The Subterranean 2.0 cipher suite

Joan Daemen, Pedro Maat Costa Massolino and Yann Rotella

Radboud University, Digital Security Department, Nijmegen

Abstract. This paper presents the Subterranean 2.0 cipher suite that can be used for hashing, MAC computation, stream encryption and several types of session authenticated encryption schemes. At its core it has a duplex object with a 257-bit state and a lightweight single-round permutation. This makes Subterranean 2.0 very well suited for low-area and low-energy implementations in dedicated hardware.

Version of the Subterranean suite: 2.0

Version of this document: 1.1, March 29, 2019

Keywords: lightweight, permutation-based crypto, deck function, XOF function, session authenticated encryption

1 Introduction

Subterranean is a cryptographic primitive to be used both for hashing and as a stream cipher and dates back to 1992 [17, 18]. With some imagination its mode can be seen as a precursor to the sponge [7] with an absorbing phase followed by a squeezing phase.

The round function of Subterranean has features that were adopted in several designs over the last three decades, including Keccak-p [12] and Xoodoo [21]. Namely, all its steps, except the addition of a constant, are bit-level shift-invariant operations, its non-linear step is χ, the mixing step is a lightweight bit-oriented mapping with a heavy inverse and it has a bit transposition step.

But the Subterranean round function also differs from Keccak-p and Xoodoo in important ways. Namely, its state is essentially one-dimensional rather than 3-dimensional, due to the particular transposition and the 257-bit state, it is not software-friendly, and it has a buffer, similar to the *belt* in belt-and-mill designs such as RadioGatún [6].

Despite the differences, it only takes some minor refurbishing to turn Subterranean into a lightweight symmetric cipher suite that can compete with new designs, at least when implemented in dedicated hardware. Refurbishing we did, and we call the result Subterranean 2.0. In short, it is Subterranean with the buffer removed and the hashing and stream encryption modes replaced by a duplex-based construction with modes on top, inspired by Xoodyak [19]. The result is very efficient in hardware but not suited for software. We believe this makes sense in resource-constrained platforms, in particular, when energy per bit is the primary concern and relatively short messages must be protected. The design of Subterranean makes no compromise to be efficient in software, giving it an exceptionally good trade-off between safety margin and hardware performance.

Subterranean 2.0 operates on a state of 257 bits. The modernization into a duplex object required updating the output extraction and the input injection. For the former, we have opted to extract a 32-bit string z per duplex call, where each bit of z is the sum of 2 state bits. For the latter, we inject a string σ of up to 32 bits per duplex call in keyed mode and up to 8 bits every two rounds in unkeyed mode. The central function of the duplex object is the application of a permutation to the state, the subsequent injection of the input string and the extraction of the output. On top of this a number of wrapper functions are defined for absorbing arbitrary-length strings, possibly combined with encryption or decryption, for performing blank rounds and for squeezing arbitrary-length strings.

Loyal to the original Subterranean, we chose for the permutation f in duplex to have only one round and so we expect there to be attacks better than generic ones, i.e., those not exploiting the specifics of f. In particular we claim 128 bits of security against multi-target attackers in keyed modes and 112 bits in unkeyed modes.

In Section 2 we specify the Subterranean duplex object, the primitive underlying the schemes we propose and the three cryptographic schemes that are specified as modes of on top of it: an eXtendable Output Function (XOF), a Doubly-Extendable Cryptographic Keyed (deck) function and a Session Authenticated Encryption (SAE) scheme. This is not meant to be exhaustive but covers most use cases: the XOF for hashing, the deck function for MAC computation, stream encryption, key derivation and more sophisticated modes such as those specified in the Xoodoo cookbook [19] and the SAE scheme for compact authenticated encryption.

In Section 3 we provide the design rationale of the Subterranean 2.0 cipher suite. In Section 4 we discuss how Subterranean should optimally be implemented. In Section 5 we discuss techniques for software optimizations that have played a role in the choice of bit positions for output and input. Finally, Section 6 discusses parameters to be used in the NIST lightweight competition.

2 Specification of the Subterranean 2.0 suite

We specify the Subterranean 2.0 suite in a bottom-up fashion, starting with the round function, input injection and output extraction in Section 2.1, the Subterranean 2.0 duplex object in Section 2.2, the XOF function in Section 2.3, the deck function in Section 2.4 and the SAE scheme in Section 2.5.

2.1 The round function R, input injection and output extraction

The round function R operates on a 257-bit state and has four steps:

$$\mathrm{R} = \pi \circ \theta \circ \iota \circ \chi \ , \tag{1}$$

where each step is there for a particular purpose: χ for non-linearity, ι for asymmetry, θ for mixing and π for dispersion.

We denote the state as s and its bits as s_i with position index i ranging from 0 to 256, where any expressions in the index must be taken modulo 257. For all $0 \leq i < 257$:

$$\begin{aligned}
\chi: \quad s_i &\leftarrow \quad s_i + (s_{i+1} + 1)s_{i+2} \ , \\
\iota: \quad s_i &\leftarrow \quad s_i + \delta_i \ , \\
\theta: \quad s_i &\leftarrow \quad s_i + s_{i+3} + s_{i+8} \ , \\
\pi: \quad s_i &\leftarrow \quad s_{12i} \ .
\end{aligned}$$

Here the addition and multiplication of state bits are in $\mathbb{F}_2$, and δ_i is a Kronecker delta: $\delta_i = 1$ if $i = 0$ and 0 otherwise. Figure 1 illustrates the round function by the computational graph of a single bit of the state.

At the core of the Subterranean duplex object is a simple (internal) *duplex call* that first applies the Subterranean round function R to the state and then injects a string σ of variable length of at most 32 bits. Before adding it into the state, it pads the string σ to 33 bits with simple padding (10^*) and hence the injection rate is 33 bits. In between duplex calls, one may extract 32-bit strings z from the state, so the extraction rate is 32 bits.

Each of the 32 bits of the extracted output z is constructed as the sum of two state bits. These are taken from 64 fixed positions that are the elements of the multiplicative

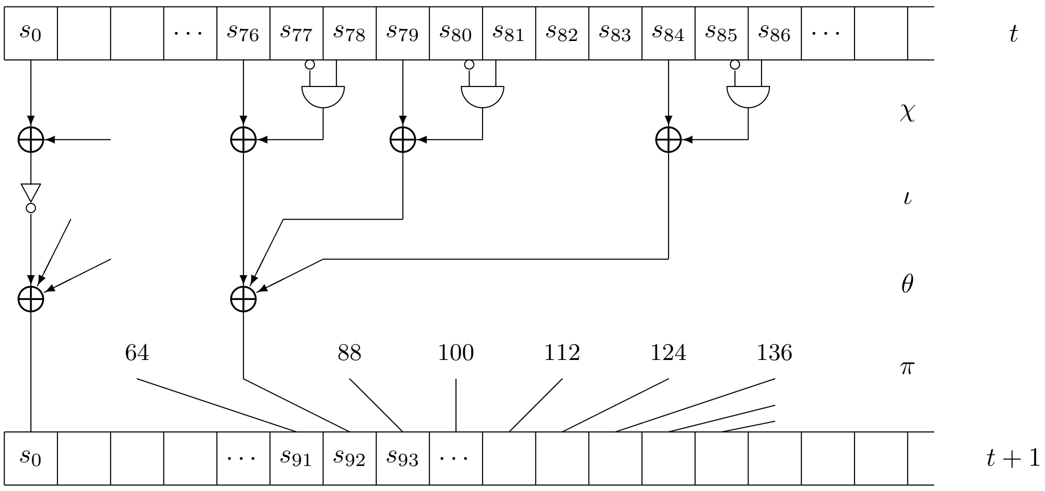

Figure 1: Subterranean round function, illustrated for bit s_{92}

subgroup of order 64 generated by $12^4 = 176$ (see Table 6). We denote this subgroup by $\mathcal{G}_{64}$. More precisely, we have that for all $0 \leq i \leq 31$, $z_i = s_{12^{4i}} + s_{-12^{4i}}$, where the minus sign is taken modulo 257: for instance $-12^0 = -1 = 256$ and $-12^4 = -176 = 81$.

The 33 bits of σ after padding are injected into the state at positions that form the first 33 powers of 12^4 in $\mathcal{G}_{64}$. For the unkeyed mode (hashing), the input σ is limited to 8 bits. This means that only the first 9 bits of padded σ can be non-zero bringing the effective injection rate to 9 bits. Those 9 bits are injected into the state at positions that form the first 9 powers of 12^4 in $\mathcal{G}_{64}$.

2.2 The Subterranean duplex object

The Subterranean duplex object has at its core two internal functions: the duplex call and the output extraction. The duplex call applies the round function and injects the input. Together with the output extraction it is specified in the previous section. On top of the duplex and extraction calls it has a thin wrapper consisting of three functions that facilitate the compact specification of cryptographic functions and schemes on top of it.

The main features of the wrapper are that it supports absorbing and squeezing of strings of arbitrary length and the integration of encryption and decryption with absorbing. It provides separators between absorbed strings by imposing its last injected block is shorter (possibly empty) than 32 bits in keyed mode and 8 in unkeyed mode. We make no security claim for the Subterranean duplex object as such but only for the schemes that consist of modes on top of the primitive.

We specify the Subterranean duplex object in Algorithm 1 and use the following conventions. Any input or output is a bit string, unless specified otherwise. We indicate the length of a bit string X by $|X|$ and the empty string by ϵ.

2.3 The Subterranean-XOF function

We specify Subterranean-XOF in Algorithm 2. It is meant to be used for unkeyed hashing and takes as input a sequence of an arbitrary number of arbitrary-length strings $M[i]$, denoted as $M[[n]]$ and returns a bit string of arbitrary length. For Subterranean-XOF we make a flat sponge claim.

Claim 1. *Subterranean-XOF satisfies a flat sponge claim [8] with capacity* 224 *bits.*

Algorithm 1 Subterranean duplex object

```
Interface: Constructor: Subterranean()
  $s \leftarrow 0^{257}$

Interface: $Y \leftarrow \text{absorb}(X, \text{op})$ with $\text{op} \in \{\text{unkeyed}, \text{keyed}, \text{encrypt}, \text{decrypt}\}$
  if op = unkeyed then $w = 8$ else $w = 32$
  Let $x[n]$ be $X$ split in $w$-bit blocks, with last block strictly shorter
  $Y \leftarrow \epsilon$
  for all blocks of $x[n]$ do
    if $\text{op} \in \{\text{encrypt}, \text{decrypt}\}$ then
      temp $\leftarrow x[i] +$ (extract($s$) truncated to |x[i]|)
      $Y \leftarrow Y||\text{temp}$
    if op = decrypt then duplex(temp) else duplex($x[i]$)
    if op = unkeyed then duplex($\epsilon$)
  return $Y$

Interface: blank($r$) with $r$ a natural number
  for $r$ times do duplex($\epsilon$)

Interface: $Z \leftarrow \text{squeeze}(\ell)$ with $\ell$ a natural number
  $Z \leftarrow \epsilon$
  while $|Z| < \ell$ do
    $Z \leftarrow Z||\text{extract}(s)$
    duplex($\epsilon$)
  return $Z$ truncated to $\ell$ bytes

Internal interface: duplex($\sigma$) with $|\sigma| \leq 32$
  $s \leftarrow \text{R}(s)$
  $x \leftarrow \sigma||1||0^{32-|\sigma|}$
  for $j$ from 0 to 32 do $s_{12^{4j}} \leftarrow s_{12^{4j}} + x_j$

Internal interface: $z \leftarrow \text{extract}(s)$
  $z \leftarrow \epsilon$
  for $j$ from 0 to 31 do $z \leftarrow z||(s_{12^{4j}} + s_{-12^{4j}})$
  return $z$
```

Basically, our claim corresponds to a security strength of 112 bits against all attacks that do not apply to a random oracle. The capacity in the claim is 24 bits short of the effective capacity $257 - 9 = 248$ bits to account for possible shortcut attacks. These are attacks that are more efficient than generic ones by exploiting Subterranean-specific properties (see section 3.4)

Algorithm 2 Subterranean-XOF

```
Interface: $Z \leftarrow \text{Subterranean-XOF}(M[[n]], \ell)$ with $M[[n]]$ a string sequence and $\ell$ a
  natural number
  $S \leftarrow \text{Subterranean}()$
  for all strings $M[i]$ in $M[[n]]$ do $S.\text{absorb}(M[i], \text{unkeyed})$
  $S.\text{blank}(8)$
  return $Z \leftarrow S.\text{squeeze}(\ell)$
```

2.4 The Subterranean-deck function

We specify Subterranean-deck in Algorithm 3. It takes as input an arbitrary-length key K and a sequence of an arbitrary number of arbitrary-length strings $M[i]$, denoted as $M[[n]]$ and returns a bit string of arbitrary length. It can readily be used as a stream cipher, a MAC function and for key derivation. The Farfalle paper [5] and the Xoodoo cookbook [19] specify several authenticated encryption modes for deck functions.

