Attacking Post-Quantum Cryptography

Leon Groot Bruinderink

E-mail: lgrootbr@gmail.com
Website: `https://www.leongb.nl`

First edition November 2019.

This research is supported by the Commission of the European Communities through the Horizon 2020 program under project number ICT–645622 (PQCRYPTO).

Printed by Gildeprint Drukkerijen, Enschede, The Netherlands.

A catalogue record is available from the Eindhoven University of Technology Library.

ISBN: 978-90-386-4932-0.

The “impossible cube” was first created by Maurits Cornelis Escher (1898–1972), a Dutch graphic artist. The front cover illustrates the following two things in my thesis. The impossible cube as a drawing is actually fine: simply some lines on a 2D page. However, once we interpret this as a 3D object we conclude that such an object cannot exist in reality. This effect illustrates scenarios where a cryptographic security model does not cover all cases that might occur in reality: there is a mismatch in the interpretation and usage of a cryptographic system and its intended purpose. Additionally the impossible cube is glowing: if we would only remove the cube from the drawing, its glow would still exist to mark the cube’s existence. This effect illustrates side-channel leakage of cryptographic implementations. The back cover illustrates that some attacks on cryptographic systems are full breaks.

However, the main purpose of abstraction is not to tell a story, but to encourage the viewer’s imagination. The cover and bookmark are designed by (in alphabetic order): Koen van Ham, Tommie Perenboom and Maijke Receveur.

Attacking Post-Quantum Cryptography

PROEFSCHRIFT

ter verkrijging van de graad van doctor aan de Technische Universiteit Eindhoven,
op gezag van de rector magnificus prof.dr.ir. F.P.T. Baaijens, voor een commissie
aangewezen door het College voor Promoties, in het openbaar te verdedigen
op dinsdag 17 december 2019 om 16:00 uur

door

Leon Groot Bruinderink

geboren te Wageningen

Dit proefschrift is goedgekeurd door de promotoren en de samenstelling van de promotiecommissie is als volgt:

voorzitter:	prof.dr. M.A. Peletier
1e promotor:	prof.dr. T. Lange
2e promotor:	prof.dr. D.J. Bernstein
copromotor:	dr. A. Hülsing
leden:	prof.dr. L. Batina (Radboud Universiteit)
	prof.dr. T. Güneysu (Ruhr-Universität Bochum)
	prof.dr. K. Paterson (ETH Zürich)
	dr. B.M.M. de Weger

Het onderzoek of ontwerp dat in dit proefschrift wordt beschreven is uitgevoerd in overeenstemming met de TU/e Gedragscode Wetenschapsbeoefening.

Acknowledgments

This thesis, that is the final product of the four years of my PhD life, would not have been possible without the guidance, support and encouragement of several persons. In these first pages of the book, I would like to take the opportunity to thank each of them.

First of all I would like to start with thanking my supervisors, and I am very lucky to have three supervisors, who guided me in every step of my PhD. Tanja, when I first stepped into your office in 2013 the first thing you told me is that you did not see me in your crypto lectures at all (oops). Despite this, you were very willing to help me with an internship/research project abroad, where I ended up spending three months in Philadelphia with Nadia. For me this illustrates how you always stand up for your students and are always willing to help in case they need it. It was during my Master's that you already convinced me of the great PhD life by taking me to the Summer School in Croatia (yes, next to the beach). You have continued encouraging me to fly everywhere around the world, attending conferences and summer schools or giving research talks, all of which contributed to me becoming a better researcher. You taught me the high standards of scientific writing and presentations, as well as the thrill and fun of starting new research collaborations. The combination of the freedom to do the research I liked and the personal support when I needed it made the past four years fly by without any issues. I am also very thankful for your extensive help in making this thesis in the good shape that it is now. Last but not least, I would like to thank you for all the spicy muffins and chocolate cakes that were not for your birthday, but were a nice treat nonetheless. Dan, I would like to thank you for helping me with the hardest questions during my PhD. Whenever I was stuck with a certain question that I could now find with the help of Google, I was lucky to have someone just a few meters away with more knowledge. I also thank you and Tanja for sharing the fun of devising both an appropriate and amusing title for scientific contributions. I did my best to keep this in mind whenever a paper was finished. Finally, I would like to thank you for all the funny yet interesting talks. Your way of presenting scientific research is a skill I can only hope to master some day.

Andy, although you have been my daily supervisor for the past years, I consider you more as a friend that helped me with basically anything. We started working together in my Master project and I am happy that several projects followed during my PhD. You taught me how to transform scientific papers, as well some parts in this thesis, from an abstract blur into something digestible. You were always there to talk about work-related or personal issues and provided me with valuable suggestions, especially in deciding which things I should or should not deem important. I cannot recall how many times I knocked on the door with the poster to make a query. Thank you for not being a random oracle, but providing very practical responses.

