Bachelor thesis
Computer Science

Radboud University

Implementing the NIST lightweight cryptography candidates Xoodyak & Subterranean 2.0

Author:
Thomas van der Burgt
thomas@thvdburgt.nl

First supervisor/assessor:
prof. dr. Joan Daemen
joan@cs.ru.nl

Second supervisor:
Pedro Maat Costa Massolino
P.Massolino@cs.ru.nl

Second assessor:
Yann Rotella
Yann.Rotella@ru.nl

July 9, 2019

Abstract

Xoodyak and Subterranean 2.0 are two cipher suites submitted to the NIST Lightweight Cryptography standardization process. In this thesis, we present implementations based on the specifications of both algorithms aimed at low power devices like the ARM Cortex-M group of chips used in various microcontrollers. To improve the performance of Subterranean 2.0 we were able to make use of an alternative and more efficient way of calculating its round function.

Contents

Chapter 1

Introduction

Combining separate encryption and authentication modes can be challenging and error-prone, especially for developers not amply familiar with cryptography. Because of that, we have seen a rise in Authenticated Encryption (AE) and Authenticated Encryption with Associated Data (AEAD) schemes in recent years. Furthermore we have seen an increase in the number of Internet of Things (IoT) and other interconnected but constrained devices. IoT devices are most commonly low-powered devices that frequently use the internet or other networks to communicate. This combination increases the demand for standardisation of a low-power encryption protocol that is capable of both encryption and authentication.

As of yet, there is no such standard low-power encryption protocol. This led the National Institute of Standards and Technology (NIST) to issue a call for Lightweight Cryptography [1], aiming to develop cryptographic algorithm standards that can work for these low-powered devices. Two of the submissions are the Xoodyak scheme and Subterranean 2.0 cipher suite. Hereinafter, the Subterranean 2.0 cipher suite shall be referred to as solely Subterranean.

To be able to see if an algorithm is suitable for the task it does not only need to provide the security they claim, but they also need to be performant enough for practical purposes. In this thesis we show how we constructed implementations in the C programming language for both Xoodyak and Subterranean and present their measured performance on an ARM Cortex-M3 processor.

1.1 Authenticated Encryption

Using schemes that provide confidentiality alone, the receiver can't be sure the data received is not tampered with by a third party. To provide authentication, allowing the receiver to check the data has been constructed by a sender possessing the key, an authentication scheme can be used together with the encryption scheme. Having to use separate schemes for both functionalities is cumbersome and allows room for mistakes.

The term AE refers to encryption schemes that provide both confidentiality and authentication assurance on the data. It usually is more complex than a scheme that provides only confidentiality or authentication, but it is easier to use and can be used with a single key. AE schemes return not only the encrypted plaintext but also a tag over the entire message that the receiver must use to be able to decrypt.

Nowadays most AE also allow associated data, which is not encrypted, to also be authenticated. These are often called AEAD schemes.

1.2 NIST's lightweight cryptography initiative

April 2018, NIST, a physical sciences laboratory part of the U.S. Department of Commerce, issued a first call for 'Lightweight Cryptography' [1]. With the development of more and more small computing devices being used in recent times, NIST noticed the need for standardisation of lightweight cryptography tailored for resource-limited devices. The call is for submissions for the lightweight cryptography standardisation process.

The criteria [3] for a submission requires the specification and implementation of an algorithm (or family of algorithms) that implement Authenticated Encryption with Associated Data (AEAD). Optionally the submission may also implement hashing functionality.

Chapter 2

Preliminaries

In this chapter, we will introduce various concepts that are prerequisites to fully understand our work. We will briefly describe the sponge construction, duplex construction, and Xoodoo-permutation. All of which consolidate in the Xoodyak primitive. Also, a short overview of the ARM Cortex-M3 architecture onto which our software implementations are tested will be provided.

2.1 The sponge and duplex constructions

The sponge construction and the closely related duplex construction are used as building blocks to implement symmetric cryptography functions. Both are constructed with a fixed-length permutation as underlying primitive.

2.1.1 The sponge construction

A sponge construction [5], shown in Figure 2.1, is a method for creating functions that consume an arbitrary-length input bitstream and produces an arbitrary-length output bitstream. It does this using a finite internal state. These constructions can be used to model or construct cryptographic primitives. The Xoodoo permutation, on which Xoodyak is built, is such a primitive.

Ordinarily, the construction takes three components:

- A state, consisting of $b = r + c$ bits, initialised with all zeroes. The value b is called the width, r is called the bitrate, and c is called the capacity.
- A permutation or transformation $f : \{0,1\}^b \rightarrow \{0,1\}^b$.
- A reversible padding function *pad* that pads the input to a multiple of r bits.

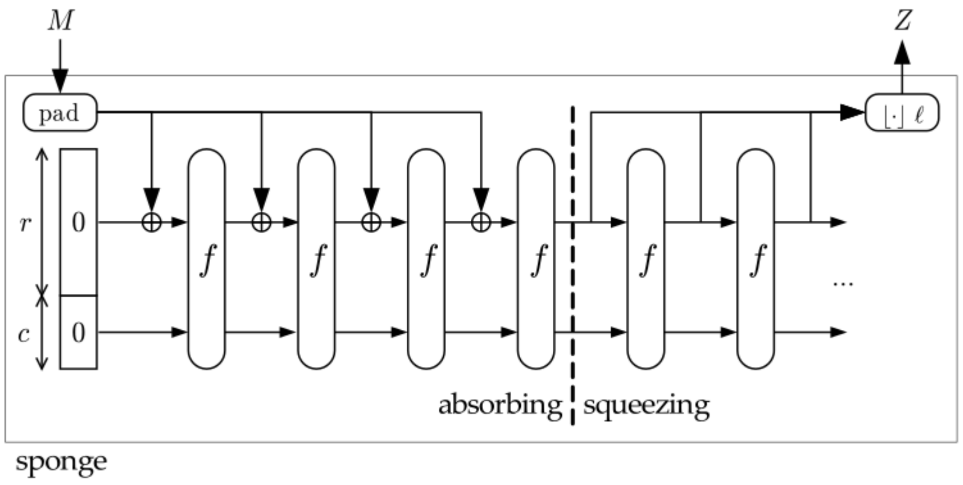

Figure 2.1: The sponge construction

The function resulting from the construction operates in two phases:

absorbing phase	The padded message is split into blocks of r bits, these blocks get XOR-ed into the first r bits of the state followed by an application of f.
squeezing phase	Returns a concatenation of r-bit sized blocks, truncated to l bits. Each block contains the first r bits of the state. After an output block is returned, the permutation f is applied to the state. The length l is chosen by the user.

