9F12BA4 F4AAB45D 50732D23 BE3347DF
    BDE6A3BD 3D313468 A76FD7F9 DD3A7EC0 B301F54F 801A5D4B A99AEBA3 E6943819

W : 00000002 00000000


block number = 2 : permutation - rotate B

A : 6A51D963 C197BCF7 DA806A15 580FAFED EB88BE44 CAEA401D 2CD089D7 AE1AA91F
    C724BF45 AF6F7EA9 E53AB434 B948B147

B : 6D3B4F1A 4A3C4D11 B5002DA0 CE9AEA51 BB995390 016D101D 3CC5039A CF80888A
    2F8F7E90 059E1FA7 2445EC37 1EA6E89D CFD28D7D A03CB910 10759DC5 02244504

C : 263316F4 DF2CE437 2C9200CA CB0A2A56 B9F12BA4 F4AAB45D 50732D23 BE3347DF
    BDE6A3BD 3D313468 A76FD7F9 DD3A7EC0 B301F54F 801A5D4B A99AEBA3 E6943819

W : 00000002 00000000


block number = 2 : permutation (j = 0)

A : 9C514AC4 3BADF179 BB75BFFD 5FB5FD4B E666B91F 318CA9B9 6D877BB1 05F34168
    FAE6F46D 5BD6CB26 479B8912 F3D10A95

B : F312556D 199302E5 3039578C E5E90D95 6EABE1C1 CCA9767C EBF2837A 650DAF82
    5A07F6B2 AF150B97 F0EFAE83 31632450 FC0BAFC0 842B7CA7 64617B88 A40288BC

C : 263316F4 DF2CE437 2C9200CA CB0A2A56 B9F12BA4 F4AAB45D 50732D23 BE3347DF
    BDE6A3BD 3D313468 A76FD7F9 DD3A7EC0 B301F54F 801A5D4B A99AEBA3 E6943819

W : 00000002 00000000


block number = 2 : permutation (j = 1)

A : 854479F2 8F09B4AA CD8F4966 7337EA94 6ED28CCD F29CF0D4 82AFF90C 1F3B6E07
    5740488B 4D92D5CC 7F5CEE46 3FA274C3

B : 43F1708B 7AF318A2 157BD5DD 77565888 75E874F6 2B3FC6CA 5746174C 0A46D438
    CEB46B69 2EDC5C7A D3AFEB9E EE0E5DCB 693A2CB3 0535F664 B592F1E3 A8C18081

C : 263316F4 DF2CE437 2C9200CA CB0A2A56 B9F12BA4 F4AAB45D 50732D23 BE3347DF
    BDE6A3BD 3D313468 A76FD7F9 DD3A7EC0 B301F54F 801A5D4B A99AEBA3 E6943819

W : 00000002 00000000


block number = 2 : permutation (j = 2)

A : 4C5EE590 B0B92F77 4905E370 1B09A00A 61A1A07E 176DFB83 9BA40FFE 69DFDBD3
    806B3232 D77D6038 0D08CEEC 290A0663

B : 0AA4FFE7 9B078670 A083857B FE885C3D 5871F383 19395D1C 18763217 F07BF785
    03368952 B52ABC88 C304273C 4A3C9FBB ADE094AB 22E9730F 99D2D2D4 8776F89F

C : 263316F4 DF2CE437 2C9200CA CB0A2A56 B9F12BA4 F4AAB45D 50732D23 BE3347DF
    BDE6A3BD 3D313468 A76FD7F9 DD3A7EC0 B301F54F 801A5D4B A99AEBA3 E6943819
```

```
W : 00000002 00000000


block number = 2 : permutation - add C to A

A : DB37C6BF 43DF675E 9CF7D94F 41A9B99A D1734ACC B578E1D8 ACACDCFA 8E54E1B9
    E6E32327 025724E8 BEFF14FC CA87F722

B : 0AA4FFE7 9B078670 A083857B FE885C3D 5871F383 19395D1C 18763217 F07BF785
    03368952 B52ABC88 C304273C 4A3C9FBB ADE094AB 22E9730F 99D2D2D4 8776F89F