We claim Subterranean-deck offers 128 bits of security against adversaries that are limited to 2^{96} data blocks, when it is loaded with min-entropy 128 bits in a single-target settings, and loaded with independent keys with min-entropy $128 + \log_2 \mu$ in a multi-target setting with μ targets.

Subterranean-deck absorbs the key in blocks of 32 bits. This makes it an application of the recent result of Bart Mennink [25]. The bottom line is that, even for a uniform key with length k and an ideal underlying permutation, the success probability of key prediction cannot be proven to be close to $N2^{-k}$ for N operations On the other hand, the absence of this bound does not imply there is an attack with success probability above $N2^{-k}$ and we do not take this into account in our claim.

Claim 2. *The advantage of an adversary in distinguishing an array of μ Subterranean-deck instances loaded with μ independent keys, each with min-entropy $128 + \log_2 \mu$ bits, from an array of μ independent random oracles. is upper bound by $(N + M)2^{-128}$, with N the total computational complexity in calls to the Subterranean round function and M the total data complexity in 32-bit input- and output blocks, with $M \leq 2^{96}$.*

The data limit for the adversary, $M < 2^{96}$ is not likely to pose a problem in the foreseeable future.

Algorithm 3 Subterranean-deck

```
Interface: $Z \leftarrow$ Subterranean-deck($K, M[[n]], \ell$) with $M[[n]]$ a string sequence and $\ell$ a
  natural number
  $S \leftarrow$ Subterranean()
  $S$.absorb($K$, keyed)
  for all strings $M[i]$ in $M[[n]]$ do $S$.absorb($M$, keyed)
  $S$.blank(8)
  return $Z \leftarrow S$.squeeze($\ell$)
```

2.5 The Subterranean-SAE authenticated encryption scheme

We specify Subterranean-SAE in Algorithm 4. It takes a nonce when starting the session and can then encipher and authenticate a sequence of messages each consisting of a plaintext and associated data. Compared to authenticated encryption modes based on Subterranean-deck, Subterranean-SAE has smaller state and is better suited to offer protection against differential power analysis (DPA). In particular, the security is based on the secrecy of a state that evolves during the session rather than a static key. Across sessions, one can derive a fresh key per session using Subterranean-deck. This protects against differential power analysis and provides fine-grained forward secrecy. If one wishes to use the same key for multiple sessions and DPA is a concern, one can absorb the nonce bit per bit, as was proposed by Taha and Schaumont in [28]. By taking as nonce the shortest binary representation of a session counter, this can be quite economical.

For Subterranean-SAE, we basically make the same security claim as for Subterranean-deck with two differences. First, we do not try to distinguish it from a random oracle, but from a random function with the same interface as Subterranean-SAE. Second, we only consider adversaries that respect nonces and that only get an error message when presenting invalid cryptograms for unwrapping.

We claim Subterranean-SAE offers 128 bits of security against adversaries that are limited to 2^{96} data blocks, when it is loaded with min-entropy 128 bits in a single-target settings, and loaded with independent keys with min-entropy $128 + \log_2 \mu$ in a multi-target setting with μ targets.

Claim 3. *Consider an adversary that respects the nonce requirement and that, when presenting invalid cryptograms for unwrapping, only gets an error message. The advantage of such an adversary in distinguishing an array of μ Subterranean-SAE instances loaded with μ independent keys, each with min-entropy $128 + \log_2 \mu$ bits, from an array of μ independent random functions with the same interface is upper bound by $(N + M)2^{-128}$, with N the total computational complexity in calls to the Subterranean round function and M the total data complexity in 32-bit input- and output blocks, with $M \leq 2^{96}$. When taking a 128-bit tag, this results in plaintext confidentiality and message integrity of security strength 128 bits.*

In nonce-misuse scenario's or when unwrapping invalid cryptograms returns more information than a simple error, we make no security claims and an attacker may even be able to reconstruct the secret state. Nevertheless we believe that this would probably a non-trivial effort, both in attack complexity as in ingenuity.

Algorithm 4 Subterranean-SAE, with τ the tag length

```
Interface: start($K, N$)
  $S \leftarrow$ Subterranean()
  $S$.absorb($K$, keyed)
  $S$.absorb($N$, keyed)
  $S$.blank(8)

Interface: $(Y, T) \leftarrow$ wrap($A, X, T'$, op) with op $\in$ {encrypt, decrypt}
  $S$.absorb($A$, keyed)
  $Y \leftarrow S$.absorb($X$, op)
  $S$.blank(8)
  $T \leftarrow S$.squeeze($\tau$)
  if op = decrypt AND ($T' \neq T$) then $(Y, T) = (\epsilon, \epsilon)$
  return $(Y, T)$
```

3 Design Rationale

In this section we give the design rationale for the Subterranean 2.0 cipher suite, following the same canvas as for the specifications in previous section. Before discussing the different aspects more in-depth, we give the design rationale in a nutshell.

3.1 Rationale in a nutshell

The round function operates at bit level with a maximum of symmetry by using very light shift-invariant operations and a minimum of sub-structures by letting it operate on a prime-sized state. When speaking of sub-structures, we are thinking of Matryoshka in Keccak-p [9] or symmetry properties in the ChaCha permutation [4, 22]. In a way, the, Subterranean round function is the nec plus ultra of weak alignment.

We extract the output z from statebits that are in positions distant from each other to make it very hard to reconstruct the secret state from a series of outputs z and to prevent measurable bias in the output stream Z.

Likewise, we inject input strings σ in the state in positions distant from each other to make it infeasible to control difference propagation in the state. In unkeyed absorbing we limit the strings σ in length to 8 bits and we apply two rounds in between input injections. The reason for this is to make the generation of state collisions infeasible.

Subterranean-XOF and Subterranean-deck each have a single absorbing phase followed by a squeezing phase. In between those phases there is a sequence of 8 blank rounds. In Subterranean-deck, these blank rounds are meant to prevent measurable correlations or exploitable differentials between input $M[i]$ and output Z. In Subterranean-XOF they are meant to make the generation of collisions in n-bit outputs, for any $n \leq 224$, infeasible. Similarly, they should prevent the generation of (2nd) pre-images for any n-bit output with $n \leq 112$ in less than 2^n operations.

More generally, in Subterranean-XOF, Subterranean-deck and Subterranean-SAE alike, the blank rounds should make the output Z (whether tag or keystream) depend on all bits of the input in a complex way, as is the case for a random oracle. Finally, the blank rounds should prevent attacks that make use of higher order differentials such as cube attacks by the fact that expressions of state bits as a function of the state 8 rounds ago has high degree and is relatively dense.

The size of the state, 257, fits nicely the ambition to offer a security of 128 bits in keyed modes and 112 bits in unkeyed mode. It is rather small but not too small to fall prey to time-memory-data-precomputation trade-offs.

3.2 The round function

The round function is just taken from the original Subterranean specified in [18], with the buffer addition removed and the non-linear step complemented. In short, it is a classical lightweight bit-oriented wide trail design, with a non-linear layer, a mixing layer, a transposition layer and a (round) constant addition. The former three are shift-invariant and the addition of the constant is just to ensure the round function itself is not shift-invariant. The state is a one-dimensional array of length 257, a prime. This had to be at least 256 as the original Subterranean targeted a security strength of 128 bits. A prime was taken to avoid the existence of exploitable symmetries.

3.2.1 The non-linear layer χ

The non-linear layer is χ, well known from Keccak-p. This is the most sparse shift-invariant mapping of algebraic degree 2 that is invertible when the state has odd length. By sparse we mean that each output bit only depends on few input bits, in this case 3 bits in neighbouring positions. The low degree is an advantage when countermeasures against differential power analysis need to be implemented, such as masking or threshold schemes. In general, computing the inverse of χ requires a recursive procedure and the consequence is that it is dense and has an algebraic degree that is proportionate to the state length. In the case of Subterranean, the inverse of χ has algebraic degree 128. This complexity helps in frustrating cryptanalysts.

3.2.2 The mixing layer θ

The mixing layer θ is similarly sparse: each output bit is the sum of 3 input bits at fixed relative offsets. The Subterranean round function was an improved version of that of the very first wide-trail design, the hash function Cellhash [20]. In Cellhash, the offsets in θ were $-3, 0, 3$: symmetric around 0 and at minimum distance so that each output bit of

$\theta \circ \chi$ depends on 9 bits. Due to this choice of offsets, there is a 2-bit input difference (resp. output mask) that leads to a 2-bit output difference (resp. input mask). This feature can be used to build n-round trails with weight $n2^n$. The choice of offsets in Subterranean avoid this problem: a 2-bit input difference leads to an output difference with at least 4 bits. The symmetry around 0 was abandoned, 3 was kept and 8 was chosen as the smallest value that gives excellent mixing properties.

For studying the algebraic properties of θ, it helps to see the state s as a binary polynomial $\sum_i s_i X^i$. The operation of θ then becomes a modular multiplication:

$$\theta\left(s(X)\right) = s(X)(1 + X^3 + X^8) \bmod (1 + X^{257}) \,.$$

We say $1 + X^3 + X^8$ is the *multiplication polynomial* of the linear shift-invariant mapping θ.

The polynomials $P(X)$ of degree smaller than 257 that are coprime to $1 + X^{257}$ form a group that we will denote by Θ. The modulus $1 + X^{257}$ is the product of $1 + X$ with 16 irreducible polynomials of degree 16 as shown in table 7. As

$$P^{2^n}(X) \bmod \left(1 + X^{257}\right) = P(X^{2^n \bmod 257}) \ ,$$

we have $P^{2^n}(X) \bmod (1+X^{257}) = P(X)$ if n is the order of 2 in $(\mathbb{Z}/257\mathbb{Z}^*, \times)$. The order of 2 happens to be 16, implying that the order of any $P(X) \in \Theta$ divides $2^{16} - 1 = 3 \cdot 5 \cdot 17 \cdot 257$. For $P(X) = 1 + X^3 + X^8$, we verified with sage that the order is $2^{16} - 1$ itself. It follows that θ has the maximum order of any element in Θ. The inverse of θ can be computed as $(1 + X^3 + X^8)^{2^{16}-2}$ and has a Hamming weight of 127. This high diffusion in the backward direction helps in frustrating cryptanalysis.

3.2.3 The transposition π

The transposition layer π puts bits that are 12 positions apart next to each other: $s_i \leftarrow s_{12i}$. This ensures that each state bit depends on 81 bits of the state 2 cycles ago. Dually, it moves neighbouring bits to positions 150 bits apart: $s_{150j} \leftarrow s_j$ as $150 \cdot 12 \bmod 257 = 1$. The result is that a single-bit difference in the state may affect 81 state bits 2 cycles later.

The order of 12 in $(\mathbb{Z}/257\mathbb{Z})^*, \times)$ is 256, or in other words, it is a generator. The consequence is that the order of the transposition π is likewise 256.