With its arbitrarily long input and output sizes, the sponge construction allows building various primitives such as a hash function, a stream cipher, or a Message Authentication Code (MAC).

2.1.2 The duplex construction

The duplex construction [8], shown in Figure 2.2, is closely related to the sponge construction. It allows for the alternating input and output of blocks. Unlike a sponge function, which is stateless between calls, the duplex construction results in an object that accepts calls that take an input string and returns an output string that depends on all formerly absorbed inputs.

The security of the duplex construction can be shown to be equivalent to that of the sponge construction [7].

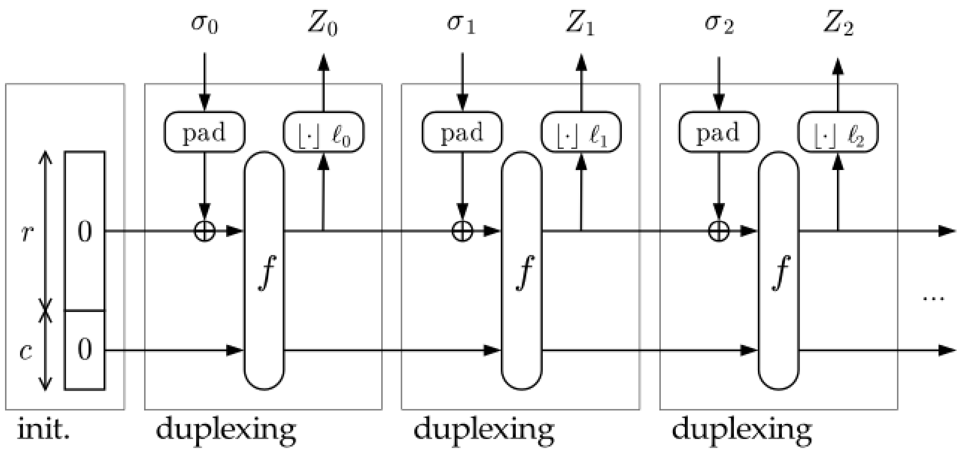

Figure 2.2: The duplex construction

2.2 The Xoodoo permutation

Xoodoo [10] was presented as a lightweight permutation (bijective mapping) with properties more favourable for low-end 32 bit CPUs compared to alternatives like Keccak-f [6] and Gimli [4]. The more heavy-weight Keccak-f uses 16-bit lanes when configured to use the smallest state (400 bits), or when using 32-bit lanes its state of 800 bits is too large to be considered light-weight. Gimli has a small state of 384 bits and lends itself well to 32-bit CPUs but its propagation properties are less than what could be expected.

Xoodoo$[n_r]$ is a family of permutations parameterized by its number of rounds n_r. The state s of Xoodoo consists of 384 bits, which is considered to consist of 3 horizontal *planes* each consisting of 4 parallel 32 bit lanes. Another way the state can be viewed is as a set of 128 *columns* of 3 bits in a 4×32 array.

The planes are index by y, from bottom to top. Lanes within a plane are index by x. Within a lane, bits are indexed by z. All this is illustrated in Figure 2.3, which for clarity uses a state with 8-bit lanes instead of 32-bit lanes.

Xoodoo$[n_r]$ is specified in Algorithm 1, the notation conventions used are specified in Table 2.1 The permutation consists of the iteration of a round function R_i which has 5 steps: a mixing layer θ, a plane shifting ρ_{west}, an addition of round constants ι, a non-linear layer χ, and another plane shifting ρ_{east}. The round constants C_i are planes with a single non-zero lane at $x = 0$, denoted as c_i.

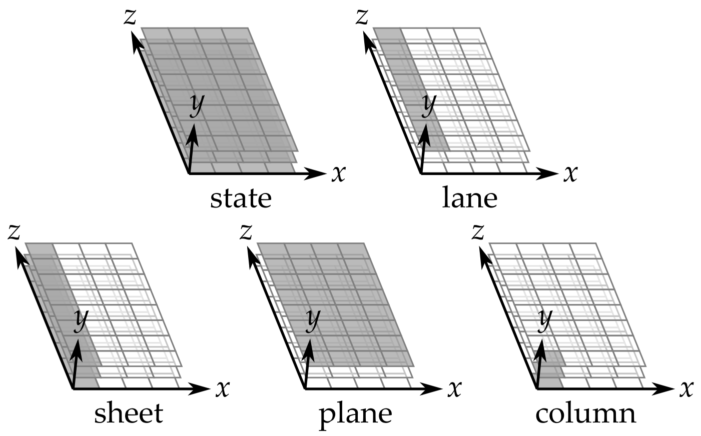

Figure 2.3: Highlighted toy version of the XOODOO state, with lanes reduced to 8 bits.

