Mystrium: Wide Block Encryption Efficient on Entry-Level Processors

Parisa Amiri Eliasi[1][0009−0003−2881−8314], Koustabh Ghosh[1][0000−0002−1820−2247], and Joan Daemen[1][0000−0002−4102−0775]

Radboud University, Nijmegen, The Netherlands
{parisa.amirieliasi,koustabh.ghosh,joan.daemen}@ru.nl

Abstract. We present a tweakable wide block cipher called Mystrium and show it as the fastest such primitive on low-end processors that lack dedicated AES or other cryptographic instructions, such as ARM Cortex-A7. Mystrium is based on the provably secure double-decker mode, that requires a doubly extendable cryptographic keyed (deck) function and a universal hash function. We build a new deck function called Xymmer that for its compression part uses Multimixer-128, the fastest universal hash for such processors, and for its expansion part uses a newly designed permutation, $\mathcal{G}_{512}$. Deck functions can also be used in modes to build encryption, authenticated encryption, and authentication schemes, and hence, Xymmer is of independent interest. The current state-of-the-art wide tweakable block cipher Adiantum-XChaCha12-AES encrypts 4096-byte messages at 11.5 cycles per byte on ARM Cortex-A7, while for Mystrium it is 6.8 cycles per byte while having a higher claimed security.

Keywords: tweakable wide block cipher · deck function · permutation-based cryptography · disk encryption.

1 Introduction

Historically block ciphers have been the main building block in symmetric key cryptography. Since block ciphers can only operate on inputs of fixed length, they are used in conjunction with a mode of operation to deal with inputs of variable length. In many modes of operation, security relies on the uniqueness of a data element called a *nonce*, typically instantiated by a counter or a freshly generated random value.

However, some practical scenarios necessitate achieving security even in the absence or misuse of a nonce. Notable examples include disk encryption, encryption in the Tor anonymity network [1], or in TLS and VPN communications. Moreover, cloud storage services, relying on nonces for data-at-rest encryption, face similar threats, potentially leading to unauthorized data exposure.

In disk encryption, additional computational demands can significantly degrade user experience and deplete battery life on mobile devices. But, achieving good performance along with security in case of nonce absence or misuse is hard to reach by leveraging traditional block ciphers and their modes. In this paper, we propose a tweakable wide block cipher [24] aimed at such use cases.

Related Works Some of the earliest examples of variable input-length supporting and length-preserving cryptographic primitives include the Hasty Pudding cipher [29] and the Mercy cipher [9]. In these designs a key innovation was the idea of *spice*, which later came to be known as *tweak*, and such ciphers are known as *tweakable wide block ciphers*. Following the results of [25], a very common approach to build a tweakable wide block cipher is to make use of an unbalanced Feistel network. Some notable examples of such wide block cipher modes include XCB [26], HEH [28], EME [20], HCTR [30], HCH [8], HSE [27], AEZ [21], FAST [7] and tweakable HCTR [14].

One recent proposal for such a tweakable wide block cipher is Adiantum [10]. Adiantum is meant for disk encryption on devices with a CPU that does not have dedicated AES instruction, such as those used in low-end smartphones.

Adiantum achieves high speed and security by using a mode HBSH (Hash, Block cipher, Stream cipher, Hash) that the authors proposed in the same paper. The HBSH mode is an unbalanced Feistel with two branches: an arbitrary length branch and a fixed length branch. It requires two calls to a universal hash function, one call to a stream cipher, and one call to a block cipher on the narrow part of the Feistel. In this mode, the bulk of the workload is processed by the universal hash functions and the underlying stream cipher. Adiantum is an instantiation of this mode where the designers use a combination of NH^T [6], Poly-1305 [2] for universal hashing, XChaCha12 [3] for the stream cipher, and AES [13] for the single block cipher application. This makes Adiantum as far as we know the most efficient tweakable wide block cipher on its target platform [10].

However, HBSH comes with a drawback: It requires three different types of cryptographic primitives. The requirement for a block cipher, even though it is only invoked once, is specially undesirable since the target platforms for Adiantum do not have any cryptographic instructions and decryption in this mode also requires the computation of the inverse of the block cipher.

Double-decker [19] is a tweakable wide block cipher mode that also make use of unbalanced Feistel network, but does not require any block ciphers. This mode is instead based on doubly extendable cryptographic keyed (deck) functions [11]. These functions take a sequence of arbitrary length strings as input and output a string of variable length. The double-decker mode has two arbitrary length branches and this mode consists of two Feistel rounds with the deck functions surrounded by two Feistel rounds of a universal hash function. The processing time in this mode is dominated by one call to the keyed hash function and two calls to the underlying deck function. Since deck functions take a sequence of arbitrary length strings, a tweak can naturally be fed to the inner rounds in these modes as can be seen in Appendix E, Fig. 8.

Our contribution and paper organization In this paper we propose a new tweakable wide block cipher named *Mystrium* based on the double-decker mode as an alternative to Adiantum. Mystrium offers better efficiency while offering more security than Adiantum (see Sections 8.1 and 9). The bound on the advantage of distinguishing Adiantum from a random tweakable length-preserving permutation is dominated by 2^{-104} [10, Theorem 1, Section 6.4], while for Mystrium

it is 2^{-126}. Mystrium uses Multimix, a keyed hash function defined on top of Multimixer-128 [17] as universal hash and a new deck function that we propose, named Xymmer, as the deck function. The paper is organized as follows.

In Section 2 we describe the notations used throughout the paper. We introduce $\mathcal{G}_{512}$, a 512-bit public permutation that is the building block for our designs in Section 3, and report on its avalanche behavior in Section 4. We then specify Multimix, a universal hash based on Multimixer-128 that takes a short key and can process any sequence of arbitrary length byte-strings in Section 5. We then introduce the deck function Xymmer in Section 6, and discuss its security in Section 7. Finally, we introduce the tweakable wide block cipher Mystrium in Section 8, and analyze the performance of both Mystrium and Xymmer implemented on a 32-bit Armv7 Cortex-A processor in Section 9.

2 Notations and preliminaries

In this paper, we use operations on bit-strings and their integer counterpart interchangeably. To that end, we use different fonts to denote bit-strings, like $\mathtt{x}$, and integers, like x.

Given $\mathtt{x} \in \{0,1\}^*$, the i-th bit in $\mathtt{x}$ is denoted by $\mathtt{x}_i$, where indexing starts from 0. $|\mathtt{x}|$ will denote the length of $\mathtt{x}$ in bits. ϵ denotes the empty string with $|\epsilon| = 0$. $\mathtt{x} \,||\, \mathtt{y}$ will be used to denote concatenation of bit-strings $\mathtt{x}$ and $\mathtt{y}$. For $\mathtt{x}, \mathtt{y}$ with $|\mathtt{x}| = |\mathtt{y}|$, $\mathtt{x} \oplus \mathtt{y}$ denotes the bitwise XOR between $\mathtt{x}$ and $\mathtt{y}$. Messages and keys in this paper are limited to byte-strings and for a byte-string $\mathtt{x}$, $\mathsf{Bytelen}(\mathtt{x})$ will denote its length in bytes.

For $\mathtt{x} \in \{0,1\}^w$, where $w > 0$ and $0 \le i < j < w$, we denote: $\mathtt{x}[i{:}j] = \mathtt{x}_i\mathtt{x}_{i+1}\ldots\mathtt{x}_{j-1}$, $\mathtt{x}[i{:}] = \mathtt{x}_i\mathtt{x}_{i+1}\ldots\mathtt{x}_{w-1}$ and $\mathtt{x}[{:}i] = \mathtt{x}_0\mathtt{x}_1\ldots\mathtt{x}_{i-1}$.

Vectors are denoted in boldface, like $\mathbf{x}$ and () will denote the empty vector. The i-th component of a vector $\mathbf{x}$ of integers is denoted as x_i. For such an n-component vector $\mathbf{x}$ and $0 \le i < j \le n-1$, we also denote $\mathbf{x}[i{:}j] = (x_i, \ldots, x_{j-1})$.

Definition 1 (integer counterpart). *Let $\mathtt{x} \in \{0,1\}^w$ for some $w > 0$. We denote its integer counterpart as $int(\mathtt{x})$ defined as $int(\mathtt{x}) = \sum_{i=0}^{w-1} \mathtt{x}_{w-i-1} 2^i$.*

Definition 2 (binary representation). *Let $w > 0$ be an integer. For any non-negative $x < 2^w$, we denote its w-bit binary representation as $bin_w(x) = \mathtt{x}_0\mathtt{x}_1\ldots,\mathtt{x}_{w-1}$ such that $int(\mathtt{x}) = x$, i.e., for $0 \le i \le w-1$, $\mathtt{x}_i = \left\lfloor \frac{x}{2^{w-i-1}} \right\rfloor \bmod 2$.*

$bin(x)$ with w omitted denotes the binary representation of x. So, $|bin(x)| = \lfloor log_2(x) \rfloor + 1$ if $x \neq 0$, and $|bin(0)| = 1$.

The integers in this paper are treated as elements of $\mathbb{Z}/2^w\mathbb{Z}$ with $w = 32$ or 64. For two elements $x, y \in \mathbb{Z}/2^{32}\mathbb{Z}$, $x \boxplus y$, $x \boxminus y$, and $x \cdot y$ denote $(x + y) \bmod 2^{32}$, $(x - y) \bmod 2^{32}$, and $(x \cdot y) \bmod 2^{32}$ respectively. Integer multiplication between two 32-bit integers, referred to as 32-bit multiplication, plays an important role in our design. The 32-bit multiplication of $x, y \in \mathbb{Z}/2^{32}\mathbb{Z}$ denoted as $x \times y$ returns their product in $\mathbb{Z}/2^{64}\mathbb{Z}$ without any modular reduction. When operating on elements of $\left(\mathbb{Z}/2^{32}\mathbb{Z}\right)^n$ for $n > 1$, binary operations like $\boxplus$, $\boxminus$, $\cdot$, and $\times$ are

defined naturally component-wise. In case of $\boxplus$, $\boxminus$, $\cdot$, the output is again an element of $\left(\mathbb{Z}/2^{32}\mathbb{Z}\right)^n$, and in case of $\times$, the output is an element of $\left(\mathbb{Z}/2^{64}\mathbb{Z}\right)^n$. Addition in the group $\mathbb{Z}/2^{64}\mathbb{Z}$ will be denoted as $\boxplus$.

On platforms with processors having the Advanced Single Instruction and Multiple Data Stream (ASIMD) instruction sets, dedicated instructions are available for operations such as addition modulo 2^{32} and 32-bit multiplication. In such platforms, bit-strings are typically stored in 128-bit registers, which can be addressed as 2 64-bit or 4 32-bit words. Such 32 or 64-bit-strings can be interpreted as their integer counterpart determined by the so-called arrangement specifier. Thus, a 128-bit vector register can interchangeably be treated as a member of $\left(\mathbb{Z}/2^{32}\mathbb{Z}\right)^4$ or $\left(\mathbb{Z}/2^{64}\mathbb{Z}\right)^2$. We can perform operations like modular addition, 32-bit multiplication, etc., efficiently on elements of these sets.

