EINDHOVEN UNIVERSITY OF TECHNOLOGY

Eindhoven University of Technology

BACHELOR

Building a quantum secure MAC from a sponge construction

Custers, Frank

Award date:
2019

Link to publication

Disclaimer
This document contains a student thesis (bachelor's or master's), as authored by a student at Eindhoven University of Technology. Student theses are made available in the TU/e repository upon obtaining the required degree. The grade received is not published on the document as presented in the repository. The required complexity or quality of research of student theses may vary by program, and the required minimum study period may vary in duration.

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain

Take down policy
If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.

Download date: 04. Nov. 2024

Eindhoven University of Technology

2WH40 Bachelor Final Project

Building a quantum secure MAC from a sponge construction

Author:
F. J. A. Custers (1013292)
f.j.a.custers@student.tue.nl
Supervisor:
Dr. Andreas Hülsing

July 9, 2019

Abstract

In this thesis we focus on the security of Message Authentication Codes (MAC), which are used to ensure data integrity and authenticity when two parties communicate via an unsafe channel. We recall that a pseudo-random function is a secure MAC [7]. We look at the family of sponge constructions, designed by the Keccak Team [5], and how applying a key to this constructions can be used to build a pseudo-random function. These methods include the inner- and outer-keyed sponges, and we look at their security in a classical setting, using notions of indistinguishability [2]. We show why the sponge family will become important in the future, by showing that the widely used CBC-MAC is broken in a quantum world, where an attacker would have access to a quantum computer. This attack is described by Santoli and Schaffner [16], and we evaluate it by performing two similar simulations. The first simulation is one with advice, where we avoid the use of the underlying quantum algorithm, called Simon's algorithm. For the second simulation we use the library called Qiskit [1] for an implementation of this algorithm. We perform a theoretical runtime analysis of the attack and confirm these results by comparing it to a practical runtime analysis based on our simulations. We also recall that by using the sponge construction we can create a quantum secure MAC, for which its security only depends on the quantum pseudo-random permutation security of the underlying keyed permutation [9].

Contents

1 Introduction **3**
- 1.1 Motivation . . . 3
- 1.2 Our contribution . . . 3
- 1.3 Content of this thesis . . . 3

2 Message authentication codes **4**
- 2.1 Tagging and verifying . . . 4
- 2.2 Security of a MAC . . . 4

3 The sponge construction **6**
- 3.1 The sponge function . . . 6
- 3.2 The keyed sponge . . . 6

4 Security of keyed sponges **8**
- 4.1 Preliminaries, tools and proof sketch . . . 8
- 4.2 Proof of security of IKS . . . 9
- 4.3 Proof of security of OKS . . . 12

5 Quantum security **14**
- 5.1 Algorithms . . . 14
- 5.2 CBC-MAC . . . 15
- 5.3 Quantum attack on CBC-MAC . . . 15
- 5.4 Quantum attack on CBC-MAC implementation . . . 17
- 5.5 Quantum security of keyed-sponge construction . . . 24
- 5.6 Quantum secure pseudo-random permutation . . . 26

6 Conclusion and recommendations **27**

A Attack with advice **28**

B Attack with Simon's algorithm **29**

1 Introduction

1.1 Motivation

The Keccak team designed a cryptographic construction called the sponge construction [5]. This very flexible construction builds a function that maps an arbitrary amount of bits to an arbitrary amount of bits. Being this flexible means that it can be used to make a great range of cryptographic primitives. An interesting application is applying a key to it, and use it to build a message authentication code. Message authentication codes ensure data authenticity and integrity. A message authentication code can also be combined with an encryption method, such that it ensure both authenticity and integrity, as well as confidentiality. These combined is called authenticated encryption, and this is necessary for secure communication. One way to show the security of message authentication codes is by making use of pseudo-random functions, which are functions that are indexed by some key, and they are indistinguishable from a truly random function. One might wonder why we need these keyed sponges to create a message authentication code, since there already exist secure methods? To answer this question, we need to look a bit into the future. The development of the quantum computer has already started, and whether this will develop into a large scale device is still an unanswered question. But better safe than sorry, and start preparing for the worst, because a large scale quantum computer introduces the ability to run quantum algorithms which are powerful enough to break many cryptographic primitives. This also includes the widely used CBC-MAC, yet interestingly enough, this attack doesn't work on a keyed sponge construction. Why not? Because a keyed sponge construction has a trick up his sleeve: a secret inner state.

1.2 Our contribution

Our contribution aims to provide answers to the following research questions:

- What defines the security of a message authentication code?
- Can a key be applied to a sponge construction to achieve a secure message authentication code, and will a proof for in a classical world hold in a quantum world?
- Will a quantum attack on CBC-MAC hold in practice?

We aim to answer the first two questions by explaining work done by various researchers [2, 5, 6, 7, 9, 12, 14, 19]. To answer the last question, we contribute by performing a theoretical runtime analysis of a known quantum attack [13, 16]. We also verify this by implementing the attack in Python, and doing a practical runtime analysis.

1.3 Content of this thesis

First we describe what a message authentication code is in chapter 2, and what the requirements are for it to be secure. Here we recall why a pseudo-random function is a secure message authentication code.
In chapter 3 we describe what the sponge construction is, by explaining the technical details. Two methods are looked at on how to apply a key to the construction, the inner- and outer-keyed sponges.
Then in chapter 4 we look at an extensive analysis on how it can be proven that the keyed sponges are secure in a classical setting. Classical setting here means that we don't allow any quantum tools. The proof that the inner-keyed sponge is a pseudo-random function is analyzed, as well as the similar results for the outer-keyed sponge.
In chapter 5 we introduce the quantum world. Some known quantum algorithms are described, with Simon's algorithm being the most important as it is used to create an attack on CBC-MAC. First we specify the details of a fixed length CBC-MAC, and then recall the attack and how Simon's algorithm works in this case. Then we simulate the attack by implementing it in Python. First a simulation with advice is explained, and then a simulation which uses Simon's algorithm on a small example is given. Then we perform a theoretical runtime analysis, and verify this by comparing it to a practical runtime analysis using our simulations. Next the quantum security of the keyed sponges is analyzed, and we recall the keyed-internal-function sponges, and the importance of the underlying permutation being a quantum secure pseudo-random permutation.
Lastly, in chapter 6 we give a conclusion.

2 Message authentication codes

Consider two parties, Alice and Bob, that want to communicate via an unsafe channel. This channel is considered unsafe because malicious attackers can intercept and change messages. Now consider that when Bob receives a message from Alice, he wants to make sure it is actually from Alice (authenticity), and that the message is identical to how Alice has sent it (integrity). This authenticity and integrity can be assured by a message authentication code:

Definition 1 (Message authentication code (MAC)[12]) *A message authentication code or MAC is a tuple of probabilistic polynomial-time algorithms* MAC=(Gen, Mac, Vrfy) *over a message space $\mathcal{M}$, fulfilling the following:*

- **Gen** *is a probabilistic algorithm that on input 1^n outputs a key k. The output space of Gen is called the key space $\mathcal{K}$.*
- **Mac** *takes as input a key $k \in \mathcal{K}$ and a message $m \in \mathcal{M}$, and outputs a tag $t \in \mathcal{T}$. The output space of Mac is called tag space $\mathcal{T}$.*
- **Vrfy** *is a deterministic algorithm that takes as inputs a key $k \in \mathcal{K}$, a message $m \in \mathcal{M}$, and a tag $t \in \mathcal{T}$, and outputs a bit $b \in \{0, 1\}$.*
- ***Correctness**: For every n, every $k \leftarrow \text{Gen}(1^n)$, and every $m \in \mathcal{M}$ it holds that $\text{Vrfy}_k(m, \text{Mac}_k(m)) = 1$.*

So a MAC consists of a key generation algorithm, a MAC algorithm and a verification algorithm. We ignore the key generation algorithm and assume Alice and Bob share a secret key. The MAC and verification algorithm rely on what is called the MAC function, and is explained in section 2.1. Then in section 2.2 we recall on what it means for a MAC function to be secure and proof that a pseudo-random function is a secure MAC.

2.1 Tagging and verifying

Consider again Alice, who wants to send a message m to Bob, via an unsafe channel. Alice and Bob share a secret key k, and decide to use a MAC. The Mac function takes as input a message and a key, and based on these it creates a tag t, so $\text{Mac}(m, k) = t$. The tag should be hard to make for someone who doesn't have the key. So when Alice wants to send her message, she first tags it by creating the tag $t_A = \text{Mac}(m, k)$. Alice sends both m and t_A to Bob. Then Bob runs a verification algorithm which takes as input the received message and tag, as well as the secret key. This verification algorithm returns whether the message and tag is a valid pair or not. If it returns that it is not, then Bob discards the message. A visual representation is given in figure 1.

2.2 Security of a MAC

For the security of a MAC we need to look at the existential unforgeability under chosen message attack (EU-CMA). The attacker is an adversary denoted by $\mathcal{A}$. In a chosen message attack, the adversary has query access to an oracle that contains the Mac function. The attacker can use this oracle to receive a tag to any input message he chooses. The attack executes the following 'game', denoted by $Exp^{\text{EU-CMA}}_{MAC,\mathcal{A}}(n)$:

1. Generate a key, $k \leftarrow \text{Gen}(1^n)$.
2. Choose a message $m \in \mathcal{M}$ and a tag $t \in \mathcal{T}$. Let $\{m_i\}_{i=1}^q$ denote $\mathcal{A}$'s queries to Mac_K.
3. If $\text{Vrfy}_k(m, t) = 1$ and $m \notin \{m_i\}_{i=1}^q$ return 1, else return 0.

The adversary breaks the MAC if it wins this game, so when it returns 1. Winning this game has a certain probability ϵ. Using this game and the winning probability, we can define the security of a MAC:

Definition 2 (MAC security, Existential Unforgability under Chosen Message Attack [12]) *A message authentication code* MAC=(Gen, Mac, Vrfy) *over a message space $\mathcal{M}$ is existentially unforgeable under a chosen message attack, or just secure, if for all efficient adversaries $\mathcal{A}$ the success probability ϵ in winning $Exp^{EU\text{-}CMA}_{MAC,\mathcal{A}}(n)$is negligible, where*

$$\epsilon = \mathbb{P}(Exp^{EU\text{-}CMA}_{MAC,\mathcal{A}}(n) = 1)$$

.

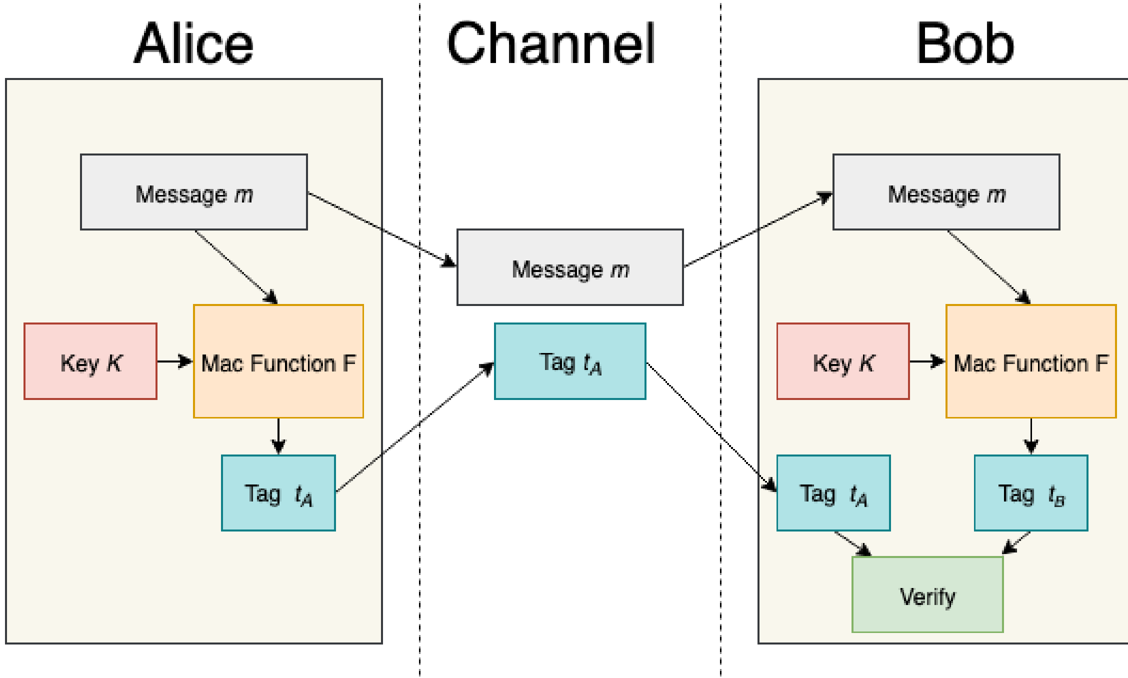

Figure 1: Visual representation of how a MAC works

Intuitively this means that for a MAC to be secure, the MAC function shouldn't give away any information. This introduces a family of functions:

Definition 3 (Pseudo random function, Bogdanov and Rosen [7]) *A function $F_k : \{0,1\}^a \to \{0,1\}^b$, indexed by a key $k \in \{0,1\}^n$ is a pseudo random function when it is:*

- ***Easy to evaluate:*** *The value $F_k(x)$ is efficiently computable given k and x.*
- ***Pseudorandom:*** *All efficient adversaries $\mathcal{A}$ can not distinguish the function F_k from a uniformly chosen random function $R : \{0,1\}^a \to \{0,1\}^b$, given query access to an oracle containing R. So the distinguishing advantage ϵ must be negligible, where*

$$\epsilon = \left|\mathbb{P}(\mathcal{A}^{F_k} = 1) - \mathbb{P}(\mathcal{A}^{R} = 1)\right|.$$

We can show that a PRF is a secure MAC:

Theorem 1 (A PRF is a secure MAC) *Let $F_K : \{0,1\}^a \to \{0,1\}^b$ be a PRF with key $k \in \{0,1\}^n$, then F_k is a secure MAC function.*

Proof. This proof is informally given by Bogdanov and Rosen [7] in section 1.1 as an application of PRFs. We need to show as in definition 2 that for all efficient adversaries $\mathcal{A}$ the probability ϵ of winning $Exp^{\text{EU-CMA}}_{F_k,\mathcal{A}}(n)$ is negligible, when given query access to an oracle containing F_k. For this the adversary needs to choose a message m and correctly guess a tag t. It follows from definition 3 that even with giving this query access, the probability of correctly guessing t does not change if we replace F_K by a random function $R : \{0,1\}^a \to \{0,1\}^b$. The probability of correctly guessing the tag using this function is $\frac{1}{2^b}$. Therefore the probability of winning the game also is $\epsilon = \frac{1}{2^b}$, which is a negligible low probability for large enough b. Therefore F_K is EU-CMA and thus a secure MAC. □

We can use this theorem later, to show if a certain function is a PRF, then it also is a secure MAC.