3.2.4 The order of the linear layer $\pi \circ \theta$

For understanding the order of the linear layer $\pi \circ \theta$ we use the following observation. For all $i \in \mathbb{Z}$, let $\theta^{(i)}$ be the linear transformation defined as $\theta^{(i)} = \pi^{-i} \circ \theta \circ \pi^i$. Then, for all $n \in \mathbb{N}$, the linear transformation $(\pi \circ \theta)^n$ can be converted into

$$\pi^n \circ \theta^{(n-1)} \circ \theta^{(n-2)} \cdots \theta^{(1)} \circ \theta^{(0)} \ .$$

Indeed, $(\pi \circ \theta)^n = \pi \circ \theta \circ \pi \circ \theta \circ \cdots \circ \pi \circ \theta = \pi^n \circ \pi^{1-n} \circ \theta \circ \pi^{n-1} \circ \pi^{2-n} \circ \theta \circ \cdots \circ \pi^1 \circ \theta$. If we take this expression for $n = 256$, the first term becomes the identity and we obtain a mapping that is the composition of 256 linear shift-invariant mappings, each with its own multiplication polynomial. From this follows that the composed mapping is also a linear shift-invariant mapping and that its order divides $2^{16} - 1$. Hence, the order of $\pi \circ \theta$ divides $2^8(2^{16} - 1)$. We checked the divisors of this integer and it turns out that the order of $\pi \circ \theta$ is 256 and the minimal polynomial of $\pi \circ \theta$ is $1 + X^{256}$.

More in general, we can prove the following lemma.

Lemma 1. *Let θ' be a linear shift-invariant mapping with multiplication polynomial P and let π' be defined as $s_i \leftarrow s_{g \times i}$ and* $\mathsf{ord}(g)$ *the multiplicative order of g in $\mathbb{Z}/257\mathbb{Z}^*$. If* $\mathsf{ord}(g) \geq 16$*, the order of $\pi' \circ \theta'$ divides* $\mathsf{ord}(g)$.

Proof. By using the technique described just above, for all $n \in \mathbb{N}$, $(\pi' \circ \theta')^n$ can be converted into

$$\pi'^n \circ \theta'^{(n-1)} \circ \theta'^{(n-2)} \dots \theta'^{(1)} \circ \theta'^{(0)}$$

where $\theta'^{(i)} = \pi'^{-i} \circ \theta' \circ \pi'^i$. Clearly, for all $i \in \mathbb{N}$, the multiplication polynomial of $\theta'^{(i)}$ is $P(X^{g^i})$ modulo $1 + X^{257}$ (see Section 5).

We have $(\pi')^{\mathsf{ord}(g)} = \mathsf{Id}$. Hence, $(\pi' \circ \theta')^{\mathsf{ord}(g)}$ is a linear shift-invariant mapping with the following polynomial Q.

$$Q = \prod_{i=0}^{n} P(X^{g^i}) \tag{2}$$

As expressed before in Section 3.2.2, $1 + X^{257}$ is the product of $1 + X$ and 16 polynomials of degree 16. We can now show that $Q = 1$, the multiplication polynomial of the identity mapping. Let α be a primitive element of $\mathbb{F}_{2^{257}}$, then if follows immediately from (2) that $Q(\alpha) = Q(\alpha^g) = Q(\alpha^{g^2}) = \dots = Q(\alpha^{g^{\mathsf{ord}(g)-1}})$. As α is a primitive element, all these powers are different elements. Thanks to the Chinese Remainder Theorem, and the factorisation of $(1 + X^{257})$ in polynomials of degree 16 or smaller (see Table 7), the result follows immediately: Q is a constant polynomial. Depending on the Hamming weight of P, we get either a constant term that is zero or one, but as P corresponds to a bijective mapping, we have $Q = 1$. Hence, $(\pi' \circ \theta')^{\mathsf{ord}(g)} = \mathsf{Id}$ if $\mathsf{ord}(g) \geq 16$. □

3.3 The number of blank rounds

The algebraic degree of the Subterranean round function is 2 and for small r the degree of r rounds is 2^r. For larger r, it was observed in, e.g., [15] that the degree lags behind from the point 2^r starts approaching the permutation width b. Still, for $r = 8$ we expect the algebraic degree to be well above 128. Together with the positioning of the input bits and the fact that there are only 32 per round, this makes it unlikely that shortcut attacks based on higher-order differentials, such as cube attacks, can be mounted.

In collision attacks in Subterranean-XOF, a non-zero difference a' in the state before the blank rounds will propagate to a non-zero difference b' in the state after the blank rounds. Due to the 8 rounds, the difference b' will depend in a complicated way on the difference a' and the absolute values of the states before the blank rounds. This implies that the best collision-generating strategy against Subterranean-XOF is to go for an internal collision during the absorbing phase.

For Subterranean-deck, differential attacks require the propagation of a difference across the blank rounds. Thanks to the excellent difference propagation properties across the rounds, we do not expect there to be 8-round differentials with a DP values anywhere near 2^{-96}, the value that would be required for exploiting it with 2^{96} pairs. Similarly, for the same reasons, we do not expect there to be 8-round correlations with amplitude above 2^{-48}, the value that would be required for exploiting it with 2^{96} input-output couples. The same reasoning is true for Subterranean-SAE for nonce-respecting adversaries that do not get unverified deciphered ciphertext. Namely, in these use cases, controllable input and output is separated by 8 blank rounds.

In the following sections we will discuss input-only attacks in the form of state collisions and output-only attacks in the form of state recovery attacks from output only and biases in the output.

3.4 State collisions in unkeyed absorbing

The single-most important security requirement of unkeyed absorbing is that it should be hard to generate state collisions. A state collision occurs when different string sequences $M[[n]]$ and $M'[[n']]$ lead to the same state value.

A string sequence $M[[n]]$ gives rise to a sequence of absorb calls, that each split the string $M[i]$ into 8-bit blocks, pad these blocks with 10^* and then sequentially inject these into the state in a series of duplex calls. The borders between the strings in the sequence are marked by the fact that the last block of each string $M[i]$ before padding is shorter than 8 bits. If the strings $M[i]$ are bit strings, an adversary can hence freely choose 9 bits between two consecutive duplex calls, as there is a blank round between each absorbing phase in unkeyed mode. In the case of byte strings, the adversary can choose from $2^8 + 1$ input values, reducing the number of freely chosen bits per duplex call to $8 + \epsilon$ with ϵ small. Our ambition is to have 112-bit security strength in the more general case of bit strings.

When generating a state collision, this may occur between string sequences $M[[n]]$ and $M'[[n']]$ that give rise to sequences of duplex calls of equal length or of different length. For duplex call sequences of equal length, we can try to find a differential from the input $M[[n]]$ to a zero difference in the state with a high differential probability. For duplex call sequences of different length, one could try to generate a *fixed point*. Those properties are discussed respectively in section 3.4.3 and section 3.4.2. Finally, one can try to find state collisions with a generic attack by just randomly trying inputs and count on the birthday bound for collisions to occur. This is the starting point of the attack explained in the following subsection.

3.4.1 Advanced inner collisions

In absorbing, we call the bit positions where the input is injected the *outer part of the state*. The inner part of the state is formed by the other bit positions. In unkeyed absorbing, the outer part consists of 9 bit positions and the inner part 248 bit positions. In keyed absorbing, the outer part is 33 bits wide and the inner part 224 bits.

In a naive version of the birthday attack, we need to try about $2^{(257+1)/2} = 2^{129}$ inputs to find a collision in the state. This can be reduced to about 2^{124} inputs if we relax the state collision requirement somewhat, by only requiring a collision in the 248-bit inner part of the state. We call this an *inner collision*. An inner collision can readily be converted into a state collision by compensating for the (possible) difference in the 9-bit outer part by choosing the last blocks in the inner-state colliding inputs. The expected number of string sequences that must be tried before an inner collision presents itself is about 2^{125}, taking about $2^{125+1} = 2^{126}$ duplex calls as there are two rounds per 8-bit block.

This is expected workload of a generic attack and decreasing it by a factor 2^{12} by exploiting specifics of the round function would break our security claim for Subterranean-XOF.

As the round function of Subterranean is rather sparse and has a degree of only 2, it is not unthinkable that this could be done. Exploiting specificities of the round function we did, and found a state-collision finding attack that takes roughly 2^{116+1} duplex calls. It is an attack on a weakened variant of Subterranean, where during unkeyed absorbing an input block is absorbed every round. This attack is exactly the reason why we chose to reduce this to one block every two rounds in unkeyed absorbing.

In a first phase of the attack, the *birthday phase*, we compute the states s obtained by absorbing many random input messages and assemble them in what we call the *birthday set*. It is the size of this set that determines the attack complexity.

In a second phase of the attack, we identify pairs of states (s, s') in the birthday set that form what we call an *advanced inner collision*. To form an advanced inner collision, the state values of the pair must satisfy certain equations and for a number of equations involving bits of (s, s') a solution must exist. The generic scheme of the attack is depicted in figure 2.

We will now derive the equations the bits of an advanced inner collision must satisfy. This allows us to estimate the required size of the birthday set and hence the attack complexity. We denote the value of the state(s), right after absorbing the last message

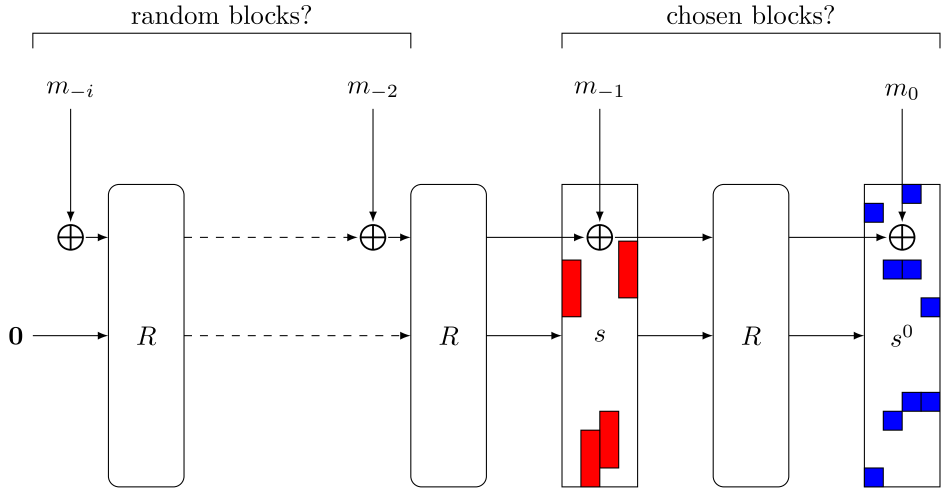

Figure 2: Finding state collisions in unkeyed absorbing. We want a collision on s^0, bits in blue are *input block bits*, only their difference matters. Bits in red represent conditions that can be satisfied by choosing the value of m_{-1}, where m_i's are message blocks concatenated with the padding bit.

blocks m_0 and m_0', s_0 and index the iteration backwards with -1, -2 etc. Our equations are in the bits of s_{-1} and s'_{-1} and for readability we will abbreviate these to simply s and s'.

For the message difference $m_0 \oplus m_0'$, we can choose 9 bits (in blue in Figure 2). This is equivalent to the statement $1 \leq j \leq 248$, $q_j(s) = q_j(s')$, where q_j are quadratic functions defined by the round function R and s and s' are state values in the birthday set. In other words, we express bits in s^0 as functions of the state in s.

An attacker can control 9 bits in both states s and s'. We denote bits of m_{-1} by $b_0, b_1, \ldots, b_7, b_8$ and bits of m'_{-1} by $b'_0, b'_1, \ldots, b'_7, b'_8$ Those bits are injected at positions $1, 176, 136, 35, 249, 134, 197, 234, 64$. Each bit b_i (or b'_i), for $0 \leq i \leq 8$ will appear in 9 equations q_j where q_j are quadratic functions defined above. By doing a Gauss pivot on the 248 equations we can minimize the number of equations where b_i or b'_i appear.