We describe the operation of splitting a nw-bit string into n w-bit substrings denoted as $\mathsf{sbin}_{n,w}()$ in Algorithm 1. Alternatively the operation of treating a nw-bit string as an element of $(\mathbb{Z}/2^w\mathbb{Z})^n$ and vice-versa, which we denote as $\mathsf{sint}_{n,w}()$ and $\mathsf{sint\_inv}_{n,w}()$ respectively are described in Algorithms 3 and 4.

We also interpret a 64-bit integer as a pair of 32-bit integers determined by its most and least significant 32-bits respectively. We denote the operation of interpreting an n-component vector of 64-bit integers as a $2n$-component vector of 32-bit integers as $\mathsf{64to32}_n()$, and describe this operation in Algorithm 2. For all the use cases, these operations are implicit with no implementation cost.

Algorithm 1: Splitting an nw-bit string in n w-bit strings $\mathsf{sbin}_{n,w}(\mathbf{x})$

Inputs : $\mathbf{x} \in \{0,1\}^{nw}$
Output: $(\mathbf{x}_0, \mathbf{x}_1, \ldots \mathbf{x}_{n-1}) \in (\{0,1\}^w)^n$

for $i \leftarrow 0$ **to** $n-1$ **do** $\mathbf{x}_i \leftarrow \mathbf{x}[iw{:}(i+1)w]$
return $(\mathbf{x}_0, \mathbf{x}_1, \ldots \mathbf{x}_{n-1})$

Algorithm 2: Converting a string of n elements of $\mathbb{Z}/2^{64}\mathbb{Z}$ to a string of $2n$ elements of $\mathbb{Z}/2^{32}\mathbb{Z}$ $\mathsf{64to32}_n(\mathbf{x})$

Inputs : $(x_0, x_1, \ldots x_{n-1}) \in \left(\mathbb{Z}/2^{64}\mathbb{Z}\right)^n$
Output: $(y_0, y_1, \ldots y_{2n-1}) \in \left(\mathbb{Z}/2^{32}\mathbb{Z}\right)^{2n}$

for $i \leftarrow 0$ **to** $n-1$ **do**
$y_{2i} \leftarrow \left\lfloor \frac{x_i}{2^{32}} \right\rfloor, y_{2i+1} \leftarrow x_i \bmod 2^{32}$
return $(y_0, y_1, \ldots, y_{2n-1})$

Algorithm 3: Splitting nw-bit string in a string of n elements of $\mathbb{Z}/2^w\mathbb{Z}$ $\mathsf{sint}_{n,w}(\mathbf{x})$

Inputs : $\mathbf{x} \in \{0,1\}^{nw}$
Output: $(x_0, x_1, \ldots x_{n-1}) \in (\mathbb{Z}/2^w\mathbb{Z})^n$

$(\mathbf{x}_0, \mathbf{x}_1, \ldots \mathbf{x}_{n-1}) \leftarrow \mathsf{sbin}_{n,w}(\mathbf{x})$
for $i \leftarrow 0$ **to** $n-1$ **do** $x_i \leftarrow \mathrm{int}(\mathbf{x}_i)$
return $(x_0, x_1, \ldots x_{n-1})$

Algorithm 4: Converting a string of n elements of $\mathbb{Z}/2^w\mathbb{Z}$ in a nw-bit string $\mathsf{sint\_inv}_{n,w}(\mathbf{x})$

Inputs : $(x_0, x_1, \ldots x_{n-1}) \in (\mathbb{Z}/2^w\mathbb{Z})^n$
Output: $\mathbf{x} \in \{0,1\}^{nw}$

for $i \leftarrow 0$ **to** $n-1$ **do** $\mathbf{x}_i \leftarrow \mathrm{bin}_w(x_i)$
return $\mathbf{x}_0 \,||\, \mathbf{x}_1 \,||\, \ldots \,||\, \mathbf{x}_{n-1}$

3 The public permutation $\mathcal{G}_{512}[r]$

We present a public permutation $\mathcal{G}_{512}[r]\colon \left(\mathbb{Z}/2^{32}\mathbb{Z}\right)^{16} \to \left(\mathbb{Z}/2^{32}\mathbb{Z}\right)^{16}$ with r Feistel rounds. We refer to the round function of $\mathcal{G}_{512}[r]$ simply as $\mathcal{G}_{512}$, i.e., $\mathcal{G}_{512}[r] = (\mathcal{G}_{512})^r$ and specify it in Algorithm 5. We also provide an informal description of $\mathcal{G}_{512}$. The input $\mathbf{q}$ can be seen as a 4×4 matrix Q with i-th column $\mathbf{q}_i$. The effect of $\mathcal{G}_{512}$ on Q is shown in Eq. 1 and Fig. 1.

Algorithm 5: The public permutation $\mathcal{G}_{512}$

Inputs : A vector $\mathbf{q} = (q_0, q_1, \ldots, q_{15}) \in \left(\mathbb{Z}/2^{32}\mathbb{Z}\right)^{16}$
Output : A vector $\mathbf{z} = (z_0, z_1, \ldots, z_{15}) \in \left(\mathbb{Z}/2^{32}\mathbb{Z}\right)^{16}$

for $i \leftarrow 0$ **to** 3 **do** $\mathbf{q}_i \leftarrow \mathbf{q}[4i : 4i + 4]$
$\mathbf{d} \leftarrow \mathsf{64to32}_4(\mathbf{q}_0 \times \mathbf{q}_1)$
for $i \leftarrow 0$ **to** 7 **do** $x_i \leftarrow d_i \boxplus q_{8+i}$
for $i \leftarrow 0$ **to** 3 **do**
  $z_{2i} \leftarrow x_{2i} \boxplus x_{(2i+4) \bmod 8} \boxplus x_{(2i+5) \bmod 8}$
  $z_{2i+1} \leftarrow x_{2i+1} \boxplus x_{(2i+4) \bmod 8} \boxplus x_{(2i+5) \bmod 8}$
end
for $i \leftarrow 0$ **to** 7 **do** $z_{i+8} \leftarrow q_i$
return z

$$\begin{pmatrix} q_0 & q_4 & q_8 & q_{12} \\ q_1 & q_5 & q_9 & q_{13} \\ q_2 & q_6 & q_{10} & q_{14} \\ q_3 & q_7 & q_{11} & q_{15} \end{pmatrix} \xrightarrow{\mathcal{G}_{512}} \begin{pmatrix} z_0 & z_4 & q_0 & q_4 \\ z_1 & z_5 & q_1 & q_5 \\ z_2 & z_6 & q_2 & q_6 \\ z_3 & z_7 & q_3 & q_7 \end{pmatrix}. \tag{1}$$

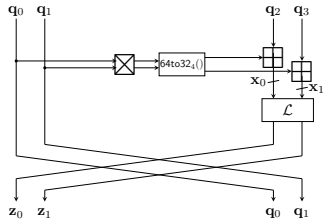


Fig. 1: $\mathcal{G}_{512}$, one round of $\mathcal{G}_{512}[r]$

We use 32-bit multiplication as the source of non-linearity in $\mathcal{G}_{512}$ and first compute 32-bit multiplication between elements of $\mathbf{q}_0$ and $\mathbf{q}_1$ component wise. Each of these products is a 64-bit integer, that we interpret as a pair of 32-bit integers: $\mathbf{d} = \mathsf{64to32}_4(\mathbf{q}_0 \times \mathbf{q}_1)$. We then compute $\mathbf{x}_0 = \mathbf{q}_2 \boxplus \mathbf{d}[0 : 4] = (x_0, x_1, x_2, x_3)$ and $\mathbf{x}_1 = \mathbf{q}_3 \boxplus \mathbf{d}[4 : 8] = (x_4, x_5, x_6, x_7)$.

We finally apply a linear transform $\mathcal{L}$ on $\mathbf{x} = (\mathbf{x}_0, \mathbf{x}_1)$, where $\mathcal{L}$ consists of an application of a 4×4 circulant matrix to distinct components of $\mathbf{x}$:

$$\begin{pmatrix} 1 & 0 & 1 & 1 \\ 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 \\ 1 & 1 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} x_0 \\ x_1 \\ x_4 \\ x_5 \end{pmatrix} = \begin{pmatrix} z_0 \\ z_1 \\ z_4 \\ z_5 \end{pmatrix} \quad \text{and} \quad \begin{pmatrix} 1 & 0 & 1 & 1 \\ 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 \\ 1 & 1 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} x_2 \\ x_3 \\ x_6 \\ x_7 \end{pmatrix} = \begin{pmatrix} z_2 \\ z_3 \\ z_6 \\ z_7 \end{pmatrix}.$$

Instead of applying the circulant matrices simply to the columns $\mathbf{x}_0$ and $\mathbf{x}_1$, our choice allows more efficient implementation while resulting in the same diffusion. These matrices are very similar to the circulant matrix employed in Multimixer-128 and branch number [13] of these matrices is 4.

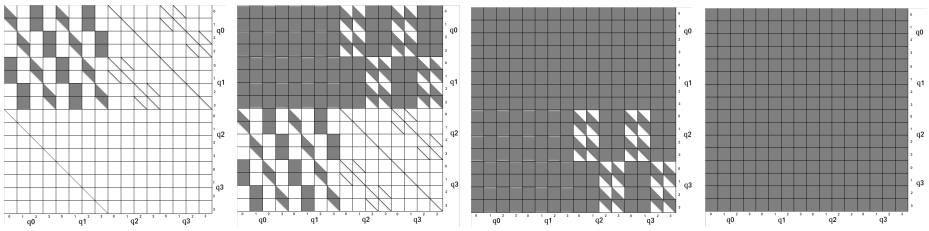
4 Avalanche tests of $\mathcal{G}_{512}[r]$

Criteria such as full diffusion, avalanche, and strict avalanche criterion (SAC) [31] are commonly used by cryptographers to estimate the vulnerability of a reduced-round cryptographic function to attacks. A function satisfies SAC if whenever

one of the input bits changes, each bit in the output changes with a probability close to 1/2, where the probability is taken over a large and random sample. These criteria are binary, meaning that they are either met or not. Therefore, one usually reports the number of rounds required to satisfy them. Here we use the avalanche probability vector defined in [11] because we are interested in figuring out how $\mathcal{G}_{512}[r]$ realizes the avalanche criteria through the rounds.

We describe the avalanche test in Appendix A, Algorithm 15. Concretely, for a cryptographic function F, we calculate a vector P_δ^F where the i^{th} entry reports the probability taken over the sample that the i-th output bit flips under F due to the input difference δ (taken bitwise modulo 2). We then build a matrix $\mathbf{V}_F$ with i-th column $P_{\delta_i}^F$, where δ_i is the binary vector with a 1 in the i-th position and 0 elsewhere. To detect small biases we take $M = 25.000.000$ samples.