3 The sponge construction

The sponge construction, designed by the Keccak team [5], is an iterated construction that is used to make a sponge function. Such a sponge function has as input a bitsequence of any length. The output is a bitsequence of any desired length. The details of the sponge function are given in section 3.1. Having a variable length input and output means that the construction is very flexible and can be used for many applications, such as hash-functions, but also keyed applications such as a stream cipher, authenticated encryption and also a MAC function. We are mainly interested in making the keyed application of making a MAC, which is explained in section 3.2.

3.1 The sponge function

A sponge function has as input a message m, which is a bitsequence of any possible length, so $m \in \{0,1\}^*$. The output z is a bitsequence that can be of infinite length, but usually is truncated to the desired length l, so $z \in \{0,1\}^l$, where $l \in \mathbb{N}$ is specified in the input. The sponge function uses a permutation f that acts on a state s which has a length of b bits. This state consists of two parts, the first part is the outer state $\hat{s}$ and the second part is the inner state $\overline{s}$. The outer state has a length of r, which is referred to as the rate. The inner state has a length of c, such that $b = r + c$, and it is referred to as the capacity. When referring to an evaluation of the sponge function, we use the following notation:

$$\textsc{Sponge}^f(m, l) = z. \tag{1}$$

The first step is padding the input message, such that its length is a multiple of the rate r. This padding method should be injective and the last block it produces should be different than 0^r. The input message is then cut up in blocks of length r. The state s is initialized by setting all b bits to zero. The padded message then enters the first phase, called the absorbing phase. In this phase the state is repeatedly updated by first XORing an input message block to the outer state and then updating the state by applying the permutation f to it. This is repeated until every input message block is processed. The next phase is the squeezing phase. In this phase the bits of the outer state are outputted, and the whole state gets updated again by applying the permutation f. This is repeated until the output has a larger length than the required l, and then the total output is truncated such that it has exactly the length of l. The sponge construction is visualized in figure 2.

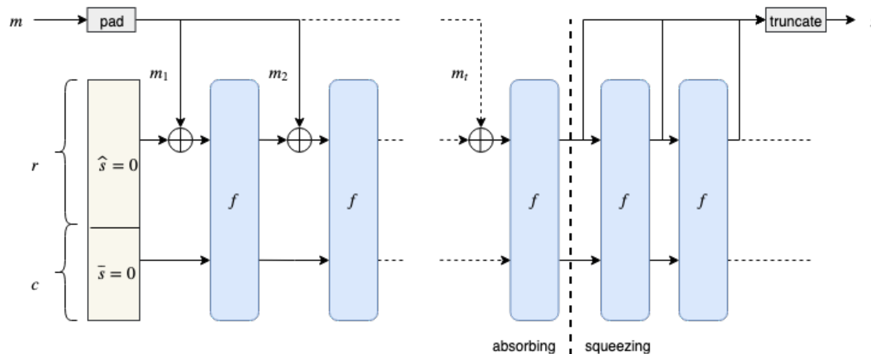

Figure 2: The sponge construction

3.2 The keyed sponge

The flexibility of the input of a sponge function leaves the option of including a key in the function. By doing so, such a keyed sponge becomes a pseudo-random function. Andreeva et al. [2] gives an overview of some methods to do this, which we will follow. One approach for including a key to the sponge construction was introduced by Chang et al. [8], which they used when building a method called EMKSC3. In this approach a key $K \in \{0,1\}^c$ is initialized as the inner state. Therefore we will refer to this as the inner-keyed sponge:

$$\text{IKS}^f_K(m, l). \tag{2}$$

To be able to express the inner-keyed sponge using the sponge function we introduce the Even-Mansour construction [10]. This construction uses the permutation f on b bits and a key K of

length b to create a cipher, which is defined as $E_K^f(x) = f(x \oplus K) \oplus K$. Since we want a key of c bits that only applies to the inner state we use the following variant:

$$E_K^f(x) = f(x \oplus (0^r||K)) \oplus (0^r||K) \tag{3}$$

. Using this construction we can define the inner-keyed sponge with the sponge function itself:

$$\text{IKS}_K^f(m, l) = \text{SPONGE}^{E_K^f}(m, l). \tag{4}$$

To see that this equivalence holds, we look at what happens when an Even-Mansour construction is called twice:

$$E_K^f(E_K^f(x)) = E_K^f(f(x \oplus K) \oplus K) = f(f(x \oplus K) \oplus K \oplus K) \oplus K = f(f(x \oplus K)) \oplus K. \tag{5}$$

Note here how the two times XOR K cancel each other out, and if the Even-Mansour function was being used even more often, the key would only appear twice: once in the nested permutations and once in the end. Now for our application the effect of the Even-Mansour construction is that the key gets XORed with the inner state before and after every call to f. So in between calls to f the key cancels itself out by XORing it twice, and thus the key only acts on the initial value of the state. Note that even though in the squeezing phase the output is gathered between calls to f, the key does not influence the output since it only acts on the inner state, and the output only takes the outer state. A visual representation of this canceling can be seen in figure 3

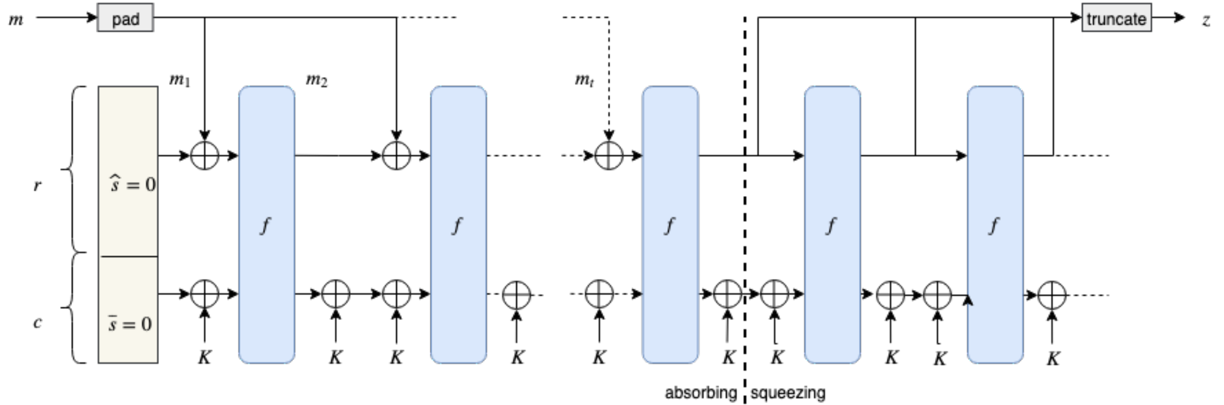

Figure 3: The inner-keyed sponge with an Even-Mansour construction

A different method to apply a key has been introduced by Bertoni et al. [6], where the key is included by simply prepending it to the message. So instead of giving m as the input, they use $K||m$ as input, where $K \in \{0, 1\}^k$ is the key. We will call this method the outer-keyed sponge, and use the following notation:

$$\text{OKS}_K^f(m, l) = \text{SPONGE}^f(K||m, l). \tag{6}$$

Note that when the key length k is longer than the rate r, then the first steps of the sponge function is to absorb the key. For security analysis reasons we now assume that k is a multiple of r, since this will somewhat simplify the security result without changing it much. We define the absorption of the key K into a root key L as a function called the key derivation KD, so we get the function:

$$\text{KD}^f(K) = L. \tag{7}$$

In the next chapter we will give a proof that both IKS and OKS are pseudo-random functions, and thus can be used to build a secure MAC. In order for this MAC to be successful, both parties need to make sure that they are using the same parameter values for the rate and capacity and that they use the same underlying permutation. These values do not provide any security and therefore can be publicly known.

4 Security of keyed sponges

The goal of applying a secret key to the sponge construction and using this to create a MAC, is to ensure the data integrity and authenticity. In order to ensure this, the MAC needs to be secure. To show this security, we proof that the keyed sponges are pseudo-random functions. For this chapter we take a look at the security of the keyed sponge constructions in a classical setting, so we ignore any attacks that use quantum methods. Performing such a security analysis is done with the indistinguishability method, where an adversary tries to tell a real system apart from an ideal system. This adversary will have a certain advantage when telling these systems apart, which will be quantified and then the security is determined by an upperbound of this advantage. By showing that this upperbound is negligible, we show that the function is a PRF. In section 4.1 the required definitions and tools are given and a general sketch for these proofs is described. In section 4.2 a detailed proof for the IKS construction is performed, in the case where an adversary has access to one single instance, which is a result of Andreeva et al. [2]. In this paper they also analysed the case where an attacker has access to multiple instances, as well as the distinguishing advantage upperbound for the OKS construction, of which results are given in section 4.3.

4.1 Preliminaries, tools and proof sketch

Proving the security of a keyed-sponge construction, in our case inner- and outer keyed, is done by finding and proving an upperbound of the distinguishing advantage of an adversary, and this can be split up in 2 parts. In the first part we derive an upperbound for the sponge construction itself, without specifying the security of the underlying permutation. Then in the second part we look at the upper bound of the advantage for distinguishing the underlying permutation from the Even-Mansour construction. The security then depends on the so called pseudo-random permutation (PRP) security of the Even-mansour construction. The advantage of an adversary is defined the following way:

Definition 4 (Distinguishing advantage, Andreeva et al. [2] definition 2) *Let $\mathcal{A}$ be an adversary whose goal is to tell apart a real system X with query access to constructions $a, b, c, ...$, and an ideal system Y with query access to constructions $s, t, u, ...$ The distinguishing advantage is defined as:*

$$\boldsymbol{Adv}_X^Y(\mathcal{A}) = \Delta_{\mathcal{A}}(X, Y) = \left|\mathbb{P}(\mathcal{A}^{a,b,c,\cdots} = 1) - \mathbb{P}(\mathcal{A}^{s,t,u,\cdots} = 1)\right|$$

Where $\Delta(X, Y)$ is a statistical distance between X and Y.

First part In the first part of the proof the real system is the IKS or OKS construction, and the ideal system is a so called random oracle [3] which has a similar interface as the construction in the real system. For the IKS the interface is an input message as a bitsequence and a desired output length, and as output a bitsequence of that length. So the random oracle outputs a bitsequence, that doesn't change for the same input message. We will denote a random oracle with this interface by $\mathcal{RO}$. For the OKS the interface also has to include the key-derivation function. The purpose of the first part is to split up the upperbound as the sum of the upperbound of the distinguishing advantage of the sponge construction itself with no key involved, and the upperbound of the distinguishing advantage of the Even-Mansour construction. To give an upperbound to the distinguishing advantage of the sponge construction we want to keep track of the amount of access that is available to the IKS/OKS construction. We do this by assigning the variable M, which counts the amount of calls to the underlying permutation of the construction. In a practical case, an attacker will be limited by this. To express the upperbound of the sponge construction itself with this variable we use the following theorem:

Theorem 2 (Bertoni et al. [4] Theorem 1) *An adversary $\mathcal{A}$ that distinguishes a sponge construction SPONGE^f with capacity c and a random oracle $\mathcal{RO}$ with M access has as upperbound $\boldsymbol{Adv}_{\text{SPONGE}^f}^{\mathcal{RO}}(\mathcal{A}) \leq \frac{M^2}{2^c}$.*

Second part In the second part the real system is the Even-Mansour construction which acts on b bits, and its ideal system is a permutation π which is uniformly chosen from the set of all permutations that act on b bits. This set we call $\mathbf{Perm}(b)$, and for choosing a random permutation from this set we use the following notation: $\pi \xleftarrow{\$} \mathbf{Perm}(b)$. When an adversary tries to distinguish the real and ideal system, it has query access to an oracle that either contains E_K^f or π, as well as query access to the permutation f and its inverse f^-. We give the adversary query access to f and f^- since an attacker could implement this himself, as this permutation would be publicly known.

Since the queries the adversary makes to f don't need access to the IKS/OKS construction, we call them offline. We keep track of the amount of offline access with variable N, which counts the amount of calls made by the adversary. In a practical case this N is restricted by the time or computing power of an attacker. Beside this offline access we also use a characteristic called the total maximum multiplicity to give the upper bound:

Definition 5 (Andreeva et al. [2], definition 1, multiplicity) *Let $\{(s_i, t_i)\}_{i=1}^{M}$ be the set of M input output pairs for f made in construction evaluations. Then the maximum forward and backward multiplicities are given by*

$$\mu_{fw} = \max_a \#\{i \in \{1, \dots, M\} | \bar{s}_i = a\},$$
$$\mu_{bw} = \max_a \#\{i \in \{1, \dots, M\} | \bar{t}_i = a\}$$

The total maximum multiplicity is given by $\mu = \mu_{fw} + \mu_{bw}$.

In order to derive an upperbound expressed in these variables, we make use of Patarin's H-Coefficiënt technique [15]. To use this technique we first define what a transcript is. A transcript τ summarizes the results of an interaction between the adversary and a system. Such a summary contains the interaction with either E_K^f or π, as well as the interaction with f, and usually the transcript contains the key K as well. Interaction with a permutation is the set of all input/output pairs. Interacting with a system X can give many different transcripts, but some are more common than others, so they follow a certain distribution, which we denote by D_X. Now a transcript τ is said to be attainable in system X if the probability of this transcript in the distribution is larger than 0, so when $\mathbb{P}(D_X = \tau) > 0$. We will denote the set of all attainable transcripts by $\mathbf{T}$. Then the H-coefficient technique states:

Theorem 3 (Andreeva et al. [2] lemma 1, H-coefficient Technique) *Let $\mathcal{A}$ be an adversary that interacts with system X and Y. Let $\boldsymbol{T} = \boldsymbol{T}_{good} \cup \boldsymbol{T}_{bad}$ be a partition of the set of attainable transcripts into "good" and "bad" transcripts. Let ϵ be such that for all $\tau \in \boldsymbol{T}_{good}$*

$$\frac{\mathbb{P}(D_X = \tau)}{\mathbb{P}(D_Y = \tau)} \geq 1 - \epsilon.$$

Then, $\boldsymbol{Adv}_X^Y(\mathcal{A}) \leq \epsilon + \mathbb{P}(D_Y \in \boldsymbol{T}_{bad})$.

To be able to use this theorem in our proofs, we have to define what it means for a transcript to be good or bad. Since we want the advantage of the adversary to be small, we will also want that both ϵ and $\mathbb{P}(D_Y \in \mathbf{T}_{bad})$ are small.

4.2 Proof of security of IKS

Theorem 4 *Let* IKS *be the inner-keyed sponge construction with rate r, capacity c and , that uses a key $K \xleftarrow{\$} \{0,1\}^c$ and a permutation $f \xleftarrow{\$} \boldsymbol{Perm}(b)$, where $b = r + c$. Then this construction is a PRF.*

Proof part 1. Let $\mathcal{A}$ be an adversary with the goal to distinguish the real system with the inner-keyed sponge IKS_K^f from the ideal system with the random oracle $\mathcal{RO}$. The adversary is limited by the online access M, offline computation access N and maximal multiplicity μ. We let the adversary have access to the permutation f. Then by definition of the advantage:

$$\mathbf{Adv}_{\text{IKS}_K^f}^{\mathcal{RO}}(\mathcal{A}) = \Delta_{\mathcal{A}}(\text{IKS}_K^f, f; \mathcal{RO}, f), \tag{8}$$

which is by (4) equal to

$$\Delta_{\mathcal{A}}(\text{IKS}_K^f, f; \mathcal{RO}, f) = \Delta_{\mathcal{A}}(\textsc{Sponge}^{E_f^K}, f; \mathcal{RO}, f). \tag{9}$$

Let $\pi \xleftarrow{\$} \mathbf{Perm}(b)$. Now we want to split this up in the advantage between:

- the sponge construction with a permutation π and the random oracle,
- the Even-Mansour construction and the permutation π.