A_y	Plane y of state A
$A_y \lll (t, v)$	Cyclic shift of A_y moving bit in (x, z) to position $(x + t, z + v)$
$\overline{A_y}$	Bitwise complement of plane A_y
$A_y + A_{y'}$	Bitwise sum (XOR) of planes A_y and $A_{y'}$
$A_y \cdot A_{y'}$	Bitwise product (AND) of planes A_y and $A_{y'}$

Table 2.1: XOODOO notational conventions

Algorithm 1 Definition of XOODOO$[n_r]$ with n_r the number of rounds

```
**Parameters:** Number of rounds $n_r$
**for** Round index $i$ from $1 - n_r$ to 0 **do**
    $A = R_i(A)$
```

Here R_i is specified by the following sequence of steps:

$$
\begin{aligned}
&\theta : \\
&\qquad P \leftarrow A_0 + A_1 + A_2 \\
&\qquad E \leftarrow P \lll (1,5) + P \lll (1,14) \\
&\qquad A_y \leftarrow A_y + E \text{ for } y \in \{0,1,2\} \\
&\rho_{\text{west}} : \\
&\qquad A_1 \leftarrow A_1 \lll (1,0) \\
&\qquad A_2 \leftarrow A_2 \lll (0,11) \\
&\iota : \\
&\qquad A_0 \leftarrow A_0 + C_i \\
&\chi : \\
&\qquad B_0 \leftarrow \overline{A_1} \cdot A_2 \\
&\qquad B_1 \leftarrow \overline{A_2} \cdot A_0 \\
&\qquad B_2 \leftarrow \overline{A_0} \cdot A_1 \\
&\qquad A_y \leftarrow A_y + B_y \text{ for } y \in \{0,1,2\} \\
&\rho_{\text{east}} : \\
&\qquad A_1 \leftarrow A_1 \lll (0,1) \\
&\qquad A_2 \leftarrow A_2 \lll (2,8)
\end{aligned}
$$

2.3 The ARM Cortex-M3 processor

The Advanced Reduced Instruction Set Computer Machine (ARM) Cortex-M3 is a processor in the Cortex-M line of processors primarily intended for microcontrollers. It used to be the case that 8-bit microcontrollers were popular, but the Cortex-M line of processors has been gaining popularity with its 32-bit wide data path, register bank, and memory interface.

The ARM Cortex-M3 processor implements the ARMv7-M architecture which is binary instruction upward compatible with the ARMv7E-M architecture used in the Cortex-M4 and Cortex-M7. There are 13 general-purpose registers and the instruction set consist of instructions from both the 16-bit Thumb-1 instruction set and the 32-bit Thumb-2 instruction set.

As such we see it as a good candidate to test how XOODYAK and Subterranean perform on microcontrollers.

Chapter 3

Xoodyak

XOODYAK [11] is a cryptographic primitive that can be used for hashing, encryption, MAC computation, and authenticated encryption. In this chapter we will describe the workings of XOODYAK and describe the process of implementing it.

3.1 The Xoodyak primitive

XOODYAK is an instance of the so-called Cyclist mode of operation on top of the XOODOO[12] permutation. The Cyclist mode can be seen as a lightweight counterpart to KEYAK's Motorist mode [9]. It is simpler than Motorist, mainly as a result of the absence of parallel variants.

The Cyclist mode of operation has two distinct modes: *hash* and *keyed* mode. Upon initialization with CYCLIST$(K, \mathrm{id}, \mathrm{counter})$, the Cyclist object starts either in *hash* mode if $K = \epsilon$ or in *keyed* mode otherwise. This yields a stateful object to which the user can make calls. It is parameterized by the permutation f, by the block sizes R_{hash}, R_{kin} and R_{kout}, and by the ratchet size ℓ_{ratchet}, all in bytes. R_{hash}, R_{kin}, and R_{kout} specify block sizes of the hash and of the input and output in keyed modes, respectively. There is no way to switch from hash to keyed mode.

XOODYAK uses Cyclist with the following parameters:

- $R_{\mathrm{hash}} = 16$ bytes
- $R_{\mathrm{kin}} = 44$ bytes
- $R_{\mathrm{kout}} = 24$ bytes
- $\ell_{\mathrm{ratchet}} = 16$ bytes

The available functions depend on which mode the object is started: The functions ABSORB() and SQUEEZE() can be called in both hash and keyed mode, whereas the functions ENCRYPT(), DECRYPT(), SQUEEZEKEY(), and

Ratchet() are restricted to keyed mode. The purpose of each function is as follows:

- Absorb(X) absorbs an input string X;
- $C \leftarrow$ Encrypt(P) enciphers P into C and absorbs P;
- $P \leftarrow$ Decrypt(C) deciphers C into P and absorbs P;
- $Y \leftarrow$ Squeeze(ℓ) produces an ℓ-byte output depending on the data absorbed so far;
- $Y \leftarrow$ SqueezeKey(ℓ) works like $Y \leftarrow$ Squeeze(ℓ) but in a different domain, for the purpose of generating a derived key;
- Ratchet() transforms the state in an irreversible way to ensure forward secrecy.

3.1.1 Hash mode

In hash mode, the Cyclist object can absorb input strings and squeeze digests at will. The simplest case goes as follows:

```
Cyclist($\epsilon, \epsilon, \epsilon$)
Absorb($x$)
$h \leftarrow$ Squeeze($n$)
```

Here h will contain an n byte long digest of string x, where n can be chosen by the user.

More complicated cases are possible, for example:

```
Cyclist($\epsilon, \epsilon, \epsilon$)
Absorb($x$)
Absorb($y$)
$h_1 \leftarrow$ Squeeze($n_1$)
Absorb($z$)
$h_2 \leftarrow$ Squeeze($n_2$)
```

Here h_1 is a digest over the two-string sequence $y \circ x$, and h_2 is a digest over $z \circ y \circ x$. The digest is over the exact sequence and not just the concatenation of the absorbed strings.

3.1.2 Keyed mode

In keyed mode, Xoodyak can do stream encryption, MAC computation, and authenticated encryption. For example, to produce a t-byte tag (MAC) over a message M using a key K:

```
$\textsc{Cyclist}(K, \epsilon, \epsilon)$
$\textsc{Absorb}(M)$
$T \leftarrow \textsc{Squeeze}(t)$
```

Encryption is done in a stream cipher-like way, hence it requires a nonce. To encrypt plaintext P with nonce N and key K, we can run the following sequence:

```
$\textsc{Cyclist}(K, \epsilon, \epsilon)$
$\textsc{Absorb}(N)$
$C \leftarrow \textsc{Encrypt}(P)$
```

To decrypt a ciphertext C, we would simply replace the last line with:

```
$P \leftarrow \textsc{Decrypt}(C)$
```

Finally, authenticated encryption can be achieved by a combination of the previous sequences. For instance, to encrypt plaintext P with nonce N, key K, and associated data A, we proceed as follows:

```
$\textsc{Cyclist}(K, \epsilon, \epsilon)$
$\textsc{Absorb}(N)$
$\textsc{Absorb}(A)$
$C \leftarrow \textsc{Encrypt}(P)$
$T \leftarrow \textsc{Squeeze}(t)$
```

Resulting in ciphertext C and authentication tag t.