We depict $\mathbf{V}_{\mathcal{G}_{512}[r]}$ as a 2-D figure in Fig. 2. It shows how $\mathcal{G}_{512}[r]$ realizes avalanche through the rounds for $r = 1, 2, 3, 4$. In this figure input and output indices are on the x and y axes respectively. The gray-scale value in pixel (x, y) shows the probability of changing the output bit at index y when the input bit in index x is changed, where white color corresponds to probability of zero and black color represents probability of one. We further explain Fig. 2 in Appendix B.



(a) $\mathcal{G}_{512}[1]$ (b) $\mathcal{G}_{512}[2]$ (c) $\mathcal{G}_{512}[3]$ (d) $\mathcal{G}_{512}[4]$

Fig. 2: The avalanche probability matrix $\mathbf{V}_{\mathcal{G}_{512}[r]}$

It can be seen that after 3 rounds, some output bits are still independent of certain input bits. After 4 rounds, $\mathbf{V}_{\mathcal{G}_{512}[4]}$ is uniformly gray, suggesting that if one bit changes in the input, each output bit changes with a probability of close to one half. To verify the divergence from 1/2 of the entries of $\mathbf{V}_{\mathcal{G}_{512}[4]}$, we compute the cumulative distribution function (CDF) of its elements and compare it with the expected distribution for a random mapping in Fig. 3. With a sample size of 25.000.000, the latter distribution would have mean 1/2 and standard deviation $1/2\sqrt{25.000.000} = 1/10.000$. These lines overlap and thus, we cannot distinguish the CDF of $\mathbf{V}_{\mathcal{G}_{512}[4]}$ from that of a random mapping.

5 Multimix

Multimixer-128 is an example of a parallel keyed hash function [15, 18, 17]. It is the parallelization of the public function $\mathcal{F}$-128: $\left(\mathbb{Z}/2^{32}\mathbb{Z}\right)^8 \rightarrow \left(\mathbb{Z}/2^{64}\mathbb{Z}\right)^8$

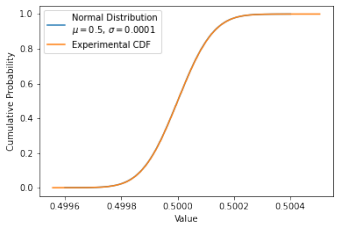


Fig. 3: Cumulative distribution function of $\mathbf{V}_{\mathcal{G}_{512}[r]}$

described in Appendix A, Algorithm 13. Multimixer-128 is a 2^{-127}-Δuniversal keyed hash function [17] under the assumption of long independent masks and is defined only over strings of elements of $\left(\mathbb{Z}/2^{32}\mathbb{Z}\right)^8$, that we call *block-strings*. Given a block-string $\mathbf{M} \in \left(\mathbb{Z}/2^{32}\mathbb{Z}^8\right)^l$, we refer to l as its length in blocks and denote it as $\mathsf{blocklen}(\mathbf{M}) = l$.

We now propose a padding scheme to convert arbitrary-length byte-string sequences into block-strings and an algorithm to generate secret mask blocks.

5.1 Injective input encoding

We first describe the injective encoding function, $\mathsf{ValueLengthEncode}()$ that encodes a byte-string $\mathtt{M}$ into a byte-string with $\mathsf{Bytelen}$ a multiple of 32. The function is inspired by the padding scheme in [23, Section 5.1]. In $\mathsf{ValueLengthEncode}(\mathtt{M})$, we append an encoding of $\mathsf{Bytelen}(\mathtt{M})$ followed by a sufficient number of 0 bytes to $\mathtt{M}$ such that 32 divides the $\mathsf{Bytelen}$ of the resulting byte-string. The encoding of $\mathsf{Bytelen}(\mathtt{M})$ is the byte-string $\mathtt{m}^0\mathtt{m}^1 \ldots \mathtt{m}^{t-1}$ for some $t > 1$ such that the MSB of $\mathtt{m}^0$ is 0, and for the remaining $\mathtt{m}^i$, it is 1. $\mathsf{Bytelen}(\mathtt{M})$ can be recovered from the encoding by $\mathsf{Bytelen}(\mathtt{M}) = \sum_{i=0}^{t-1} (\mathrm{int}(\mathtt{m}^i) \bmod 2^7) 2^{7(t-i-1)}$. We specify $\mathsf{ValueLengthEncode}(\mathtt{M})$ in Algorithm 6.

Since Multimixer-128 takes block-strings as input and Xymmer as a deck function must be able to process a sequence of byte-strings, we specify the injective encoding of a sequence of byte-strings into a block-string that we call $\mathsf{SequenceToBlockString}()$ in Algorithm 7. For a sequence of byte-strings $(\mathtt{M}_0; \mathtt{M}_1; \ldots; \mathtt{M}_{n-1})$, we first create a byte-string $\mathtt{M}$ by appending the result of $\mathsf{ValueLengthEncode}(\mathtt{M}_i)$ to ϵ for each M_i in the sequence. We then convert each 32-byte chunk of $\mathtt{M}$ into elements of $(\mathbb{Z}/2^{32}\mathbb{Z})^8$. Clearly, the strings can be recovered from the encoding recursively one by one from the back.

5.2 Mask derivation

We propose $\mathsf{Maskgen}_\kappa(\mathtt{K})$ that outputs a string of secret masks $\mathbf{K} \in \left(\left(\mathbb{Z}/2^{32}\mathbb{Z}\right)^8\right)^\kappa$ from a short secret key $\mathtt{K}$ with $32 \leq \mathsf{Bytelen}(\mathtt{K}) \leq 63$. In $\mathsf{Maskgen}_\kappa(\mathtt{K})$, $\mathtt{K}$ is

Algorithm 6: Encoding of a byte-string into a byte-string with Bytelen a multiple of 32 ValueLengthEncode()

Inputs : A byte-string $\mathtt{M}$
Output : a byte-string $\mathtt{M}'$ where 32 divides $\mathsf{Bytelen}(\mathtt{M}')$

$\mathtt{b} \leftarrow \mathrm{bin}(\mathsf{Bytelen}(\mathtt{M}))$
$\mathtt{m} \leftarrow \epsilon$
while $|\mathtt{b}| > 7$ **do**
  $\mathtt{x} \leftarrow \mathtt{b}[(|\mathtt{b}| - 7):]$
  $\mathtt{b} \leftarrow \mathtt{b}[:(|\mathtt{b}| - 7)]$
  $\mathtt{m} \leftarrow 1 \,||\, \mathtt{x} \,||\, \mathtt{m}$
end
$\mathtt{m} \leftarrow \mathrm{bin}_8(\mathrm{int}(\mathtt{b})) \,||\, \mathtt{m}$
$\mathtt{M}' \leftarrow \mathtt{M} \,||\, \mathtt{m}$
while 32 *does not divide* $\mathsf{Bytelen}(\mathtt{M}')$ **do** $\mathtt{M}' \leftarrow \mathtt{M}' \,||\, \mathrm{bin}_8(0)$
return $\mathtt{M}'$

Algorithm 7: Encoding a sequence of byte-strings to a block-string SequenceToBlockString()

Inputs : Sequence of byte-strings $(\mathtt{M}_0; \mathtt{M}_1; \ldots; \mathtt{M}_{n-1})$
Output : A string $\mathbf{M} \in \left(\left(\mathbb{Z}/2^{32}\mathbb{Z}\right)^8\right)^*$

$\mathtt{M} \leftarrow \epsilon$
for $i \leftarrow 0$ **to** $n-1$ **do** $\mathtt{M} \leftarrow \mathtt{M} \,||\, \mathsf{ValueLengthEncode}(\mathtt{M}_i)$
$\ell \leftarrow \frac{|\mathtt{M}|}{256}$
$(\mathtt{M}_0, \mathtt{M}_1, \ldots, \mathtt{M}_{\ell-1}) \leftarrow \mathsf{sbin}_{\ell,256}(\mathtt{M})$
$\mathbf{M} \leftarrow (\mathsf{sint}_{8,32}(\mathtt{M}_0), \mathsf{sint}_{8,32}(\mathtt{M}_1), \ldots, \mathsf{sint}_{8,32}(\mathtt{M}_{\ell-1}))$
return $\mathbf{M}$

first padded by ValueLengthEncode(K). It is then interpreted as an element of $\left(\mathbb{Z}/2^{32}\mathbb{Z}\right)^{16}$, and processed by $\mathcal{G}_{512}[6]$ to generate the first two mask-string blocks. To generate more mask blocks, we apply $\mathcal{G}_{512}[4]$ sequentially each time extracting two blocks. We describe it in Algorithm 8 and illustrate it in Fig. 4.

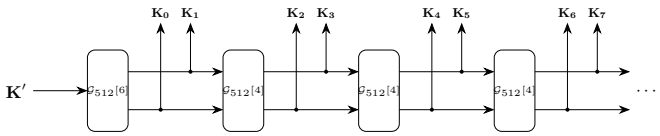


Fig. 4: Description of $\mathsf{Maskgen}_\kappa(\mathtt{K})$

5.3 Specification of Multimix

By generating a sequence of secret masks via $\mathsf{Maskgen}_\kappa()$ from a short secret key K, and by making use of SequenceToBlockString() over a sequence of byte-strings,

Algorithm 8: Mask generation algorithm $\mathsf{Maskgen}_\kappa()$

Inputs : A byte-string $\mathtt{K}$ with $32 \leq \mathsf{Bytelen}(\mathtt{K}) \leq 63$, An integer $\kappa \geq 1$
Output : A string $\mathbf{K} \in \left(\left(\mathbb{Z}/2^{32}\mathbb{Z}\right)^8\right)^\kappa$

$\mathbf{K}' \leftarrow \mathcal{G}_{512}[6]\left(\mathsf{sint}_{16,32}(\mathsf{ValueLengthEncode}(\mathtt{K}))\right)$
$\mathbf{K} \leftarrow (\mathbf{K}')$
while $\lceil \kappa/2 \rceil > 1$ **do**
  $\mathbf{K}' \leftarrow \mathcal{G}_{512}[4](\mathbf{K}')$
  $\mathbf{K}$.append$(\mathbf{K}')$
  $\lceil \kappa/2 \rceil \leftarrow \lceil \kappa/2 \rceil - 1$
end
$\mathbf{K} \leftarrow \left(\mathbf{K}_0[0:8], \mathbf{K}_0[8:16], \ldots, \mathbf{K}_{\lceil \kappa/2 \rceil}[0:8], \mathbf{K}_{\lceil \kappa/2 \rceil}[8:16]\right)$
$\mathbf{K} \leftarrow \mathbf{K}[0{:}\kappa]$
return $\mathbf{K}$

we obtain a keyed hash function on top of Multimixer-128 that takes a short key and can process any non-empty sequence of arbitrary length byte-strings. We call this function Multimix and describe it in Algorithm 9.