To do this we use the triangle inequality, which is visualized in an analogy in figure 4.

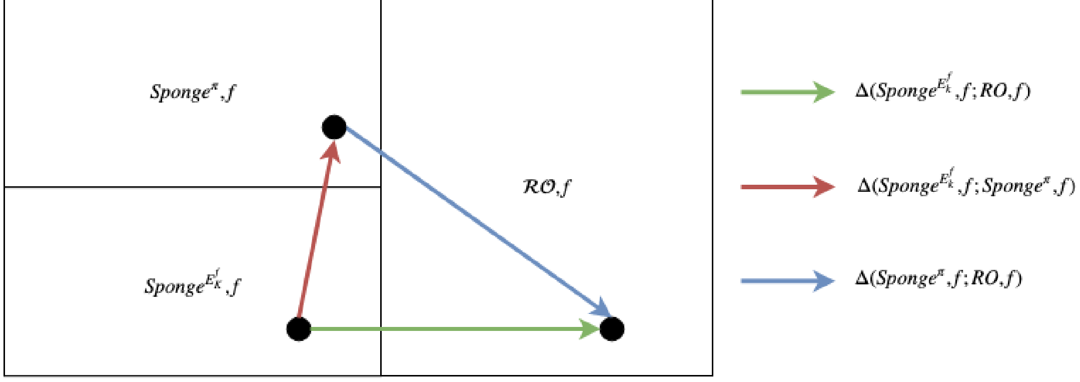

Figure 4: Visualization of the triangle inequality for splitting up the distinguishing advantage of an adversary

Note here that $\Delta_{\mathcal{A}}(\textsc{Sponge}^{E_K^f}, f; \textsc{Sponge}^{\pi}, f) = \Delta_{\mathcal{A}}(E_K^f, f; \pi, f)$, which follows from the fact that the only difference between a sponge with the Even-Mansour construction and a sponge with a random permutation, is the underlying permutation. So distinguishing these two sponges is the same as distinguishing an Even-Mansour construction from a random permutation. Using this we get, continued from (9):

$$\Delta_{\mathcal{A}}(\textsc{Sponge}^{E_f^K}, f; \mathcal{RO}, f) \leq \Delta_{\mathcal{A}}(\textsc{Sponge}^{\pi}, f; \mathcal{RO}, f) + \Delta_{\mathcal{A}}(E_f^K, f; \pi, f) \tag{10}$$

For the advantage of distinguishing the sponge construction with permutation π and the random oracle, having query access to permutation f doesn't add anything, so we get $\Delta_{\mathcal{A}}(\textsc{Sponge}^{\pi}, f; \mathcal{RO}, f) = \Delta_{\mathcal{A}}(\textsc{Sponge}^{\pi}; \mathcal{RO})$. Since this is the only part limited by M, we can apply theorem 2 to get $\Delta_{\mathcal{A}}(\textsc{Sponge}^{\pi}; \mathcal{RO}) \leq \frac{M^2}{2^c}$. This finishes part 1 of the proof, and the intermediate conclusion is:

$$\mathbf{Adv}^{\mathcal{RO}}_{\mathrm{IKS}_K^f}(\mathcal{A}) \leq \frac{M^2}{2^c} + \Delta_{\mathcal{A}}(E_f^K, f; \pi, f). \tag{11}$$

A visual representation of splitting up the task of the adversary is given in figure 5.

Proof part 2. In this part of the proof we want to give an upperbound to $\Delta_{\mathcal{A}}(E_f^K, f; \pi, f)$, which is the distinguishing advantage of the adversary between the Even-Mansour construction in the real system, and the permutation π in the ideal system. The adversary has access to the permutation f and its inverse f^{-}. Since the adversary has been limited by the online access M, there are exactly M input/output pairs of the interaction with the Even-Mansour construction or the permutation π. These interactions are summarized in the transcript $\tau_M = \{(s_i, t_i)\}_{i=1}^M$. The adversary is also limited by the offline computation access N, so there are exactly N interactions with the permutation f, which are summarized in the transcript $\tau_N = \{(x_i, y_i)\}_{i=1}^N$. A full summary of the interactions of $\mathcal{A}$ is defined by $\tau = (\tau_M, \tau_N, K)$, which also includes the key K. We want to use the H-coefficient technique, so we will have to define when a transcript is considered good or bad. A transcript τ is considered bad if in the real world there are 2 equal inputs to the permutation f, one from the Even-Mansour construction and one directly to f. In terms of our transcripts, a transcript τ is considered bad when there exists an input/output pair (s, t) in τ_M and an input/output pair (x, y) in τ_N such that $s \oplus (0^r || K) = x$, since the Even-Mansour construction uses $s \oplus (0^r || K)$ as input to f. To see why this is considered bad, we look at what happens when we evaluate s in the Even-Mansour construction and x in f, and XOR them:

$$\begin{aligned} E_K^f(s) \oplus f(x) &= f(s \oplus (0^r || K)) \oplus (0^r || K) \oplus f(x) \\ &= f(x) \oplus (0^r || K) \oplus f(x) \\ &= 0^r || K. \end{aligned} \tag{12}$$

This will evaluate to just the key K. Note the following equivalence:

$$\begin{aligned} & s \oplus (0^r || K) = x \\ \Longleftrightarrow\ & f(s \oplus (0^r || K)) = f(x) \\ \Longleftrightarrow\ & (f(s \oplus (0^r || K)) \oplus (0^r || K)) \oplus (0^r || K) = f(x) \\ \Longleftrightarrow\ & t \oplus (0^r || K) = y \end{aligned} \tag{13}$$

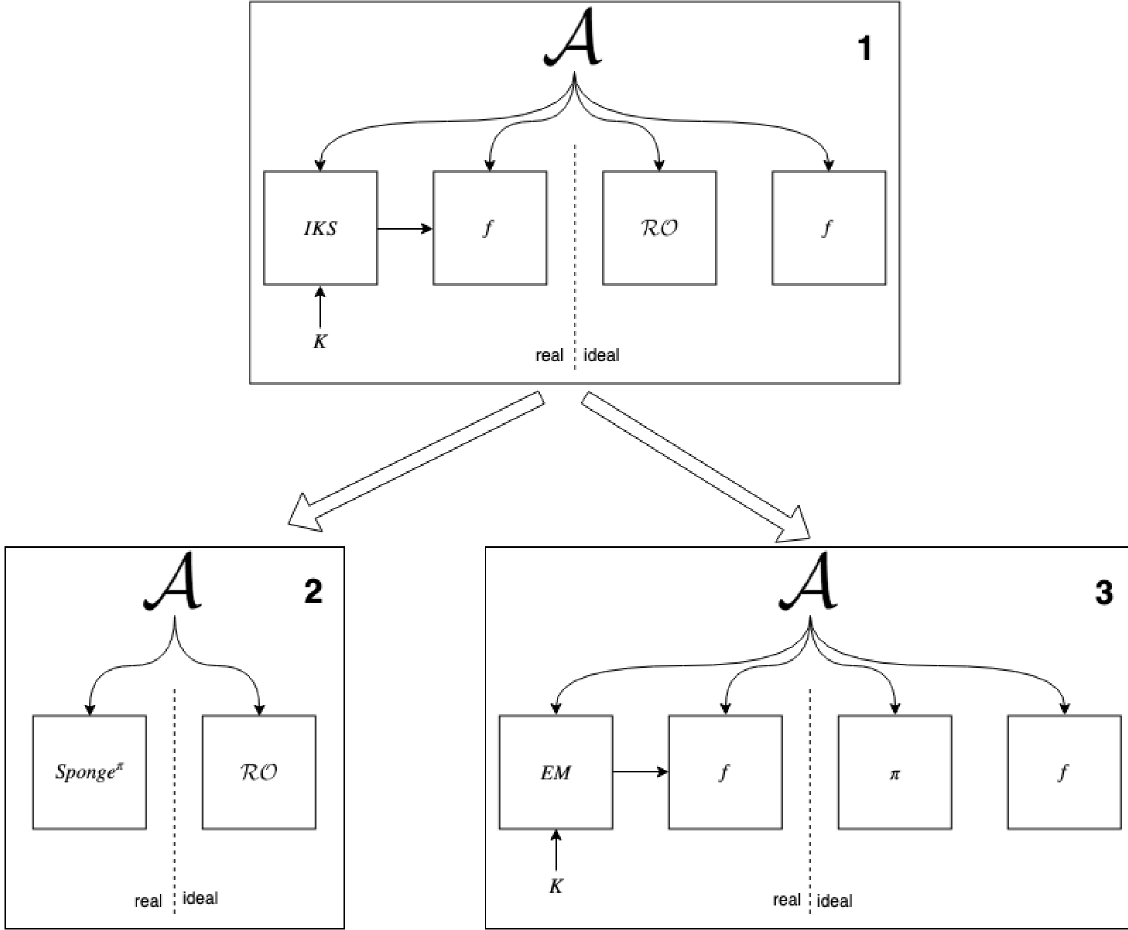

Figure 5: Visual representation of how an adversary has to distinguish the IKS with the random oracle $\mathcal{RO}$ (1), and how this is split up in distinguishing the sponge construction with permutation π and the random oracle (2), and distinguish the Even-Mansour construction and the permutation π (3).

So we also consider transcript to be bad when there exists an input/output pair (s, t) in τ_M and an input/output pair (x, y) in τ_N such that $t \oplus (0^r||K) = y$. This implies that in a good transcript every input/output pair in both τ_N and τ_M is a unique evaluation of f. For notation we now denote the real system by $X = (E_K^f, f)$ and the ideal system by $Y = (\pi, f)$. Next we want to know a bound to $\mathbb{P}(D_Y \in \mathbf{T}_{bad})$. For this we will use our maximal multiplicity $\mu = \mu_{fw} + \mu_{bw}$. First we look at the condition for a transcript being bad that states $s \oplus (0^r||K) = x$, which we can split up in the two conditions, one for the inner and one for the outer part:

- $\bar{s} \oplus 0^r = \bar{x}$, which is equivalent to $\bar{s} = \bar{x}$
- $\hat{s} \oplus K = \hat{x}$.

If we fix an input/output pair (x, y), then by definition there are only μ_{fw} pairs (s, t) such that the first condition is met, and thus also μ_{fw} pairs for the second condition. There are N pairs of (x, y) so there are at most $\mu_{\text{fw}}N$ pairs (s, t) that meet the second condition. Every pair has probability $\frac{1}{2^c}$ meeting the second condition, so in total we find the probability $\frac{\mu_{\text{fw}}N}{2^c}$. Now for the second condition for a transcript being bad, $t \oplus (0^r||K) = y$, we can do exactly the same but now with the backward multiplicity. In total we find:

$$\mathbb{P}(D_Y \in \mathbf{T}_{bad}) \leq \frac{\mu_{\text{fw}}N}{2^c} + \frac{\mu_{\text{bw}}N}{2^c} = \frac{\mu N}{2^c} \tag{14}$$

Next we want to know $\mathbb{P}(D_X = \tau)$ and $\mathbb{P}(D_Y = \tau)$ for all $\tau \in \mathbf{T}_{good}$. For that we use the following computations:

$$\mathbb{P}(D_X = \tau) = \frac{|comp_X(\tau)|}{|\Omega_X|}, \text{ and } \mathbb{P}(D_Y = \tau) = \frac{|comp_Y(\tau)|}{|\Omega_Y|}, \tag{15}$$

where:

- Ω_X is the set of all oracles for τ in the real system, so a combination of all possible E_K^f, f and K. The Even-Mansour construction is a b-bit permutation, that already states f, so their space has $2^b!$ permutations. The key space has size 2^c, so in total $|\Omega_X| = 2^b! \cdot 2^c$.
- Ω_Y is the set of all oracles for τ in the ideal system , so a combination of all possible π, f and K. Both π and f are b-bit permutations so their space has $2^b!$ permutations. The key space has size 2^c, so in total $|\Omega_Y| = (2^b!)^2 \cdot 2^c$.
- $comp_X(\tau)$ is the set of all oracles that are compatible with τ in the real system. We know that τ is a good transcript, thus every input/output pair in it is unique. Since there's $M+N$ unique input/output pairs in τ_M and τ_N, that leaves $(2^b - (M+N))!$ possible permutation for f such that they are all still unique in the transcript, so $|comp_X(\tau)| = (2^b - M - N)!$.
- $comp_Y(\tau)$ is the set of all oracles that are compatible with τ in the ideal system. Again we have the uniqueness of the input/output pairs that we need to preserve. For the permutation π we're left with $(2^b - M)!$ permutations such that every π evaluation is unique, and for f we're left with $(2^b - N)!$ permutations such that every f evaluation is unique. So in total we get $|comp_Y(\tau)| = (2^b - M)! \cdot (2^b - N)!$.

Now we want to know the following:

$$\begin{aligned}\frac{\mathbb{P}(D_X = \tau)}{\mathbb{P}(D_Y = \tau)} &= \frac{|comp_X(\tau)| \cdot |\Omega_Y|}{|comp_Y(\tau)| \cdot |\Omega_X|} \\ &= \frac{(2^b - M - N)! \cdot (2^b!)^2 \cdot 2^c}{(2^b - M)! \cdot (2^b - N)! \cdot 2^b! \cdot 2^c} \\ &= \frac{(2^b - M - N)! \cdot 2^b!}{(2^b - M)! \cdot (2^b - N)!} \\ &= \frac{(2^b - M - N)! \cdot ((2^b) \cdot (2^b - 1) \cdot \ldots \cdot (2^b - N + 1)) \cdot (2^b - N)!}{((2^b - M) \cdot (2^b - M - 1) \cdot \ldots \cdot (2^b - M - N + 1)) \cdot (2^b - M - N)! \cdot (2^b - N)!} \\ &= \frac{(2^b) \cdot (2^b - 1) \cdot \ldots \cdot (2^b - N + 1)}{(2^b - M) \cdot (2^b - M - 1) \cdot \ldots \cdot (2^b - M - N + 1)} \\ &\geq 1\end{aligned} \tag{16}$$

Note that in the last step, both the numerator and denominator are a product of N numbers, where pairwise the number of the numerator is larger than the denominator. Now we can apply theorem 3 with $\epsilon = 0$ and find that

$$\Delta_{\mathcal{A}}(E_f^K, f; \pi, f) \leq \epsilon + \mathbb{P}(D_Y \in \mathbf{T}_{bad}) = \frac{\mu N}{2^c}. \tag{17}$$

Which finishes part 2 of the proof.
In total we get from the combination of (11) and (17) that

$$\mathbf{Adv}^{\mathcal{RO}}_{\mathrm{IKS}_K^f}(\mathcal{A}) \leq \frac{M^2 + \mu N}{2^c}. \tag{18}$$

Since M, μ and N are fixed for any adversary, we can simply increase the capacity c of any such length that the distinguishing advantage of the adversary becomes negligible small. This means c is our security parameter, and the requirements of IKS being a PRF as in definition 3 are met. □

We've shown that IKS is a PRF, and by theorem 1 it can be used to make a secure MAC. Note that in the proof, the adversary is given access to only one instance of the inner-keyed sponge construction. In reality, an attacker can have multiple targets. In the case of n instances, there are n keys: $K_1, \ldots K_n \xleftarrow{\$} \{0,1\}^c$. Also, the adversary is limited in online access $M_1, \ldots M_n$. Beside this all the other settings are exactly the same in the above analyzed case $n = 1$. If we let $M = M_1 + \ldots + M_n$ then we will actually find the exact same upperbound of the distinguishing advantage, and thus the inner-keyed sponge construction stays a PRF. The proof is somewhat similar, and we refer the reader to section 4.2 of Andreeva et al. [2] for the proof.