C : 263316F4 DF2CE437 2C9200CA CB0A2A56 B9F12BA4 F4AAB45D 50732D23 BE3347DF
    BDE6A3BD 3D313468 A76FD7F9 DD3A7EC0 B301F54F 801A5D4B A99AEBA3 E6943819

W : 00000002 00000000


block number = 2 : subtract M from C

A : DB37C6BF 43DF675E 9CF7D94F 41A9B99A D1734ACC B578E1D8 ACACDCFA 8E54E1B9
    E6E32327 025724E8 BEFF14FC CA87F722

B : 0AA4FFE7 9B078670 A083857B FE885C3D 5871F383 19395D1C 18763217 F07BF785
    03368952 B52ABC88 C304273C 4A3C9FBB ADE094AB 22E9730F 99D2D2D4 8776F89F

C : F401E6C7 A8F7B004 FF58C893 66A6C7F5 5189C53F 883F49F4 E003BEB6 49BFD56E
    456F2D48 3CB0B9EF A76FD7F9 DD3A7EC0 B301F54F 801A5D4B A99AEBA3 E6943819

W : 00000002 00000000


block number = 2 : swap B with C

A : DB37C6BF 43DF675E 9CF7D94F 41A9B99A D1734ACC B578E1D8 ACACDCFA 8E54E1B9
    E6E32327 025724E8 BEFF14FC CA87F722

B : F401E6C7 A8F7B004 FF58C893 66A6C7F5 5189C53F 883F49F4 E003BEB6 49BFD56E
    456F2D48 3CB0B9EF A76FD7F9 DD3A7EC0 B301F54F 801A5D4B A99AEBA3 E6943819

C : 0AA4FFE7 9B078670 A083857B FE885C3D 5871F383 19395D1C 18763217 F07BF785
    03368952 B52ABC88 C304273C 4A3C9FBB ADE094AB 22E9730F 99D2D2D4 8776F89F

W : 00000002 00000000


n0_final = 0 : add M to B

A : DB37C6BF 43DF675E 9CF7D94F 41A9B99A D1734ACC B578E1D8 ACACDCFA 8E54E1B9
    E6E32327 025724E8 BEFF14FC CA87F722

B : 263316F4 DF2CE437 2C9200CA CB0A2A56 B9F12BA4 F4AAB45D 50732D23 BE3347DF
    BDE6A3BD 3D313468 A76FD7F9 DD3A7EC0 B301F54F 801A5D4B A99AEBA3 E6943819

C : 0AA4FFE7 9B078670 A083857B FE885C3D 5871F383 19395D1C 18763217 F07BF785
    03368952 B52ABC88 C304273C 4A3C9FBB ADE094AB 22E9730F 99D2D2D4 8776F89F

W : 00000002 00000000


n0_final = 0 : xor counter W into A

A : DB37C6BD 43DF675E 9CF7D94F 41A9B99A D1734ACC B578E1D8 ACACDCFA 8E54E1B9
    E6E32327 025724E8 BEFF14FC CA87F722

B : 263316F4 DF2CE437 2C9200CA CB0A2A56 B9F12BA4 F4AAB45D 50732D23 BE3347DF
    BDE6A3BD 3D313468 A76FD7F9 DD3A7EC0 B301F54F 801A5D4B A99AEBA3 E6943819
```

```
C : 0AA4FFE7 9B078670 A083857B FE885C3D 5871F383 19395D1C 18763217 F07BF785
    03368952 B52ABC88 C304273C 4A3C9FBB ADE094AB 22E9730F 99D2D2D4 8776F89F

W : 00000002 00000000


n0_final = 0 : permutation - rotate B

A : DB37C6BD 43DF675E 9CF7D94F 41A9B99A D1734ACC B578E1D8 ACACDCFA 8E54E1B9
    E6E32327 025724E8 BEFF14FC CA87F722

B : 2DE84C66 C86FBE59 01945924 54AD9614 574973E2 68BBE955 5A46A0E6 8FBF7C66
    477B7BCD 68D07A62 AFF34EDF FD81BA74 EA9F6603 BA970034 D7475335 7033CD28