We clarify this by explaining this in detail for equations where bits b_2, b'_2, b_5 and b'_5 intervene. Those bits are respectively injected at position 136 and 134. This yields equations of the form:

$$\begin{cases}
q_{124}(s) + q_{124}(s') &= b_5 s_{133} + b'_5 s'_{133} \\
q_{125}(s) + q_{125}(s') &= b_5 s_{135} + b'_5 s'_{135} \\
q_{126}(s) + q_{126}(s') &= b_5 + b'_5 + b_2 s_{135} + b'_2 s'_{135} \\
q_{127}(s) + q_{127}(s') &= b_2 s_{137} + b'_2 s'_{137} \\
q_{128}(s) + q_{128}(s') &= b_2 + b'_2 \\
q_{129}(s) + q_{129}(s') &= b_5 s_{133} + b'_5 s'_{133} \\
q_{130}(s) + q_{130}(s') &= b_5 s_{135} + b'_5 s'_{135} \\
q_{131}(s) + q_{131}(s') &= b_5 + b'_5 + b_2 s_{135} + b'_2 s'_{135} \\
q_{132}(s) + q_{132}(s') &= b_5 s_{133} + b'_5 s'_{133} + b_2 s_{137} + b'_2 s'_{137} \\
q_{133}(s) + q_{133}(s') &= b_5 s_{135} + b'_5 s'_{135} + b_2 + b'_2 \\
q_{134}(s) + q_{134}(s') &= b_5 + b'_5 + b_2 s_{135} + b'_2 s'_{135} \\
q_{135}(s) + q_{135}(s') &= b_2 s_{137} + b'_2 s'_{137} \\
q_{136}(s) + q_{136}(s') &= b_2 + b'_2
\end{cases}$$

By doing a Gauss pivot, we can say that this system of equations is equivalent to

$$\begin{cases} q'_{124}(s)+q'_{124}(s') &= 0 \\ q'_{125}(s)+q'_{125}(s') &= 0 \\ q'_{126}(s)+q'_{126}(s') &= 0 \\ q'_{127}(s)+q'_{127}(s') &= 0 \\ q'_{128}(s)+q'_{128}(s') &= 0 \\ q'_{129}(s)+q'_{129}(s') &= 0 \\ q'_{130}(s)+q'_{130}(s') &= 0 \\ q'_{131}(s)+q'_{131}(s') &= 0 \\ q'_{132}(s)+q'_{132}(s') &= b_5s_{133}+b'_5s'_{133} \\ q'_{133}(s)+q'_{133}(s') &= b_5s_{135}+b'_5s'_{135} \\ q_{134}(s)+q_{134}(s') &= b_5+b'_5+b_2s_{135}+b'_2s'_{135} \\ q_{135}(s)+q_{135}(s') &= b_2s_{137}+b'_2s'_{137} \\ q_{136}(s)+q_{136}(s') &= b_2+b'_2 \end{cases}$$

where $q'_{133}=q_{133}+q_{136}$, $q'_{132}=q_{135}$, $q'_{131}=q_{131}+q_{134}$, $q'_{130}=q_{130}+q'_{133}$, $q'_{129}=q_{129}+q'_{132}$, $q'_{128}=q_{128}+q_{136}$, $q'_{127}=q_{127}+q_{135}$, $q'_{126}=q_{126}+q_{134}$, $q'_{125}=q_{125}+q'_{133}$ and $q'_{124}=q_{124}+q'_{132}$.

Except for this specific behaviour of bits b_2 and b_5 that are injected at positions 136 and 134, the injected bits yield a system of 3×7 equations, concatenated with equations exclusively in bits of s and s'. The system of equations we obtain (after applying the Gauss pivot and reordering the equations) has the following form

$$\begin{cases} q'_1(s)+q'_1(s') &= b_0+b'_0 \\ q'_2(s)+q'_2(s') &= b_0s_0+b'_0s'_0 \\ q'_3(s)+q'_3(s') &= b_0s_2+b'_0s'_2 \\ q'_4(s)+q'_4(s') &= b_1+b'_1 \\ q'_5(s)+q'_5(s') &= b_1s_{175}+b'_1s'_{175} \\ q'_6(s)+q'_6(s') &= b_1s_{177}+b'_1s'_{177} \\ & \cdots \\ q'_{19}(s)+q'_{19}(s') &= b_7+b'_7 \\ q'_{20}(s)+q'_{20}(s') &= b_7s_{63}+b'_7s'_{63} \\ q'_{21}(s)+q'_{21}(s') &= b_7s_{65}+b'_7s'_{65} \\ q'_{22}(s)+q'_{22}(s') &= b_5s_{133}+b'_5s'_{133} \\ q'_{23}(s)+q'_{23}(s') &= b_5s_{135}+b'_5s'_{135} \\ q_{24}(s)+q_{24}(s') &= b_5+b'_5+b_2s_{135}+b'_2s'_{135} \\ q_{25}(s)+q_{25}(s') &= b_2s_{137}+b'_2s'_{137} \\ q_{26}(s)+q_{26}(s') &= b_2+b'_2 \\ q'_{27}(s) &= q'_{27}(s') \\ q'_{28}(s) &= q'_{28}(s') \\ & \cdots \\ q'_{248}(s) &= q'_{248}(s') \end{cases}$$

where q'_j, for all $0\le j\le 27$, are quadratic functions exclusively in bits of s and s'. The q'_j functions differs from q_j functions as we did a Gauss pivot. Satisfying this system is equivalent to building a collision in the state s^0. So how do we proceed?

We see that the last 221 equations just express equality of quadratic expressions in bits of s and s' respectively. So we first try to find pairs in the birthday set that satisfy these equations. To do so, we can think of storing the birthday set in a hash table, where order the states and corresponding inputs according to the 221-bit values defined by $(q'_{27}(s)||q'_{28}||\cdots||q'_{248})$. These can be seen as coordinates and we call them birthday set coordinates.

When we find a pair (s, s') that has colliding birthday set coordinates, we try to find values of b_i and b'_i for i from 0 to 8 so that the first 26 equations are also satisfied. We now evaluate the probability that such values can be found.

For any i with i different from 2 and 5, bits b_i and b'_i for different i occur in non-overlapping equations and there are 3 equations per couple (b_i, b'_i). An exception to this are equations in bits b_2, b'_2, b_5 and b'_5 that we will address afterwards. We will now work out the case for b_0 and b'_0 that corresponds to the first 3 equations.

- If $s_0 = s'_0$ and $s_2 = s'_2$, then only the difference $b_0 + b'_0$ matters. This difference is uniquely determined by the first equation and this equation can be satisfied by choosing $b_0 + b'_0$. The next two equations are then both satisfied with probability 2^{-2}.

- If $s_0 = s'_0 + 1$ or $s_2 = s'_2 + 1$, then the difference $b_0 + b'_0$ matters for the first equation, but the absolute value also matters. This uniquely determines the value of b_0 and b'_0 by using only 2 equations. The last equation is then satisfied with probability 2^{-1}.

Hence, the first three equations can be satisfied with probability

$$\frac{1}{4} \times \frac{1}{4} + \frac{3}{4} \times \frac{1}{2} = \frac{7}{16}\,.$$

This probability is the same for equations where b_1, b_3, b_4, b_6, b_7 and b_8 intervene.

For the five equations where b_2, b'_2, b_5 and b'_5 intervene, the previous analysis does not hold. To obtain the probability that these equations can be satisfied, we can look into the following 8 different events:

- $s_{137} = s'_{137}$, $s_{135} = s'_{135}$, $s_{133} = s'_{133}$;
- $s_{137} \neq s'_{137}$, $s_{135} = s'_{135}$, $s_{133} = s'_{133}$;
- $s_{137} = s'_{137}$, $s_{135} \neq s'_{135}$, $s_{133} = s'_{133}$;
- $s_{137} = s'_{137}$, $s_{135} = s'_{135}$, $s_{133} \neq s'_{133}$;
- $s_{137} \neq s'_{137}$, $s_{135} \neq s'_{135}$, $s_{133} = s'_{133}$;
- $s_{137} \neq s'_{137}$, $s_{135} = s'_{135}$, $s_{133} \neq s'_{133}$;
- $s_{137} = s'_{137}$, $s_{135} \neq s'_{135}$, $s_{133} \neq s'_{133}$;
- $s_{137} \neq s'_{137}$, $s_{135} \neq s'_{135}$, $s_{133} \neq s'_{133}$.

By using the same arguments as before, the probability of satisfying the five equations can be expressed as

$$\frac{1}{8}\left(\frac{1}{8} + \frac{1}{4} + \frac{3}{8} + \frac{1}{4} + \frac{1}{2} + \frac{1}{2} + \frac{3}{8} + \frac{1}{2}\right) = \frac{23}{64}\,.$$

So given a pair (s, s') with colliding birthday set coordinates, the probability that we can find trailing blocks m_{-1}, m'_{-1} and a difference in m_0 that lead to a state collision is:

$$p' = \left(\frac{7}{16}\right)^7 \left(\frac{23}{64}\right) \approx 2^{-10}\,.$$

Hence, on the average we would need to find about 2^{10} pairs with colliding birthday coordinates. In a birthday set of size 2^w the expected number of pairs that collide in birthday coordinates would be $\binom{2^w}{2}2^{-221} \approx 2^{2w-1-221} = 2^{2w-222}$. Setting this to 2^{10} gives us the required size of the birthday set: 2^{116}. Hence the computational complexity is roughly 2^{116} duplex calls.

The complexity of this attack is probably dominated by storing the states in the hash table in their order of birthday set coordinates. Assuming $n \log(n)$ complexity and skipping over the constant factor, this would correspond to $116 \times 2^{116} \approx 2^{123}$ operations. Moreover, it would require an amount of memory enough to store 2^{116} states each taking a few hundred bits.

In order to reduce the birthday set in this attack on weakened Subterranean unkeyed absorbing to size below 2^{112}, an attacker would have to make use of sets of two trailing blocks m_{-2} and m_{-1} and one final difference. Equations would become of degree 4 and there would be many more equations involving the input bits. However, as the safety margin between the claimed security strengt of 112 bits, and the complexity of this attack is small, we think the reduction of injecting 9 bits every round to 9 bits every two rounds is justified.

To modify this attack for the nominal unkeyed absorbing in Subterranean so that its birthday set would have size below 2^{112}, an attacker would have to construct equations spanning 4 Subterranean rounds. We believe this to be infeasible.

3.4.2 Fixed points

A fixed point consists of a state value s, reached after absorbing a first sequence of input blocks, and a second sequence of input blocks such that after absorbing that second sequence with duplex calls the state has again value s. Such a fixed point would allow generating an infinite set of colliding input sequences. Finding such a fixed point with a generic attack has the same complexity as generically finding a state collision and we believe there are no shortcut attacks that would reduce the expected complexity by a factor 2^8.

3.4.3 Differential properties

For duplex call sequences of equal length one may try to generate an inner collision by exploiting a differential or trail with high differential probability (DP). As an inner collision must be obtained in 248 bits of the state and the adversary can choose 9 bits per duplex call in each of the two input block sequences, it is unlikely that starting from some given state, there exist colliding input sequences of less than $248/(2 \cdot 9) \approx 14$ blocks. Clearly, there are $2^{9 \cdot 14} = 2^{126}$ input block sequences and just trying them all would just be a generic attack. Doing this in less calls in a systematic way would require controlling the propagation of the difference through the rounds and hence having some kind of high probability differential in Subterranean from the input blocks to the state. We believe such differentials do simply not exist.

3.5 State-recovery attacks

Subterranean-SAE is very similar to the CAESAR competition candidate Ketje Jr [10,11] that was attacked last year [23]. This attack is a state recovery attack on a weakened version of Ketje Jr, where the weakening consists of an increase of the rate during the wrap calls from the nominal 16 bits to 32 bits. The attack focuses on 4 consecutive rounds on Ketje Jr v1. The feasibility of the attack strongly depends on the bit positions of the outer part. In Ketje Jr the outer part covers full (5-bit) *rows* and the nonlinear mapping operates at row level. This means that if in a state at the input (resp. output) of χ all bits of a row are known, one can compute the bits in that row at the output (resp. input) of χ. This fact allows an attacker to link the information between 4 consecutive rounds. In Ketje Jr v2 the definition of the outer part was changed and no longer contains full rows, greatly reducing the applicability of the attack.

In Subterranean the squeezing rate is 32 bits, so at first sight the attacks in [23] may be a concern. However, two factors already make it much harder to pull of for Subterranean than for the weakened version of Ketje Jr v1:

- Subterranean has a 257-bit state and Ketje Jr only a 200-bit state.
- In Ketje Jr the χ mapping is applied on *rows* of 5 bits. In Subterranean, χ is applied on all state bits arranged in a single circle.