I would like to thank Lejla Batina, Tim Güneysu, Kenny Paterson and Benne de Weger for joining my PhD defense committee. I would like to thank you all for taking the time to read this thesis, providing valuable comments and suggestions and for traveling to Eindhoven for the defense. A special thanks to Benne for being my mentor during the first year of my study in mathematics and convincing me to keep on trying.

I was lucky to have worked with many outstanding researchers during my PhD. Without the fruitful discussions and collaboration, this thesis and the published works would not have been possible. I thank Daniel J. Bernstein, Joachim Breitner, Jan Czajkowski, Daniel Genkin, Nadia Heninger, Andreas Hülsing, Tanja Lange, Lorenz Panny, Peter Pessl, Christian Schaffner, Dominique Unruh, Christine van Vredendaal and Yuval Yarom.

I am grateful for all the friends and colleagues in Eindhoven who made the life of a PhD student actually feel as busy as the normal student life with all the activities that are organized. I thank in no particular order Anita, Andy, Christine, Gustavo, Chloe, Lorenz, Manos, Francisco, Thijs, Jake, Florian, ("wajo") Daan, Ale, Boris, Benne, Davide, Mahdi, Stefan, Niels, Frank, Niek, Bouke, Meilof, Simon, Chitchanok and Tony. A special thanks to Christine, Francisco, Lorenz, Niels and Frank for being my office mates during some time of my PhD and dealing with my daily buzz.

During the past four years I had the opportunity to travel a lot oversees to attend conferences or summer schools. On almost every occasion there was one or more of the 'Radboud crew' to join. I thank (wrong) Joost, (right) Joost, Ko and Pedro for all the fun on the abroad trips.

I thank all my friends for their support in my PhD, but I am more thankful for all the jokes, pranks, drinks, dinners, cosy unions, holidays and other social events that turned off my brain. A special thanks to Tommie, Koen and Maaike, for helping me with the design of the cover of this thesis. Probably the best part in making this thesis was to view my work with some abstract-inspirational-creative-design-thinking.

I thank my parents, Eric and Audrey, and my brother Patrick, for their infinite love and support during my PhD, but also during all the previous years of my educational life. Without their encouragement I would have been stuck with a VMBO advice and would have never come this far. I also thank my "in-laws" Wieke, Hilde, Pleun and JW for all the cosy weekends and holidays, as well as everything else they have done for me, ranging from taking the car to the garage to taking care of our 'inimini' fish when we are on holidays.

I want to end these acknowledgements with the person I owe the most thanks to. Fleur, a large part of our time together was during my pursuit of getting the PhD degree and I could not have done it without you by my side. You brought me to (or picked me up at) the airport, made dinner in times of deadlines, rehearsed my presentations with me and made our home the warmest place to go back to after a day of work. You are my 'ski-buddy' to pick me up when I fall, my 'skydive-buddy' to celebrate the highs, my 'diving-buddy' to support me in the lows, and my 'wine-buddy' to enjoy the finer things in life. I am looking forward to so many more adventures together with you. *It's always better when we're together*.

Eindhoven, November 2019
Leon Groot Bruinderink

Contents

List of Algorithms

List of Figures

List of Tables

Introduction

*A (not so) long time ago in a galaxy (not so) far,
far away....*

CRYPTO WARS

Episode 0 and 1
The cryptanalyst strikes back

It is a dark time for cryptography.
Although QUANTUM COMPUTERS
have been deflected,
Imperial troops have driven the
cryptosystems from their
mathematical models
and pursued them across
the galaxy with more attacks.

Evading the dreaded Imperial
MODEL-MISMATCH ATTACKS,
a group of freedom fighters led by
PQCRYPTO has submitted several
post-quantum schemes to the
remote ice world called NIST.

The evil lord Darth Promovendus,
obsessed with finding more secret keys,
has dispatched thousands of
remote SIDE-CHANNEL ATTACKS into
the far reaches of the internet....

This thesis covers several attacks on post-quantum cryptography, the new cryptography able to resist attacks by large quantum computers. The opening crawl should make clear that new is not always better.

Although the literal meaning of cryptography is “secret writing”, many cryptographic applications and protocols go far beyond secretly transforming messages. There are various security goals that cryptography is able to provide and to achieve these goals, there are several evaluation criteria. The first building block for cryptography is some computationally hard mathematical problem, e.g. factoring large numbers or computing discrete logarithms. For example, these could be used to build so-called trapdoor one-way functions: functions that are easy to compute in the forward direction (e.g. encryption of a message), but computationally hard to compute in the backward direction unless a certain secret key is known (e.g. decryption of a ciphertext). In post-quantum cryptography, we are only interested in mathematical problems that are computationally hard, even for large quantum computers.