C : 0AA4FFE7 9B078670 A083857B FE885C3D 5871F383 19395D1C 18763217 F07BF785
    03368952 B52ABC88 C304273C 4A3C9FBB ADE094AB 22E9730F 99D2D2D4 8776F89F

W : 00000002 00000000


n0_final = 0 : permutation (j = 0)

A : 007E853B BCF7132E 2EC6EDCC 6A95766A CCBF887C 97B7F87A 64D079C4 87051CBF
    63B380CE FF88C2D0 01B3E368 6987989E

B : 999E2DCA 3470467A 25CF15A8 A7C8F6D3 9DD29047 B93FD52F 2FA2C7F7 67841B8D
    12BA88AB D1D7C9EB A1AA8128 6D7B1388 2ABFB6C3 3626ECB8 7FB7B458 750D13C5

C : 0AA4FFE7 9B078670 A083857B FE885C3D 5871F383 19395D1C 18763217 F07BF785
    03368952 B52ABC88 C304273C 4A3C9FBB ADE094AB 22E9730F 99D2D2D4 8776F89F

W : 00000002 00000000


n0_final = 0 : permutation (j = 1)

A : 56AD8642 4F53444B 300FBED9 C2D9B8B8 17B6A359 BDFE6C80 C448E63E ABA7A6C3
    6A8DE8A1 80AF4351 B1F5774E D8E1B1C7

B : 2E0852E3 F0095852 B3F9B371 08E8EEC7 AED737D1 0D2F16F1 114F075F E8167922
    8C2768EB 13032863 8CA54377 E7D06057 BD363120 2E4C4A0F C4D87171 BE427EB6

C : 0AA4FFE7 9B078670 A083857B FE885C3D 5871F383 19395D1C 18763217 F07BF785
    03368952 B52ABC88 C304273C 4A3C9FBB ADE094AB 22E9730F 99D2D2D4 8776F89F

W : 00000002 00000000


n0_final = 0 : permutation (j = 2)

A : 463B12A4 9EEF72EE 84CA8F14 65710BEA B1AF37FE CC7E7CA8 ED166423 8B798937
    7BA4A533 237B5AC1 724F6BEE D51A399B

B : 6664EEFF 70CC036A AF492E12 D96F8B3C E46A82F8 7B4EA0F3 59AB7E55 4AA20650
    561E19D6 1587D391 0BA31D33 BB26B667 FE37388D 801C3120 040076F2 56613B09

C : 0AA4FFE7 9B078670 A083857B FE885C3D 5871F383 19395D1C 18763217 F07BF785
    03368952 B52ABC88 C304273C 4A3C9FBB ADE094AB 22E9730F 99D2D2D4 8776F89F

W : 00000002 00000000


n0_final = 0 : permutation - add C to A

A : 1677073B AFE6FB03 5BF4E5AF B83D9650 282A845F 32CBF964 56820437 07776805
    B4E598B0 2D046C41 639CF8A1 4A6765C2
```

```
B : 6664EEFF 70CC036A AF492E12 D96F8B3C E46A82F8 7B4EA0F3 59AB7E55 4AA20650
    561E19D6 1587D391 0BA31D33 BB26B667 FE37388D 801C3120 040076F2 56613B09

C : 0AA4FFE7 9B078670 A083857B FE885C3D 5871F383 19395D1C 18763217 F07BF785
    03368952 B52ABC88 C304273C 4A3C9FBB ADE094AB 22E9730F 99D2D2D4 8776F89F

W : 00000002 00000000


n0_final = 0 : subtract M from C

A : 1677073B AFE6FB03 5BF4E5AF B83D9650 282A845F 32CBF964 56820437 07776805
    B4E598B0 2D046C41 639CF8A1 4A6765C2

B : 6664EEFF 70CC036A AF492E12 D96F8B3C E46A82F8 7B4EA0F3 59AB7E55 4AA20650
    561E19D6 1587D391 0BA31D33 BB26B667 FE37388D 801C3120 040076F2 56613B09