From the reduction in effectiveness of attacks from Ketje Jr v1 and Ketje Jr v2, we learned that it is a good idea to choose the positions of output bits *far from each other.* Our choice of output positions does exactly that when seen in the context of state-recovery attacks. On top of that, we added another hurdle to frustrate state-recovery attacks: instead of taking as output string z the concatenation of state bits, we construct each of its bits as the sum of 2 state bits. This is inspired by the stream cipher Trivium [16], where the output bit consists of the sum of three state bits. Hence, to obtain 32 output bits, we take 64 state bits and add them in pairs. While information-technically an attacker gets the same amount of information per round as in the case of taking 32 consecutive state bits, it is much harder to exploit this information. In particular, most techniques used in [23] no longer work and it appears that Subterranean has a comfortable safety margin with respect to state recovery attacks.

3.6 Bias in the keystream

In this section we investigate biases in the keystream. Here, we refer to biases using linear combination of output bits such as recently found in AEGIS [26] or MORUS [2]. For Subterranean this would mean the following. We write an output stream Z as a sequence of 32-bit blocks $z_0, z_1, z_2 \dots$ and define a mask sequence U that consists of a sequence of, say, n masks $u_0, u_1, u_2 \dots u_{n-1}$. Each mask is a 32-bit block. We can now compute a parity of a sequence Z using that mask. We write $U^{\mathrm{T}} Z = \sum_{0 \le i < n} u_i^{\mathrm{T}} z_i$. We can *shift* this mask to a later position in Z and compute a similar parity. We write $U^{\mathrm{T}}(Z \ll q) = \sum_{0 \le i < n} u_i^{\mathrm{T}} z_{q+i}$. Each parity is a single bit. If we have a keystream Z of length m, this parity can be computed on $m - n$ positions. If we have multiple keystreams Z of length m_i, this parity can be computed on $\sum_i (m_i - n)$ positions.

For AEGIS and MORUS, masks were found so that $U^{\mathrm{T}} Z$ exhibits a bias, i.e., its value has higher probability to be 0 than 1 or vice versa. This is equivalent to saying that $U^{\mathrm{T}} Z$ is correlated to 0 with positive or negative correlation C. To detect a bias with some given correlation C, one needs about C^{-2} samples, so for example to detect a bias with correlation 1/1000 we need about a million samples.

For any given mask U we can form the algebraic expression of $U^{\mathrm{T}} Z$. In Subterranean, every bit of Z is the sum of two state bits. However, typically U spans multiple blocks and the bits of Z are bits of state values separated by rounds. Still, in principle, one can express each state bit as an algebraic expression of the bits of the state one round earlier. This can be applied recursively and $U^{\mathrm{T}} Z$ can be expressed fully as a sum of monomials each containing bits of the state value used to generate z_0. Now, the value of the correlation of $U^{\mathrm{T}} Z$ with 0 is determined by a property of this algebraic function. Namely, if it contains at least one degree-1 monomial s_i that does not appear in any other monomial, the correlation is 0. In this respect, the choice of the bit positions for generating the output helps in avoiding measurable correlations.

The 64 bit positions used for generating the output are the elements of the multiplicative subgroup $\mathcal{G}_{64}$ of $\mathbb{Z}/257\mathbb{Z}$ of order 64 generated by $12^4 = 176$.

The relations between state bits at time $t+1$ and time t are of the following form:

$$s^{t+1}_{150*i} = s^t_i + \delta_i + s^t_{i+3} + s^t_{i+8} + (s^t_{i+1} + 1)(s^t_{i+2}) + (s^t_{i+4} + 1)(s^t_{i+5}) + (s^t_{i+9} + 1)(s^t_{i+10})\,. \quad (3)$$

So when combining bits of z_{t+1} with bits of z_t, in the expression of each bit of s^{t+1} there is always one bit with position $i \in 12\mathcal{G}_{64}$ (namely s_i^t), while the bits of s^t will be in positions in $\mathcal{G}_{64}$. A bit of z_{t+2} expressed in bits of s^t will contain at least one bit in position $i \in 144\mathcal{G}_{64}$. Finally, the expression of a bit of z_{t+3} will contain at least one bit in position $i \in 176\mathcal{G}_{64}$.

During squeezing, the attacker knows 32 sums of 2 state bits each, taken from positions in the multiplicative subgroup generated by 176. Hence, the attacker knows the value of $s^{t+1}_{176^i} + s^{t+1}_{-176^{-i}}$. But, all those bits belong to $\mathcal{G}_{64}$, and we know that they can be expressed as a function of s^t (see equation 3). We have that for all $0 \leq i \leq 63$,

$$s^{t+1}_{176^i} + s^{t+1}_{-176^i} = s^t_{12*176^i} + s^t_{-12*176^i} + q(s^t)\,.$$

By construction, we know that $12 * 176^i$ and $-12 * 176^i$ do not belong to $\mathcal{G}_{64}$. Those bits remain then unknown and their presence guarantees that linear biases on the keystream can only be found for masks U spanning more rounds. More precisely, any mask U for which $U^{\mathrm{T}}Z$ would exhibit non-zero bias has a span of at least 4 blocks. We believe this eliminates measurable bias in Z.

3.7 Time-Memory-Data Trade-offs

When squeezing an output, Subterranean behaves like a stream cipher and hence it may be subject to Time-Memory trade-off attacks as specified in [3]. Those attacks can recover the internal state given resources $M = T = N/2$ with M the memory complexity and T the time complexity and N the stream cipher state space. As for Subterranean $N = 2^{257}$, these attacks are not a threat for our claimed security strength of 128 bits. In 2000, Biryukov and Shamir [13] improved the trade-off by bringing the data complexity D and computation in the pre-processing phase P into the equation. The invariances of their trade-off become $TP = N$ and $MD = N$. As we limit the data complexity to $D < 2^{96}$ and have $N = 2^{257}$, this still does not jeopardize the claimed security strength of 128 bits.

4 Implementation of Subterranean

We did 3 software implementations and 1 hardware/software co-design for FPGAs or ASICs. The software implementations are:

- reference code in C,
- a clone of the reference code in Python,
- memory-compact code in C.

The hardware/software co-design uses the memory-compact code for the mode layers and a Verilog implementation of the Subterranean permutation and some I/O management.

In the following subsections we will discuss these implementtions.

4.1 Software code

The reference code in C stores each state bit in a byte, thus making it easier to handle the round function π step and performing input injection and output extractions. This code is very close to the specifications in Algorithms 1, 2, 3 and 4 and therefore easier to understand and debug. It does diverges in the way the absorb function is structured. This function is split along the 4 options unkeyed, keyed, encryption and decryption, for understandability.

We wrote a clone of the reference code in Python. We used this code for debugging and now make it available. In this code the state is stored into a list of integers, where each integer is a state bit. Like the C reference code, this code is meant for understanding the Subterranean 2.0 suite, and not for performance.

4.1.1 Memory-compact code

As a proof of concept, we implemented the Subterranean 2.0 suite that packs sets of 8 statebits in bytes, thus showing it is possible to work with less memory. This byte-packing requires all the bit-oriented transformations to be done with the use of bit masks and shifts. While the low-level state handling functions is quite different from the reference code, the mode-level functions for XOF, deck and SAE are quite close to those in the reference C code and Algorithms 2, 3 and 4. However, in the functions of absorb and duplex the architecture is different from the one described in Algorithm 1. This is because in the memory-compact code, all the state handling operations are done in the duplex and extract calls, therefore having a clear separation of the functions that handles the state. This separation makes it easier to make the hardware/software co-design, because the hardware will be the one handling the state directly. Finally, the code only accepts inputs which are byte multiples, as the NIST submission interface only accepts byte multiples.

While the memory compact code will work for any architecture that can process bytes, it is still open to write code for words of 32 or 64 bits.

4.2 Hardware architecture

It makes sense to implement the Subterranean duplex object as a hardware core, and this can be done in different ways. The entire Subterranean 2.0 suite, including XOF, deck and SAE could be done as a hardware core with an internal buffer and state machine or with a hardware/software co-design. In the co-design strategy the serial and communications tasks are usually done by a software on a CPU, and the computation tasks themselves are done in the hardware core. The hardware/software co-design could also be integrated in a SoC, therefore having the state handling operations done in a special circuit, while the communication and serial tasks done in software. We design as a proof of concept a hardware circuit only to handle the state and a software code to handle the communication based on the memory-compact code. In order to better understand, we split the Subterranean algorithm operations into the ones that handle the state and ones that do not.

In the hardware implementation we adopted a different interpretation of the low level Subterranean duplex object from Algorithm 1, as (partly) defined in Algorithm 5. In this alternative definition, duplex is subdivided in duplexSimple, duplexEncrypt and duplexDecrypt, while extract becomes squeezeSimple. In the alternative definitions, the padding responsibility is given to the function caller, while some extra functionality is given to the duplex and extract. The duplexEncrypt and duplexDecrypt are made in order to also perform the output extraction and input injection inside the duplex function, while still not handling the padding. The squeezeSimple performs the extract and the blank duplex call in the Algorithm 1. We made this separation to avoid the conditional execution of the original duplex and to have in the hardware core only simple functions that handle the state. The memory-compact software implementations follows a similar approach, except the padding is handled by the inner functions instead of the caller.

Our hardware and software co-design architecture is split into 4 parts: the permutation round, the registers with a simple interface , the AXI4-Lite slave interface [1] and the entire SoC system.

Figure 3 shows the Subterranean round architecture. This architecture is basically the same as the one shown in Figure 1, the only difference is the addition of the input σ in the θ step. This is done because the 3 input XORs that are done in the θ step can be done

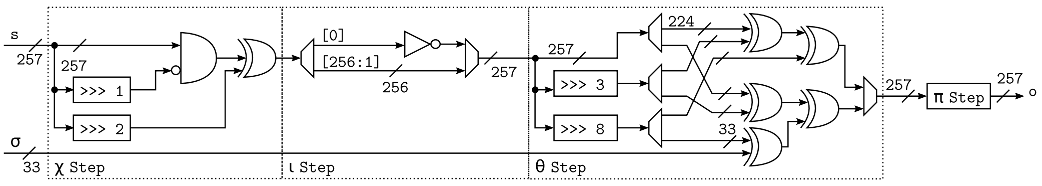

Figure 3: Subterranean round hardware architecture.

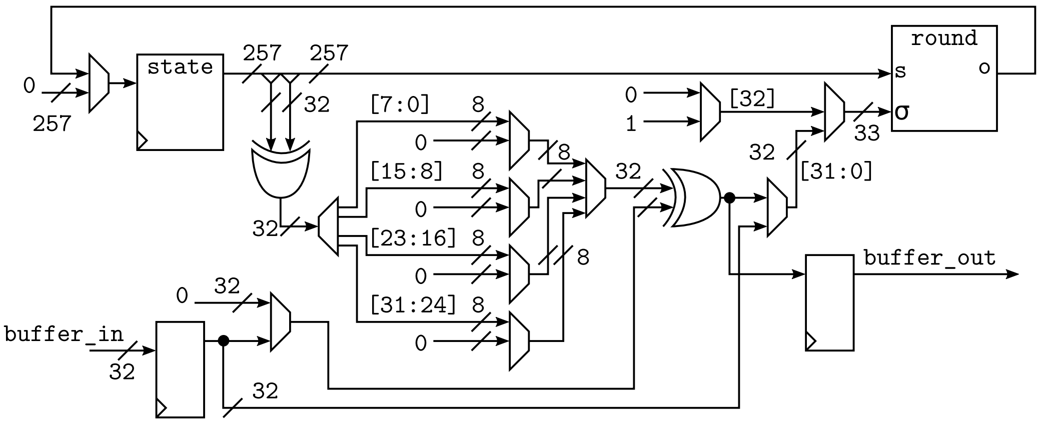

Figure 4: Subterranean round with registers and duplex logic.

together with the buffer addition, thus using 4 input XOR. By adding the input buffer before the π step, the total delay between registers in the round circuit is not affected the addition of the buffer.

Figure 4 adds registers and the functions described in Algorithm 5. There are 3 registers, the 257-bit state buffer and the 32-bit input and output buffers. The state register can be initialized with all zeroes or the evaluation of the round function applied to the content of the state register. The round function will use the value received in buffer in together with the state to compute the duplex empty, duplex simple and duplex encrypt, just as seen in Algorithm 5. The corner cases are with the duplex decrypt and the squeeze simple. In the duplex decrypt last message it is necessary to force some bits of the state to be zero, therefore the padding in "buffer in" is kept and is used during the round function computation. In case of squeeze simple, the value to be added in the round comes from buffer in, and then the circuit force the XOR to happen only with the state and then writes in the buffer out.