To evaluate the security of public-key cryptographic primitives, some security model is required to describe the exact setting. This model describes what type of primitive we are considering: e.g. a public-key encryption scheme or a digital signature scheme. It also describes the attacker’s capabilities: e.g. we often make the distinction whether keys are used once or can be reused. Also the attacker’s goal is described in the model: a signature forgery does not make sense in the analysis of an encryption scheme. The model also has some level of abstraction: some models are more idealized (e.g. Random Oracle Models) than other models (e.g. Standard Model). Ultimately such models are used in provable security: linking the problem of breaking the cryptographic primitive (i.e. the attacker reaches his goal) to the problem of solving the underlying mathematical problem. However, there are scenarios where the security model does not cover all cases that occur in reality: e.g. incorrect usage of the cryptographic primitive by accident. Such a mismatch between the security model and reality may lead to several attacks that are not covered by the security model or security proof.

The models in provable security usually do not consider attacks on cryptographic implementations: this is another separate evaluation category. There are several new attack vectors possible when we move from the mathematics on the white-board to actually implementing cryptography in practice. What happens when we perform cryptographic operations on our devices (e.g. laptop, mobile phone, smart-card) having certain physical properties? Depending on the implementation and the device, these physical properties can leak certain values during a cryptographic operation, that are supposed to be kept secret. This opens the door to attacks that do not solve the underlying mathematical problem, but instead use this additional physical information as a short-cut to find the secret key. As cryptography is used every day on any device connected to the internet, these so-called side-channel attacks are very important to take into consideration in practice.

This thesis covers several attacks (and fixes) on post-quantum cryptographic primitives. The knowledge of these attacks might weaken the trust we have in the security level of the primitive, but can also achieve the opposite. The attacks, together with the countermeasures, could actually improve the understanding of the primitive and may thus ultimately lead to improved quality. This thesis is built up of two parts, each part covering attacks and countermeasures. For each part we will briefly discuss the content of its chapters.

Part I: Model-mismatch attacks

The first part of this thesis focuses on model-mismatch attacks: what happens when (some of) the assumptions made in the security model become invalid? There are several reasons that such a model mismatch might occur. For example: the concept of quantum computing introduces a whole new computing paradigm, possibly invalidating the (previous) assumptions on the capabilities of the attacker, that now has a (large enough) quantum computer. On the other hand, model mismatch might also occur when cryptography is incorrectly used: possibly due to accidental events in applications or simply because the security model did not match with the cryptographic scheme itself. Part I is built up as follows:

- **Chapter 1** gives an introduction to security models in (post-quantum) cryptography and poses the research questions that will be answered in Chapters 2, 3, and 4.

- **Chapter 2** examines possible fixes to the security model of certain commitment schemes: the security proofs of these schemes are no longer valid when dealing with quantum attackers. One way to mitigate this, is showing that hash-functions have some desired quantum property: the collapsing property. We show that the sponge construction, which is, among others, the construction behind the standardized hash function SHA3, is collapsing for specific instantiations. This work could strengthen the belief that the sponge construction is post-quantum secure. The results of this chapter are based on the paper *The post-quantum security of the sponge construction* [CGH$^+$18], which was published at PQCrypto 2018 and is joint work with Jan Czajkowski, Andreas Hülsing, Christian Schaffner and Dominique Unruh.

- **Chapter 3** analyzes (accidental) reuse of a hash-based one-time signature key-pair. Although the security reductions only guarantee security under single-message attacks, the two-message attacks are in some cases still of exponential complexity, at least asymptotically. As we show what this actually means for concrete parameters, this chapter can provide meaningful insights to standardization bodies like NIST, that need to analyze the risks associated with these model-mismatch attacks on hash-based signatures. The results of this chapter are based on the paper *"Oops, I did it again" – Security of One-Time Signatures under Two-Message Attacks* [GH17], which was published at SAC 2017 and is joint work with Andreas Hülsing.

- **Chapter 4** examines whether the error-correction codes used in the lattice-based public-key encryption scheme HILA5 can prevent reaction attacks. There are similar attacks where the additional attacker capabilities in the IND-CCA model allowed for breaking a IND-CPA-secure encryption scheme. This chapter can strengthen the understanding about which applications can handle an IND-CPA-secure encryption scheme and which require an IND-CCA-secure scheme. The results of this chapter are based on the paper *"HILA5 Pindakaas": On the CCA security of lattice-based encryption with error correction* [BGLP18], which is published at AFRICACRYPT 2018 and is joint work with Daniel J. Bernstein, Tanja Lange and Lorenz Panny.

- **Chapter 5** concludes the first part about model-match attacks and summarizes the answers to the research questions posed in Chapter 1. We pose a list of open problems that are related to the work covered in Part I.

Part II: Side-channel attacks

The second part of this thesis covers side-channel attacks. Post-quantum cryptography introduces new cryptographic primitives, requiring new algorithms and implementations. It is not straightforward to achieve a secure implementation, but at the same time it is also not straightforward to perform a side-channel attack. Part II covers several side-channel analyses of post-quantum cryptographic schemes and is built up as follows:

- **Chapter 6** describes preliminaries on lattice-based cryptography and side-channel attacks, specifically cache attacks and fault attacks. Similar to Part I, we pose several research questions in this chapter that will be answered in Chapters 7, 8, 9, and 10.