C : D873CFBA 64D2523D 734A4D44 9A24F9DC F00A8D1E ACCDF2B3 A806C3AA 7C088514
    8ABF12DD B4AA420F C304273C 4A3C9FBB ADE094AB 22E9730F 99D2D2D4 8776F89F

W : 00000002 00000000


n0_final = 0 : swap B with C

A : 1677073B AFE6FB03 5BF4E5AF B83D9650 282A845F 32CBF964 56820437 07776805
    B4E598B0 2D046C41 639CF8A1 4A6765C2

B : D873CFBA 64D2523D 734A4D44 9A24F9DC F00A8D1E ACCDF2B3 A806C3AA 7C088514
    8ABF12DD B4AA420F C304273C 4A3C9FBB ADE094AB 22E9730F 99D2D2D4 8776F89F

C : 6664EEFF 70CC036A AF492E12 D96F8B3C E46A82F8 7B4EA0F3 59AB7E55 4AA20650
    561E19D6 1587D391 0BA31D33 BB26B667 FE37388D 801C3120 040076F2 56613B09

W : 00000002 00000000


n0_final = 1 : add M to B

A : 1677073B AFE6FB03 5BF4E5AF B83D9650 282A845F 32CBF964 56820437 07776805
    B4E598B0 2D046C41 639CF8A1 4A6765C2

B : 0AA4FFE7 9B078670 A083857B FE885C3D 5871F383 19395D1C 18763217 F07BF785
    03368952 B52ABC88 C304273C 4A3C9FBB ADE094AB 22E9730F 99D2D2D4 8776F89F

C : 6664EEFF 70CC036A AF492E12 D96F8B3C E46A82F8 7B4EA0F3 59AB7E55 4AA20650
    561E19D6 1587D391 0BA31D33 BB26B667 FE37388D 801C3120 040076F2 56613B09

W : 00000002 00000000


n0_final = 1 : xor counter W into A

A : 16770739 AFE6FB03 5BF4E5AF B83D9650 282A845F 32CBF964 56820437 07776805
    B4E598B0 2D046C41 639CF8A1 4A6765C2

B : 0AA4FFE7 9B078670 A083857B FE885C3D 5871F383 19395D1C 18763217 F07BF785
    03368952 B52ABC88 C304273C 4A3C9FBB ADE094AB 22E9730F 99D2D2D4 8776F89F

C : 6664EEFF 70CC036A AF492E12 D96F8B3C E46A82F8 7B4EA0F3 59AB7E55 4AA20650
    561E19D6 1587D391 0BA31D33 BB26B667 FE37388D 801C3120 040076F2 56613B09

W : 00000002 00000000


n0_final = 1 : permutation - rotate B
```

```
A : 16770739 AFE6FB03 5BF4E5AF B83D9650 282A845F 32CBF964 56820437 07776805
    B4E598B0 2D046C41 639CF8A1 4A6765C2

B : FFCE1549 0CE1360F 0AF74107 B87BFD10 E706B0E3 BA383272 642E30EC EF0BE0F7
    12A4066D 79116A55 4E798608 3F769479 29575BC1 E61E45D2 A5A933A5 F13F0EED

C : 6664EEFF 70CC036A AF492E12 D96F8B3C E46A82F8 7B4EA0F3 59AB7E55 4AA20650
    561E19D6 1587D391 0BA31D33 BB26B667 FE37388D 801C3120 040076F2 56613B09

W : 00000002 00000000


n0_final = 1 : permutation (j = 0)

A : 584547D8 E4ABC4A4 1B18E9F8 DBF7B600 C3F3A594 8810C99E 9D3395A0 1212F94B
    489A7A9F 711209B5 447379E5 B27EE663

B : 77781FB2 AEEBEFC9 87F452FC 382285C6 F2013BAC 039F5284 AA900B87 33FAC75B
    922D89BA 7CCF22E0 277F8A0A 336C316E F5140FA5 D768B0FE AFB5714C C6765424

C : 6664EEFF 70CC036A AF492E12 D96F8B3C E46A82F8 7B4EA0F3 59AB7E55 4AA20650
    561E19D6 1587D391 0BA31D33 BB26B667 FE37388D 801C3120 040076F2 56613B09