4.3 Proof of security of OKS

Proving the security of the outer-keyed sponge construction is similar to the inner-keyed sponge construction. The settings are the same as in the previous section, but now instead of IKS we

have OKS with a key $K \xleftarrow{\$} \{0,1\}^k$, where k is the key length which is a multiple of the rate r. The upperbound of the distinguishing advantage of an adversary has been formalized by Andreeva et al.[2] in section 5. The first step in the security proof is similar and will also find part of the upperbound of $\frac{M^2}{2^c}$. The second part of the proof is sightly different though, because the key derivation function, as in (7) plays a role. This key derivation function needs to be included in the transcript, which will change the upperbound:

$$\mathbf{Adv}^{\mathcal{RO}}_{\mathrm{OKS}^f_K}(\mathcal{A}) \leq \frac{M^2 + 2\mu N}{2^c} + \lambda(N) + \frac{2(\frac{k}{r})N}{2^b}, \tag{19}$$

where $\lambda(N) \leq \min\left\{\frac{N^2}{2^{c+1}} + \frac{N}{2^k}, \frac{1}{2^b} + \frac{N}{2^{\frac{1}{2} - \frac{\log_2(3b)}{2r} - \frac{1}{r}}}\right\}$. This upperbound looks much more complicated, but important here is the fact that it can be made negligible small by choosing the capacity c and key length k large enough. In the general case where the adversary has access to n instances of the OKS construction, the distinguishing advantage upperbound becomes:

$$\mathbf{Adv}^{\mathcal{RO}}_{\mathrm{OKS}^f_K}(\mathcal{A}) \leq \frac{M^2 + 2\mu N}{2^c} + n\lambda(N) + \frac{\binom{n}{2}}{2^k} + \frac{2(\frac{k}{r})(nN + \binom{n}{2})}{2^b}. \tag{20}$$

The conclusion in this case is the same, since also this advantage becomes negligible small for large enough k and c. So we can conclude that the outer-keyed sponge construction is also safe against a forgery attack with chosen plaintext.

5 Quantum security

In the previous chapter we've described the security of the various keyed sponge constructions in a classical setting. This showed that the constructions are safe to use right now, but that doesn't mean it will be safe to use them in the future. The reason for this is the development of quantum computers. Quantum computers actually already exist, but they're computers used in a lab, which are on a very small scale. They're extremely expensive to make and are complex to use. The usefulness of quantum computers is that they don't use the classical 0 and 1 bits, but they use qubits which are in a superposition of 0 and 1. The properties of these qubits can be used to make various quantum algorithms, which can be useful for attacks on various cryptographic primitives that are currently being widely used. Because of this quantum-secure cryptography arose, which aims for secure cryptography in a quantum world. In section 5.1 some of the most important quantum algorithms are described. Then in section 5.2 we describe a common construction which is used to make a MAC, called CBC-MAC. We look specifically at this attack since it has a similar construction compared to the sponge construction, and we describe a quantum attack in section 5.3, that shows that this construction is broken in a quantum world. We also demonstrate an implementation of this attack in section 5.4. Then in section 5.5 and 5.6 we argue about the security of the keyed sponge construction in a quantum world.

5.1 Algorithms

Quantum attacks rely on applying quantum algorithms in a smart way. The most well known algorithm was developed in 1994 by Peter W. Shor, and therefore is called Shor's algorithm [17]. Then another noteworthy algorithm was developed by Lov K. Grover, in 1996 [11]. But what we're most interested in this work is Simon's problem, which is solved by Simon's algorithm, developed by Daniel R. Simon in 1994 [18].

Shor's algorithm [17] Shor's algorithm is an algorithm that factos integers. Integer factorization is finding the prime composition of an integer N. So far no classical algorithm has been found that can do this in polynomial time, but it also has not been proven that such an algorithm doesn't exist. Though quantumly, Shor's algorithm is able to factorize an integer N in polynomial time, namely in $\mathcal{O}((\log N)^2(\log\log N)(\log\log\log N))$ runtime. Some cryptographic primitives rely on the difficulty of integer factorization, such as the widely used RSA scheme. Shor's algorithm can thus be used to create an attack on the RSA cryptosystem and other cryptosystems that rely on integer factorization and the discrete logarithm problem.

Grover's algorithm [11] Grover's algorithm is a quantum algorithm which acts on a black box function. Such a black box function f has a domain of size N, so it has N different possible inputs. Now if we know that for some unknown input x, the output is y, how hard is it to find x? Classically, we will simply have to try all x, and in the worst case the last input we try results in y. This gives a complexity of $\mathcal{O}(N)$. But Grover's algorithm is able to do this faster, namely with a complexity of $\mathcal{O}(\sqrt{N})$. This means it is quadratically faster than the classical approach. If a cryptographic function uses a key K of size n, so $K \in \{0,1\}^n$, than classically it would take roughly 2^n attempts to find K. But with Grover's algorithm it would take roughly $\sqrt{2^n} = 2^{n/2}$ attempts. Therefore it is advised to simply double the currently advised key length, to prevent attacks that use this algorithm.

Simon's problem and algorithm [18] Simon's algorithm is a quantum algorithm which solves the following problem:

Definition 6 (Simon's problem) *Given a black box function $f : \{0,1\}^n \to \{0,1\}^n$, and the promise that there exists some $s \in \{0,1\}^n$ such that for all $x, y \in \{0,1\}^n$ we have $f(x) = f(y) \iff x \oplus y \in \{0^n, s\}$, find s.*

We have that $x \oplus y = 0^n$ if and only if $x = y$. In the case that $x \neq y$ we have that from $x \oplus y = s$ it follows that $y = x \oplus s$, and thus $f(y) = f(x \oplus s)$. But we also know that $f(y) = f(x)$. So if we have found an s then we can use it for the fact that for every x, we have that $f(x) = f(x \oplus s)$. Clasically this problem is very hard to solve since you need to find a pair of inputs x and y such that $f(x) = f(y)$. This is similar to finding a collision of f, which is as hard as the birthday bound, so in the worst case an algorithm that finds these inputs has a complexity of $\mathcal{O}(2^{n/2})$, and this worst case also holds for the average case. Simon's algorithm however is able to solve it in just $\mathcal{O}(n)$. How the algorithm works is described with an example in section 5.4.

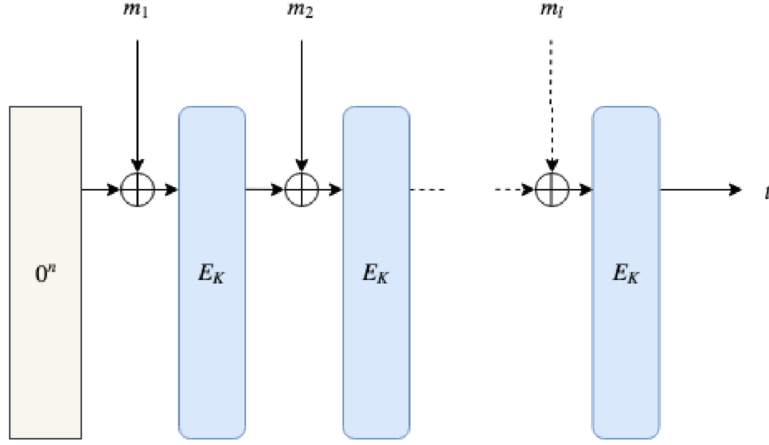

Figure 6: Visual representation of the fixed length CBC-MAC

5.2 CBC-MAC

Probably the most common method to make a Message Authentication Code is the method called 'cipher block chaining message authentication code', or for short CBC-MAC. It builds a MAC based on a block cipher. It is actually very similar to the absorbing phase of the sponge construction, where the key is applied to the block cipher used. The difference though is that CBC-MAC has no inner state. Let E_K be a block cipher that uses the key K and acts on n bits, so $E_K : K \times \{0,1\}^n \to \{0,1\}^n$. In practice the block cipher often used is the Advanced Encryption Standard (AES), with $n = 128$ and a key of length 128, 192 or 256 bits. Let m be a message with length a multiple of n, then we can cut it in l blocks, so $m = m_1||m_2||\ldots||m_l$. Then CBC-MAC is defined as:

$$\text{CBC-MAC}^l_{E_K}(m_1||m_2||\ldots||m_l) = E_K(E_K(\ldots E_K(E_K(m_1) \oplus m_2) \oplus \ldots) \oplus m_l) =: t. \tag{21}$$

Note that in this definition the message has to be of length $l \cdot n$, and therefore it is called fixed length CBC-MAC. A visual representation is shown in figure 6. The reason that we only allow fixed length messages, is because variable length messages allow to break this basic construction. A MAC construction is said to be broken when an attacker is able to make $q+1$ tags by querying only q messages. So if an attacker queries message m and m' and gets tags t and t' respectively, then what he can do is make a third message $m'' = m||(m'_1 \oplus t)||m'_2||\ldots||m'_x$. Now we look at what happens if we evaluate m'':

$$\begin{aligned}
\text{CBC-MAC}_{E_K}(m'') &= \text{CBC-MAC}_{E_K}(m||m'_1 \oplus t||m'_2||\ldots||m'_x) \\
&= E_K(E_K(\ldots E_K(E_K(\text{CBC-MAC}_{E_K}(m) \oplus (m'_1 \oplus t)) \oplus m'_2) \oplus \ldots) \oplus m'_x) \\
&= E_K(E_K(\ldots E_K(E_K(t \oplus (m'_1 \oplus t)) \oplus m'_2) \oplus \ldots) \oplus m'_x) \\
&= E_K(E_K(\ldots E_K(E_K(m'_1) \oplus m'_2) \oplus \ldots) \oplus m'_x) \\
&= t'.
\end{aligned} \tag{22}$$

So this means that t' is a valid tag for m'', and thus the attacker has made valid tags for three messages by only making two queries, and therefore CBC-MAC with variable length input is broken. This attack can be countered by for example prepending the length of the message, or by encrypting the last block by a different key.

5.3 Quantum attack on CBC-MAC

Being able to break CBC-MAC would have serious consequences of any system using this as message authentication code, since it enables the attacker to identify as an other person and also allows to change messages. So it will cause authenticity and integrity to not hold anymore. For the quantum attack we analyze we will only look at the fixed length version, but this attack can also be adapted to work for the length prepending alternative. Both Kaplan et al. [13] and Santoli et al. [16] describe an attack using Simon's algorithm. They actually had contact in the early phase but both finished their attack independently. The setting of the attack is that we have to make a valid tag for a message without querying the message itself. To do this we have the ability to make queries to the fixed length CBC-MAC construction, which returns us a tag that we can use to our

advantage. The attack is a chosen-prefix attack, so every message we query has a certain prefix. The attack will exploit the construction itself, and not the underlying block cipher, so instead of using a block cipher with a key, E_K, we can just assume that this is some random permutation.

The attack We want to attack a fixed length CBC-MAC construction which accepts inputs of length $l \cdot n$, where l is the amount of blocks, and n is the length of 1 block. The block cipher is a pseudorandom permutation π with domain and range of size n. The prefix is fixed and denoted by $\alpha_1 \in \{0,1\}^n$, which is exactly one block long. We have query access to CBC-MAC_π^l which returns a tag $t \in \{0,1\}^n$ on input $m \in \{0,1\}^{l \cdot n}$. The attack follows these steps:

1. Choose a second prefix $\alpha_0 \in \{0,1\}^n$ such that $\alpha_0 \neq \alpha_1$. This prefix will be used in future evaluations.
2. Define the following function for $j = 1, \dots, l-1$:

$$\begin{aligned} g_j : \{0,1\} \times \{0,1\}^n &\longrightarrow \{0,1\}^n \\ b, x &\longmapsto \text{CBC-MAC}_\pi^l(\alpha_b || 0^{(j-1)n} || x || 0^{(l-j-1)n}) \end{aligned} \tag{23}$$

   Note that in the input to g_j, most input blocks are simply zero, and XORing zero does nothing, so we use the notation π^m when the permutation is applied m times. Using this we get the equivalence

$$\text{CBC-MAC}_\pi^l(\alpha_b || 0^{(j-1)n} || x || 0^{(l-j-1)n}) = \pi^{l-j}(\pi^j(\alpha_b) \oplus x). \tag{24}$$

   For two inputs (b_u, u) and (b_v, v) to have equivalent outputs we get:

$$\begin{aligned} g_j(b_u, u) = g_j(b_v, v) &\iff \pi^{l-j}(\pi^j(\alpha_{b_u}) \oplus u) = \pi^{l-j}(\pi^j(\alpha_{b_v}) \oplus v) \\ &\iff \pi^j(\alpha_{b_u}) \oplus u = \pi^j(\alpha_{b_v}) \oplus v \\ &\iff u \oplus v = \pi^j(\alpha_{b_u}) \oplus \pi^j(\alpha_{b_v}) \\ &\iff \begin{cases} u \oplus v = 0^n & \text{if } b_u = b_v \\ u \oplus v = \pi^j(\alpha_0) \oplus \pi^j(\alpha_1) & \text{if } b_u \neq b_v \end{cases} \end{aligned} \tag{25}$$

3. We can now use Simon's algorithm for $j = 1, \dots, l-1$ with g_j as black box function to find $s = 1 || (\pi^j(\alpha_0) \oplus \pi^j(\alpha_1))$. Then by dropping the first bit we define $z^j := \pi^j(\alpha_0) \oplus \pi^j(\alpha_1)$.
4. Create the message $m = \alpha_1 || z^1 || z^2 || \dots || z^{l-1}$, and let its tag be

$$t = \begin{cases} \text{CBC-MAC}_\pi^l(\alpha_0 || 0^{(l-1)\cdot n}) = \pi^l(\alpha_0) & \text{if } l \text{ is even} \\ \text{CBC-MAC}_\pi^l(\alpha_1 || 0^{(l-1)\cdot n}) = \pi^l(\alpha_1) & \text{if } l \text{ is odd} \end{cases}. \tag{26}$$