- **Chapter 7** is an intermezzo chapter out of the realm of post-quantum cryptography. Instead, we look at implementations of RSA where we analyze a common belief: that using the sliding-window method for exponentiation (with implemented window sizes) does not leak enough bits for a full break. However, when we delve deeper we find that a lot more information than previously known can be gained for the left-to-right version. The attack in this chapter should let everyone reconsider statements about security under side-channel leakage. It also provides motivation for the necessity to understand side-channel attacks on post-quantum cryptography, which is the topic of the remaining chapters of this part. The results of this chapter are based on the paper *Sliding right into disaster – Left-to-right sliding windows leak* [BBG$^+$17b], which was published at CHES 2017 and is joint work with Daniel J. Bernstein, Joachim Breitner, Daniel Genkin, Nadia Heninger, Tanja Lange, Christine van Vredendaal and Yuval Yarom.

- **Chapter 8** analyzes possible side-channel attacks that exploit leakage from samplers for the discrete Gaussian distribution. Many lattice-based schemes, including a promising digital signature scheme called BLISS, use noise sampled from this distribution. However, as it is not straightforward to sample from this distribution, it is also not straightforward to (efficiently) protect the implementations against side-channel attacks. At the same time it is not obvious what exact information can be gained in a cache attack. This work should help designers (and implementers) of lattice-based cryptography make well-informed decisions on whether to use the discrete Gaussian distribution, or use another distribution instead. The results of this chapter are based on the paper *Flush, Gauss, and Reload – A Cache Attack on the BLISS Lattice-Based Signature Scheme* [GHLY16], which was published at CHES 2016 and is joint work with Andreas Hülsing, Tanja Lange and Yuval Yarom.

- **Chapter 9** considers attacks on an improved version of BLISS called BLISS-B. Despite the known vulnerabilities in BLISS, the improved version seems protected against side-channel attacks. At the same time, BLISS-B is the only version implemented in e.g. strongSwan, and is thus the only real target to test practical asynchronous cache-attacks on lattice-based signature schemes. The work in this chapter shows that cache-attacks are not only possible, but also practical, and are therefore a real threat that has to be mitigated. The results in this chapter are based on the paper *To BLISS-B or not to be – Attacking strongSwan's Implementation of Post-Quantum Signatures* [PGY17], which was published at CCS 2017 and is joint work with Peter Pessl and Yuval Yarom.

- **Chapter 10** investigates the applicability of differential fault attacks on deterministic lattice-based signature schemes. To remove the dependency on well-generated random numbers, several schemes including Dilithium and qTESLA are deterministic. It is already known that differential fault attacks are possible on deterministic elliptic-curve based signature schemes, but it is unclear whether this is also possible for lattice-based schemes. This chapter broadens the knowledge on possible attack vectors on deterministic lattice-based signature schemes, especially when implemented on small devices (e.g. smartcards). The results in this chapter are based on the paper *Differential Fault Attacks on Deterministic Lattice Signatures* [GP18], which was published in TCHES 2018-3 and is joint work with Peter Pessl.

- **Chapter 11** finally concludes the second part with an overview of the answers to the research questions, summarizing the most important results and posing several open questions that are related to the work covered in Part II.

PART I

MODEL-MISMATCH ATTACKS

CHAPTER 1

Security models in post-quantum cryptography

1.1 — Problem description

An important aspect in the evaluation of (post-quantum) cryptography is provable security. The goal of provable security is to relate the security of a cryptographic system to the complexity of a solver for some computationally hard mathematical problem. In the most desirable case (the tight case), we can then say that breaking the scheme is as hard as solving the underlying mathematical problem. However, such security proofs/reductions do not say anything about security of cryptographic implementations, e.g. they do not consider side-channel attacks[1]: we handle this scenario in Part II of this thesis.

The setting and requirements in provable security is covered by the model: e.g. what type of scheme are we considering (e.g. digital signatures or encryption) and what type of adversary are we talking about (i.e. what is the attacker's goal). There exist several models in (post-quantum) cryptography. As quantum computing introduces a new computing paradigm, some of these models might have to be reconsidered. On the other hand, the formal models that are used for analyzing the security of post-quantum cryptography might not cover all cases that could occur in real life scenarios, e.g. in the form of a mismatch. This requires further analysis of these models in search for consequences and possible fixes. In Part I of this thesis, we address some of the issues that arise with (quantum) security models in post-quantum cryptography.