W : 00000002 00000000


n0_final = 1 : permutation (j = 1)

A : 8D8AFAD0 08370C73 EC83D0D7 A7D3D97D F752013A BA0F211B 940FB1EF D7DEDC36
    9D82DEEF D0DD8110 DE2A4F3F F49CC319

B : D8D7ACC7 5B52C8E5 BF580A9D D0966821 867F5649 281CDBE7 74F5A7CF 6C96B250
    562E165A 0E56B64C 5D833B3C 3EF4445E E285E18E EB21BF19 349AAC89 A4CD8B80

C : 6664EEFF 70CC036A AF492E12 D96F8B3C E46A82F8 7B4EA0F3 59AB7E55 4AA20650
    561E19D6 1587D391 0BA31D33 BB26B667 FE37388D 801C3120 040076F2 56613B09

W : 00000002 00000000


n0_final = 1 : permutation (j = 2)

A : D5BE9831 F3143340 7EE33692 DC9B0CA8 455306CC 05D59A83 4DED5952 2BE65A56
    815F2F95 3DFF9C03 4F46B77F F84EB239

B : 5A03ACF2 5A44DF04 9819C294 417DB69A 26BFCB5D 5CD27B71 68F786F3 FA4997F7
    16F0D587 E68709E4 0914D0D5 A9F12D15 BBAB1377 14431DCF D98C1192 4E2A5AC7

C : 6664EEFF 70CC036A AF492E12 D96F8B3C E46A82F8 7B4EA0F3 59AB7E55 4AA20650
    561E19D6 1587D391 0BA31D33 BB26B667 FE37388D 801C3120 040076F2 56613B09

W : 00000002 00000000


n0_final = 1 : permutation - add C to A

A : C0B614DD 3C1ADDC4 EB1A0C0F E9903001 BFC5D361 A6C32650 4F8FD140 407E23F0
    60977788 76BF715E 60395D23 619DC4B3

B : 5A03ACF2 5A44DF04 9819C294 417DB69A 26BFCB5D 5CD27B71 68F786F3 FA4997F7
    16F0D587 E68709E4 0914D0D5 A9F12D15 BBAB1377 14431DCF D98C1192 4E2A5AC7

C : 6664EEFF 70CC036A AF492E12 D96F8B3C E46A82F8 7B4EA0F3 59AB7E55 4AA20650
    561E19D6 1587D391 0BA31D33 BB26B667 FE37388D 801C3120 040076F2 56613B09

W : 00000002 00000000
```

```
n0_final = 1 : subtract M from C

A : C0B614DD 3C1ADDC4 EB1A0C0F E9903001 BFC5D361 A6C32650 4F8FD140 407E23F0
    60977788 76BF715E 60395D23 619DC4B3

B : 5A03ACF2 5A44DF04 9819C294 417DB69A 26BFCB5D 5CD27B71 68F786F3 FA4997F7
    16F0D587 E68709E4 0914D0D5 A9F12D15 BBAB1377 14431DCF D98C1192 4E2A5AC7

C : 3433BED2 3A96CF37 820FF5DB 750C28DB 7C031C93 0EE3368A E93C0FE8 D62E93DF
    DDA6A361 15075918 0BA31D33 BB26B667 FE37388D 801C3120 040076F2 56613B09

W : 00000002 00000000


n0_final = 1 : swap B with C

A : C0B614DD 3C1ADDC4 EB1A0C0F E9903001 BFC5D361 A6C32650 4F8FD140 407E23F0
    60977788 76BF715E 60395D23 619DC4B3

B : 3433BED2 3A96CF37 820FF5DB 750C28DB 7C031C93 0EE3368A E93C0FE8 D62E93DF
    DDA6A361 15075918 0BA31D33 BB26B667 FE37388D 801C3120 040076F2 56613B09

C : 5A03ACF2 5A44DF04 9819C294 417DB69A 26BFCB5D 5CD27B71 68F786F3 FA4997F7
    16F0D587 E68709E4 0914D0D5 A9F12D15 BBAB1377 14431DCF D98C1192 4E2A5AC7