Ratchet

At any time in keyed mode, the user can call RATCHET(). This causes part of the state to be overwritten with zeroes, thereby making it computationally infeasible to compute the state value before the call to RATCHET(). This forward secrecy and prevents an attacker from recovering the secret key prior to the application of the ratchet.

```
$\textsc{Cyclist}(K, \text{id}, \epsilon)$
$\textsc{Absorb}(N)$
$\textsc{Absorb}(A)$
$C \leftarrow \textsc{Encrypt}(P)$
$\textsc{Ratchet}()$ {either here ...}
$T \leftarrow \textsc{Squeeze}(t)$
$\textsc{Ratchet}()$ {... or here}
```

3.2 Implementation in code

During the research, the XOODYAK specification was still under heavy development. As such, no (public) test vectors were available making it difficult to assess the correctness of our code. To minimise the probability of error and because of the relatively small size of the specification, we decided to construct two separate codebases. This allowed us to compare the results from both and quickly identify discrepancies when they arose.

Both codebases use the publicly available reference code[1] for the XOODOO permutation written in the C programming language. This way all discrepancies found were known to originate from our XOODYAK part of the codebase. And because the reference code is architecture-independent we were not only able to run it on an ARM architecture, but also on the x86–64 architecture we used for developing.

The first codebase used the Rust programming language. To interface with the XOODOO permutation code written in C first a small wrapper was written. This way the Cyclist mode part of XOODYAK was able to be written in pure Rust.

The second and final codebase is written in the C programming language and later became our final product.

[1] https://github.com/XKCP/XKCP/tree/master/lib/low/Xoodoo/Reference

Chapter 4

Subterranean 2.0

Like XOODYAK, Subterranean [12] is a cryptographic primitive. It can be used for both hashing and a stream cipher. Based on a design dating back to 1992 [2] its mode can be seen as a precursor to the sponge construction. Like a sponge construction, it has an absorb phase, followed by a squeezing phase.

Although its design lends it to be efficiently implemented in hardware it is not well suited for software. We describe how we can partly overcome this during implementation at the end of this chapter.

4.1 The Subterranean 2.0 cipher suite

This section introduces the workings of Subterranean, and presents the cryptographic Extendible Output Function (XOF) [13] function, Doubly-Extendable Cryptographic Keyed (deck) [10] function, and Session Authenticated Encryption (SAE) scheme built on top of it.

4.1.1 The state and round function

Subterranean has a one-dimensional state with a size of 257 bits. The round function R operates on this state and consists of four steps:

$$R = \pi \circ \theta \circ \iota \circ \chi$$

We denote the state as s and its bits as s_i with position index i ranging from 0 to 256. Any calculations done on 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 are in $\mathbb{F}_2$, and δ_i is a Kronecker delta:

$$\delta_i = \begin{cases} 1 & \text{if } i = 0 \\ 0 & \text{if } i \neq 0. \end{cases}$$

Figure 4.1 illustrates the round function by the computational graph of a single bit of the state.

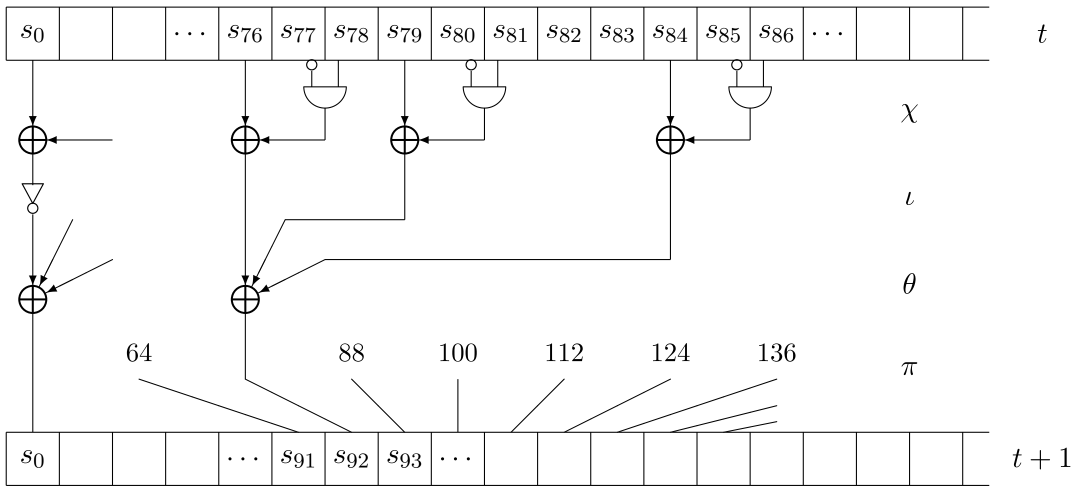

Figure 4.1: Subterranean round function, fully illustrated for bit s_{92}

4.1.2 The Subterranean duplex object

At the core of the Subterranean duplex object are two functions, the *duplex call* and the output extraction. On top of these internal interfaces it has a thin wrapper consisting of three functions.

The duplex call first applies the 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, making the extraction rate 32 bits. Each of the 32 bits of the extracted output z is constructed as the sum of two state bits (see Table 4.1). These are taken from 64 fixed positions that are the elements of the multiplicative subgroup of order 64 generated by $12^4 = 176$. 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 $-12^4 = -176 = 81$.

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 4.1: Mapping between state bits and output bits.

The 33 bits of σ after padding are injected into the state at positions that form the first 33 powers of 12^4 in $\mathcal{G}_{64}$ (see Table 4.2). For the unkeyed mode (hashing), the length of 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 $\mathcal{G}_{64}$. Between injections the round function will be called twice instead of once when in unkeyed 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 4.2: Input bits for Subterranean duplex object. Bits 0 to 8 are for the unkeyed mode and all bits are taken for the keyed mode.