Figure 5 adds the components to make the circuit be able to interact with the AXI4-Lite interface. Only the slave interface is done, since the hardware circuit does not need to use an external memory or other components.

In order to differentiate operations and input sizes, the circuit makes use of the address system in the bus. Each operation is tied to a certain address, and the size of the data is also tied to the address. Since for the bus operation is easier to perform read and write operations into direct 32-bit aligned addresses, all commands are 32 bits aligned. Then in order to support up to 5 different byte sizes (empty, 1, 2, 3 and 4) the commands were split into operation on a full buffer (4 bytes) or in a incomplete buffer. In case of the incomplete buffer operations, then we use 2 extra bits to indicate the size in the buffer. So for example, if the address of incomplete buffer encryption is 0x50, then 0x50, 0x54, 0x58 and 0x5C means buffer with 0, 1, 2, 3 bytes respectively.

The entire AXI4-Lite system can be instantiated in a ASIC or FPGA with the same bus and a communication master which should be the CPU. In our case, we tested with

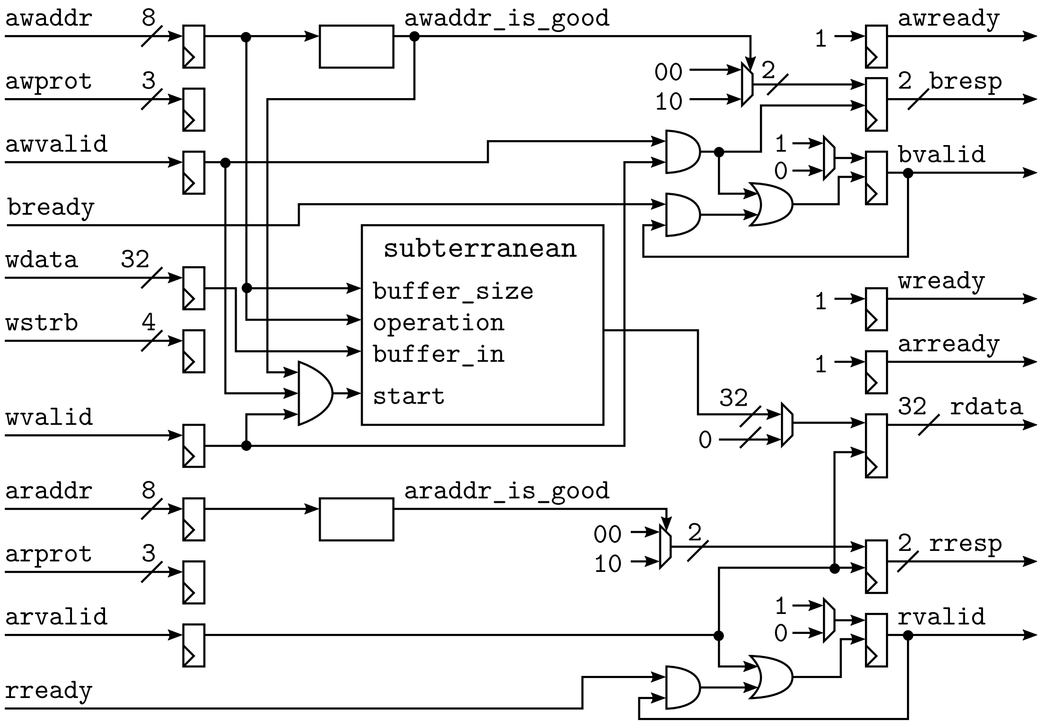

Figure 5: Subterranean simple encapsulated into a AXI4-Lite slave interface.

the Xilinx Zynq SoC, which is a ARM SoC, with external peripherals such as Ethernet, USB, UART etc and the Xilinx FPGA interconnected with different buses, but more importantly an AXI4. With the AXI4 bus interconnecting the FPGA and the CPU, this helps to test and evaluate IP designs that are supposed to be used in a AXI4 bus.

Figure 6 shows how we interconnected the ARM CPU and the Subterranean core. The Zynq has a central interconnect which connects the SoC peripherals, the ARM CPU and the FPGA as well, there are other connections like interrupts which are not shown. In our case the CPU is the master of the AXI4 communication, which then is connected to the AXI4 slave port in the FPGA. Inside the FPGA, we need to instantiate an intermediate component that is the AXI Interconnect. This extra component performs the conversion between the AXI4 and the AXI4-Lite protocol.

If we implemented Subterranean with a AXI4 interface, then we could directly connect to the central interconnect in the chip. However, the AXI4 interface is quite complex, and therefore it could be done for another implementation.

4.3 Hardware results

The circuits described above where tested and described in Verilog language, and also tested on the Zedboard with the Zynq SoC from Xilinx. The tests applied the KAT produced by the software implementation done in C, which are compared with the reference software KAT. The hardware software co-design was done through Vivado 2017.4 tool, where the software runs on the Zynq ARM CPU that sends the hardware commands through the AXI interface in the FPGA. In this implementation we needed 298 Slices that can be split into 763 LUT and 877 flip-flops. The FPGA was set at 200 MHz (5 ns period), but our circuit could theoretically operate up to 217 MHz.

We also synthesized the same circuit in ASIC cells with the open FreePDK 45nm [27] and the open source tool Yosys 0.8 [29]. Yosys does not perform the entire ASIC development

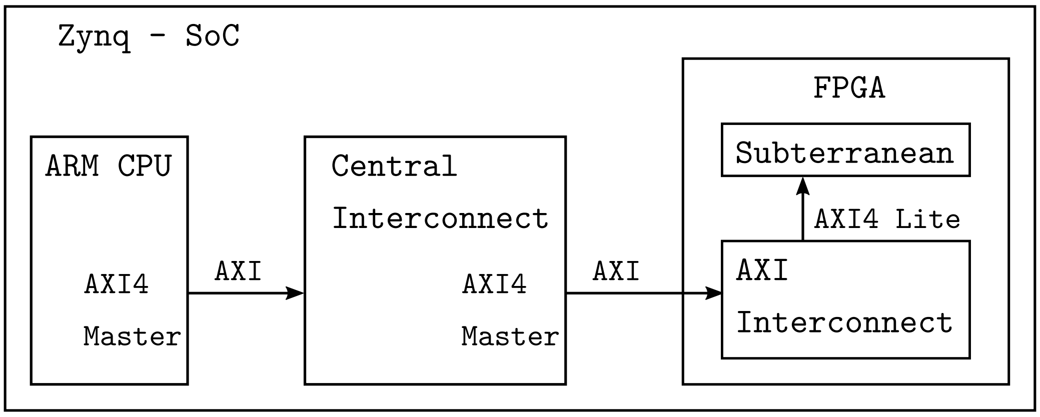

Figure 6: Subterranean core in the Xilinx Zynq FPGA. The communication is bidirectional, the arrows point from communication master to slave.

stack, but gives results for only the gates necessary to implement (therefore no wiring taken into account). It also does not have a fully time oriented synthesis, which tries different combinations and optimizations until it meets the timing requirements. However, in order to have a resources estimation and possible circuit delay this is enough.

The results for the ASIC cells are summarized in Table 1. It shows that by adding the registers plus duplex logic, the total area doubled. Which is easy to verify by knowing a flip-flop with no resets/sets occupies 4.25 GE, while the NAND is only 1 GE [27]. Therefore, serial architectures for this scenario are extremely discouraged, since the registers are the biggest bottlenck, and reducing the round logic will give minor optimization gains. However, maybe the construction of a circuit that can perform 2, 3, 4 or more rounds in one cycle might be preferable, since it will give some latency gains while performing XOF computations.

Table 1 also shows the critical circuit path results for this technology. Because the round function does not have a big dependency the results are less than 1 ns. Thus adding a register between the computations might not be interesting in this case. In a extreme scenario where someone would need to reduce the critical path, they could be added after the ι step.

By inserting the AXI4-Lite interface into our circuit, it increased the amount of resources by approximately 16%. While this is a reasonable amount, if we needed to add a full AXI4 interface in our design, this number would be bigger. Therefore, designers should also taken into account the communication and environment where the solution will be used.

Table 1: FreePDK 45nm [27] area and delay results for Subterranean duplex.

	Area (μm^2)	Area (GE)	Critical path (ns)
Round logic (Fig. 3)	4602	2452	0.290
Duplex logic (Fig. 4)	9161	4880	0.385
AXI4 Lite (Fig. 5)	10655	5676	0.413

4.4 Reference software results

Even though our reference code is a guide in order to test and understand, we still evaluate it in terms of memory consumption and timing. All the results were obtained on the virtual machine running on a Intel Core i5-4570 running at 3.2GHz with Hyperthread on and Windows 7. The virtual machine is configured with 16 GB of RAM, 2 CPUs and

Debian OS, while the host has 32 GB RAM. While obtaining results on a virtual machine might not be ideal, a small comparison test between compiling and running in the VM against the Windows, showed the results has less than 10% penalty.

The stack estimation is done with GCC internal stack usage estimation [14] for each function, with a script to account the theoretically [24] maximum amount. Table 2 shows the results for both reference code stack usage. We can easily see the byte reference code needs almost one third of the bytes in contrast of the bit reference code.

Table 2: Subterranean-XOF, Subterranean-deck and Subterranean-SAE memory consumption in bytes.

	XOF	deck	SAE-encryption	SAE-decryption
Bit Reference	680	680	744	760
Memory compact	248	248	264	296

In case of cycles we got the following results on Tables 3 and 4. From the results we can clearly conclude that encryption and decryption share closer times and the bit reference implementation is bigger, but faster. This raises the doubt if the byte code could be faster than the bit code by optimizing for an architecture word size, like 32 or 64 bits, and how much gain can we get from an direct assembly implementation.

Table 3: Subterranean-XOF time in cycles. “mlen” is message length. The last column is a linear regression of the results.

Subterranean-XOF with 256 bits output					
mlen	1 byte	16 bytes	256 bytes	4096 bytes	x bytes
Bit Ref.	15774	21857	114994	1605706	$15545 + 388x$
Memory compact	12528	28625	311177	4853451	$9866 + 1182x$

Table 4: Subterranean-SAE time in cycles. “mlen” is the plaintext/ciphertext with no tag length, and “adlen” is the associated data length. The last column is a linear regression of the results.

Subterranean-SAE with 128 bits tag				
mlen = 1 byte | adlen =	1 byte	16 bytes	256 bytes	x bytes
Bit Ref. Enc.	30916	30996	44544	$30947 + 52x$
Bit Ref. Dec.	30776	31044	44513	$30971 + 52x$
Memory compact Enc.	21621	21888	61183	$20566 + 157x$
Memory compact Dec.	19244	23072	60887	$20153 + 156x$
mlen = 16 bytes | adlen =	1 byte	16 bytes	256 bytes	adlen = x bytes
Bit Ref. Enc.	30973	31315	44473	$31246 + 52x$
Bit Ref. Dec.	31141	31313	44659	$31075 + 52x$
Memory compact Enc.	22414	25760	61938	$22660 + 154x$
Memory compact Dec.	22836	24126	62675	$22445 + 155x$
mlen = 256 bytes | adlen =	1 byte	16 bytes	256 bytes	x bytes
Bit Ref. Enc.	46230	46293	59605	$45988 + 52x$
Bit Ref. Dec.	47086	47120	60282	$47011 + 52x$
Memory compact Enc.	71252	71887	107683	$69565 + 155x$
Memory compact Dec.	68677	72615	108859	$69155 + 156x$

5 Optimizing software implementation

The choice of the 64 bit positions for extracting output as the elements of $\mathcal{G}_{64}$ is not only motivated by symmetry reasons and optimal diffusion as explained in section 3.6. The choice is also beneficial for optimizing the software performance of Subterranean duplex object.

The steps χ and θ lend themselves reasonably well for software implementations: they can be implemented with shifts and bitwise Boolean instructions. Also ι does not pose a problem. The bit permutation π on the other hand requires the manipulation of individual bits and something that is costly on all CPUs.