Algorithm 9: The keyed hash function Multimix()

Inputs : Sequence $(\mathtt{M}_0; \mathtt{M}_1; \ldots, \mathtt{M}_{n-1})$
  A secret key $\mathtt{K}$ with $32 \leq \mathsf{Bytelen}(\mathtt{K}) \leq 63$
Output : A digest $\mathbf{h} \in \left(\mathbb{Z}/2^{64}\mathbb{Z}\right)^8$

$\mathbf{M} \leftarrow \mathsf{SequenceToBlockString}(\mathtt{M}_0; \mathtt{M}_1; \ldots; \mathtt{M}_{n-1})$
$\ell \leftarrow \mathsf{blocklen}(\mathbf{M})$
$\mathbf{K} \leftarrow \mathsf{Maskgen}_\ell(\mathtt{K})$
$\mathbf{h} \leftarrow (0,0,0,0,0,0,0,0)$
for $i \leftarrow 0$ **to** $\ell - 1$ **do** $\mathbf{h} \leftarrow \mathbf{h} \boxplus \mathcal{F}\text{-}128(\mathbf{M}_i \boxplus \mathbf{K}_i)$
return $\mathbf{h}$

6 The deck function Xymmer

Xymmer is a deck function and hence must support the corresponding interface. A deck function F takes as input a secret key $\mathtt{K}$ and a sequence of an arbitrary number of byte-strings $(\mathtt{M}_0; \ldots; \mathtt{M}_{n-1})$, and produces an arbitrary-length bit-string $\mathtt{Z}$ that shall be hard to distinguish from a random string. We write:

$$\mathtt{Z} = 0^\ell \oplus F_{\mathtt{K}}\left(\mathtt{M}_0, \ldots, \mathtt{M}_{n-1}\right) \ll d\,.$$

In this notation the output length is determined by the context, i.e., its length is ℓ bits as it is added to the all-zero string 0^ℓ and the optional expression $\ll d$ at the right denotes that these bits are taken from the offset $d \in \mathbb{N}$ in its output. If it is absent, bits are taken from offset 0.

A deck function should allow efficient incremental computing. By keeping state, appending an extra string to the input sequence costs only the processing of this extra string. Similarly, like an extendable output function (XOF), asking for more output bits for a given input should also be efficient.

6.1 Informal description and design rationale for Xymmer

Xymmer first compresses its input to a 512-bit internal state by applying Multimix. The state is then expanded to an output bit-string of arbitrary length. As Multimix in the compression phase, we designed the expansion phase to support parallelism.

The expansion phase takes the state at the output of the compression phase, denoted as $\mathbf{q}^{(0)}$, and generates output blocks in a sequence of *stages*. In each stage $i \geq 1$ it derives 32 output blocks from the state $\mathbf{q}^{(i)}$ and then updates the state to $\mathbf{q}^{(i+1)}$. The derivations of output blocks and state update make use of the same transformation. This transformation consists of applying $\mathcal{G}_{512}[6]$ with a feed forward, making it infeasible to compute state values $\mathbf{q}^{(i)}$ from output blocks or from $\mathbf{q}^{(j)}$ with $j > i$. The inputs to this transformation are diversified with encoded indices. The encoding of an index $0 \leq i \leq 32$ is denoted as $\mathsf{Enc}(i)$.

The expansion phase of Xymmer can be seen as an instantiation of the Kirby construction [23] with the 512-bit permutation $P = \mathcal{G}_{512}[6]$. That construction has excellent generic security as it provably achieves birthday-bound security in the ideal permutation model provided its inputs form a prefix-free set.

Kirby takes as input a key, an ID and a sequence of *input blocks*. The value of $\mathbf{q}^{(0)} \boxplus \mathsf{Enc}(32)$ serves as key in our Kirby instance, the ID is empty and the sequences of encoded indices serve as input block sequences. For any $i \geq 1$ and $0 \leq j \leq 31$, output block $\mathbf{z}^{(i,j)}$ can be seen as the output of a Kirby construction where the input block sequence is $\mathsf{Enc}(32)\mathsf{Enc}(32)\ldots\mathsf{Enc}(32)\mathsf{Enc}(j)$, i.e., $i-1$ times $\mathsf{Enc}(32)$ followed by $\mathsf{Enc}(j)$. As there clearly is domain separation between intermediate blocks ($= \mathsf{Enc}(32)$) and final blocks ($\mathsf{Enc}(j)$ with $j < 32$) these sequences form a prefix-free set. For $\mathsf{Enc}(i)$ we use the scheme specified in Definition 3, that aims at preventing differential attacks discussed in Section 7.3.

6.2 Specification of Xymmer

Before we specify Xymmer, we define the index encoding function $\mathsf{Enc}(i)$ and the transformation $\mathsf{FF}[6]$ it uses.

Definition 3. *The index encoding* $\mathsf{Enc}\colon \mathbb{Z}/32\mathbb{Z} \to \left(\mathbb{Z}/2^{32}\mathbb{Z}\right)^{16}$ *is defined as*

$$\mathsf{Enc}(i) = (c_0, c_1, \ldots, c_{15}) \text{ with } c_j = \begin{cases} (i+1)\cdot 251 & , j \in \{0,1,2,3\} \\ 2^i & , j \in \{4,5,6,7\} \\ 0 & , j \in \{8,\ldots,15\}\,. \end{cases}$$

Addition with 2^i and $(i+1)\cdot A$ for some $A \in \mathbb{Z}/2^{32}\mathbb{Z}$ can be implemented very efficiently in software using bitwise left shift by an offset i on $\mathrm{bin}_{32}(1)$ and

by repeated additions of A respectively. In the ARMv7-A architecture, we can only store constants smaller than 256 with minimal cost. Our choice of $A = 251$ is a good choice with respect to the attacks discussed in Section 7.3.

After addition by $\mathsf{Enc}(i)$, the internal state is input to a transformation that we denote as $\mathsf{FF}[6]$ and that is specified in Algorithm 10. In short, $\mathsf{FF}[6]$ applies $\mathcal{G}_{512}[6]$ to its input and returns the bitwise addition modulo 2 of the binary representation of its input and the $\mathcal{G}_{512}[6]$ output.

Algorithm 10: The transformation $\mathsf{FF}[6]\,(\mathbf{q})$

Inputs : A vector $\mathbf{q} \in \left(\mathbb{Z}/2^{32}\mathbb{Z}\right)^{16}$
Output : A binary string $\mathbf{z} \in \{0,1\}^{512}$

$\mathbf{q}' \leftarrow \mathcal{G}_{512}[6](\mathbf{q})$
$\mathsf{z} \leftarrow \mathsf{sint\_inv}_{16,32}(\mathbf{q}) \oplus \mathsf{sint\_inv}_{16,32}(\mathbf{q}')$
return z

We describe Xymmer in Algorithm 11 and illustrate it in Fig. 5.

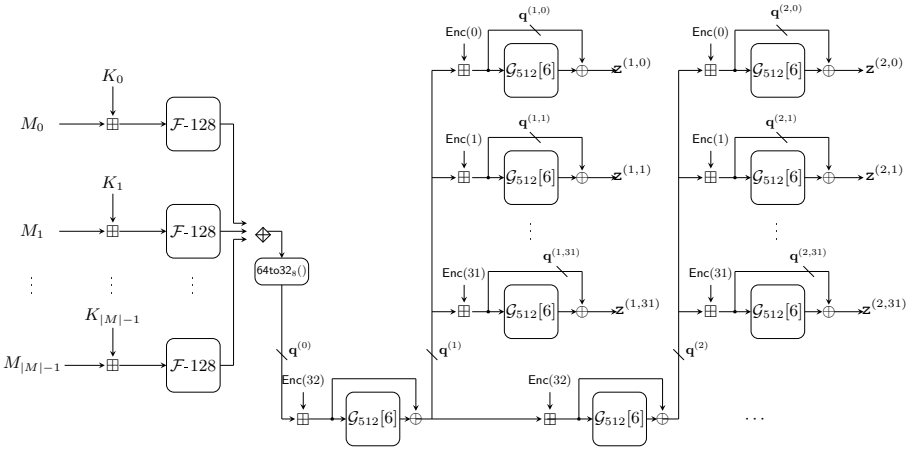


Fig. 5: Deck function Xymmer

6.3 Security Claim

We make a security claim that expresses an upper bound on the advantage of distinguishing Xymmer with a secret key from a random oracle. The security of Xymmer is limited by the ε-universality of Multimix in the compression phase and by the generic security of Kirby with a 512-bit state in a single-user setting for the expansion phase.

Algorithm 11: The deck function Xymmer

Inputs : Sequence of byte-strings $(\mathtt{M}_0; \mathtt{M}_1; \ldots; \mathtt{M}_{n-1})$
A secret key $\mathtt{K}$ with $32 \leq \mathsf{Bytelen}(\mathtt{K}) \leq 63$
Requested offset $d \in \mathbb{N}$
Output : An arbitrarily long bit-string $\mathbf{z}$

$\mathbf{q}^{(0)} \leftarrow \mathsf{64to32}_8(\mathrm{Multimix}_{\mathtt{K}}(\mathtt{M}_0; \mathtt{M}_1; \ldots; \mathtt{M}_{n-1}))$
$\mathbf{z} \leftarrow \epsilon$
for $i \leftarrow 1$ **to** ∞ **do**
  $\mathbf{q}^{(i)} \leftarrow \mathsf{sint}_{16,32}\left(\mathsf{FF}[6]\left(\mathbf{q}^{(i-1)} \boxplus \mathsf{Enc}(32)\right)\right)$
  for $j \leftarrow 0$ **to** 31 **do** $\mathbf{z}^{(i,j)} \leftarrow \mathsf{FF}[6]\left(\mathbf{q}^{(i)} \boxplus \mathsf{Enc}(j)\right)$
end
$\mathbf{z} \leftarrow \mathbf{z}^{(1,0)} \,||\, \mathbf{z}^{(1,1)} \,||\, \mathbf{z}^{(1,2)} \ldots [d{:}]$
return $\mathbf{z}$

We quantify the resources of the adversary with the following metrics:

- N: the amount of computation expressed in the number of evaluations of $\mathcal{G}_{512}[6]$ or equivalent,
- M: the total amount of input and output of construction queries expressed in the number of blocks,
- q: the number of construction queries.

Similar to the security bounds in [12, 23], in our claim we require that $\mathtt{K}$ be sampled from a distribution $\mathcal{D}_{\mathrm{Key}}$ with at least 256 bits of min-entropy, where min-entropy $\mathcal{H}_{\min}(\mathcal{D}_{\mathrm{Key}}) = \max\limits_{\mathtt{x} \in \{0,1\}^{8\kappa}} -\log_2(\Pr(\mathtt{K} \leftarrow \mathcal{D}_{\mathrm{Key}}) : \mathtt{K} = \mathtt{x})$.

Here $\mathtt{K} \leftarrow \mathcal{D}_{\mathrm{Key}}$ represents that $\mathtt{K}$ is generated using the distribution $\mathcal{D}_{\mathrm{Key}}$.

Claim. The advantage of an adversary in distinguishing Xymmer from a random oracle, where $\mathtt{K}$ is sampled from a distribution with min-entropy of 256-bits and M is limited by $M \leq 2^{128}$, is upper bounded by

$$\frac{q^2}{2^{128}} + \frac{N}{2^{256}}\,. \tag{2}$$

First term in Eq. 2 expresses the effort to find collision at the digest of Multimix and the second term accounts for the effort to find the key by exhaustive search.