On top of the two core internal functions of the duplex object — the duplex call and the output destruction — is a thin wrapper consisting of three functions. These facilitate a compact specification of cryptographic functions and the schemes on top of it. The main features of the wrapper are supporting the 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 that the last injected block is shorter than (and possibly empty) 32 bits in keyed mode and 8 in unkeyed mode.

The specification of the Subterranean duplex object can be found in Algorithm 2. 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.

Algorithm 2 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 $\leftarrow x[i] + (\text{extract}(s)$ 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 $z$
```

4.1.3 The Subterranean-XOF function

The Subterranean-XOF function (Algorithm 3) 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.

Algorithm 3 Subterranean-XOF

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

4.1.4 The Subterranean-deck function

The Subterranean-deck function (Algorithm 4) takes as input an arbitrary-length key K and a sequence of an arbitrary number of arbitrary-length strings $M[i]$, denoted as $M[[n]]$. It returns a bit string of arbitrary length. It can readily be used as a stream cipher, a MAC function, or for key derivation.

Algorithm 4 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$(\ell)$
```

4.1.5 The Subterranean-SAE authenticated encryption scheme

The Subterranean-SAE object (Algorithm 5) takes a nonce when starting the session and can then be used encipher and authenticate a sequence of messages each consisting of plaintext and associated data. Compared to authenticated encryption modes based on Subterranean-deck, Subterranean-SAE has smaller a 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 DPA and provides fine-grained forward secrecy.