1.2 — Cryptographic security models

The study of (quantum) algorithms for breaking cryptography only gives an upper bound on the cost of an attack. A far better security guarantee would be a lower bound on the cost of any (possibly unknown) attack. Let us take the example of a signature scheme. Ideally, one would like to be able to make a statement such as: "if an attacker $\mathcal{A}$ can efficiently forge signatures of the scheme (for some definition of forgery), then $\mathcal{V}^{\mathcal{A}}$ can efficiently solve some underlying mathematical problem". We call $\mathcal{V}^{\mathcal{A}}$ the "reduction", or the algorithm that uses the signature forger $\mathcal{A}$ as a subroutine. If this reduction is tight (meaning the complexity of $\mathcal{V}^{\mathcal{A}}$ is asymptotically identical to the complexity of $\mathcal{A}$), we can say that the problem of solving the underlying mathematical problem is a lower bound for the problem of forging signatures. Instead of trying to find issues with the signature scheme, we can simply concentrate on determining the complexity of the underlying mathematical problem.

[1] The area of leakage resilient cryptography does attempt to prove security under some model of side-channel leakage, but most schemes cannot be proven secure in such models or are not practical to use.

In order to prove such lower bounds, we have to use some form of abstraction: the model. The security model describes the setting in which the security proof/reduction is deemed valid. There are several dimensions in the model that together describe the exact setting. First, we need to describe the exact functionalities of the cryptographic primitive: e.g. what functionalities are provided by a digital signature scheme or a public-key encryption scheme. Then, the adversary model describes the type of adversary we are considering: classical adversaries (i.e. only having classical computers) or quantum adversaries (i.e. having access to a large quantum computer).

To make a statement such as the one in the previous paragraph, we also need to define the attacker's goal in the model (i.e. what does a forgery mean). In digital signatures, we often consider the case that an attacker can learn digital signatures for arbitrary messages of his choice. The number of times he can learn these signatures is bounded by some number, which is usually related to some security parameter. However, many hash-based signature schemes are only secure for one single use: i.e. the accompanying reduction is given for a one-time attack model. In general the attacker's goal is to construct a valid signature on a new message that he did not learn yet, i.e. a valid signature forgery. This is called the EU-CMA setting (we give a more formal definition of this notion in Chapter 3). Other goals may include for example full key-recovery.

Of course, an attacker model for forgeries does not make sense for public-key encryption. In encryption there are two main attack models: the IND-CPA and the IND-CCA setting. In both settings, the attacker can learn ciphertexts for arbitrary messages of his choice. However, in the IND-CCA setting, the attacker can also learn decryption outputs corresponding to arbitrary ciphertexts of his choice. The attacker should then choose two messages (not learned earlier), from which he receives back the corresponding ciphertext of one of them (chosen at random). The attacker is again able to learn new ciphertexts (and new decryption outputs in the IND-CCA setting) at this point. The attacker's goal is to decide which one of his messages got encrypted and we call the scheme IND-CPA (or IND-CCA) secure if the advantage of the attacker over picking a message at random is negligible (and by guessing he has a probability of 1/2 of guessing correct). Note that any IND-CCA-secure scheme is also IND-CPA secure, but not the other way around. The most important difference between these two settings in real life is that for an IND-CCA secure encryption scheme, the (public) key can be cached and reused for multiple encryptions, i.e. the owner of the public key can reuse his keys to receive (and decrypt) multiple messages. This scenario changes if someone uses an IND-CPA-secure scheme that is not IND-CCA secure: apparently the scheme cannot handle multiple decryption queries of arbitrary ciphertexts. However, this becomes a problem (if the keys are cached and reused) as many applications leak some information about a decryption procedure. We will show in Chapter 4 that leaking even one bit of information per decryption query can be devastating for an IND-CPA-secure scheme. One way to securely use an IND-CPA scheme is to use the keys only once: renew a key pair after every usage (i.e. decryption of a message). It turns out that this distinction is particularly important in post-quantum cryptography: most basic public-key encryption schemes are only secure in the IND-CPA setting (e.g. based on lattices, codes or isogenies) and require additional steps to be secure in the IND-CCA setting, e.g. by using the FO-transform [FO99].

Last, the security model should also describe the "type" of proof. In the so-called standard model, we assume some building block has some property P. This can be the computationally hard mathematical problem, but also any other property. If this property

P suffices to prove/show the reduction (from breaking P to the security of some scheme), we have a proof in the standard model. In the next section, we describe the Random Oracle Model (ROM), which is often used to prove security of a system that uses a hash-function as a building block. This is a more idealized model than the standard model, but also turns out to be quite useful in various scenarios.

1.3 — The (quantum) random oracle model

(Cryptographic) hash functions are one of the most basic building blocks in cryptography. They are virtually used everywhere: as cryptographically secure checksums to verify integrity of software or data packages, as building blocks in security protocols, including TLS, SSH, IPSEC, as part of any efficient variable-input-length signature scheme, in transformations for CCA-secure encryption (i.e. the FO-transform [FO99]), and many more. This automatically means that the accompanying security model and proof/reduction has to accommodate the hash function. It depends on the setting which property of the hash-function is required, but a cryptographic hash function $H : \{0,1\}^* \to \{0,1\}^n$ is a function that should have all of the following properties:

- Collision resistance: given the function H, it is computationally infeasible to find two inputs $x \neq y$ with $H(x) = H(y)$.
- Preimage resistance: given the function H and a target y, it is computationally infeasible to find any input x with $H(x) = y$.
- Second preimage resistance: given the function H and an input x, it is computationally infeasible to find another input y such that $H(x) = H(y)$.
- Pseudo-randomness: given the function H, for any input x the output $H(x)$ seems to be chosen at random from the output space if x is unknown (i.e. is unpredictable).