Now we have to verify if the produced tag t is a valid tag for the message m. We do this by induction with the base cases $l = 3$ and $l = 4$. We start with $l = 3$:

$$\begin{aligned} \text{CBC-MAC}_\pi^3(\alpha_1 || z^1 || z^2) &= \pi(\pi(\pi(\alpha_1) \oplus z^1) \oplus z^2) \\ &= \pi(\pi(\pi(\alpha_1) \oplus (\pi(\alpha_0) \oplus \pi(\alpha_1))) \oplus z^2) \\ &= \pi(\pi(\pi(\alpha_0)) \oplus z^2) \\ &= \pi(\pi^2(\alpha_0) \oplus z^2) \\ &= \pi(\pi^2(\alpha_0) \oplus (\pi^2(\alpha_0) \oplus \pi^2(\alpha_1))) \\ &= \pi(\pi^2(\alpha_1)) \\ &= \pi^3(\alpha_1) \\ &= t \end{aligned} \tag{27}$$

And for the base case $l = 4$, using the previous result, we get:

$$\begin{aligned} \text{CBC-MAC}_\pi^4(\alpha_1 || z^1 || z^2 || z^3) &= \pi(\pi(\pi(\pi(\alpha_1) \oplus z^1) \oplus z^2) \oplus z^3) \\ &= \pi(\pi^3(\alpha_1) \oplus z^3) \\ &= \pi(\pi^3(\alpha_1) \oplus (\pi^3(\alpha_0) \oplus \pi^3(\alpha_1))) \\ &= \pi(\pi^3(\alpha_0)) \\ &= \pi^4(\alpha_0) \\ &= t \end{aligned} \tag{28}$$

If we have $l+1$ even then we know l is odd so for the inductive step we get:

$$\begin{aligned}
\text{CBC-MAC}_{\pi}^{l+1}(\alpha_1||z^1||z^2||\dots||z^{l-1}||z^l) &= \pi(\text{CBC-MAC}_{\pi}^{l}(\alpha_1||z^1||z^2||\dots||z^{l-1}) \oplus z^l) \\
&= \pi(\pi^l(\alpha_1) \oplus z^l) \\
&= \pi(\pi^l(\alpha_1) \oplus (\pi^l(\alpha_0) \oplus \pi^l(\alpha_1))) \\
&= \pi(\pi^l(\alpha_0)) \\
&= \pi^{l+1}(\alpha_0) \\
&= \text{CBC-MAC}_{\pi}^{4}(\alpha_{|}|0^{l\cdot n}).
\end{aligned} \tag{29}$$

If we have $l+1$ odd then we know l is even so for the inductive step we get:

$$\begin{aligned}
\text{CBC-MAC}_{\pi}^{l+1}(\alpha_1||z^1||z^2||\dots||z^{l-1}||z^l) &= \pi(\text{CBC-MAC}_{\pi}^{l}(\alpha_1||z^1||z^2||\dots||z^{l-1}) \oplus z^l) \\
&= \pi(\pi^l(\alpha_0) \oplus z^l) \\
&= \pi(\pi^l(\alpha_0) \oplus (\pi^l(\alpha_0) \oplus \pi^l(\alpha_1))) \\
&= \pi(\pi^l(\alpha_1)) \\
&= \pi^{l+1}(\alpha_1) \\
&= \text{CBC-MAC}_{\pi}^{4}(\alpha_{|}|1^{l\cdot n}).
\end{aligned} \tag{30}$$

And thus by induction we've shown that the attack indeed creates a valid tag for a message that has not been queried. Therefore by definition CBC-MAC is broken in a quantum world. However this attack itself is only useful with specific settings, since it requires a prefix, and that is not always ensured in a real world attack. Beside this, we can argue whether the forged message we create is of any use. The message is made up of z^j which are created by applying permutations to α_0 and α_1, and since we're free to choose α_0, we have some influence on what the message looks like. But we have no information on the permutation, or in the real world, a block cipher with a key. We don't know the key so we have no idea how our z^j will look like, and whether this translates to something useful. However the largest issue with the attack is that it requires the honest users to use a quantum computer. This is because it needs to be able to make queries in a superposition. For the honest users there is no requirement to use a quantum computer, as it obviously suffices to use a classical computer for CBC-MAC to work. But showing that there are methods to break the construction is enough, and will likely lead to more powerful attacks that give the attacker more influence of what the forged message looks like.

5.4 Quantum attack on CBC-MAC implementation

In order to get a better understanding of Simon's algorithm and the above described attack, we will demonstrate the attack by implementing it in Python, with the help of the library Numpy and a library called Qiskit [1] for simulation of quantum algorithms. Obviously we don't own a large scale quantum computer, therefore we will demonstrate it with a small example, namely with $n = 3$. But first we perform a simulation with advice, which means we avoid having to use Simon's algorithm, and forge the z^j simply by using the permutation π.

Simulation with advice In order to simulate we need to define the block length n and the amount of blocks l. After that we can generate the prefix α_1, as well as α_0 which is free to chose, and we let both have a length of one block. For the CBC-MAC we need to decide what cipher it uses, and as described in the attack, we never attack the cipher nor the key, so we can simply choose a permutation π. The permutation we choose is the 'roll' function which is implemented in Numpy, which shifts every bit one or more place. For example, with an input of 3 bits and a shift

of 1 place, the permutation is given as:

$$\begin{aligned}\pi : \{0,1\}^3 &\longrightarrow \{0,1\}^3 \\ (0,0,0) &\mapsto (0,0,0) \\ (0,0,1) &\mapsto (1,0,0) \\ (0,1,0) &\mapsto (0,0,1) \\ (0,1,1) &\mapsto (1,0,1) \\ (1,0,0) &\mapsto (0,1,0) \\ (1,0,1) &\mapsto (1,1,0) \\ (1,1,0) &\mapsto (0,1,1) \\ (1,1,1) &\mapsto (1,1,1).\end{aligned} \tag{31}$$

This permutation suffices for this attack, but if we wanted to simulate other attacks we would need an actual block cipher with a key. The CBC-MAC function has as input the block length n, the amount of blocks l and a message m, which is an array of $l \cdot n$ zeroes and ones. The first step is cutting the message in l blocks of length n. The function creates a variable output, which is initialized as an array of zeroes, and then the input message blocks are repeatedly XORed and the permutation is applied to update the output. After all the message blocks are processed, the output is returned which is the tag represented by an array of one block length. We also create a verify function which takes as input a message and tag, and returns 'True' if it is a valid tag, and 'False' if it isn't. Now to forge the message we use the advice and simply make the z^j by using the permutation. In a real scenario an attacker wouldn't have access to this since you need the key for it. Based on the parity of l we also create the tag by using the CBC-MAC function, and we use the verify function to verify if we indeed forged a correct tag. The code is given in appendix A, with an example where $l = 100$ and $n = 64$, which will always return that the forged message and tag is a valid pair. The chosen prefixes α_0 and α_1 are uniformly random chosen, so this shows that the attack doesn't depend on the chosen prefixes. Furthermore, the parameters l and n are kept variable and can be easily changed. Also the permutation can be changed by plugging in any other permutation, so even some keyed block cipher. By playing around with these parameters and permutation and checking whether the code still returns that the forged message and tag are a valid pair, we can get a decent idea whether the attack works.

Simulation with Simon's algorithm The full simulation doesn't make use of the advice as described above, since an attacker does not have query access to the block cipher directly, since he would need the key for this. Instead we use an implementation of Simon's algorithm, which has been implemented by the Qiskit quantum library[1]. Beside this the attack looks exactly the same as in the simulation with advice. We can only have few qubits for Simon's algorithm, so the permutation we choose has to be small, therefore we choose for the block length $n = 3$. We also don't want to simulate messages that are too long, so for the amount of blocks we choose $l = 6$. We need to apply Simon's algorithm on the function g_j for every $j = 1, \ldots, l-1$, so we need to apply it $l-1$ times. This is done in Qiskit by a truth table oracle, which first has to be generated. Note that g_j is a function with $m = n+1 = 4$ bits in the input, and with $n = 3$ bits in the output. For notation we now let $g_j(b||x) := g_j(b,x)$. The quantum circuit that is being used in Simon's algorithm is shown in figure 7, where $H^{\otimes 4}$ is the Hadamard transform on 4 qubits, defined as:

$$H^{\otimes 4} |a\rangle = \sum_{b \in \{0,1\}^4} (-1)^{a \cdot b} |b\rangle , \tag{32}$$

where $a \cdot b = a_1 * b_1 + \ldots + a_4 * b_4$ with $*$ the symbol for the product. The U_{g_j} in the circuit is a unitary operator, defined as:

$$U_{g_j} |a\rangle |b\rangle = |a\rangle |b \oplus g_j(a)\rangle . \tag{33}$$

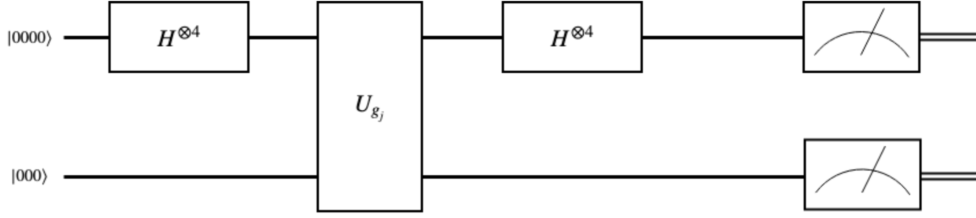

Figure 7: The quantum circuit that is used for Simon's algorithm

The first step in Simon's algorithm is to make two registers, the first with m zero qubits and the second with n zero qubits:

$$|\psi\rangle = |0000\rangle\,|000\rangle = \left|0^4\right\rangle\left|0^3\right\rangle. \tag{34}$$

Then the first Hadamard transform $H^{\otimes 4}$ is applied on the first register. We get:

$$\begin{aligned}|\psi\rangle &= (H^{\otimes 4}\left|0^4\right\rangle)\left|0^3\right\rangle \\ &= \left(\frac{1}{\sqrt{2^4}}\sum_{x\in\{0,1\}^4}(-1)^{x\cdot 0000}\,|x\rangle\right)\left|0^3\right\rangle \\ &= \left(\frac{1}{4}\sum_{x\in\{0,1\}^4}|x\rangle\right)\left|0^3\right\rangle.\end{aligned} \tag{35}$$

Next the unitary operator U_{g_j} is applied. It takes the value x of the first register, and applies the function on this value and stores it in the second register by XORing it, so we get:

$$\begin{aligned}|\psi\rangle &= \frac{1}{4}\sum_{x\in\{0,1\}^4}U_{g_j}\,|x\rangle\left|0^3\right\rangle \\ &= \frac{1}{4}\sum_{x\in\{0,1\}^4}|x\rangle\left|0^3\oplus g_j(x)\right\rangle \\ &= \frac{1}{4}\sum_{x\in\{0,1\}^4}|x\rangle\,|g_j(x)\rangle.\end{aligned} \tag{36}$$

Then the second register is measured, which returns a $g_j(x)$ chosen uniform over all $x \in \{0,1\}^4$. Note that exactly one of x or $x \oplus s$ as input results in $g_j(x)$, so the first register collapses:

$$|\psi\rangle = \frac{1}{\sqrt{2}}\left(|x\rangle + |x\oplus s\rangle\right)|g_j(x)\rangle. \tag{37}$$

After measuring, we apply the second Hadamard transform on the first register and get:

$$\begin{aligned}|\psi\rangle &= \frac{1}{\sqrt{2}}\left(H^{\otimes 4}\left(|x\rangle + |x\oplus s\rangle\right)\right)|g_j(x)\rangle \\ &= \frac{1}{\sqrt{2}}\left(H^{\otimes 4}\,|x\rangle + H^{\otimes 4}\,|x\oplus s\rangle\right)|g_j(x)\rangle \\ &= \frac{1}{\sqrt{2}}\left(\frac{1}{4}\sum_{y\in\{0,1\}^4}(-1)^{x\cdot y}\,|y\rangle + \frac{1}{4}\sum_{y\in\{0,1\}^4}(-1)^{(x\oplus s)\cdot y}\,|y\rangle\right)|g_j(x)\rangle \\ &= \frac{1}{4\sqrt{2}}\sum_{y\in\{0,1\}^4}\left((-1)^{x\cdot y}\,|y\rangle + (-1)^{(x\oplus s)\cdot y}\,|y\rangle\right)|g_j(x)\rangle.\end{aligned} \tag{38}$$

We can use the fact that $(-1)^{(x\oplus s)\cdot y} = (-1)^{(x\cdot y)\oplus(s\cdot y)} = (-1)^{x\cdot y}(-1)^{s\cdot y}$, and thus can further simplify:

$$\begin{aligned}|\psi\rangle &= \frac{1}{4\sqrt{2}}\sum_{y\in\{0,1\}^4}\left((-1)^{x\cdot y}\,|y\rangle + (-1)^{x\cdot y}(-1)^{s\cdot y}\,|y\rangle\right)|g_j(x)\rangle \\ &= \frac{1}{4\sqrt{2}}\sum_{y\in\{0,1\}^4}(-1)^{x\cdot y}\left(1+(-1)^{s\cdot y}\right)|y\rangle\,|g_j(x)\rangle.\end{aligned} \tag{39}$$

Now we measure the first register to get a y. Note that the probability of measuring some y can be zero, namely when the amplitude is zero:

$$\begin{aligned} & (-1)^{x\cdot y}\left(1+(-1)^{s\cdot y}\right)=0 \\ \iff & 1+(-1)^{s\cdot y}=0 \\ \iff & y\cdot s\equiv 1 \mod 2. \end{aligned} \tag{40}$$

So for the y we measure we know that $y \cdot s \equiv 0 \mod 2$, and every y that meets this condition has equal probability of being measured. This gives us information about s, but this is not enough yet. Since we have $y \cdot s = y_1 * s_1 + y_2 * s_2 + y_3 * s_3 + y_4 * s_4$, and we know y and want to know s, we can see this as a equation with four unknowns, namely s_1 to s_4. So what we can do is run everything multiple times to get different values of y such that $y \cdot s \equiv 0 \mod 2$. If we repeat this until we have three (linear independent) equations, then we can solve this set of equations to get s. Note that in our case $s = 1||\pi^j(\alpha_0) \oplus \pi^j(\alpha_1)$, so we have to drop the first bit, which should always be 1, to get our $z^j = \pi^j(\alpha_0) \oplus \pi^j(\alpha_1)$. We do this for all $j = 1, \ldots, l-1$ and can forge the message. The Python code for this is given in appendix B, which requires the packages numpy, Qiskit and itertools. Executing the code returns that the forged message and tag are a valid pair.