7 Security analysis of Xymmer

In this section, we follow the notations used in Algorithm 11. For $i \geq 0$, we refer to $\mathbf{q}^{(i)}$ as the i-th stage. We further denote for $i \geq 1$, $0 \leq j \leq 31$:

$$\mathbf{q}^{(i,j)} = \mathbf{q}^{(i)} \boxplus \mathsf{Enc}(j),\ \mathbf{w}^{(i,j)} = \mathcal{G}_{512}[6](\mathbf{q}^{(i,j)}),\ \text{and}\ \mathtt{w}^{(i,j)} = \mathsf{sint\_inv}_{16,32}(\mathbf{w}^{(i,j)})\,. \tag{3}$$

For any fixed $i \geq 1$ and $0 \leq j \neq k \leq 31$, we refer to $\mathbf{q}^{(i,j)}$, $\mathbf{q}^{(i,k)}$ or $\mathbf{w}^{(i,j)}$, $\mathbf{w}^{(i,k)}$ or $\mathbf{z}^{(i,j)}$, $\mathbf{z}^{(i,k)}$ as states belonging to the i-th stage.

Xymmer initialized with a secret key K can be distinguished from a random function in several ways that we describe in the following subsections.

7.1 Guessing $\mathbf{q}^{(0)}$, achieving collision at $\mathbf{q}^{(0)}$ for independent masks

Multimixer-128 is a 2^{-127}-Δuniversal hash function under the assumption of independent masks. So the probability of a collision at its output for a pair of messages is upper bounded by 2^{-127} . Moreover, its universality is also valid over messages of different length. Now, if we choose a pair of messages $\mathbf{M}_0$ and $(\mathbf{M}_0, \mathbf{M}_1, \ldots, \mathbf{M}_l)$, i.e., a single-block message and an $(l+1)$-block message whose first block is the same as the first message, then the probability of predicting that their output difference under Multimixer-128 is Δ is given by the number of pre-images of block length l of Δ under Multimixer-128. Due to the universality, this probability is upper bounded by 2^{-127}. Therefore for any l, the probability of predicting an output of Multimixer-128 is upper bounded by 2^{-127}.

The assumption of independent masks does not hold for the keyed hash function Multimix. Multimix is 2^{-127}-Δuniversal provided that an attacker is unable to exploit the dependencies between the secret masks in the string $\mathbf{K} = (\mathbf{K}_0, \mathbf{K}_1, \ldots, \mathbf{K}_{\kappa-1})$ resulting from $\mathsf{Maskgen}_\kappa()$ to predict output differences with probability greater than 2^{-127}. By dependency we mean dependency inside some mask block $\mathbf{K}_i$, or in between different mask blocks $\mathbf{K}_i$ and $\mathbf{K}_j$. We now provide strong heuristic arguments to show that this is indeed the case.

7.2 Exploiting dependency in mask derivation

The universality bound of Multimixer-128 is determined by the maximum differential probability across all differentials and image probability across all outputs to $\mathcal{F}$-128, the public function of Multimixer-128. Here image probability of an output refers to the fraction of its pre-images. Due to the specification of $\mathsf{Maskgen}_\kappa()$, attackers targeting single or two-block messages must exploit dependencies within mask blocks $\mathbf{K}_0$ or in $\mathbf{K}_0, \mathbf{K}_1$ respectively. Any such dependencies would stem from lack of entropy in the key x and the failure of $\mathcal{G}_{512}[6]$ to diffuse that. According to [17, Lemma 10], the solution set of single-block differentials with a differential probability of 2^{-128} consists of keys $\mathbf{K}_0 \in \left(\mathbb{Z}/2^{32}\mathbb{Z}\right)^8$ where 4 of the 8 components are constants. Similar conditions hold for the set of pre-images of outputs with image probability close to 2^{-127} [17, Lemma 8]. For two-block messages, the conditions are characterized by linear conditions on eight out of the sixteen components of the mask blocks $\mathbf{K}_0, \mathbf{K}_1$. Thus, improving such differential attacks would require very strong dependencies between the mask blocks. Dependencies in $\mathbf{K}_0$, $\mathbf{K}_1$ or between $\mathbf{K}_0$ and $\mathbf{K}_1$ are very intricate since $\mathcal{G}_{512}[6]$ significantly scrambles its inputs.

We thoroughly investigated potential attacks for exploiting dependencies, but found absolutely no feasible means by which an attacker can improve differen-

tial attacks utilizing mask dependencies. In fact, we could not even detect such dependencies when $\mathcal{G}_{512}[4]$ takes as input a key with only 128-bits of entropy.

The differential probability of other single or two-block differentials is at most 2^{-160}, significantly smaller than 2^{-128}. Thus improving differential attacks using these differentials would require significantly more dependency in the mask blocks, making it considerably more challenging for attackers to target them.

Over more than 2 blocks, the attacker needs to exploit not only $\mathcal{G}_{512}[6]$, but also at least a single instance of $\mathcal{G}_{512}[4]$. So using more blocks to exploit possible dependencies is even more challenging. Many cryptographic constructions, e.g., Xoofff [11] only make use of very simple and linear mask generation algorithm. To the best of our knowledge, there have not been any practical attacks in such schemes exploiting dependency in the mask bits due to linear mask generation. Knudsen et al. have also reported in case of iterated ciphers that complex mask schedules lead to a high chance of attaining uniform distribution [22]. Based on these arguments, we do not expect dependencies in the mask block sequence due to $\mathsf{Maskgen}()$ to lead to collision-generating attacks with success probability above 2^{-127} per pair.

In fact, we think that using only $\mathcal{G}_{512}[2]$ instead of $\mathcal{G}_{512}[4]$ inside $\mathsf{Maskgen}()$ would also lead to a secure mask generation algorithm. But we are very conservative in our design and for a significant safety margin, we use $\mathcal{G}_{512}[4]$ instead. We welcome further cryptanalysis on our design. Finally, we claim that Multimix is a 2^{-127}-Δuniversal hash function.

7.3 Recovering internal state $\mathbf{q}^{(i)}$ from output blocks

An attacker may try to recover an internal state $\mathbf{q}^{(i)}$ with $i \geq 1$ allowing them to gain full knowledge on the output starting from that stage leading to a distinguishing attack.

We first look at the difficulty of recovering $\mathbf{q}^{(i)}$ from a single output block. This amounts to inverting the transformation $\mathsf{FF}[6]$. To better understand this attack, we look at the difficulty of inverting $\mathsf{FF}[1]$, i.e., the transformation with a single round of $\mathcal{G}_{512}$, and provide the details of this attack in Appendix C. Even, our best attack on inverting $\mathsf{FF}[1]$ requires 2^{256} guesses. Over multiple rounds, this attack becomes even more infeasible. Thus, we claim that an attacker can not retrieve the internal state from a single output block.

The attacker may also combine multiple output blocks in order to recover the state $\mathbf{q}^{(i)}$. We first let the attacker try to recover the internal states from two output blocks. The best strategy for the attacker is to choose the output blocks from the same stage since in addition to the output blocks and consequently their bitwise difference modulo 2, the attacker also knows the difference between the corresponding internal states modulo 2^{32}. More concretely, for $i \geq 1$ and $0 \leq j \neq k \leq 31$ the attacker knows

$$\mathbf{z}^{(i,j)} \oplus \mathbf{z}^{(i,k)} = (\mathtt{w}^{(i,j)} \oplus \mathtt{q}^{(i,j)}) \oplus (\mathtt{w}^{(i,k)} \oplus \mathtt{q}^{(i,k)}) \,,$$
$$\mathbf{q}^{(i,j)} \boxminus \mathbf{q}^{(i,k)} = \Big(\mathbf{q}^{(i)} \boxplus \mathsf{Enc}(j)\Big) \boxminus \Big(\mathbf{q}^{(i)} \boxplus \mathsf{Enc}(k)\Big) = \mathsf{Enc}(j) \boxminus \mathsf{Enc}(k) \,.$$

By slightly abusing notation, we denote $(\mathbf{a}^{(j,k)}, \mathbf{b}^{(j,k)}) = \mathbf{q}^{(i,j)} \boxminus \mathbf{q}^{(i,k)}$, where $\mathbf{a}^{(j,k)}, \mathbf{b}^{(j,k)} \in \left(\mathbb{Z}/2^{32}\mathbb{Z}\right)^8$. By Definition 3, this means

$$\mathbf{b}^{(j,k)} = \mathbf{0} \text{ and for } n \in \{0,1,2,3\}, \mathbf{a}_n^{(j,k)} = (j \boxminus k) \cdot 251 \ , \mathbf{a}_{n+4}^{(j,k)} = 2^j \boxminus 2^k \ .$$

Since $\mathbf{b}^{(j,k)} = \mathbf{0}$, the attacker knows xor-difference between the last 256-bits of $\mathtt{w}^{(i,j)}$ and $\mathtt{w}^{(i,k)}$. So, the attacker has full knowledge of input difference modulo 2^{32} and 256-bits out of the 512-bit bitwise output difference modulo 2 to $\mathcal{G}_{512}[6]$. We first show that an attacker cannot exploit knowledge of the input difference modulo 2^{32} to mount differential attacks.

In $\mathcal{G}_{512}$ we first compute the 4 32-bit multiplications involving $\mathbf{q}_n^{(i,j)}$ and $\mathbf{q}_{n+4}^{(i,j)}$ for $n \in \{0,1,2,3\}$. This means that we introduce an input difference $((j \boxminus k) \cdot 251, 2^j \boxminus 2^k)$ to each of these 32-bit multiplications at the beginning of the first round of $\mathcal{G}_{512}[6]$. The differential properties of 32-bit multiplication have been studied in [17]. Given any input difference to 32-bit multiplication, the maximum possible value of differential probability over all output differences for that particular input difference can be computed efficiently. The input difference $((j \boxminus k) \cdot 251, 2^j \boxminus 2^k)$ has been chosen such that for all $\binom{32}{2}$ values of $0 \leq j \neq k \leq 31$, maximum differential probability of these differentials for any output difference is upper bounded by $2^{-39,97}$.

This means that for the internal states $\mathbf{q}^{(i,j)}$ and $\mathbf{q}^{(i,k)}$, the output difference after the first round of $\mathcal{G}_{512}$ can only be predicted with a probability at most 2^{-159}. Due to the high nonlinearity of $\mathcal{G}_{512}$, following any plausible differential trail over 5 more rounds is completely infeasible for any practical attacker.

The attacker may try to utilize the knowledge of 256 bits of the difference at the output of $\mathcal{G}_{512}[6]$. But even knowledge of full xor-difference at both input and output to $\mathcal{G}_{512}[6]$ leads to no meaningful attacks since their propagation through $\mathcal{G}_{512}[6]$ defined over 32-bit integers cannot be exploited feasibly.