A perfect random function has all the above properties, so ideally we would like a cryptographic hash function to behave like a random function. However, since real hash functions have a compact description, they will never be perfectly random: if it would be perfectly random, the function description would be of exponential size and computing any output would take exponential time. Despite this, in some security proofs/reductions a hash-function is used to deterministically compute random numbers (e.g. to generate an arbitrary number of random bits from a secret seed). We can abstract away this predicament in a more idealized model: the "Random Oracle Model (ROM)" [BR93]. In this model, we assume there exists a perfectly random function $O : \{0,1\}^* \to \{0,1\}^n$ called the Random Oracle, which is publicly available and can be queried efficiently (i.e. although the function is perfectly random, storage/computation costs are neglected). Given any input x, a deterministic output $y = O(x)$ is returned by the oracle, sampled uniform at random from the output space. We can attempt to prove the security of any scheme that uses cryptographic hash functions in the ROM: we simply replace the hash function by the random oracle. In this case, a security reduction can give lower bounds on the number of queries an attacker (both $\mathcal{A}$ or $\mathcal{V}^{\mathcal{A}}$ from the previous section) has to perform to the random oracle in order to show the reduction: this defines the query complexity. However, when the mentioned scheme is used in real-life, the random oracle is replaced by a (cryptographically secure) hash function again, as random oracles do not exist in real life. This means the proofs in the ROM only give heuristic security arguments: they cannot prove the security of the scheme that is used in real life, but do allow to verify if the construction is solid.

Quantum computing We call the ROM the classical setting, as the attacker only has *classical* access to the random oracle. Despite the fact that quantum cryptanalysis only mildly affects the security of hash functions, any construction in the ROM instantiates the random oracle with a cryptographic hash function in real life. But this instantiation would enable a quantum attacker to evaluate the hash function on quantum states, i.e. qubits in superposition. In particular, one superposition query could evaluate the hash function on all its inputs: this is not captured in the definition of the query complexity. To capture this issue, the random oracle needs to be replaced by a quantum-accessible oracle: the Quantum Random Oracle Model (QROM) [BDF+11] (we give a more formal definition in Chapter 2). This model seems to be the appropriate model for analyzing security in the so-called post-quantum setting. In this case, an attacker can query the random oracle O with qubits: i.e. on input $|x\rangle\,|y\rangle$, the oracle returns $|x\rangle\,|y \oplus O(x)\rangle$ (we also give more background on quantum computing in Chapter 2). This means any security proof in the ROM needs to be replaced by a new security proof in the QROM, which turns out to be significantly harder.

In the ROM, it is often trivial to quantify the amount of information an attacker obtains with each query. For example, if an attacker wants to find a collision, he needs to keep querying the oracle in order to find a collision. This means that most proofs consist of simple counting arguments to show lower bounds on the query complexity. In the quantum setting however, an attacker can query the oracle on a superposition over all messages. In the collision example, this would mean the attacker immediately gains (several) collisions as part of the superposition output. Luckily, to actually retrieve a collision from this superposition state is still non-trivial. In fact, also in the quantum setting there exist examples where it is possible to find lower bounds for quantum algorithms, bounding the amount of information the attacker obtains with each oracle query. The most notable example of this is Grover's algorithm [Gro96], which is proven to be optimal: the upper bound matches the lower bound.

Not only do the security proofs (in the standard or random oracle model) turn out to be harder if an attacker has quantum access, in some cases they actually turn out to be impossible. In previous work, Unruh [Unr16b] showed that the security model for some commitment schemes did not capture the intuition of what it means to be a secure commitment scheme when an attacker has quantum-access to the random oracle. He furthermore showed how to adapt/fix this: i.e. change the security requirements and attacker capabilities of the model. A new notion for hash functions was introduced: collapsing hash functions. The collapsing definition is a purely quantum definition (i.e. it does not exist in the classical setting) and can be viewed as a strengthening of collision-resistance. In other words, in the quantum setting, the security requirements were not strong enough and had to be strengthened.

1.4 — Challenges and research questions

In Part I of the thesis, we will look at several research questions related to mismatch of (quantum) models and reality in post-quantum cryptography. We will now present the research questions that will be answered in this part, accompanying the following two chapters.

Q1: Is the sponge construction (and thus a hash function like SHA3) collapsing?