Runtime analysis With the simulations we verified that we indeed get a correct tag, but for the attack to work we also need to verify that it is forged in a 'fast' time. Fast here means in polynomial time, with as variables the block length n and the amount of blocks l. The attack makes many queries to CBC-MAC, so we first look at its runtime. CBC-MAC first cuts the message in l blocks, having complexity $\mathcal{O}(l)$, then for every block we XOR and apply an n-bit permutation. XORing n bits has complexity $\mathcal{O}(n)$, and the complexity of an n-bit permutation depends on the permutation. So in total CBC-MAC has a runtime of $\mathcal{O}(l) + l \cdot (\mathcal{O}(n) + c)$. To simplify this analysis we assume that the permutation has a complexity of $\mathcal{O}(n)$, resulting in a total runtime for CBC-MAC of $\mathcal{O}(l \cdot n)$. The reason that we are allowed to make this simplification is because the attacker doesn't choose the permutation, and choosing a permutation of this complexity will give more accurate results of the runtime analysis since it allows quicker simulations. Then for the runtime of the attack:

- In step 1 we make α_0, which simply has complexity $\mathcal{O}(n)$.
- In step 2 we make the function g_j, which requires an invocation to CBC-MAC for every $j = 1, \ldots, l-1$, so it has complexity $\mathcal{O}(l) \cdot \mathcal{O}(l \cdot n) = \mathcal{O}(l^2 \cdot n)$.
- In step 3 we forge z^j for every $j = 1, \ldots, l-1$ by applying Simon's algorithm. The function g_j has as input $n+1$ bits, so Simon's algorithm has a complexity $\mathcal{O}(n+1)$, so in total this step has a complexity $\mathcal{O}(l) \cdot \mathcal{O}(n+1) = \mathcal{O}(l \cdot n)$.
- In step 4 we forge the message by concatenating l messages, which has complexity $\mathcal{O}(l)$. Making a tag requires a single query to CBC-MAC, and checking the parity of l has complexity $\mathcal{O}(1)$. So in total this step has complexity $\mathcal{O}(l) + \mathcal{O}(l \cdot n) \cdot \mathcal{O}(1) = \mathcal{O}(l \cdot n)$.

So in total we find that the attack has a runtime of

$$\mathcal{O}(n) + \mathcal{O}(l^2 \cdot n) + \mathcal{O}(l \cdot n) + \mathcal{O}(l \cdot n) = \mathcal{O}(l^2 \cdot n). \tag{41}$$

We want to verify this runtime as much as possible, though we're limited in steps 2 and 3 since Qiskits Simon's algorithm only allows enough qubits for up to $n = 5$. However we can verify that step 1 and 4 together have runtime of $\mathcal{O}(l \cdot n)$. We do this by using our attack with simulation, and keep track how much time it takes for the attack to execute, minus the time it took to calculate the advice part. First we fix the amount of blocks at $l = 50$, and we let n increase from 100 up to 10000 in steps of 100 each. The hypothesis is then a runtime of $\mathcal{O}(n)$. The results are shown in figure 8, with n on the horizontal axes and the time in microseconds on the vertical axes. Note that there are some outliers with a higher time than expected. The reason that these occur is most likely because of unexpected background processes. Therefore these outliers can be ignored.

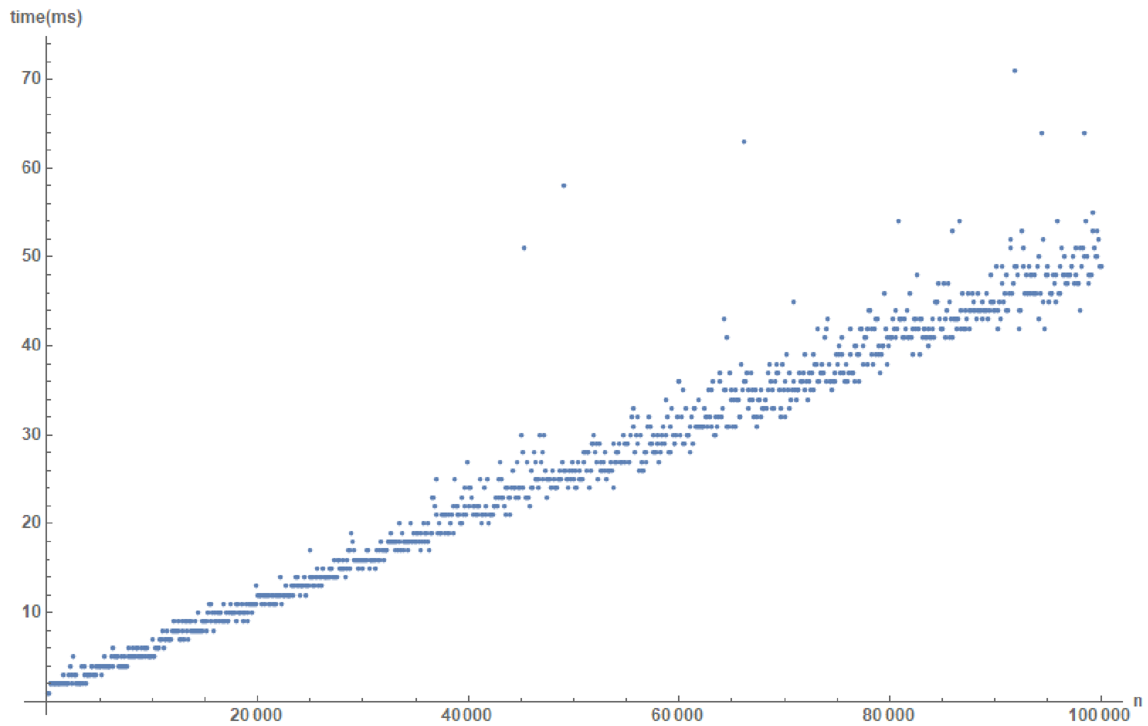

Figure 8: Plot showing runtime of CBC-MAC attack steps 1 and 4, with fixed $l = 50$ and increasing n.

In order to verify our hypothesis, we normalize the data by dividing every time value by its corresponding value of n. Now the hypothesis is true, if the plotted data is eventually constant. The plot is shown in figure 9.

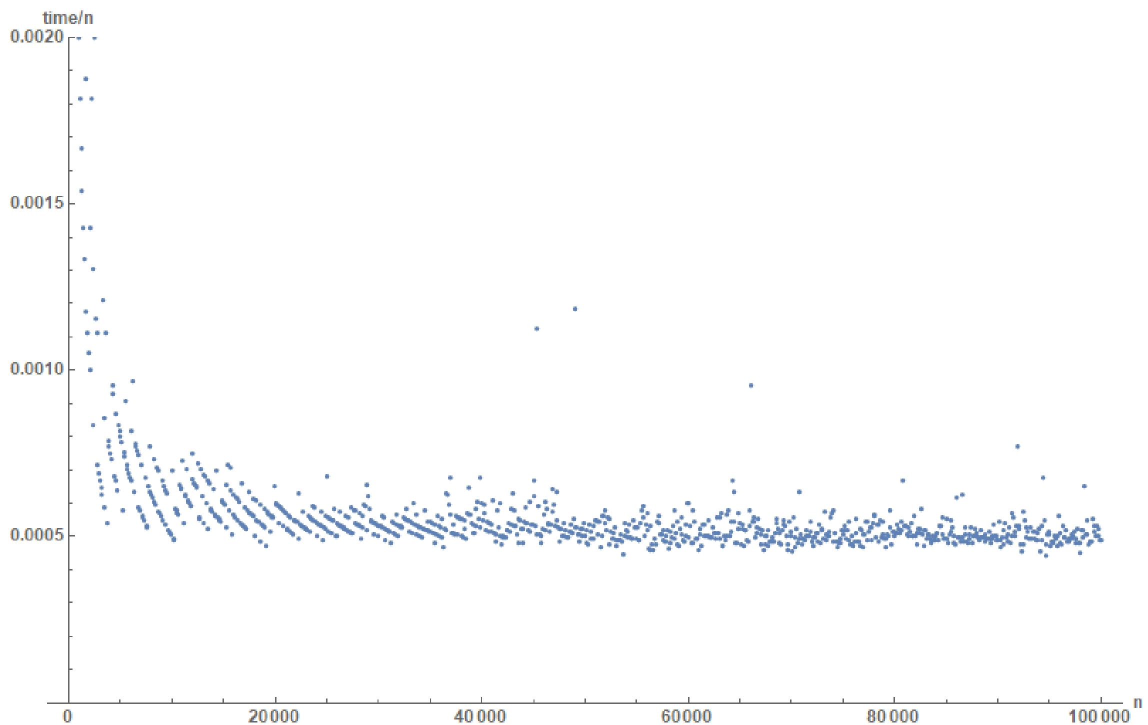

Figure 9: Plot showing same data as in figure 8, but with values divided by n to fit the hypothesis of the runtime $\mathcal{O}(n)$ for fixed l.

The results show that for a fixed l, the runtime is $\mathcal{O}(n)$, since the data is constant. The lines that occur for smaller values are due to rounding errors when measuring the time. Next we can do the same thing, but fix $n = 10$ and let l increase. The same plots are shown in figure 10 and 11, and they confirm that for a fixed n, the runtime is $\mathcal{O}(l)$.

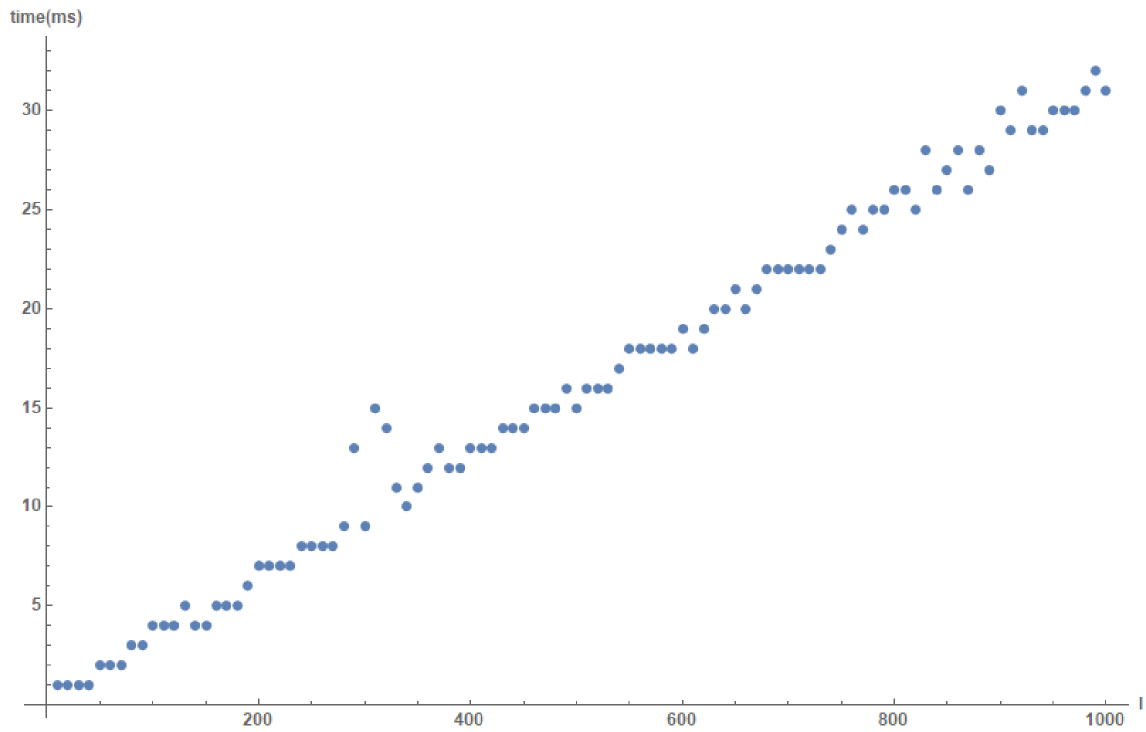

Figure 10: Plot showing runtime of CBC-MAC attack steps 1 and 4, with fixed $n = 10$ and increasing l.

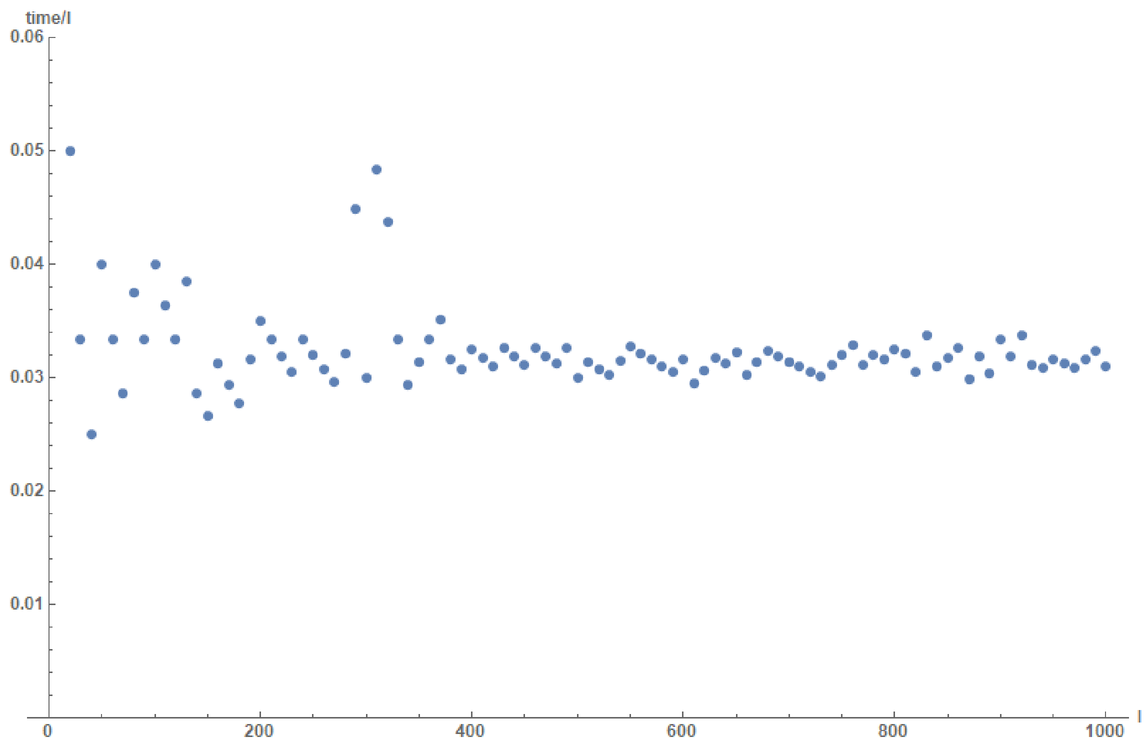

Figure 11: Plot showing same data as in figure 10, but with values divided by l to fit the hypothesis of the runtime $\mathcal{O}(l)$ for fixed n.

These two results together do not yet confirm the runtime of $\mathcal{O}(l \cdot n)$. To confirm this we let both n and l increase. The results are shown in a 3D plot in figure 12.