W : 00000002 00000000


n0_final = 2 : add M to B

A : C0B614DD 3C1ADDC4 EB1A0C0F E9903001 BFC5D361 A6C32650 4F8FD140 407E23F0
    60977788 76BF715E 60395D23 619DC4B3

B : 6664EEFF 70CC036A AF492E12 D96F8B3C E46A82F8 7B4EA0F3 59AB7E55 4AA20650
    561E19D6 1587D391 0BA31D33 BB26B667 FE37388D 801C3120 040076F2 56613B09

C : 5A03ACF2 5A44DF04 9819C294 417DB69A 26BFCB5D 5CD27B71 68F786F3 FA4997F7
    16F0D587 E68709E4 0914D0D5 A9F12D15 BBAB1377 14431DCF D98C1192 4E2A5AC7

W : 00000002 00000000


n0_final = 2 : xor counter W into A

A : C0B614DF 3C1ADDC4 EB1A0C0F E9903001 BFC5D361 A6C32650 4F8FD140 407E23F0
    60977788 76BF715E 60395D23 619DC4B3

B : 6664EEFF 70CC036A AF492E12 D96F8B3C E46A82F8 7B4EA0F3 59AB7E55 4AA20650
    561E19D6 1587D391 0BA31D33 BB26B667 FE37388D 801C3120 040076F2 56613B09

C : 5A03ACF2 5A44DF04 9819C294 417DB69A 26BFCB5D 5CD27B71 68F786F3 FA4997F7
    16F0D587 E68709E4 0914D0D5 A9F12D15 BBAB1377 14431DCF D98C1192 4E2A5AC7

W : 00000002 00000000


n0_final = 2 : permutation - rotate B

A : C0B614DF 3C1ADDC4 EB1A0C0F E9903001 BFC5D361 A6C32650 4F8FD140 407E23F0
    60977788 76BF715E 60395D23 619DC4B3

B : DDFECCC9 06D4E198 5C255E92 1679B2DF 05F1C8D5 41E6F69D FCAAB356 0CA09544
    33ACAC3C A7222B0F 3A661746 6CCF764D 711BFC6E 62410038 EDE40800 7612ACC2

C : 5A03ACF2 5A44DF04 9819C294 417DB69A 26BFCB5D 5CD27B71 68F786F3 FA4997F7
```

```
    16F0D587 E68709E4 0914D0D5 A9F12D15 BBAB1377 14431DCF D98C1192 4E2A5AC7

W : 00000002 00000000


n0_final = 2 : permutation (j = 0)

A : 25BB41ED 030944B0 7931C842 0EE6CB55 3C62EA04 E3473D7E EFF28F08 CEAD8919
    33F792D7 6F755271 1F07D243 4A4C0B99

B : 2FE67E6A F508D465 02104DAA F31C6214 C87E8451 9F752FBB E958165A 28135C6E
    AB513550 DECEFB91 94340330 6C2D18FC 387346CE 3874BB3F 5D0627BC 1D3C6D2E

C : 5A03ACF2 5A44DF04 9819C294 417DB69A 26BFCB5D 5CD27B71 68F786F3 FA4997F7
    16F0D587 E68709E4 0914D0D5 A9F12D15 BBAB1377 14431DCF D98C1192 4E2A5AC7

W : 00000002 00000000


n0_final = 2 : permutation (j = 1)

A : F68087D5 875B00CB 60A8CE06 512155E1 E2935957 1B3E03D6 3C0BA829 2353F2E5
    3888C584 8EE110C8 56764BCD 2BE6E607

B : 38A3B18C 1DFC4008 B8CF609C AFD76D35 578A32D8 4FF4B040 7B399887 843FA124
    5FDD128B C5390817 B73F3798 76849BE6 6D8A2B34 94288A57 79F818AE E6D4D746

C : 5A03ACF2 5A44DF04 9819C294 417DB69A 26BFCB5D 5CD27B71 68F786F3 FA4997F7
    16F0D587 E68709E4 0914D0D5 A9F12D15 BBAB1377 14431DCF D98C1192 4E2A5AC7