In light of recent results by Unruh [Unr16b], it is desirable to find hash functions that are collapsing. Unruh [Unr16b] also showed that the random oracle is collapsing. However, this has little relevance for real world hash functions. A practical hash function is typically constructed by iteratively applying some elementary building block in order to hash large messages. So even if we are willing to model the elementary building block as a random oracle, the overall hash function construction should arguably not be modeled as a random oracle. For hash functions based on the Merkle-Damgård construction, it was already shown [Unr16a] that if the elementary building block (the compressing function) is a random oracle, the resulting hash function is collapsing. However, the SHA3 construction is based on the sponge construction, which leaves us with the question whether we can prove that the sponge construction is collapsing? This would actually *prevent* the model-mismatch attacks by quantum attackers and is the focus of Chapter 2.

Q2: What can we say about the security of hash-based one-time signatures, when a user accidentally signs twice?

One-time signatures (OTS) are called one-time, because the accompanying reductions only guarantee security under single-message attacks. However, this does not imply that efficient attacks are possible under two-message attacks. Especially in the context of hash-based OTS, this leads to the question if accidental reuse of a one-time key pair leads to immediate loss of security or to graceful degradation? In Chapter 3, we investigate this question for three prominent hash-based OTS: Lamport's scheme, its optimized variant, and WOTS. This model mismatch is analyzed in various scenarios, i.e. for different attacker goals and capabilities.

Q3: Can error-correcting codes prevent the reaction attack?

Many lattice-based encryption schemes (as well as many other post-quantum encryption schemes) are probabilistic: there is a chance that correctly constructed ciphertexts are decrypted incorrectly. This opens the door to so-called reaction attacks and emphasizes the importance of the security claims/models accompanying the encryption scheme. More specifically, such probabilistic schemes usually only guarantee security in the IND-CPA setting. An idea to prevent such reaction attacks is to add an error-correcting code to the encryption scheme: this should lower the probability of failures drastically. This is used in HILA5 [Saa17a], a scheme submitted to the NIST competition as an IND-CCA-secure scheme. Does error-correction eliminate the event of decryption errors and thereby prevent the model mismatch via reaction attacks? We investigate this possibility in Chapter 4.

Chapter 2

Post-quantum sponges

2.1 — Overview

Context. For modern hash functions like SHA2 or SHA3 there exist proofs that show security properties of the hash function, assuming that an internal building block of the construction has a certain property. For example, the SHA3 hash function is an instantiation of the sponge construction [BDPV07]. For sponge functions it was shown [BDPV07] that if the internally used permutation (or function) behaves like a random permutation (or function), then the sponge function achieves one of the strongest security properties possible: indifferentiability from a random oracle. This property guarantees that there do not exist any attacks that perform better than generic attacks against SHA3 as long as the permutation used in SHA3 behaves like a random permutation. Sadly, the security reduction for sponge functions does not carry over to the quantum setting, as the arguments are query-based and do not work against a quantum adversary A. Such an adversary can use a quantum circuit implementing SHA3 and can thereby query the function in superposition. In particular, A could execute SHA3 on the uniform superposition over all messages of a certain length, possibly helping A to distinguish SHA3 from a random oracle. This leaves us with no security argument for SHA3 besides the absence of attacks which is an unfortunate situation.

One of the most important properties of a hash function H is collision-resistance. Intuitively, collision-resistance guarantees, in some sense, that given $H(x)$ the value x is effectively determined. Of course, information-theoretically, x is not determined, but in many situations, we can treat the preimage x as unique, because we will see another value with the same hash with negligible probability. For example, collision-resistant hashes can be used to extend the message space of signature schemes (by signing the hash of the message), or to create commitment schemes (e.g., sending $H(x\|r)$ for random r, commits the sender to x; the sender cannot change his mind about x because he cannot find another preimage).

In the post-quantum setting,[1] however, it was shown by Unruh [Unr16b] that collision-resistance is weaker than expected: for example, the commitment scheme sketched in the previous paragraph is not binding. In fact, the attack is much stronger: it is possible for an attacker to send a hash value h, then to be given a value x, and then to send a random value r such that $h = H(x\|r)$, thus opening the commitment to any desired

[1] We mean a situation in which the protocols and primitives that are studied run on a conventional computer, but the attacker can perform quantum computations, see Section 1.3.

value – even if H is collision-resistant against quantum adversaries.[2] This contradicts the intuitive requirement that $H(x)$ determines x.

Fortunately, Unruh [Unr16b] also presented a strengthened security definition for post-quantum secure hash functions: collapsing hash functions. Roughly speaking, a hash function is collapsing if, given a superposition of values m, measuring $H(m)$ has the same effect as measuring m (at least from the point of view of a computationally limited observer: we give a formal definition in Section 2.2). Collapsing hash functions serve as a drop-in replacement for collision-resistant ones in the post-quantum setting: Unruh showed that several natural classical commitment schemes (namely the scheme sketched above, and the statistically-hiding schemes from [HM96]) become post-quantum secure when using a collapsing hash function instead of a collision-resistant one. The collapsing property also directly implies collision-resistance.