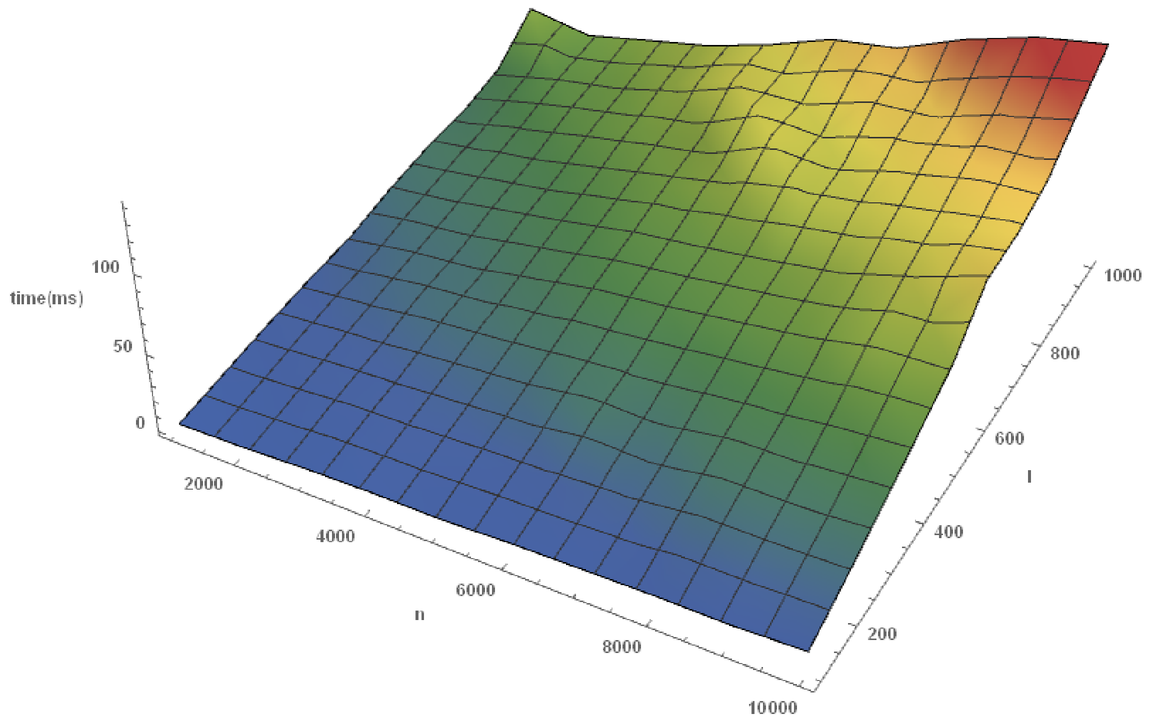

Figure 12: Plot showing runtime of CBC-MAC attack steps 1 and 4, with increasing n and l.

A normalized plot, where the values are divide by $l \cdot n$, is shown in figure 13.

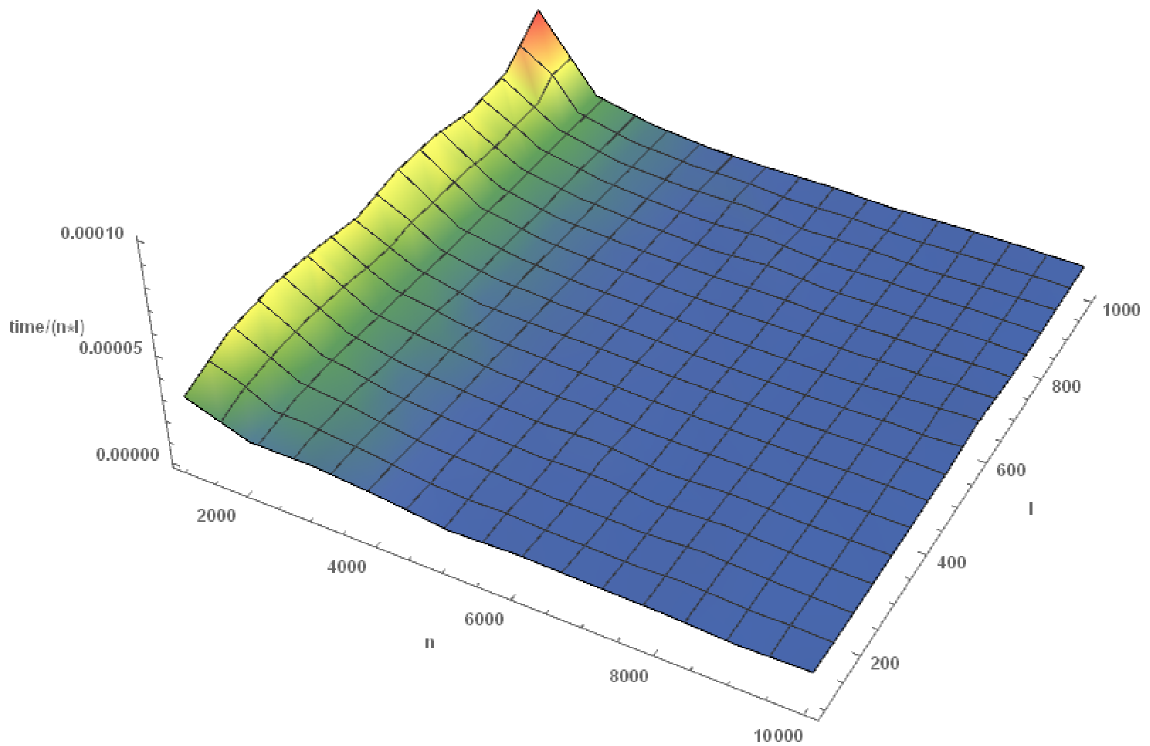

Figure 13: Plot showing same data as in figure 12, but with values divided by $l \cdot n$ to fit the hypothesis of the runtime $\mathcal{O}(l \cdot n)$.

From this figure we can conclude that the runtime of steps 1 and 4 is $\mathcal{O}(l \cdot n)$, since the data is constant when both l and n increase. Next we want to verify the runtime of steps 2 and 3. We can not verify that it is $\mathcal{O}(l^2 \cdot n)$, since we have insufficient data of n. Therefore we need to fix $n = 3$ and let l increase. We use the full simulation and only track the time of the part where we use Simon's algorithm. The results are shown in figure 14.

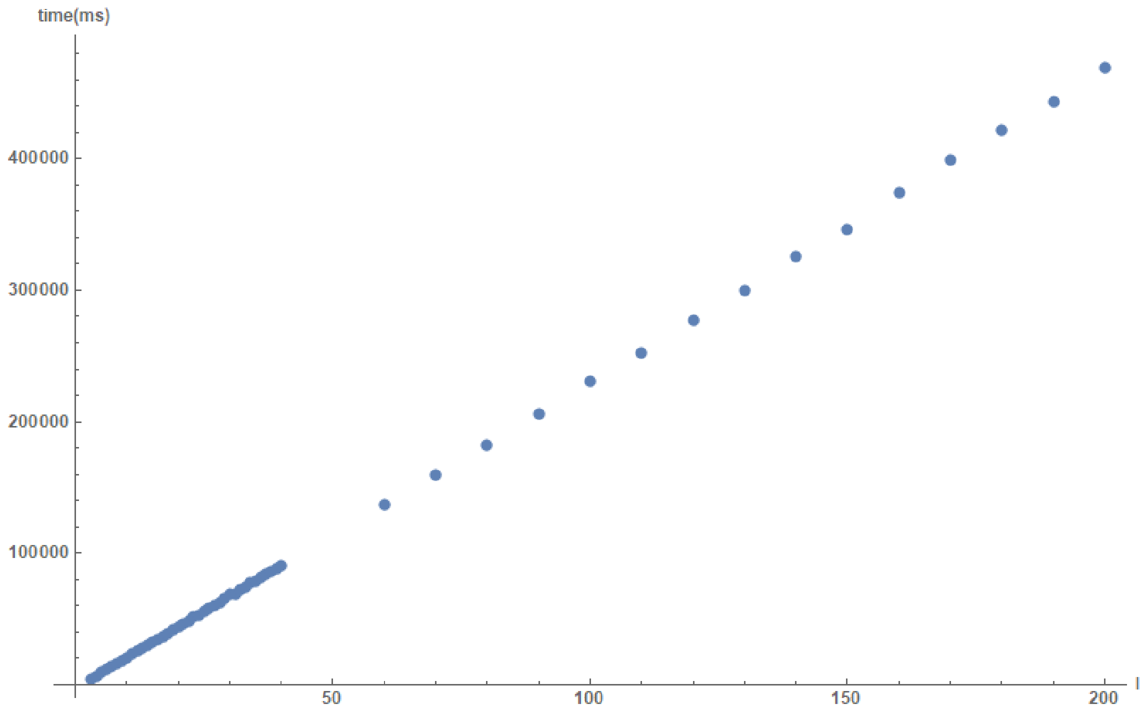

Figure 14: Plot showing runtime of CBC-MAC attack steps 2 and 3 (Simon's algorithm), with fixed $n = 3$ and increasing l.

Somewhat surprisingly, the results show a linear runtime, namely one of $\mathcal{O}(l)$. The reason for this is actually simple. Step 2, which theoretically dominates the runtime with $\mathcal{O}(l^2 \cdot n)$, is quite fast. We can make this conclusion based on our previous tests where we saw that for small n and l (less than 100) an invocation to CBC-MAC will just take a few microseconds. For step 3, where we apply Simon's algorithm, our implementation is much slower, namely multiple seconds and even minutes for larger l. So because of the large constants, step 3 with runtime $\mathcal{O}(l)$ dominates over step 2 with runtime $\mathcal{O}(l^2)$. To overcome this, we would need to simulate with much larger values of l, but unfortunately that takes too long to simulate. So even though theoretically the total runtime is $\mathcal{O}(l^2 \cdot n)$, in practice it would be much closer to $\mathcal{O}(l \cdot n)$.

5.5 Quantum security of keyed-sponge construction

Since CBC-MAC and the keyed-sponge constructions are very similar, a first intuition would be to try a similar attack on the keyed-sponges. However this attack would fail, and the reason for this is the inner state of the sponge. If we look at the security proof of the inner and outer-keyed sponges in section 4, we used the fact that we can express them both in a sponge construction with a keyed permutation, namely with the Even-Mansour construction as the keyed permutation. Then we showed that the security depends on two parts, one from distinguishing the sponge construction from a random oracle, and the second from distinguishing the Even-Mansour construction from a pseudo-random permutation. We can apply the same approach in a quantum setting, where we now let an adversary have superposition access to the keyed sponge construction and the underlying permutation. Having superposition access increases the power of an adversary. The goal of the adversary still is to distinguish the keyed sponge from a random oracle. By splitting it up in two parts, we need to show that both are a negligible function. The first part where we distinguish a random sponge from a random oracle still gives us a negligible function as upperbound, as shown by Czajkowski et al. [9]:

Theorem 5 (Special case of theorem 8 of Czajkowski et al. [9]) SPONGE^f *with rate* r *and capacity* c *is quantum indistinguishable from a random oracle* $\mathcal{RO}$*. An adversary* $\mathcal{A}$ *which makes*

q quantum queries to $\textsc{Sponge}^f$ *with maximal input length of* $m \cdot r$ *and maximal output length* $z \cdot r$ *has a negligible distinguishing advantage:*

$$|\mathbb{P}(\mathcal{A}^{|\textsc{Sponge}^f\rangle} = 1) - \mathbb{P}(\mathcal{A}^{|\mathcal{RO}\rangle} = 1)| \leq \frac{\pi^2 \cdot (2q(m+z-2))^3}{6 \cdot 2^c}$$

From this theorem we see again that the advantage is negligible, since the numerator is polynomial in the amount of queries and input and output length, and the denominator is exponential in the capacity, which we can choose sufficiently large.
The second part of the distinguishing advantage however isn't secure anymore. The reason for this is that the Even-Mansour construction is broken in a quantum setting, as shown by Kuwakado and Morii [14]. The attack on this construction also uses Simon's algorithm, with the following function:

$$\begin{aligned} g : \{0,1\}^n &\longrightarrow \{0,1\}^n \\ x &\mapsto E_K^f(x) \oplus f(x) = f(x \oplus K) \oplus K \oplus f(x). \end{aligned} \tag{42}$$

For this function we have that:

$$\begin{aligned} & g(x) = g(y) \\ \iff & f(x \oplus K) \oplus K \oplus f(x) = f(y \oplus K) \oplus K \oplus f(y) \\ \iff & f(x \oplus K) \oplus f(x) = f(y \oplus K) \oplus f(y) \\ \iff & f(x \oplus K) \oplus f(y) = f(y \oplus K) \oplus f(x) \\ \iff & x \oplus K = y \\ \iff & x \oplus y = K \end{aligned} \tag{43}$$

And here we have for Simon's promise that $s = K$, and we actually learn the key K. Therefore the Even-Mansour construction is unsafe and it is not advised to use this keyed permutation. This however doesn't directly lead to an attack on IKS or OKS, since in a real attack scenario an attacker doesn't have direct query access to the Even-Mansour construction. Though it does show that the security analysis doesn't hold anymore. In order to create a quantum secure keyed-sponge construction, we need to have that the underlying permutation is quantumly indistinguishable from a random permutation. This introduces a new class of keyed sponges, namely where the key is applied only in the permutation. We will refer to this class as the keyed-internal-function sponge and it has been introduced by Czajkowski et al. [9]. We will refer to it as KFS:

$$\mathrm{KFS}_K^f(m, l) = \textsc{Sponge}^{f_K}(m, l) \tag{44}$$

So instead of a permutation f, we used a keyed permutation f_K. For this class of keyed sponges to be secure we need the permutation to be a quantum secure pseudo-random permutation:

Definition 7 (Quantum secure pseudo-random permutation (qPRP) [9])
Let $f_k : \{0,1\}^k \times \{0,1\}^n \longrightarrow \{0,1\}^n$ *be a keyed permutation that acts on* n *bits, where the key has length* k*. Let* $\pi : \{0,1\}^n \longrightarrow \{0,1\}^n$ *be a permutation that acts on* n *bits, which is uniformly chosen from the set of all permutations, so* $\pi \xleftarrow{\$} \textbf{\textit{Perm}}(n)$*. Then* f_k *is said to be a quantum secure pseudo-random permutation if the advantage of an adversary* $\mathcal{A}$ *that tries to distinguish* f_k *from* π*, by having superposition query access, is a negligible function* ϵ*:*

$$|\mathbb{P}(\mathcal{A}^{|f_k\rangle} = 1) - \mathbb{P}(\mathcal{A}^{|\pi\rangle} = 1)| \leq \epsilon$$

We can now show that the keyed-internal-function sponge is secure in a quantum world.