Algorithm 5 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 $(Y, T)$
```

4.2 Procrastination of π

Originally Subterranean was designed as a hardware cipher, as such how it would perform in software was not considered critical. For the Subterranean 2.0 cipher suite submitted to NIST software performance is considered more relevant, and as such techniques to optimise this were explored. The bottleneck of (software) performance in the cipher suite is the round function, therefore this was the focal point when searching for optimisations.

The χ and θ steps of the Subterranean round function lend themselves well for software implementation: they can be implemented using shifts and Boolean instructions. Likewise, the ι step does not pose a problem, it can simply be implemented using a single Boolean instruction. The bit-permutation π, on the other hand, requires the manipulation of individual bits and is hard to get at low cost.

In this section, we present a technique to avoid implementing π altogether. Instead of executing π we execute variants of χ and θ that operate on an alternative state representation. This surrogate state changes with every round. In essence, we push π out in front of us.

4.2.1 The basic concept

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

$$R^2 = \pi \circ \gamma \circ \pi \circ \gamma$$

Now let $\gamma^{(1)}$ be defined as $\pi^{-1} \circ \gamma \circ \pi$. Then

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

This can be generalised to any number of rounds: let $\gamma^{(n)}$ be defined as $\pi^{-n} \circ \gamma \circ \pi^n$, then we have:

$$R^n = \pi^n \circ \gamma^{(n)} \circ \gamma^{(n-1)} \circ \cdots \circ \gamma^{(1)} \circ \gamma^{(0)}$$

So clearly π^n can be procrastinated indefinitely.

Of course, this is of no use if we can't think of an efficient version of $\gamma^{(n)}$. So what does $\gamma^{(n)}$ look like? It can be seen as the sequence of three variant functions:

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

where $\theta^{(n)}$, $\iota^{(n)}$, and $\chi^{(n)}$ are defined along the same lines as $\gamma^{(n)}$:

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

4.2.2 What do $\theta^{(n)}$, $\iota^{(n)}$, and $\chi^{(n)}$ look like?

To answer this question we will first take a look at $\chi^{(n)}$. Let $a = \pi^n(s)$, $b = \chi(a)$, $c = \pi^{-n}(b)$. This way c can be directly expressed as a function on the state s, and $\chi^{(n)} = c$. Using the definitions of π and χ we have:

$$\begin{aligned}
a_i &= s_{12^n i} \\
b_i &= a_i + (a_{i+1} + 1)a_{i+2} \\
c_i &= b_{12^{-n} i}
\end{aligned}$$

From $c_i = b_{12^{-n} i}$ follows that $b_i = c_{12^n i}$. Substitution of this and a_i in $b_i = a_i + (a_{i+1} + 1)a_{i+2}$ gives:

$$c_{12^n i} = s_{12^n i} + (s_{12^n i + 12^n} + 1)s_{12^n i + 2 \cdot 12^n}$$

We can now write q as a shortcut of $12^n i$, yielding:

$$\chi^{(n)} : \quad c_q = s_q + (s_{q+12^n} + 1)s_{q+2\cdot 12^n}$$

So $\chi^{(n)}$ is simply χ with both offset multiplied by 12^n. By repeating the same exercise we can conclude that $\theta^{(n)}$ is θ with both offsets 3 and 8 multiplied by 12^n. Because ι only operates on the bit with index zero whose position is not moved by π it is not affected by this.

4.2.3 Localising the state bits

Rather than working with the regular state where the indices of the state bits are fixed we have a kind of evolving state representation in which these indices move around with every round. This means that for every round with input or output we need to be able to localise the bit indices of these parts of the state.

We know that $s = \pi^n(s')$ where s is the regular state and s' is the evolving state we are working with. Expanding the definition of π we get $s_i = s'_{12^n i}$ where i is the index of a bit in the regular state. Note that 12^n can be pre-computed, this way this operation consists of a lookup and a multiplication. We do have an alternative way of calculating the position of a bit s_i. The multiplicative group of integers modulo 257 is isomorphic with the additive group of integers modulo 256, which is likely a less expensive operation in software. It happens to be the case that 12 is a generator of the multiplicative group modulo 257, so for every index $i > 0$ we have $i = 12^q$. Here the values of q can be pre-computed for all indices. After n rounds: $s_i = s'_{12^n 12^q} = s'_{12^{q+n}}$. So we can compute the indices after n rounds by adding a constant n and then exponentiation (or, as noted before, a lookup). Which of the two approaches is faster is not immediately clear.

4.2.4 Applying an offset to the state

In computer hardware, we most likely can't operate on the state as a single entity of 257 bits. Modern processors generally have a word size that is a power-of-two multiple of a byte (8, 16, 32, ...). Therefore we chose for our implementation to save the bit s_0 on its own, allowing the rest of the state to be stored in an array A of 32 bytes. Let w be the word size, assumed to be a byte multiple, as such $w \mid 256$, Let array A consist of elements of length w, then:

$$s_i = A[a][j] \quad \text{for } i \neq 0 \qquad \text{where} \quad \begin{array}{c} a = \lfloor (i-1)/w \rfloor \\ j = (i-1) \bmod w \end{array}$$

Here a is the index of the word in A that contains s_i, and j is the index of s_i in this memory unit. For ease of calculation we also store s_0 in a word where all bits are set to s_0:

$$z[j] = s_0 \quad \text{for } 0 \leq j < w$$

In both χ^n and θ^n bitwise operations are done using a state s_{i+o} that is offset by $o = c \cdot 12^n$ to s_i where $c \in \{1, 2, 3, 8\}$ as defined in χ^n and θ^n.

To do bitwise operations on A we make use of arrays A^{+o} similar to A with versions of the state offset by o. For example, we can apply $\theta : s_i \leftarrow s_i + s_{i+3} + s_{i+8}$ on A as follows:

$$\theta : \quad A \leftarrow A + A^{+3} + A^{+8}$$

Note we have to calculate the effect of θ on bit $s_0 = z$ separately.

The elements of A^{+o} can be divided into three groups based on their index a and the offset o:

- There is an element that contains the bit with index $i = -o$ that maps to z. Let a_z be the index in A^{+o} of this element, we can calculate it as follows:
  $$a_z = \lfloor (-o-1)/b \rfloor$$
  Note that $c \neq 0$ and as such $o = c \cdot 12^n \neq 0 \implies -o \neq 0$.
- The elements with an index $a < a_z$.