On the other hand, we can also make the very strong assumption that the attacker has knowledge of the difference modulo 2^{32} at the output of $\mathcal{G}_{512}[6]$. We note that this can happen if we replace the bitwise addition inside $\mathsf{FF}[6]$ with addition modulo 2^{32}. In our attack, we look at the propagation of the input difference in the forward direction and of the output difference in the backward direction to meet in the middle. We provide details of our attack in Appendix D. We can only meaningfully attack at most 3 rounds of $\mathcal{G}_{512}[6]$. Our attack is not scalable over more rounds, and hence we claim that knowledge of difference modulo 2^{32} at input and output of $\mathcal{G}_{512}[6]$ does not aid the attacker.

While using internal states belonging to different stages makes any potential attack much harder, combining multiple blocks belonging to the same stage leads to an over-defined system of equations and we could not find any such attack to improve over the attacks utilizing two output blocks.

7.4 Cryptanalytic attacks involving input and output to Xymmer

The class of attacks that exploit Xymmer output dependencies from its input include differential, linear, and integral attacks. These are unlikely to succeed as

the data path between input and output takes the composition of $\mathcal{F}$-128 with at least two iterations of $\mathcal{G}_{512}[6]$, with its 512-bit state and powerful avalanche behavior after only 4 rounds. For differential attacks, an attacker can only obtain a collision after the compression phase with a probability upper bounded by 2^{-127}. Assuming there is no collision, any other differential attack is infeasible since that requires difference propagation over at least 12 rounds of $\mathcal{G}_{512}$. Moreover, $\mathcal{F}$-128 and $\mathcal{G}_{512}[6]$ treat the state as an array of 32-bit integers, and linear and integral attacks work best at the bit level, but cannot be applied to arrays of 32-bit integers, which does not even have a field structure. We welcome further cryptanalysis on our design.

8 The tweakable wide block cipher Mystrium

Double-decker is a construction for building a tweakable wide block cipher from deck functions and universal hash functions [19]. It generalizes the Farfalle-WBC construction, a wide block cipher construction proposed by Bertoni et al. [5], and the global construction is an instantiation of the HHFHFH mode [4].

Double-decker, described in Appendix E and illustrated in Fig. 8, operates as a four-round Feistel network with two arbitrary length branches. It involves two Feistel rounds of deck functions surrounded by two rounds of a keyed hash function. It takes as input a key `K`, a tweak W, and a message `P`, and transforms it to a ciphertext `C` the same length as `P`. To prove the security bound of a double-decker construction, the deck functions should have independent keys. Instead of using two keys, it is also possible to use a single key and apply domain separation between the inputs to these functions, as that gives deck functions with two independent inputs.

Mystrium is an instantiation of the double-decker mode, where one of the arbitrary length branches is empty. It uses Multimix as the keyed hash function and Xymmer as the deck function. In Mystrium, a plaintext `P` is split into 3 branches $\texttt{P} = \texttt{T} \mid\mid \texttt{U} \mid\mid \texttt{V}$ with $|T| = |U| = 512$ and $|V| > 0$. This means that Mystrium does not require different keys as the deck functions and the keyed hash functions all have inputs with different length, thus ensuring domain separation. We specify encryption in Mystrium in Algorithm 12 and illustrate Mystrium in Fig. 6. Since Mystrium follows the Feistel construction, the decryption is straightforward and we specify the decryption in Appendix A, Algorithm 14.

8.1 Security claim

Our security claim for Mystrium expresses how hard it is to distinguish it from a randomly chosen tweakable wide block cipher. We base it on filling in our claimed advantage for Xymmer and the ε-Δuniversality of Multimix in the bound for double-decker proven in [19]. We quantify the resources of the adversary with the following metrics:

- N: the amount of computation expressed in the number of evaluations of $\mathcal{G}_{512}[6]$ or equivalent,

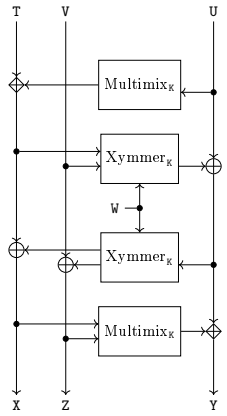


Fig. 6: Specification of Mystrium

Algorithm 12: Encryption in Mystrium

Inputs : Plaintext $\mathtt{P}$: $\mathsf{Bytelen}(\mathtt{P}) > 128$,
Tweak $\mathtt{W}$: $0 < \mathsf{Bytelen}(\mathtt{W}) \leq 63$,
Key $\mathtt{K}$: $32 \leq \mathsf{Bytelen}(\mathtt{K}) \leq 63$
Output: Ciphertext $\mathtt{C}$, same length as $\mathtt{P}$
$\mathtt{T} \leftarrow \mathtt{P}[0{:}512]$
$\mathtt{U} \leftarrow \mathtt{P}[512{:}1024]$
$\mathtt{V} \leftarrow \mathtt{P}[1024{:}]$
$\mathtt{L} \leftarrow \mathsf{sint\_inv}_{8,64}\left(\mathsf{sint}_{8,64}(\mathtt{T}) \boxplus \mathrm{Multimix}_{\mathtt{K}}(\mathtt{U})\right)$
$\mathtt{R} \leftarrow \mathtt{U} \oplus \mathrm{Xymmer}_{\mathtt{K}}(\mathtt{W}; \mathtt{L} \,||\, \mathtt{V}) \lll 0$
$\mathtt{X} \leftarrow \mathtt{L} \oplus \mathrm{Xymmer}_{\mathtt{K}}(\mathtt{W}; \mathtt{R}) \lll 0$
$\mathtt{Z} \leftarrow \mathtt{V} \oplus \mathrm{Xymmer}_{\mathtt{K}}(\mathtt{W}; \mathtt{R}) \lll 512$
$\mathtt{Y} \leftarrow \mathsf{sint\_inv}_{8,64}(\mathsf{sint}_{8,64}(\mathtt{R}) \boxplus \mathrm{Multimix}_{\mathtt{K}}(\mathtt{X} \,||\, \mathtt{Z}))$
$\mathtt{C} \leftarrow \mathtt{X} \,||\, \mathtt{Y} \,||\, \mathtt{Z}$
return $\mathtt{C}$

- M: the total amount of input and output of construction queries expressed in the number of 256-bit blocks,
- q: the number of construction queries,
- $q_{\mathtt{W}}$: the number of construction queries for a given tweak $\mathtt{W}$.

Claim. The advantage of an adversary in distinguishing Mystrium, where $\mathtt{K}$ is sampled from a distribution with min-entropy 256-bits and $M \leq 2^{128}$, from a randomly chosen wide tweakable block cipher is upper bounded by

$$\frac{q^2 + \sum_{\mathtt{W}} q_{\mathtt{W}}^2}{2^{127}} + \frac{N}{2^{254}}\,.$$

Since $\sum_{\mathtt{W}} q_{\mathtt{W}} = q$, it follows that $\frac{q^2+\sum_W q_W^2}{2^{127}} \leq \frac{q^2}{2^{126}}$. Thus the distinguishing advantage for Mystrium is dominated by the term $q^2/2^{126}$, a significant improvement over the claimed security bound of Adiantum, which is dominated by $q^2/2^{104}$ [10, Theorem 1, Section 6.4].

9 Implementation and benchmarks

The Cryptography Extensions including dedicated AES instructions were introduced for the first time as part of the ARMv8-A architecture. However, a substantial number of low-end devices, such as smartphones aimed at developing markets and smartwatches, still have CPUs that lack these extensions. To address this gap, Adiantum, a wide tweakable block cipher, has been introduced yielding satisfactory performance on such platforms.

We wrote optimized code for Multimixer-128, $\mathcal{G}_{512}[r]$, Xymmer, Mystrium, and $\mathsf{Maskgen}_{\kappa}()$ on the 32-bit ARMv7 Cortex-A processors. While designing them we had Advanced SIMD (ASIMD) instruction set in mind to optimize the

performance and efficiency of the computations and at the same time minimally impacting power consumption. Here we used the NEON vector instruction set that is available on the ARM Cortex-A7 processor in the Broadcom BCM2836 chipset used in the Raspberry Pi 2 model B single-board computer. NEON vector operations are designed to simultaneously process multiple data elements of identical size and type within vectors. These data types, which can be signed/unsigned 8-bit, 16-bit, 32-bit, 64-bit, are determined by what is referred to as the arrangement specifier. The NEON vector operations used in the implementations work on data elements of the size of 32-bit, as our input words in the design are defined with the same length.

Our performance evaluation was primarily focused on 4096-byte messages, although we also conducted tests on 512-byte messages and other message sizes. In fact the reason behind selecting these special sizes of message is that that 512-byte disk sectors were standard until the shift to Advanced Format in 2010, with modern large-capacity hard drives and flash drives now predominantly using 4096-byte sectors. In Linux systems, the standard page and file system allocation unit size is also 4096 bytes.

For comparative analysis, Mystrium was evaluated against Adiantum. Table 1 reports the benchmarking results. Mask derivation in not taken into account in the benchmarking of Adiantum. Thus we benchmark Mystrium with and without taking mask derivation into account. We employed the most efficient constant-time implementations available[1] or developed for the platform, with performance-critical sections predominantly written in assembly language, often utilizing NEON instructions.

Algorithm	Input length in bytes	
	512	4096
Mystrium including mask derivation	13.121	9.766
Mystrium	9.969	6.765
Adiantum-XChaCha8-AES (ENC)[a]	17.032	9.652
Adiantum-XChaCha12-AES (ENC)[a]	19.401	11.559
Adiantum-XChaCha20-AES (ENC)[a]	24.862	15.871

[a] mask derivation is not included in the benchmarking.

Table 1: Performance on ARM Cortex-A7 in cycles per byte.

In Fig. 7 (top), we report on the number of cycles required for computations in Xymmer with 32 bytes of input and various output sizes. Both axes are presented on a base-2 logarithmic scale. The blue curve denotes the empirical benchmark results, exhibiting a linear gradual increase in cycles per number of output bytes. The red line represents the theoretical model, where the number of cycles is predicted to increase by a constant, namely base-2 logarithm of 3. It can be seen that for outputs larger than 2^{10} the empirical benchmarking closely follows the theoretical model. Fig. 7 (bottom) presents a similar analysis, this

[1] `https://github.com/google/adiantum/tree/master`

time correlating the number of cycles to the input byte size on a logarithmic scale when the output of Xymmer is 64 bytes. The slope of the theoretical line is $\log_2 1.3$.