5.1 Procrastination of π

In this section we present a technique to avoid implementing π: instead of executing π, we execute variants of χ and θ that operate on the state in an alternative representation that changes every round. Basically, we push π in front of us, hence the name π *procrastination.*

Let γ be shortcut notation for $\theta \circ \iota \circ \chi$. Then two rounds can be expressed as:

$$\mathrm{R}^2 = \pi \circ \gamma \circ \pi \circ \gamma$$

Let now $\gamma^{(1)}$ be defined as $\pi^{-1} \circ \gamma \circ \pi$. Then we have for two rounds:

$$\mathrm{R}^2 = \pi \circ \pi \circ \gamma^{(1)} \circ \pi^{-1} \circ \pi \circ \gamma = \pi^2 \circ \gamma^{(1)} \circ \gamma \ .$$

This can be generalized to any number of rounds. Let $\gamma^{(j)} = \pi^{-j} \circ \gamma \circ \pi^j$, then we have for all $n \in \mathbb{N}$,

$$\mathrm{R}^n = \pi^n \circ \gamma^{(n-1)} \circ \gamma^{(n-2)} \circ \ldots \gamma^{(1)} \circ \gamma^{(0)} \ . \tag{4}$$

So clearly, the π steps can be procrastinated eternally. The remaining question is now: what does $\gamma^{(j)}$ look like? It can be seen as the sequence of three variant functions:

$$\begin{aligned}\gamma^{(j)} &= \pi^{-j} \circ \theta \circ \iota \circ \chi \circ \pi^j \\ &= \pi^{-j} \circ \theta \circ \pi^j \circ \pi^{-j} \circ \iota \circ \pi^j \circ \pi^{-j} \circ \chi \circ \pi^j \\ &= \theta^{(j)} \circ \iota^{(j)} \circ \chi^{(j)} \ ,\end{aligned}$$

with $\theta^{(j)}$, $\iota^{(j)}$ and $\chi^{(j)}$ defined along the same lines as $\gamma^{(j)}$.

All functions $\chi^{(j)}$, $\iota^{(j)}$ and $\theta^{(j)}$ have a simple description. For instance, consider $\chi^{(j)}$.

Let $b = \pi^j(a)$, $c = \chi(b)$ and $d = \pi^{-j}(c)$. We can now directly express d as a function of a.

$y = \pi(x)$ is defined as $y_i = x_{12i}$ So $b_i = a_{12^j i}$ and $d_i = c_{12^{-j} i}$ or equivalently: $c_i = d_{12^j i}$ Substitution in $c_i = b_i + (b_{i+1} + 1)b_{i+2}$ gives:

$$d_{12^j i} = a_{12^j i} + (a_{12^j i + 12^j} + 1)a_{12^j i + 2 \cdot 12^j} \ .$$

We can now write q as a shorthand for $12^j i$, yielding:

$$d_q = a_q + (a_{q+12^j} + 1)a_{q+2 \cdot 12^j} \ .$$

So $\chi^{(j)}$ is simply χ with both offsets 1 and 2 multiplied by 12^j. By doing the same exercise, we have that $\theta^{(j)}$ is θ with both offsets 3 and 8 multiplied by 12^j. ι only operates on the bit in position 0. As this position is not moved by π, ι is not affected by this.

So the consequence is that we have a kind of evolving state representation, implying that all bit positions (except the bit at 0) are moving for every cycle by multiplication by 12 modulo 257.

5.2 Localizing positions in the evolving state

For input injection and output extraction, we need to locate the corresponding bit positions in a register containing the (moving) state. The state, denoted by s^t, is stored in a register R. We indicate the bits of s^t in the usual way, and the bits in the register by $\mathsf{R}[i]$. Using equation (4), we have that for all $0 \leq i \leq 256$ and any $t \in \mathbb{N}$,

$$s_i^t = \mathsf{R}[12^t i]\,.$$

Hence, due to the procrastination, bit i of the state after t rounds can be found in the cell of R with index $12^t i$.

If the input or output positions were unrelated, we would have to locate the positions in R for each bit separately. This would require keeping track of the positions and each time multiplying them by 12 modulo 257. This could be done by a table-lookup in a 256-byte table but still would be an expensive operation.

The positions we need to locate in R are the 64 output bit position and the 33 input bit positions in keyed mode (reduced to 9 bits in unkeyed mode) that are a subset of those 64 positions. Moreover, as explained above, we also have to compute the value of offsets of $\chi^{(t)}$ and $\theta^{(t)}$, i.e., the offsets 1, 2, 3 and 8, multiplied by 12^t at round $t \in \mathbb{N}$.

Our choice of the output and input bit positions simplifies this task. $(\mathbb{Z}/257\mathbb{Z}^*, \times)$ is a group of order 256 and can be generated with 12. Bit positions are evolving by the multiplication by 12. By choosing $\mathcal{G}_{64}$ generated by $12^4 = 176$, the 64 bit positions are evolving in a nice way.

At round 0, bits are $v_{64} = [1, 176, 136, \ldots, 92]$. Those are the elements of $\mathcal{G}_{64}$ and ordered with the successive powers of the generator $12^4 = 176$. At round 1, bit positions are now defined by the coset of $\mathcal{G}_{64}$ multiplied by 12 and ordered in the same way as before, that is $[12, 56, 90, \ldots, 76]$. At round 2, the positions evolve to a second coset of $\mathcal{G}_{64}$, the one multiplied by $12^2 = 144$: $[144, 158, 52, \ldots, 141]$. At round 3, they evolve to the last coset of $\mathcal{G}_{64}$, the one multiplied by $12^3 = 186$: $[186, 97, 110, \ldots, 150]$. At round 4, the cycle returns to v_{64}, but cyclically shifted over one position: $[176, 136, 35, \ldots, 92, 1]$.

In a software implementation, we can hardcode the four 64-byte arrays, leading to a reduction of the workload. Four possible choices would have fit the requirements for security and software performance, $\mathcal{G}_{64}$ and its cosets: $12\mathcal{G}_{64}$, $144\mathcal{G}_{64}$ or even $186\mathcal{G}_{64}$. Nevertheless, we also need to access the value of offsets 1, 2, 3 and 8 multiplied by 12^t at each round $t \in \mathbb{N}$. Offsets 1, 2 and 8 are in $\mathcal{G}_{64}$, so it is natural to choose $\mathcal{G}_{64}$ as bit positions for offsets.

Eventually, we explain how we add the 64 bits to obtain the 32 bits of z. To every bit in a position i in R, we add bit at position $-i$ in R to compute the output bit. By doing so, the shift operation applied on the 64 bit vector commutes with the addition used to obtain the output, leading to another reduction in workload.

Offset 3 does not belong to $\mathcal{G}_{64}$, and we will need this offset to compute θ. $3 * 12^n$ can be computed using either the multiplication by 12 at each round or a lookup table.

Finally, we need to choose either 9 bits or 33 bits for input depending on the unkeyed or keyed mode. In order to lower the complexity, we took the 9 (respectively the 33) first powers of 12^4, that we already need to locate.

6 Parameter choices for the NIST lightweight competition

In this section we specify the set of parameters for both the hash function and the authenticated encryption scheme in the context of the NIST lightweight competition. Both concrete instances are modes on top of the Subterranean duplex object that can be shared.

6.1 Hash function

The hash function of the Subterranean 2.0 suite submitted to the NIST lightweight competition, is the following. Subterranean-XOF, with the input restricted to a string sequence of a single arbitrary-length byte string and the output length fixed to 256. The security claim is given in Claim 1

6.2 Authenticated Encryption

The authenticated encryption scheme of the Subterranean 2.0 suite submitted to the NIST lightweight competition is the following. Subterranean-SAE, with a key K of length 128 bits and a tag length $\tau = 128$ and associated data and plaintext limited to byte strings. The security claim is given in Claim 3 and it implies that the amount of data available to the attacker shall be limited to 2^{96} bits.

6.3 Advantages and limitations

In this section we describe the advantages and limitations of Subterranean 2.0 cipher suite. We discuss first the advantages and then the limitations, in a bottom-up fashion.

We start with the advantages:

- The round function:
  - It is optimized for propagation without compromising for software implementation resulting in an exceptionally good security build up with respect to cost (area, energy)
  - It has a dense inverse
  - It is extremely simple and therefore an attractive target for cryptanalysis
  - It has ultimate weak alignment, ensuring that large classes of attacks cannot be mounted
  - It has low gate delay allowing fast and energy-efficient dedicated hardware implementations
  - It has algebraic degree two, very suitable for protection against DPA such as masking and threshold implementations
  - It has small state while still offering both hashing and authenticated encryption (and more)
  - It is historically important as it clearly is the *mother of all sponges*
- Its architecture:
  - Duplex does not have a fixed key as block ciphers do and hence offers better leakage resilience features
  - The basic duplex object is simple and can be used in other use cases. In other words, the design is modular.
  - The whole cipher suite is so simple that it can be easily memorized.
  - SAE supports session-based authenticated encryption, or seen from a different perspective, supports intermediate tags, to limit the amount of buffering needed at unwrapping end required when not releasing unverified deciphered ciphertext.

The obvious limitations are:

- Operation of duplex is strictly serial.
- The round function is not really suited for software implementations.

7 Acknowledgements

We thank Gilles Van Assche, Michaël Peeters, Seth Hoffert, Ronny Van Keer for inspiring the division in a duplex object and modes on top of it, Bart Mennink for his advice on the provable security aspects and Thomas van der Burgt for useful discussions related to the optimized software implementations.

Joan Daemen and Yann Rotella are supported by the European Research Council under the ERC advanced grant agreement under grant ERC-2017-ADG Nr. 788980 ESCADA.

Pedro Maat Costa Massolino is supported by the Technology Foundation STW (project 13499 - TYPHOON), from the Dutch government.

We thank Xilinx for the Vivado and ISE software licenses and donating the ZedBoard. They were useful to obtain the FPGA results and evaluating the hardware design. Also, we would like to thank the side-channel lab in Radboud University for the infrastructure supplied during the ZedBoard tests.

References

[1] ARM, *AMBA AXI and ACE protocol specification AXI3, AXI4, and AXI4-Lite ACE and ACE-Lite*, 2011.

[2] Tomer Ashur, Maria Eichlseder, Martin M. Lauridsen, Gaëtan Leurent, Brice Minaud, Yann Rotella, Yu Sasaki, and Benoît Viguier, *Cryptanalysis of MORUS*, Advances in Cryptology - ASIACRYPT 2018 - 24th International Conference on the Theory and Application of Cryptology and Information Security, Brisbane, QLD, Australia, December 2-6, 2018, Proceedings, Part II (Thomas Peyrin and Steven D. Galbraith, eds.), Lecture Notes in Computer Science, vol. 11273, Springer, 2018, pp. 35–64.

[3] Steve Babbage, *A space/time trade-off in exhaustive search attacks on stream ciphers*, European Convention on Security and Detection, no. 408, IEEE Conference Publication, 1995.

[4] D. J. Bernstein, *Chacha, a variant of Salsa20*, Workshop Record of SASC 2008, 2008.

[5] G. Bertoni, J. Daemen, S. Hoffert, M. Peeters, G. Van Assche, and R. Van Keer, *Farfalle: parallel permutation-based cryptography*, IACR Trans. Symmetric Cryptol. **2017** (2017), no. 4, 1–38.

[6] G. Bertoni, J. Daemen, M. Peeters, and G. Van Assche, RADIOGATÚN*, a belt-and-mill hash function*, Second Cryptographic Hash Workshop, Santa Barbara, August 2006, http://radiogatun.noekeon.org/.

[7] ______, *Sponge functions*, Ecrypt Hash Workshop 2007, May 2007, also available as public comment to NIST from http://www.csrc.nist.gov/pki/HashWorkshop/Public_Comments/2007_May.html.

[8] ______, *Cryptographic sponge functions*, January 2011, https://keccak.team/files/SpongeFunctions.pdf.

[9] ______, *The* KECCAK *reference*, January 2011, http://keccak.noekeon.org/.

[10] G. Bertoni, J. Daemen, M. Peeters, G. Van Assche, and R. Van Keer, *CAESAR submission:* KETJE *v1.1*, March 2014, https://keccak.team/ketje.html.

[11] ______, *CAESAR submission:* KETJE *v2*, September 2016, https://keccak.team/ketje.html.

[12] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche, *The making of KECCAK*, Cryptologia **38** (2014), no. 1, 26–60.