- The elements with an index $a > a_z$.

Construction of the memory units of A^{+o}

We now look at these three element groups separately, determining how they can be calculated based on z and bit shifted elements of A.

While the bits in an element $A^{+o}[a]$ are in the same order as in A, they come from (at most) two elements $P, Q \in A$, and if $a = a_z$ there is a single bit that comes from z.

The element in A^{+o} with index a_z

Let $j_z = (-o-1) \bmod w$ be the index within $A^{+o}[a_z]$ of the bit that maps to z. The $p = j_z$ number of bits before index j_z map to bits in the last memory unit P of A, and the $q = w - p - 1$ number of bits after index j_z map to bits in the first memory unit Q of A. Thus:

$$\begin{aligned} p &= j_z & P &= A[|A|-1] \\ q &= w - p - 1 & Q &= A[0] \end{aligned}$$

To construct $A^{+o}[a_z]$ we perform an OR operation on masked versions of z, P, and Q where P and Q have their bits shifted to the correct position (see Figure 4.2):

$$A^{+o}[a_z] = ((P \gg w - p) \;\&\; M_P) \mid (z \;\&\; M_z) \mid ((Q \ll w - q) \;\&\; M_Q)$$

Here M_P is a mask with only the first p bits set to one, M_Q is a mask with only the last q bits set to one, and M_z is a mask with only the bit at index j_z set to one.

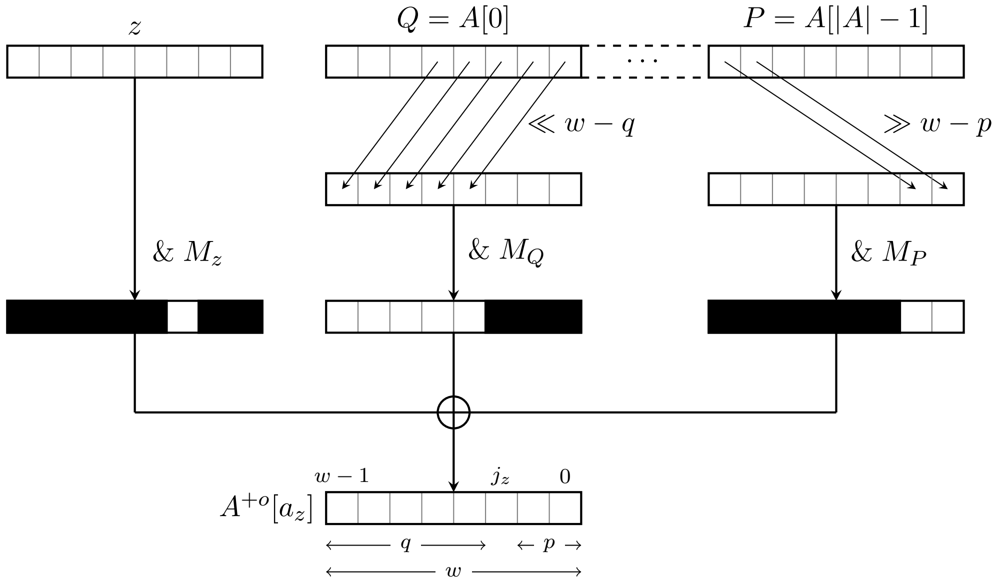

Figure 4.2: Construction of $A^{+o}[a_z]$

Elements in A^{+o} with an index $a < a_z$

The bits in this kind of memory unit map to at most two memory units in A. The first $p = w - (o \bmod w)$ bits map to memory unit $P = A[a + \lfloor o/w \rfloor]$. The last $q = o \bmod w$ bits map to memory unit $Q = A[a + \lceil o/w \rceil]$. Hence:

$$\begin{aligned} p &= w - (o \bmod w) & P &= A[a + \lfloor o/w \rfloor] \\ q &= o \bmod w = w - p & Q &= A[a + \lceil o/w \rceil] \end{aligned}$$

As before, we perform an OR operation on masked versions of P and Q where the bits have been shifted to the correct position (see Figure 4.3):

$$B[a] = ((P \gg q) \;\&\; M_P) \mid ((Q \ll p) \;\&\; M_Q) \qquad \text{for } a < a_z$$

Where M_P is a mask with only the first p bits set to one and M_Q is a mask with only the last q bits set to one.

Elements in A^{+o} with an index $a > a_z$

The construction of these elements take place in a very similar manner as above; we only need to adjust the offset we use to calculate with to take into account that the bit at index zero is not included in the array A. As such let $o' = o - 1$.

Again the bits in this kind of memory unit map to at most two memory units in A. The first $p = b - (o' \bmod b)$ bits map to memory unit $P = A[a+$

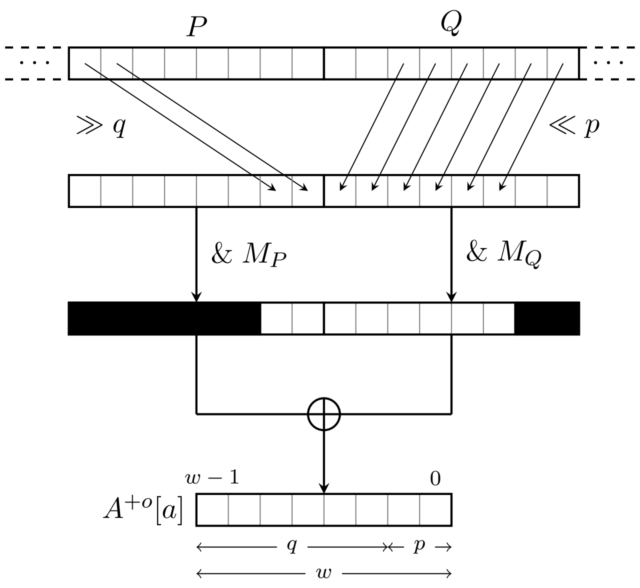

Figure 4.3: Construction of $A^{+o}[a]$ where $a \neq a_z$

$\lfloor o'/b \rfloor]$. The last $q = o' \bmod b$ bits map to memory unit $Q = A[a + \lceil o'/b \rceil]$. Hence:

$$\begin{aligned} p &= b - (o' \bmod b) & P &= A[a + \lfloor o'/b \rfloor] \\ q &= o' \bmod b = b - p & Q &= A[a + \lceil o'/b \rceil] \end{aligned}$$

Constructing the element (again see Figure 4.3):

$$B[a] = ((P \gg q)\ \&\ M_P) \mid ((Q \ll p)\ \&\ M_Q) \quad \text{for } a > a_z$$

Where M_P is a mask with only the first p bits set to one and M_Q is a mask with only the last q bits set to one.

Findings based on the above

We made some observations based which would make implementation easier. Let $p_{<a_z}$ be p when $a < a_z$ etcetera. First, let's have a closer look at p_{a_z}:

$$\begin{aligned} p_{a_z} = j_z &= (-o - 1) \bmod w \\ &= (-o + 256) \bmod w \\ &= -o \bmod w \\ &= (w - (o \bmod w)) \bmod w \\ &= p_{<a_z} \bmod w \end{aligned}$$

Since $p_{<a_z} \in [0, w]$ this means $p_{a_z} = p_{<a_z}$ when $p_{<a_z} \neq w$. We can see that when $p_{<a_z} = w$ then $p_{a_z} = 0$.

And for q_{a_z} we can conclude:

$$\begin{aligned} q_{a_z} &= w - p_{a_z} - 1 \\ &= w - (-o \bmod w) - 1 \\ &= (-1 \bmod w) - (-o \bmod w) \\ &= ((-1 \bmod w) + (o \bmod w)) \bmod w \\ &= (o - 1 \bmod w) \\ &= q_{>a_z} \end{aligned}$$

We also saw before that the calculations for memory units with an index $a > a_z$ is the same as for memory units with index $a < a_z$ just with an offset that has one subtracted. Using this information we are able to calculate an offset state in a quite efficient way.

4.3 Implementation

Again, because of the active development of Subterranean during our research no test vectors were available. After implementing a non-procrastinated version in both the C and Rust programming languages, also a procrastinated version in both languages was created. Like before the outputs of these versions were compared with each other to make sure discrepancies were found and corrected. After access to a Python implementation version was given by the authors of Subterranean, was this also put alongside for comparison.

Chapter 5

Benchmarks

Benchmarks were done on a STM32F103C8[1] microcontroller which incoreperates an ARM Cortex-M3 core. This was connected to an ST-LINK/V2[2], an in-circuit debugger and programmer that also functioned as the power source and allowed interfacing via USB.

The performance measurement used is the number of cycles needed to complete a certain operation. Cycle counting was done using functions from the `libopencm3`[3] firmware library: `dwt_enable_cycle_counter` and `dwt_read_cycle_counter`[4]. Both are small wrappers around the instructions needed to activate and read out the *ARM Data Watchpoint and Trace Unit Current PC Sampler Cycle Count Register*[5].

To read out the values returned by above functions and as a general debugging tool we used the on-chip debugging tool `OpenOCD`[6].

After the publication of XOODYAK the KECCAK Team released their reference implementation of XOODYAK in the C programming language. This can be found in the eXtended Keccak Code Package (XKCP)[7]. As such, we decided to test these three codebases, all written in the C programming language:

1. The XKCP version of XOODYAK;
2. Our version of XOODYAK;
3. Our procrastinated version of Subterranean.

All were compiled and linked using version 9.1.0 of `arm-none-eabi-gcc`, part of the GNU ARM embedded toolchain.

[1] https://www.st.com/en/microcontrollers-microprocessors/stm32f103c8.html
[2] https://www.st.com/en/development-tools/st-link-v2.html
[3] http://libopencm3.org/
[4] http://libopencm3.org/docs/latest/lm4f/html/dwt_8c.html
[5] http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0337e/ch11s05s01.html
[6] http://openocd.org/
[7] https://github.com/XKCP/XKCP/

5.1 Hash performance