W : 00000002 00000000


n0_final = 2 : permutation (j = 2)

A : 36D5FFBC E625E21B 4A8D142E 5248F193 D464E432 04E263D6 F1FAED97 A6D485AB
    FD8BB6ED 1C71434C B56A93B4 FD980801

B : 7F6F7E67 F00AD712 2F6635B3 5168B559 663E65F3 86337D64 4301DADF A5C84C25
    94213EDB 716F8C06 607B7D59 B4224D98 D9601F7A CBDFA81C B9655D17 CFCE5973

C : 5A03ACF2 5A44DF04 9819C294 417DB69A 26BFCB5D 5CD27B71 68F786F3 FA4997F7
    16F0D587 E68709E4 0914D0D5 A9F12D15 BBAB1377 14431DCF D98C1192 4E2A5AC7

W : 00000002 00000000


n0_final = 2 : permutation - add C to A

A : 706F3E32 22946DE1 15E78C72 2CE64CAC 5E568D8A 9C96B1AC 8F9951F0 B0FAA007
    E3443293 15CCF7A7 0D0736D8 4930715B

B : 7F6F7E67 F00AD712 2F6635B3 5168B559 663E65F3 86337D64 4301DADF A5C84C25
    94213EDB 716F8C06 607B7D59 B4224D98 D9601F7A CBDFA81C B9655D17 CFCE5973

C : 5A03ACF2 5A44DF04 9819C294 417DB69A 26BFCB5D 5CD27B71 68F786F3 FA4997F7
    16F0D587 E68709E4 0914D0D5 A9F12D15 BBAB1377 14431DCF D98C1192 4E2A5AC7

W : 00000002 00000000


n0_final = 2 : subtract M from C

A : 706F3E32 22946DE1 15E78C72 2CE64CAC 5E568D8A 9C96B1AC 8F9951F0 B0FAA007
    E3443293 15CCF7A7 0D0736D8 4930715B

B : 7F6F7E67 F00AD712 2F6635B3 5168B559 663E65F3 86337D64 4301DADF A5C84C25
```

```
    94213EDB 716F8C06 607B7D59 B4224D98 D9601F7A CBDFA81C B9655D17 CFCE5973

C : 27D27CC5 240FAAD1 6AE08A5D DD1A5439 BE5864F8 F0671108 F8881886 85D62586
    9E795F12 E6068F6B 0914D0D5 A9F12D15 BBAB1377 14431DCF D98C1192 4E2A5AC7

W : 00000002 00000000


n0_final = 2 : swap B with C  (final state)

A : 706F3E32 22946DE1 15E78C72 2CE64CAC 5E568D8A 9C96B1AC 8F9951F0 B0FAA007
    E3443293 15CCF7A7 0D0736D8 4930715B

B : 27D27CC5 240FAAD1 6AE08A5D DD1A5439 BE5864F8 F0671108 F8881886 85D62586
    9E795F12 E6068F6B 0914D0D5 A9F12D15 BBAB1377 14431DCF D98C1192 4E2A5AC7

C : 7F6F7E67 F00AD712 2F6635B3 5168B559 663E65F3 86337D64 4301DADF A5C84C25
    94213EDB 716F8C06 607B7D59 B4224D98 D9601F7A CBDFA81C B9655D17 CFCE5973

W : 00000002 00000000


Hash value :

H : 7F6F7E67 F00AD712 2F6635B3 5168B559 663E65F3 86337D64 4301DADF A5C84C25
    94213EDB 716F8C06 607B7D59 B4224D98 D9601F7A CBDFA81C B9655D17 CFCE5973


Hash value (byte array):

H : 67 7E 6F 7F 12 D7 0A F0 B3 35 66 2F 59 B5 68 51
    F3 65 3E 66 64 7D 33 86 DF DA 01 43 25 4C C8 A5
    DB 3E 21 94 06 8C 6F 71 59 7D 7B 60 98 4D 22 B4
    7A 1F 60 D9 1C A8 DF CB 17 5D 65 B9 73 59 CE CF
```