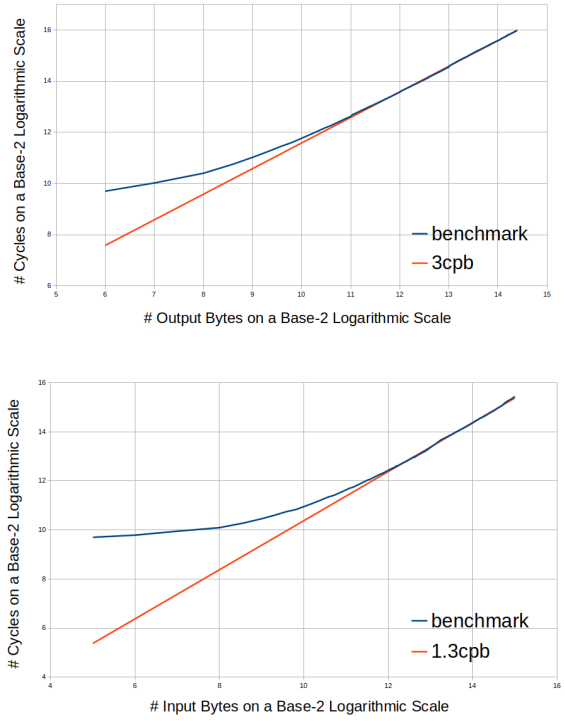


Fig. 7: Benchmarking of Xymmer. Number of cycles plotted against varying output sizes for a single block input (top) and input sizes for a single block output (bottom) measured in bytes.

Software implementations can be found at `https://github.com/parisaeliasi/Mystrium`

10 Conclusion

We propose a tweakable wide block cipher, called Mystrium, that is very efficient on software. Mystrium outperforms the state of the art Adiantum, designed for similar platforms, in both security and efficiency. In order to build Mystrium, we design a deck function called Xymmer that is also very fast on software and is of independent interest. We conduct preliminary cryptanalysis on Xymmer. As future work, it would be interesting to conduct a more fine-grained security analysis of Xymmer.

References

1. The Tor project: Anonymity online, https://www.torproject.org/
2. Bernstein, D.J.: The Poly1305-AES Message-Authentication Code. In: Gilbert, H., Handschuh, H. (eds.) FSE 2005. LNCS, vol. 3557, pp. 32–49. Springer (2005)
3. Bernstein, D.J.: ChaCha, a variant of Salsa20 (Workshop Record of SASC 2008)
4. Bernstein, D.J., Nandi, M., Sarkar, P.: Some challenges in heavyweight cipher design. Presented at Seminar on Symmetric Cryptography, Schloss Dagstuhl (2016), https://cr.yp.to/talks/2016.01.15/slides-djb-20160115-a4.pdf
5. Bertoni, G., Daemen, J., Hoffert, S., Peeters, M., Assche, G.V., Keer, R.V.: Farfalle: parallel permutation-based cryptography. ToSC **2017**(4), 1–38 (2017)
6. Black, J., Halevi, S., Krawczyk, H., Krovetz, T., Rogaway, P.: UMAC: Fast and Secure Message Authentication. In: Wiener, M.J. (ed.) Advances in Cryptology - CRYPTO '99, USA, Proceedings. LNCS, vol. 1666, pp. 216–233. Springer (1999)
7. Chakraborty, D., Ghosh, S., López, C.M., Sarkar, P.: FAST: disk encryption and beyond. Cryptology ePrint Archive (2017)
8. Chakraborty, D., Sarkar, P.: HCH: A new tweakable enciphering scheme using the hash-counter-hash approach. IEEE Trans. Inf. Theory **54**(4), 1683–1699 (2008)
9. Crowley, P.: Mercy: A fast large block cipher for disk sector encryption. In: Schneier, B. (ed.) FSE, 2000, USA. LNCS, vol. 1978, pp. 49–63. Springer (2000)
10. Crowley, P., Biggers, E.: Adiantum: length-preserving encryption for entry-level processors. IACR Trans. Symmetric Cryptol. **2018**(4), 39–61 (2018)
11. Daemen, J., Hoffert, S., Assche, G.V., Keer, R.V.: The design of Xoodoo and Xoofff. IACR Trans. Symmetric Cryptol. **2018**(4), 1–38 (2018)
12. Daemen, J., Mennink, B., Assche, G.V.: Full-state keyed duplex with built-in multi-user support. In: Takagi, T., Peyrin, T. (eds.) ASIACRYPT 2017, China, 2017. vol. 10625, pp. 606–637. Springer (2017)
13. Daemen, J., Rijmen, V.: The Design of Rijndael - The Advanced Encryption Standard (AES), 2nd Edition. Information Security and Cryptography, Springer (2020)
14. Dutta, A., Nandi, M.: Tweakable HCTR: A BBB secure tweakable enciphering scheme. In: Chakraborty, D., Iwata, T. (eds.) INDOCRYPT 2018, India, 2018, Proceedings. vol. 11356, pp. 47–69. Springer (2018)
15. Fuchs, J., Rotella, Y., Daemen, J.: On the security of keyed hashing based on public permutations. In: CRYPTO 2023, USA, Proceedings, Part III. LNCS, vol. 14083, pp. 607–627. Springer (2023)
16. Ghosh, K., Eliasi, P., Daemen, J.: Multimixer-156: Universal keyed hashing based on integer multiplication and cyclic shift. INDOCRYPT 2023, Proceedings (2024)
17. Ghosh, K., Eliasi, P.A., Daemen, J.: Multimixer-128: Universal keyed hashing based on integer multiplication. IACR Trans. Symmetric Cryptol. **2023**(3) (2023)
18. Ghosh, K., Fuchs, J., Eliasi, P.A., Daemen, J.: Universal hashing based on field multiplication and (near-)MDS matrices. In: AFRICACRYPT 2023, Tunisia, Proceedings. LNCS, vol. 14064, pp. 129–150. Springer (2023)
19. Gunsing, A., Daemen, J., Mennink, B.: Deck-Based Wide Block Cipher Modes and an Exposition of the Blinded Keyed Hashing Model. ToSC **2019**(4), 1–22 (2019)
20. Halevi, S., Rogaway, P.: A parallelizable enciphering mode. In: Okamoto, T. (ed.) CT-RSA 2004, USA, Proceedings. LNCS, vol. 2964, pp. 292–304. Springer (2004)
21. Hoang, V.T., Krovetz, T., Rogaway, P.: Robust authenticated-encryption AEZ and the problem that it solves. In: Oswald, E., Fischlin, M. (eds.) EUROCRYPT 2015, Bulgaria, Proceedings, Part I. LNCS, vol. 9056, pp. 15–44. Springer (2015)

22. Knudsen, L.R., Mathiassen, J.E.: On the role of key schedules in attacks on iterated ciphers. In: ESORICS 2004, France. LNCS, vol. 3193, pp. 322–334. Springer (2004)
23. Lefevre, C., Belkheyar, Y., Daemen, J.: Kirby: A robust permutation-based PRF construction. IACR Cryptol. ePrint (2023), https://eprint.iacr.org/2023/1520
24. Liskov, M., Rivest, R.L., Wagner, D.: Tweakable block ciphers. In: Advances in Cryptology—CRYPTO 2002, USA, Proceedings. pp. 31–46. Springer (2002)
25. Luby, M., Rackoff, C.: How to construct pseudorandom permutations from pseudorandom functions. SIAM J. Comput. **17**(2), 373–386 (1988)
26. McGrew, D.A., Fluhrer, S.R.: The security of the extended codebook (XCB) mode of operation. In: SAC 2007, Canada. LNCS, vol. 4876, pp. 311–327. Springer (2007)
27. Minematsu, K., Matsushima, T.: Tweakable enciphering schemes from hash-sum-expansion. In: INDOCRYPT 2007. LNCS, vol. 4859, pp. 252–267. Springer (2007)
28. Naor, M., Reingold, O.: A pseudo-random encryption mode (1997)
29. Schroeppel, R.: Hasty Pudding Cipher Specification (1998), http://richard.schroeppel.name/hpc/hpc-spec
30. Wang, P., Feng, D., Wu, W.: HCTR: A Variable-Input-Length enciphering mode. In: CISC 2005, China, Proceedings. LNCS, vol. 3822, pp. 175–188. Springer (2005)
31. Webster, A.F., Tavares, S.E.: On the design of s-boxes. In: Conference on the theory and application of cryptographic techniques. pp. 523–534. Springer (1985)

A Algorithms

Algorithm 13: $\mathcal{F}$-128($\mathbf{X}$) [17]

Inputs : $\mathbf{X} = (x_0, x_1, x_2, x_3, y_0, y_1, y_2, y_3) \in \left(\mathbb{Z}/2^{32}\mathbb{Z}\right)^8$
Output : $\mathbf{Z} = (z_0, z_1, z_2, z_3, z_4, z_5, z_6, z_7) \in \left(\mathbb{Z}/2^{64}\mathbb{Z}\right)^8$

for $i \leftarrow 0$ **to** 3 **do**
  $u_i \leftarrow x_i \boxplus x_{i+1} \boxplus x_{i+2}$
  $v_i \leftarrow y_{i+1} \boxplus y_{i+2} \boxplus y_{i+3}$
end
for $i \leftarrow 0$ **to** 3 **do**
  $z_i \leftarrow x_i \times y_i$
  $z_{i+4} \leftarrow u_i \times v_i$
end
$\mathbf{Z} \leftarrow (z_0, z_1, z_2, z_3, z_4, z_5, z_6, z_7)$
return $\mathbf{Z}$

Algorithm 14: Decryption in Mystrium($\mathtt{P}, \mathtt{W}, \mathtt{K}$)

Inputs : A ciphertext $\mathtt{C}$
A tweak $\mathtt{W}$ with $0 < \mathsf{Bytelen}(\mathtt{W}) \leq 63$
A secret key $\mathtt{K}$ with $32 \leq \mathsf{Bytelen}(\mathtt{K}) \leq 63$
Output : A plaintext $\mathtt{P}$ such that Mystrium($\mathtt{P}, \mathtt{W}, \mathtt{K}$) = $\mathtt{C}$

$\mathtt{X} \leftarrow \mathtt{C}[0{:}512]$
$\mathtt{Y} \leftarrow \mathtt{C}[512{:}1024]$
$\mathtt{Z} \leftarrow \mathtt{C}[1024{:}]$
$\mathtt{R} \leftarrow \mathsf{sint\_inv}_{8,64}\,(\mathsf{sint}_{8,64}(\mathtt{Y}) \boxplus \text{Multimix}(\mathtt{X} \,||\, \mathtt{Z}))$
$\mathtt{V} \leftarrow \mathtt{Z} \oplus \text{Xymmer}(\mathtt{W}, \mathtt{R}) \ll 512$
$\mathtt{L} \leftarrow \mathtt{X} \oplus \text{Xymmer}(\mathtt{W}, \mathtt{R}) \ll 0$
$\mathtt{U} \leftarrow \mathtt{R} \oplus \text{Xymmer}(\mathtt{W}, \mathtt{L} \,||\, \mathtt{V}) \ll 0$
$\mathtt{T} \leftarrow \mathsf{sint\_inv}_{8,64}\,(\mathsf{sint}_{8,64}(\mathtt{L}) \boxplus \text{Multimix}(\mathtt{U}))$
$\mathtt{P} \leftarrow \mathtt{T} \,||\, \mathtt{U} \,||\, \mathtt{V}$
return $\mathtt{P}$

Algorithm 15: Computation of the avalanche matrix $\mathbf{V}_F$

Parameters: a transformation F over $\mathbb{Z}_2^b$, an input difference Δ and number of samples M.
Output : the avalanche probability matrix $\mathbf{V}_F$.