Hashing performance of both Xoodyak and Subterranean was tested by splitting the hashing operation into its two phases: the absorbing and the squeezing phase.

5.1.1 Absorbing phase

The absorbing phase consists of initialisation and absorption of data of various lengths: 1, 16, 256, and 4096 bytes. Results of these benchmarks can be found in Table 5.1

input[a]	XKCP[b]	XKCP[c]	Xoodyak[b]	Xoodyak[c]	Subterranean[b]	Subterranean[c]
1	267	267	297	297	5325	5325
16	300	19	330	21	46508	2907
256	76035	298	75739	296	703103	2747
4096	1287795	315	1282219	314	11215228	2739

[a] length, in bytes
[b] in cycles
[c] in cycles per byte of input, rounded up

Table 5.1: Hash absorbing phase benchmark results

The performance of both Xoodyak variants is as good as identical, which was to be expected as both are written in the same language and don't make use of any architecture-specific optimisations. The small difference in the number of cycles between an input length of 1 and 16 bytes using Xoodyak is because both will result in a single input block for $R_{\text{hash}} = 16$.

Subterranean with its hashing input block length of 8 bits requires more input blocks than Xoodyak per the same length of input. This results in a quicker stabilisation of the number of cycles per byte. We do see quite a large gap in performance, where Subterranean is slower than Xoodyak by about a factor of 10. This is exaggerated because of the double round function of Subterranean when absorbing in unkeyed mode.

5.1.2 Squeezing phase

The squeezing phase consists of squeezing out a digest of various lengths, again 1, 16, 256, and 4096 bytes. In the case of Subterranean this is preceded with 8 blank rounds as per the Subterranean-XOF algorithm (see Algorithm 3). Results of these benchmarks can be found in Table 5.2

We see that the performance difference between Xoodyak and Subterranean has decreased, most likely due to the larger output blocks of 32 bits that Subterranean uses for extraction.

digest[a]	XKCP[b]	XKCP[c]	XOODYAK[b]	XOODYAK[c]	Subterranean[b]	Subterranean[c]
1	5017	5017	5023	5023	11484	11484
16	5026	314	5032	315	21800	1363
256	79740	312	80106	313	201755	789
4096	1275180	312	1281306	313	3074057	751

[a] length, in bytes
[b] in cycles
[c] in cycles per byte of digest, rounded up

Table 5.2: Hash squeezing phase benchmark results

5.2 AEAD performance

Authenticated encrypting performance of both XOODYAK and Subterranean was tested by applying the authenticated encryption-variants of both cipher suits on plain text and additional data of various lengths.

The lengths in bytes of the plain text and additional data strings were: 1, 16, 256, and 4096. As we normally don't expect the size of the additional data to exceed the size of the plain text we only measured combinations of plain text and additional data where this is not the case. Results of the measurements can be found in Table 5.3

plaintext[a]	AD[a]	XKCP[b]	XKCP[c]	XOODYAK[b]	XOODYAK[c]	Subterranean[b]	Subterranean[c]
1	1	20649	20649	20725	20725	56142	56142
16	1	20722	1296	20801	1301	71356	4460
16	16	20755	1298	20834	1303	80219	5014
256	1	72942	285	72689	284	309814	1211
256	16	72975	286	72722	285	318604	1245
256	256	98620	386	98313	385	455021	1778
4096	1	908462	222	902929	221	4116862	1006
4096	16	908495	222	902962	221	4125627	1008
4096	256	934140	229	928553	227	4261670	1041
4096	4096	1384559	339	1378004	337	6433291	1571

[a] bytes
[b] cycles
[c] cycles per byte of plaintext, rounded up

Table 5.3: AEAD benchmark results

Again we see a difference in performance, be it not as stark as when absorbing data when hashing. Overall we can expect that XOODYAK is faster by about a factor of 5 for sufficiently large input.

Chapter 6

Conclusions

As a result of our work we have implementations of both Xoodyak and Subterranean than can both offer acceptable performance on ARM. As was to be expected is the performance of Subterranean worse than that of Xoodyak, but still respectable enough that it should not be outright dismissed as impractical based on this. We do believe that the efficiency of both implementations can still be improved. Foremost by making use of optimisations for the targeted architecture. Our work should be a good basis for such further work.

Bibliography

[1] NIST issues first call for 'lightweight cryptography' to protect small electronics. https://www.nist.gov/news-events/news/2018/04/nist-issues-first-call-lightweight-cryptography-protect-small-electronics.

[2] *Proceedings 1993 International Conference on Computer Design: VLSI in Computers & Processors, ICCD '93, Cambridge, MA, USA, October 3-6, 1993*. IEEE Computer Society, 1993.

[3] NIST. Submission requirements and evaluation criteria for the lightweight cryptography standardization process, April 2018.

[4] Daniel J Bernstein, Stefan Kölbl, Stefan Lucks, Pedro Maat Costa Massolino, Florian Mendel, Kashif Nawaz, Tobias Schneider, Peter Schwabe, François-Xavier Standaert, Yosuke Todo, et al. Gimli: a cross-platform permutation. In *International Conference on Cryptographic Hardware and Embedded Systems*, pages 299–320. Springer, 2017.

[5] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. Sponge functions. In *ECRYPT hash workshop*, volume 2007. Citeseer, 2007.

[6] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. Keccak sponge function family main document. *Submission to NIST (Round 2)*, 3(30), 2009.

[7] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. Cryptographic sponge functions, 2011.

[8] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. Duplexing the sponge: single-pass authenticated encryption and other applications. In *International Workshop on Selected Areas in Cryptography*, pages 320–337. Springer, 2011.

[9] Guido Bertoni, Joan Daemen, Michaël Peeters, Gilles Van Assche, and Ronny Van Keer. Keyak v2. *CAESAR Submission*, 2015.

[10] Joan Daemen, Seth Hoffert, Gilles Van Assche, and Ronny Van Keer. Xoodoo cookbook. Technical report, Cryptology ePrint Archive: Report 2018/767, 2018. https://eprint. iacr. org . . . , 2018.

[11] Joan Daemen, Seth Hoffert, Gilles Van Assche, and Ronny Van Keer. Xoodyak, a lightweight cryptographic primitive, February 2019.

[12] Joan Daemen, Pedro Maat Costa Massalino, and Yann Rotella. The subterranean 2.0 cipher suite, February 2019.

[13] Morris J Dworkin. Sha-3 standard: Permutation-based hash and extendable-output functions. Technical report, 2015.