[13] Alex Biryukov and Adi Shamir, *Cryptanalytic time/memory/data tradeoffs for stream ciphers*, Advances in Cryptology - ASIACRYPT 2000, 6th International Conference on the Theory and Application of Cryptology and Information Security, Kyoto, Japan, December 3-7, 2000, Proceedings (Tatsuaki Okamoto, ed.), Lecture Notes in Computer Science, vol. 1976, Springer, 2000, pp. 1–13.

[14] Eric Botcazou, Cyrille Comar, and Olivier Hainque, *Compile-time stack requirements analysis with gcc*, https://www.adacore.com/uploads/technical-papers/Stack_Analysis.pdf.

[15] Christina Boura, Anne Canteaut, and Christophe De Cannière, *Higher-order differential properties of keccak and* Luffa, Fast Software Encryption - 18th International Workshop, FSE 2011, Lyngby, Denmark, February 13-16, 2011, Revised Selected Papers (Antoine Joux, ed.), Lecture Notes in Computer Science, vol. 6733, Springer, 2011, pp. 252–269.

[16] Christophe De Cannière, *Trivium: A stream cipher construction inspired by block cipher design principles*, Information Security, 9th International Conference, ISC 2006, Samos Island, Greece, August 30 - September 2, 2006, Proceedings (Sokratis K. Katsikas, Javier López, Michael Backes, Stefanos Gritzalis, and Bart Preneel, eds.), Lecture Notes in Computer Science, vol. 4176, Springer, 2006, pp. 171–186.

[17] Luc J. M. Claesen, Joan Daemen, Mark Genoe, and G. Peeters, *Subterranean: A 600 mbit/sec cryptographic VLSI chip*, Proceedings 1993 International Conference on Computer Design: VLSI in Computers & Processors, ICCD '93, Cambridge, MA, USA, October 3-6, 1993, IEEE Computer Society, 1993, pp. 610–613.

[18] J. Daemen, *Cipher and hash function design strategies based on linear and differential cryptanalysis, PhD thesis*, K.U.Leuven, 1995.

[19] J. Daemen, S. Hoffert, G. Van Assche, and R. Van Keer, *Xoodoo cookbook*, IACR Cryptology ePrint Archive **2018** (2018), 767.

[20] Joan Daemen, René Govaerts, and Joos Vandewalle, *A framework for the design of one-way hash functions including cryptanalysis of damgård's one-way function based on a cellular automaton*, Advances in Cryptology - ASIACRYPT '91, International Conference on the Theory and Applications of Cryptology, Fujiyoshida, Japan, November 11-14, 1991, Proceedings (Hideki Imai, Ronald L. Rivest, and Tsutomu Matsumoto, eds.), Lecture Notes in Computer Science, vol. 739, Springer, 1991, pp. 82–96.

[21] Joan Daemen, Seth Hoffert, Gilles Van Assche, and Ronny Van Keer, *The design of xoodoo and xoofff*, IACR Transactions on Symmetric Cryptology **2018** (2018), no. 4, 1–38.

[22] Niels Ferguson, Stefan Lucks, and Kerry A. McKay, *Symmetric states and their structure: Improved analysis of cubehash*, Cryptology ePrint Archive, Report 2010/273, 2010, https://eprint.iacr.org/2010/273.

[23] Thomas Fuhr, María Naya-Plasencia, and Yann Rotella, *State-recovery attacks on modified ketje jr*, IACR Trans. Symmetric Cryptol. **2018** (2018), no. 1, 29–56.

[24] Peter McKinnis, *Worst case stack*, https://github.com/PeterMcKinnis/WorstCaseStack.

[25] Bart Mennink, *Key prediction security of keyed sponges*, IACR Trans. Symmetric Cryptol. **2018** (2018), no. 4, 128–149.

[26] Brice Minaud, *Linear biases in AEGIS keystream*, Selected Areas in Cryptography - SAC 2014 - 21st International Conference, Montreal, QC, Canada, August 14-15, 2014, Revised Selected Papers (Antoine Joux and Amr M. Youssef, eds.), Lecture Notes in Computer Science, vol. 8781, Springer, 2014, pp. 290–305.

[27] James E. Stine, Ivan Castellanos, Michael Wood, Jeff Henson, Fred Love, W. Rhett Davis, Paul D. Franzon, Michael Bucher, Sunil Basavarajaiah, Julie Oh, and Ravi Jenkal, *Freepdk: An open-source variation-aware design kit*, Proceedings of the 2007 IEEE International Conference on Microelectronic Systems Education (Washington, DC, USA), MSE '07, IEEE Computer Society, 2007, pp. 173–174.

[28] M. M. I. Taha and P. Schaumont, *Side-channel countermeasure for SHA-3 at almost-zero area overhead*, 2014 IEEE International Symposium on Hardware-Oriented Security and Trust, HOST 2014, IEEE Computer Society, 2014, pp. 93–96.

[29] Clifford Wolf, *Yosys Open SYnthesis Suite*, http://www.clifford.at/yosys/.

A Tables

Table 5: Input bits for Subterranean duplex object. Bits 0 to 8 are for the unkeyed mode and all bits are taken for the keyed mode.

i	j	i	j	i	j	i	j	i	j
0	1	8	64	16	241	24	4	32	256
1	176	9	213	17	11	25	190		
2	136	10	223	18	137	26	30		
3	35	11	184	19	211	27	140		
4	249	12	2	20	128	28	225		
5	134	13	95	21	169	29	22		
6	197	14	15	22	189	30	17		
7	234	15	70	23	111	31	165		

Table 6: Mapping between state bits and input/output bits.

i	j	i	j	i	j	i	j
0	$(1,256)$	8	$(64,193)$	16	$(241,16)$	24	$(4,253)$
1	$(176,81)$	9	$(213,44)$	17	$(11,246)$	25	$(190,67)$
2	$(136,121)$	10	$(223,34)$	18	$(137,120)$	26	$(30,227)$
3	$(35,222)$	11	$(184,73)$	19	$(211,46)$	27	$(140,117)$
4	$(249,8)$	12	$(2,255)$	20	$(128,129)$	28	$(225,32)$
5	$(134,123)$	13	$(95,162)$	21	$(169,88)$	29	$(22,235)$
6	$(197,60)$	14	$(15,242)$	22	$(189,68)$	30	$(17,240)$
7	$(234,23)$	15	$(70,187)$	23	$(111,146)$	31	$(165,92)$

Table 7: Factors of $1 + X^{257}$

$X + 1$

$X^{16} + X^{12} + X^{11} + X^{8} + X^{5} + X^{4} + 1$

$X^{16} + X^{13} + X^{8} + X^{3} + 1$

$X^{16} + X^{13} + X^{12} + X^{10} + X^{8} + X^{6} + X^{4} + X^{3} + 1$

$X^{16} + X^{14} + X^{12} + X^{11} + X^{8} + X^{5} + X^{4} + X^{2} + 1$

$X^{16} + X^{14} + X^{13} + X^{11} + X^{10} + X^{9} + X^{8} + X^{7} + X^{6} + X^{5} + X^{3} + X^{2} + 1$

$X^{16} + X^{14} + X^{13} + X^{12} + X^{10} + X^{8} + X^{6} + X^{4} + X^{3} + X^{2} + 1$

$X^{16} + X^{14} + X^{13} + X^{12} + X^{11} + X^{9} + X^{8} + X^{7} + X^{5} + X^{4} + X^{3} + X^{2} + 1$

$X^{16} + X^{15} + X^{8} + X + 1$

$X^{16} + X^{15} + X^{13} + X^{9} + X^{8} + X^{7} + X^{3} + X + 1$

$X^{16} + X^{15} + X^{13} + X^{11} + X^{10} + X^{8} + X^{6} + X^{5} + X^{3} + X + 1$

$X^{16} + X^{15} + X^{13} + X^{12} + X^{10} + X^{9} + X^{8} + X^{7} + X^{6} + X^{4} + X^{3} + X + 1$

$X^{16} + X^{15} + X^{14} + X^{8} + X^{2} + X + 1$

$X^{16} + X^{15} + X^{14} + X^{12} + X^{10} + X^{8} + X^{6} + X^{4} + X^{2} + X + 1$

$X^{16} + X^{15} + X^{14} + X^{13} + X^{9} + X^{8} + X^{7} + X^{3} + X^{2} + X + 1$

$X^{16} + X^{15} + X^{14} + X^{13} + X^{11} + X^{10} + X^{8} + X^{6} + X^{5} + X^{3} + X^{2} + X + 1$

$X^{16} + X^{15} + X^{14} + X^{13} + X^{12} + X^{11} + X^{8} + X^{5} + X^{4} + X^{3} + X^{2} + X + 1$

Algorithm 5 Subterranean duplex hardware state handling function

```
**HW Interface:** $s \leftarrow$ Initialize()
  **return** $0^{257}$

**HW Interface:** duplexSimpleFull($x$) with $|x| = 32$
  $s \leftarrow \mathrm{R}(s)$
  **for** $j$ from 0 to 31 **do** $s_{12^{4j}} \leftarrow s_{12^{4j}} + x_j$
  $s_{12^{4\cdot 32}} \leftarrow s_{12^{4\cdot 32}} + 1$

**HW Interface:** duplexSimpleIncomplete($x$) with $|x| = 32$
  $s \leftarrow \mathrm{R}(s)$
  **for** $j$ from 0 to 31 **do** $s_{12^{4j}} \leftarrow s_{12^{4j}} + x_j$

**HW Interface:** $Y \leftarrow$ duplexEncryptFull($x$) with $|x| = 32$
  $z \leftarrow \epsilon$
  **for** $j$ from 0 to 31 **do** $z \leftarrow z||(s_{12^{4j}} + s_{-12^{4j}})$
  $Y \leftarrow x + z$
  $s \leftarrow \mathrm{R}(s)$
  **for** $j$ from 0 to 31 **do** $s_{12^{4j}} \leftarrow s_{12^{4j}} + x_j$
  $s_{12^{4\cdot 32}} \leftarrow s_{12^{4\cdot 32}} + 1$
  **return** $Y$

**HW Interface:** $Y \leftarrow$ duplexEncryptIncomplete($x$) with $|x| = 32$
  $z \leftarrow \epsilon$
  **for** $j$ from 0 to 31 **do** $z \leftarrow z||(s_{12^{4j}} + s_{-12^{4j}})$
  $Y \leftarrow x + z$
  $s \leftarrow \mathrm{R}(s)$
  **for** $j$ from 0 to 31 **do** $s_{12^{4j}} \leftarrow s_{12^{4j}} + x_j$
  **return** $Y$

**HW Interface:** $Y \leftarrow$ duplexDecryptFull($x$) with $|x| = 32$
  $z \leftarrow \epsilon$
  **for** $j$ from 0 to 31 **do** $z \leftarrow z||(s_{12^{4j}} + s_{-12^{4j}})$
  $Y \leftarrow x + z$
  $s \leftarrow \mathrm{R}(s)$
  **for** $j$ from 0 to 31 **do** $s_{12^{4j}} \leftarrow s_{12^{4j}} + Y_j$
  $s_{12^{4\cdot 32}} \leftarrow s_{12^{4\cdot 32}} + 1$
  **return** $Y$

**HW Interface:** $Y \leftarrow$ duplexDecryptIncomplete($x, |\sigma|$) with $|x| = 32$
  $z \leftarrow \epsilon$
  **for** $j$ from 0 to 31 **do** $z \leftarrow z||(s_{12^{4j}} + s_{-12^{4j}})$
  $Y \leftarrow x + z$
  $s \leftarrow \mathrm{R}(s)$
  temp $\leftarrow Y||$($x$ only the last $(32 - |\sigma|)$ bits)
  **for** $j$ from 0 to 31 **do** $s_{12^{4j}} \leftarrow s_{12^{4j}} + \mathrm{temp}_j$
  **return** $Y$

**HW Interface:** $Y \leftarrow$ squeezeSimple($x$)
  $Y \leftarrow \epsilon$
  **for** $j$ from 0 to 31 **do** $Y \leftarrow Y||(s_{12^{4j}} + s_{-12^{4j}})$
  $s \leftarrow \mathrm{R}(s)$
  **for** $j$ from 0 to 32 **do** $s_{12^{4j}} \leftarrow s_{12^{4j}} + x_j$
  **return** $Y$
```