Theorem 6 ([9]) *Let* KFS_K^f *be a keyed-internal-function sponge, with rate* r*, capacity* c *and underlying permutation* f_k *which is qPRP with advantage* ϵ*, then* KFS_K^f *is a quantum secure function.*

Proof. Let $\mathcal{A}$ be an adversary with the goal to distinguish real world with KFS_K^f from the ideal world with a random oracle $\mathcal{RO}$, by having superposition query access, in which the maximal input length is $m \cdot r$ bits and the maximal output length is $z \cdot r$. Then for the advantage we have:

$$\begin{aligned} \mathbf{Adv}_{|\mathrm{KFS}_K^f\rangle}^{|\mathcal{RO}\rangle}(\mathcal{A}) &= \left|\mathbb{P}(\mathcal{A}^{|\mathrm{KFS}_K^f\rangle} = 1) - \mathbb{P}(\mathcal{A}^{|\mathcal{RO}\rangle} = 1)\right| \\ &= \left|\mathbb{P}(\mathcal{A}^{|\textsc{Sponge}^{f_K}\rangle} = 1) - \mathbb{P}(\mathcal{A}^{|\mathcal{RO}\rangle} = 1)\right|. \end{aligned} \tag{45}$$

Now let $\pi \xleftarrow{\$} \mathbf{Perm}(n)$ be a qPRP, then by using triangle inequality:

$$\begin{aligned}\mathbf{Adv}^{|\mathcal{RO}\rangle}_{|\text{KFS}^f_K\rangle}(\mathcal{A}) &= \left|(\mathbb{P}(\mathcal{A}^{|\textsc{Sponge}^{f_K}\rangle} = 1) - \mathbb{P}(\mathcal{A}^{|\textsc{Sponge}^{\pi}\rangle} = 1)) - (\mathbb{P}(\mathcal{A}^{|\textsc{Sponge}^{\pi}\rangle} = 1) + \mathbb{P}(\mathcal{A}^{|\mathcal{RO}\rangle} = 1))\right| \\ &\leq \left|(\mathbb{P}(\mathcal{A}^{|\textsc{Sponge}^{f_K}\rangle} = 1) - \mathbb{P}(\mathcal{A}^{|\textsc{Sponge}^{\pi}\rangle} = 1))\right| + \left|(\mathbb{P}(\mathcal{A}^{|\textsc{Sponge}^{\pi}\rangle} = 1) + \mathbb{P}(\mathcal{A}^{|\mathcal{RO}\rangle} = 1))\right| \\ &\leq \left|(\mathbb{P}(\mathcal{A}^{|f_K\rangle} = 1) - \mathbb{P}(\mathcal{A}^{|\pi\rangle} = 1))\right| + \left|(\mathbb{P}(\mathcal{A}^{|\textsc{Sponge}^{\pi}\rangle} = 1) + \mathbb{P}(\mathcal{A}^{|\mathcal{RO}\rangle} = 1))\right|.\end{aligned} \tag{46}$$

Here the first part is the definition of f_K being a qPRP, and for the second part we can apply theorem 5, and we get:

$$\mathbf{Adv}^{|\mathcal{RO}\rangle}_{|\text{KFS}^f_K\rangle}(\mathcal{A}) \leq \epsilon + \frac{\pi^2 \cdot (2q(m+z-2))^3}{6 \cdot 2^c}. \tag{47}$$

So the upperbound of the distinguishing advantage of the adversary is a negligible function, even with superposition query access, and therefore KFS^f_K is secure in a quantum world. □

5.6 Quantum secure pseudo-random permutation

In the previous section we have concluded that a keyed-internal function sponge is secure in a quantum world if and only if the underlying keyed permutation is a quantum secure pseudo-random permutation, which is defined in definition 7. We however didn't specify how we can construct such a qPRP. A recent work of Zhandry [19] describes how a qPRP depends only on the existence of a one way function. The steps are to first make a pseudo-random generator from any one way function. This can then be used to build a pseudo-random function. Using this function in a 4-round feistel network, a qPRP is build. The Advanced Encyption Standard (AES) has become the most popular block cipher, and has been declared as the encryption standard by the National Institute of Standards and Technology (NIST). AES has 3 different versions, AES-128, AES-192 and AES-256,where the number represents the key length being used, and all versions are a permutation on 128 bits. Clasically, AES is a PRP, so it would be most interesting to verify if it also is a qPRP. The most powerful attacks on AES are the ones that utilize Grover's algorithm, which has the effect that all known classical algorithm can be sped-up quadratically. Because of this it is unadvisable to use the AES-128 version, and also the AES-192 is in danger, though the AES-256 still has an extremely small advantage for any efficient quantum adversary and might thus be considered a qPRP.

6 Conclusion and recommendations

Sponge functions have the option of applying keys, which can be used to build a MAC. Two methods are the inner and outer-keyed sponge constructions. Both of them are secure in a classical setting, which follows from a proof that they are pseudo-random functions by using notions of indistinguishability. Currently CBC-MAC is being widely used as a MAC function, and we've shown that with the help of Simon's algorithm, which is a quantum algorithm that requires a large scale quantum computer to be efficient, this function is broken. We verify by simulation that this is a valid attack. However, it is only possible in specific settings, namely when all messages have a prefix. Furthermore for the attack to be successful, the attacker needs a lot of query access to the CBC-MAC function, which isn't always possible in a real scenario. So the challenge for further research could be looking for more powerful attacks, that can be used in a more general case. CBC-MAC being vulnerable is enough reason to look for better alternatives, such as keyed sponges. A good alternative is using sponge functions where the underlying permutation is a keyed permutation. The security of the keyed-internal-function sponge then fully depends on the underlying permutation being a quantum secure pseudo-random permutation. Therefore a quantum secure MAC would for example be the keyed-internal-function sponge with as underlying permutation AES-256.

A Attack with advice

```
import numpy as np
block_length = 64  # length of a single block
block_count = 100  # total amount of blocks, including prefix
alpha_1 = np.random.randint(2, size=block_length)  # the prefix of one block
print("the prefix is", alpha_1)


# permutation function (underlying cipher of cbc-mac, no key needed for this application)
def permutation(b):
    return np.roll(b, 1)


# fixed length CBC-MAC, m=message, l=amount of blocks, n=length per block
def cbcmac(m, l, n):
    message_blocks = np.split(m, l)
    output = np.full(n, 0)
    for i in range(0, l):
        output = np.bitwise_xor(output, message_blocks[i])
        output = permutation(output)
    return output


# verifies message and tag, return True if correct, False if incorrect
def verify(m_ver, t_ver):
    return np.array_equal(cbcmac(m_ver, block_count, block_length), t_ver)


alpha_0 = np.random.randint(2, size=block_length)  # free to choose alpha_0 of one block
while np.array_equal(alpha_0, alpha_1):  # make sure alpha_0 and alpha_1 are different
    alpha_0 = np.random.randint(2, size=block_length)
print("chosen alpha_0 is ", alpha_0)


# start forging a message
z = [alpha_1]

for i in range(1, block_count):  # use advice to avoid Simon's algorithm
    z_0 = alpha_0
    z_1 = alpha_1
    for j in range(i):
        z_0 = permutation(z_0)
        z_1 = permutation(z_1)
    z.append(np.bitwise_xor(z_0, z_1))

m_att = np.array(z).flatten()
print('The forged message is ', m_att)

# check if block length is even or odd, forge tag
if block_count % 2 == 0:
    t = cbcmac(np.append(alpha_0, np.full((block_count-1) * block_length, 0)),
               block_count, block_length)
else:
    t = cbcmac(np.append(alpha_1, np.full((block_count-1) * block_length, 0)),
               block_count, block_length)
print('The forged tag is ', t)
print("verifying the forged message: ", verify(m_att, t))
```

B Attack with Simon's algorithm

```
import numpy as np
import itertools
from qiskit import BasicAer
from qiskit.aqua import QuantumInstance
from qiskit.aqua.algorithms import Simon
from qiskit.aqua.components.oracles import TruthTableOracle
block_length = 3  # length of a single block
block_count = 6  # total amount of blocks, including prefix


# permutation function (underlying cipher of cbc-mac, no key needed for this application)
def permutation(p):
    return np.roll(p, 1)


# fixed length CBC-MAC, m=message, l=amount of blocks, n=length per block
def cbcmac(m, l, n):
    message_blocks = np.split(m, l)
    output = np.full(n, 0)
    for i in range(0, l):
        output = np.bitwise_xor(output, message_blocks[i])
        output = permutation(output)
    return output


# verifies message and tag, return True if correct, False if incorrect
def verify(m_ver, t_ver):
    return np.array_equal(cbcmac(m_ver, block_count, block_length), t_ver)


# define the function g for every j in 1,...,block_length-1
def g(j, b, x):
    message = np.full((block_count, block_length), 0)
    if b == 0:
        message[0] = alpha_0
    else:
        message[0] = alpha_1
    message[j] = x
    return cbcmac(message.flatten(), block_count, block_length)


alpha_1 = np.array([0, 1, 1])
alpha_0 = np.array([1, 0, 1])
# alpha_1 = np.random.randint(2, size=block_length)  # the prefix of one block
# alpha_0 = np.random.randint(2, size=block_length)  # free to choose alpha_0 of one block
while np.array_equal(alpha_0, alpha_1):  # make sure alpha_0 and alpha_1 are different
    alpha_0 = np.random.randint(2, size=block_length)
print("the prefix alpha_1 is", alpha_1)
print("chosen alpha_0 is ", alpha_0)

# start forging a message
z = [alpha_1]

# create the list of all 0/1 inputs
perms = [np.array(x) for x in itertools.product('01', repeat=block_length)]

# run Simon's algorithm on g() via truth table oracles
for j in range(1, block_count):
    bitmaps = ['' for i in range(0, block_length)]  # bitmap used for truth table oracle
    for b in [0, 1]:
        for x in perms:
            gResults = g(j, b, np.array(x))
            for i in range(0, block_length):
                bitmaps[i] += str(gResults[i])
    oracle = TruthTableOracle(bitmaps)
    simon = Simon(oracle)
    backend = BasicAer.get_backend('qasm_simulator')
    result = list(simon.run(QuantumInstance(backend, shots=1024))['result'])
    del result[0]  # always 1
    result = np.array([int(i) for i in result])
    z.append(result)
    print("Simon's algorithm returns: ", result)

m_att = np.array(z).flatten()
```

```python
print('The forged message is ', m_att)

if block_count % 2 == 0:  # check if block length is even or odd, forge tag
    t = cbcmac(np.append(alpha_0, np.full((block_count-1) * block_length, 0)),
               block_count, block_length)
else:
    t = cbcmac(np.append(alpha_1, np.full((block_count-1) * block_length, 0)),
               block_count, block_length)
print('The forged tag is ', t)
print("verifying the forged message: ", verify(m_att, t))
```

References

[1] Gadi Aleksandrowicz, Thomas Alexander, Panagiotis Barkoutsos, Luciano Bello, Yael Ben-Haim, David Bucher, Francisco Jose Cabrera-Hernádez, Jorge Carballo-Franquis, Adrian Chen, Chun-Fu Chen, Jerry M. Chow, Antonio D. Córcoles-Gonzales, Abigail J. Cross, Andrew Cross, Juan Cruz-Benito, Chris Culver, Salvador De La Puente González, Enrique De La Torre, Delton Ding, Eugene Dumitrescu, Ivan Duran, Pieter Eendebak, Mark Everitt, Ismael Faro Sertage, Albert Frisch, Andreas Fuhrer, Jay Gambetta, Borja Godoy Gago, Juan Gomez-Mosquera, Donny Greenberg, Ikko Hamamura, Vojtech Havlicek, Joe Hellmers, Łukasz Herok, Hiroshi Horii, Shaohan Hu, Takashi Imamichi, Toshinari Itoko, Ali Javadi-Abhari, Naoki Kanazawa, Anton Karazeev, Kevin Krsulich, Peng Liu, Yang Luh, Yunho Maeng, Manoel Marques, Francisco Jose Martín-Fernández, Douglas T. McClure, David McKay, Srujan Meesala, Antonio Mezzacapo, Nikolaj Moll, Diego Moreda Rodríguez, Giacomo Nannicini, Paul Nation, Pauline Ollitrault, Lee James O'Riordan, Hanhee Paik, Jesús Pérez, Anna Phan, Marco Pistoia, Viktor Prutyanov, Max Reuter, Julia Rice, Abdón Rodríguez Davila, Raymond Harry Putra Rudy, Mingi Ryu, Ninad Sathaye, Chris Schnabel, Eddie Schoute, Kanav Setia, Yunong Shi, Adenilton Silva, Yukio Siraichi, Seyon Sivarajah, John A. Smolin, Mathias Soeken, Hitomi Takahashi, Ivano Tavernelli, Charles Taylor, Pete Taylour, Kenso Trabing, Matthew Treinish, Wes Turner, Desiree Vogt-Lee, Christophe Vuillot, Jonathan A. Wildstrom, Jessica Wilson, Erick Winston, Christopher Wood, Stephen Wood, Stefan Wörner, Ismail Yunus Akhalwaya, and Christa Zoufal. Qiskit: An open-source framework for quantum computing, 2019.

[2] Elena Andreeva, Joan Daemen, Bart Mennink, and Gilles Van Assche. Security of keyed sponge constructions using a modular proof approach. In *International Workshop on Fast Software Encryption*, pages 364–384. Springer, 2015.

[3] Mihir Bellare and Phillip Rogaway. Random oracles are practical: A paradigm for designing efficient protocols. In *Proceedings of the 1st ACM conference on Computer and communications security*, pages 62–73. ACM, 1993.

[4] Guido Bertoni, Joan Daemen, Michael Peeters, and Gilles Van Assche. On the indifferentiability of the sponge construction. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 181–197. Springer, 2008.

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

[6] Guido Bertoni, Joan Daemen, Michael Peeters, and Gilles Van Assche. On the security of the keyed sponge construction. In *Symmetric Key Encryption Workshop*, volume 2011, 2011.

[7] Andrej Bogdanov and Alon Rosen. Pseudorandom functions: Three decades later. In *Tutorials on the Foundations of Cryptography*, pages 79–158. Springer, 2017.

[8] Donghoon Chang, Morris Dworkin, Seokhie Hong, John Kelsey, and Mridul Nandi. A keyed sponge construction with pseudorandomness in the standard model. In *The Third SHA-3 Candidate Conference (March 2012)*, volume 3, page 7, 2012.

[9] Jan Czajkowski, Andreas Hülsing, and Christian Schaffner. Quantum indistinguishability of random sponges. 2019.

[10] Shimon Even and Yishay Mansour. A construction of a cipher from a single pseudorandom permutation. *Journal of cryptology*, 10(3):151–161, 1997.

[11] Lov K Grover. A fast quantum mechanical algorithm for database search. *arXiv preprint quant-ph/9605043*, 1996.

[12] Andreas Hülsing. Introduction to the theory of secret key cryptography. `https://huelsing.net/wordpress/wp-content/uploads/2019/06/20190617_symmetric-1.pdf`, 2019.

[13] Marc Kaplan, Gaëtan Leurent, Anthony Leverrier, and María Naya-Plasencia. Breaking symmetric cryptosystems using quantum period finding. In *Annual International Cryptology Conference*, pages 207–237. Springer, 2016.

[14] Hidenori Kuwakado and Masakatu Morii. Security on the quantum-type even-mansour cipher. In *2012 International Symposium on Information Theory and its Applications*, pages 312–316. IEEE, 2012.

[15] Jacques Patarin. The "coefficients H" technique. In *International Workshop on Selected Areas in Cryptography*, pages 328–345. Springer, 2008.

[16] Thomas Santoli and Christian Schaffner. Using simon's algorithm to attack symmetric-key cryptographic primitives. *arXiv preprint arXiv:1603.07856*, 2016.

[17] Peter W Shor. Algorithms for quantum computation: Discrete logarithms and factoring. In *Proceedings 35th annual symposium on foundations of computer science*, pages 124–134. Ieee, 1994.

[18] Daniel R Simon. On the power of quantum computation. *SIAM journal on computing*, 26(5):1474–1483, 1997.

[19] Mark Zhandry. A note on quantum-secure prps. *arXiv preprint arXiv:1611.05564*, 2016.