Initialize a b -bit vector p of probabilities p_i to all zeroes
Initialize a $b \times b$ -bit matrix $\mathbf{V}_F$ of probabilities $v_{i,j}$ to all zeroes
for *M randomly generated states S* **do**
  for *all bits j in the state A* **do**
    Complement bit j of the state A to acquire the new state A_n
    Compute $B = F(A) \oplus F(A_n)$
    for *all state bit positions i* **do**
      $p_i = p_i + B_i/M$
    end
    $\mathbf{V}_F[:j] = p_i$
  end
end
return $\mathbf{V}_F$

B Explanation of the patterns in Fig. 2

Writing out the output after one round of $\mathcal{G}_{512}$ explicitly we have

$$
\begin{aligned}
z_0 &= q_8 \boxplus q_{12} \boxplus q_{13} \boxplus d_0 \boxplus d_4 \boxplus d_5 \,, \\
z_1 &= q_9 \boxplus q_{12} \boxplus q_{13} \boxplus d_1 \boxplus d_4 \boxplus d_5 \,, \\
z_2 &= q_{10} \boxplus q_{14} \boxplus q_{15} \boxplus d_2 \boxplus d_6 \boxplus d_7 \,, \\
z_3 &= q_{11} \boxplus q_{14} \boxplus q_{15} \boxplus d_3 \boxplus d_6 \boxplus d_7 \,, \\
z_4 &= q_8 \boxplus q_9 \boxplus q_{12} \boxplus d_0 \boxplus d_1 \boxplus d_4 \,,
\end{aligned}
$$

$$
\begin{aligned}
z_5 &= q_8 \boxplus q_9 \boxplus q_{13} \boxplus d_0 \boxplus d_1 \boxplus d_5 \ , \\
z_6 &= q_{10} \boxplus q_{11} \boxplus q_{14} \boxplus d_2 \boxplus d_3 \boxplus d_6 \ , \\
z_7 &= q_{10} \boxplus q_{11} \boxplus q_{14} \boxplus d_2 \boxplus d_3 \boxplus d_6 \ , \\
\mathbf{z}_2 &= \mathbf{q}_0 \ , \\
\mathbf{z}_3 &= \mathbf{q}_1 \ .
\end{aligned}
$$

Changing bits only in the input word q_0 propagates to changes in the words d_0, d_1 after the 32-bit multiplication. Thus, if the word q_0 changes, then only the output words z_0, z_1, z_4, z_5, z_8 would change. The words z_0 and z_1 are solely dependent on d_0 and d_1 respectively. As a results, we see only the structures ◺ and ◹ corresponding to these output words respectively in Fig. 2a. On the other hand, z_4, z_5 are dependent on $d_0 \boxplus d_1$ and $z_8 = q_0$. As a result, for the output words z_4, z_5, we see the structure ■ and for the output word z_8 we only see the diagonal structure, ◹ .

The changes in other output words after one round can also be explained similarly. The difference propagation through subsequent rounds in Fig. 2 can be viewed by following the propagation of differences through the rounds. Due to the Feistel structure of $\mathcal{G}_{512}$, we also note that the lower half of $\mathbf{V}_{\mathcal{G}_{512}[r]}$ is the same as the upper half of $\mathbf{V}_{\mathcal{G}_{512}[r-1]}$ for $r \geq 1$ as can be seen in Fig. 2. Finally we see that after 4 rounds the avalanche criterion is satisfied. This means that the output bits of $\mathcal{G}_{512}[6]$ depend non-trivially on the input bits and this gives us a lot of confidence in choosing $\mathcal{G}_{512}[6]$ as the permutation in Xymmer.

C Recovering state $\mathbf{q}^{(i)}$ from a single output block

First we look at $r = 1$. For notational simplicity let, $\mathsf{FF}[1](\mathbf{q}) = \mathbf{z}$ and $\mathbf{w} = \mathcal{G}_{512}[1](\mathbf{q})$. Then we naturally denote $\mathtt{q} = \mathsf{sint\_inv}_{16,32}(\mathbf{q})$ and $\mathtt{w} = \mathsf{sint\_inv}_{16,32}(\mathbf{w})$. We then have, $(\mathtt{q}_0, \mathtt{q}_1, \mathtt{q}_2, \mathtt{q}_3) = \mathsf{sbin}_{4,128}(\mathtt{q})$ and $(\mathtt{w}_0, \mathtt{w}_1, \mathtt{w}_2, \mathtt{w}_3) = \mathsf{sbin}_{4,128}(\mathtt{w})$. Since $\mathcal{G}_{512}[1]$ is a Feistel, $\mathtt{w}_2 = \mathtt{q}_0$ and $\mathtt{w}_3 = \mathtt{q}_1$. So, $\mathsf{FF}[1](\mathbf{q}) = \mathtt{w} \oplus \mathtt{q} = (\mathtt{q}_0 \oplus \mathtt{w}_0, \mathtt{q}_1 \oplus \mathtt{w}_1, \mathtt{q}_0 \oplus \mathtt{q}_2, \mathtt{q}_1 \oplus \mathtt{q}_3)$. The attacker can trivially guess the states $\mathtt{q}_0$ and $\mathtt{q}_1$ to recover all the internal states. But this is expected to require 2^{256} guesses.

Now, let us assume that the attacker only guesses $\mathtt{q}_0$. For every guess of $\mathtt{q}_0$, the attacker obtains $\mathtt{w}_0$ and $\mathtt{q}_2$. But, it is very hard for the attacker to retrieve any more useful information. Indeed for $i \in \{0, 1, 2, 3\}$, let $\mathsf{sint}_{4,32}(\mathtt{w}_i) = \mathbf{w}_i = (w_{4i}, w_{4i+1}, w_{4i+2}, w_{4i+3})$ and $\mathsf{sint}_{4,32}(\mathtt{q}_i) = \mathbf{q}_i = (q_{4i}, q_{4i+1}, q_{4i+2}, q_{4i+3})$. It can be easily verified that

$$
d_0 \boxminus d_1 = w_0 \boxplus q_9 \boxminus (w_1 \boxplus q_8) \ , \ d_2 \boxminus d_3 = w_2 \boxplus q_{11} \boxminus (w_3 \boxplus q_{10}) \ . \tag{4}
$$

Since the entries at the right hand side of Eq. 4 is known to the attacker, the attacker can learn the difference between the integers representing the first 32 and the last 32-bits of the product $q_i \times q_{i+4}$ for $i = 0, 1$. But, we could not find any attack that can exploit this information without guessing d_{2i} or d_{2i+1}, while guessing each of d_{2i} or d_{2i+1} increases the cost of the attack substantially. The

attacker can additionally guess part of $\mathtt{q}_1$, but that will only lead to information on more such differences and will not help the attacker to find the whole internal state. We could not find any attack that allows the user to obtain the internal state with less than 2^{256} guesses.

D Propagation of modular difference through $\mathcal{G}_{512}[6]$

We denote the modular difference at the output of $\mathcal{G}_{512}[6]$ as: $(\mathbf{x}^{(j,k)}, \mathbf{y}^{(j,k)}) = \mathbf{w}^{(i,j)} \boxminus \mathbf{w}^{(i,k)}$.

In our approach, we look at the backward propagation of the difference $(\mathbf{x}^{(j,k)}, \mathbf{y}^{(j,k)})$ and the forward propagation of the difference $(\mathbf{a}^{(j,k)}, \mathbf{0})$ through different rounds of $\mathcal{G}_{512}[r]$, and we try to meet in the middle to recover the internal state. 32-bit multiplication followed by $\mathsf{64to32}_4()$ is the only source of non-linearity in our functions. While $\mathsf{64to32}_4()$ is a weakly non-linear function [16, Section 3.2], we provide further advantage to the attacker by disregarding its effect in the propagation of the differences. So, we will look at the difference propagations before and after the application of 32-bit multiplications over different rounds.

We first look at $\mathcal{G}_{512}[1]$. It can be easily seen from Fig. 1 that the differences before and after the multiplications are given by $\mathbf{a}$ and $\mathcal{L}^{-1}(\mathbf{x})$ respectively. With the knowledge of the input difference and output difference to the four 32-bit multiplication, the attacker can easily recover the internal states $\mathbf{q}_n^{(i)}$ and $\mathbf{q}_{n+4}^{(i)}$ for $n \in \{0, 1, 2, 3\}$. They can do this by looking at the solution sets to the corresponding differentials to 32-bit multiplication and for most differentials, the solution set is actually expected to be very small [17, Section 4]. This recovers the left limb of the Feistel and by looking at the output, the attacker can easily obtain the right limb to recover the whole internal state.

We now look at $\mathcal{G}_{512}[2]$. The difference before the multiplications in the second round is $\mathbf{y}$ and the difference after the multiplication is $\mathcal{L}^{-1}(\mathbf{x}) \boxminus \mathbf{a}$. Thus, the attacker can apply similar strategy to recover the entire internal state.

For $\mathcal{G}_{512}[3]$, the only useful information that the attacker can recover is that the difference after the multiplication in second round is $\mathcal{L}^{-1}(\mathbf{y}) \boxminus \mathbf{a}$, but does not learn both the input and output differences between multiplications in any round. Knowledge of only the output difference can still be used to restrict the set of all possible input differences to these multiplication, which can lead to an attack. But, the complexity of the attack increases substantially.

Over four rounds or more, the attacker cannot even know the output differences to any multiplication. Thus we claim that even if the attacker has knowledge of difference modulo 2^{32} after $\mathcal{G}_{512}[6]$, which is already infeasible to obtain for the attacker, our construction is still secure.

E Double Decker construction

In this construction, the message $\mathtt{P}$ is first split into the inputs to the two strings $\mathtt{U}$ and $\mathtt{V}$ using a split function. The strings $\mathtt{U}$ and $\mathtt{V}$ are further split into $\mathtt{U}_\mathtt{L}$, $\mathtt{U}_\mathtt{R}$, $\mathtt{V}_\mathtt{L}$

and $\mathtt{V_R}$ with the outside branches length n, so $|\mathtt{U_L}| = |\mathtt{V_R}| = n$. This implies that the input strings should at least have $2n$ bits. The operation of this construction is demonstrated in Fig. 8. For Mystrium, we set $\mathtt{V_L} = \epsilon$ as demonstrated by the dashed lines.

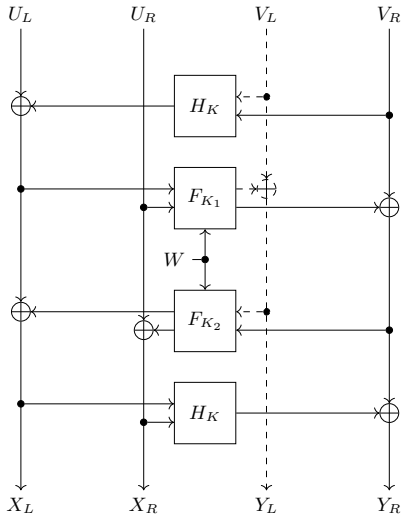


Fig. 8: The double-decker construction [19]