In light of these results, it is desirable to find hash functions that are collapsing. Unruh [Unr16b] showed that the random oracle is collapsing. (That is, a hash function $H(x) := O(x)$ is collapsing when O is a random oracle.) However, this has little relevance for real-world hash functions: A practical hash function is typically constructed by iteratively applying some elementary building block (e.g., a "compression function") in order to hash large messages. So even if we are willing to model the elementary building block as a random oracle, the overall hash function construction should arguably not be modeled as a random oracle.[3]

For hash functions based on the Merkle-Damgård (MD) construction (such as SHA2 [Nat15]), Unruh [Unr16a] showed: If the compression function is collapsing, so is the hash function resulting from the MD construction. In particular, if we model the compression function as a random oracle (as is commonly done in the analysis of practical hash functions), we have that a hash function based on the MD construction is collapsing (and thus suitable for use in a post-quantum setting).

However, not all hash functions are constructed using MD. Another popular construction is the sponge construction [BDPV07], underlying for example the current international hash function standard SHA3 [NIS14], but also other hash functions such as Quark [AHMN10], Photon [GPP11], Spongent [BKL$^+$13], and Gluon [BDM$^+$12]. The sponge construction builds a hash function H from a block function[4] f. In the classical setting, we know that the sponge construction is collision-resistant if the block function f is modeled as a random oracle, or a random permutation, or an invertible random permutation [BDPA08]. In particular, it was shown that the sponge construction is indifferentiable from a random oracle *in the classical setting*. Together with the fact that the random oracle is collision-resistant, collision-resistance of the sponge construction follows. However, their proof (i.e. the proof in [BDPA08]) does not carry over to the post-quantum setting: their proof relies on the fact that queries performed by the adversary to the block function are classical (i.e., not in superposition between different values). As first argued

[2] More precisely, [Unr16b] shows that relative to certain oracles, a collision-resistant hash function exists that allows such attacks. In particular, this means that there cannot be a relativizing proof that the commitment scheme is binding assuming a collision-resistant hash function.

[3] For example, hash functions using the Merkle-Damgård construction are not well modeled as a random oracle. If we use $\mathit{MAC}(k, m) := H(k\|m)$ as a message authentication code (MAC) with key k, we have that *MAC* is secure (unforgeable) when H is a random oracle, but easily broken when H is a hash function built using the Merkle-Damgård construction because of length-extension attacks.

[4] It is not called a compression function, since the domain and range of f are identical.

in [BDF$^+$11], random oracles and related objects should be modeled as functions that can be queried in superposition of different inputs. (Namely, with a real hash function, an adversary can use a quantum circuit implementing SHA3 and can thereby query the function in superposition. The adversary could evaluate the sponge on the uniform superposition over all messages of a certain length, possibly helping him to, e.g., find a collision.) Thus, it leaves us with the question whether the sponge construction (and thus hash functions like SHA3) is collapsing.

Summary. In this chapter we tackle the question whether the sponge construction is collapsing in the post-quantum setting. We show that:

- If the block function f is a random function or a random one-way permutation, then the projections of f on its inner and outer part are collapsing.
- If the inner part of the block function f is collapsing, then a step function that we define (used later in the proof) is collapsing.
- If the outer and inner part of the block function f are collapsing, and furthermore the inner part is zero-preimage resistant, then the sponge construction is collapsing.
- Several corollaries are implied, regarding the collision resistance, preimage resistance and second preimage resistance of the sponge construction.

It should be stressed that we *do not* show that the sponge construction is collapsing (or even collision-resistant) if the block function f is an *efficiently invertible* random permutation. In this case, it is trivial to find zero-preimages by applying the inverse permutation to 0. This means that the present result cannot be directly used to show the security of, say, SHA3, because SHA3 uses an efficiently invertible permutation as block function. Our results apply to hash functions where the block function is not (efficiently) invertible, e.g., Gluon [BDM$^+$12]. But we believe that these results are also a first step towards understanding the sponge construction for invertible block functions, and towards showing the post-quantum security of SHA3.

Merger of two papers. Concurrent to the work that this chapter is based on (namely [CGHS17]), there was another publication [Unr17] on the same subject. After the decision to merge the two works, many changes were applied that improved readability and even some results. Despite of this, the chapter is still largely based on the work from [CGHS17], as the author of this thesis had a lot more influence on that work. In Section 2.5 we compare the main differences between this chapter and the published, merged version.

Organization. In Section 2.2, we give the necessary background on quantum computing, the definition of collapsing and introduce the sponge construction. In Section 2.3, we begin with proving several Lemmas and preconditions, that we will later use in the full proof. In Section 2.4 we give the full proof for the collapsing property for the sponge construction. Finally, we compare the results in this chapter with the results of the published version in Section 2.5.

2.2 — Preliminaries

In this section we briefly introduce quantum computing as needed for this chapter. We revisit terminology and notations of sponge constructions, which are used throughout this chapter. Next we recall the definition of collapsing and two related results.

Basic notations. We say that $\varepsilon = \varepsilon(n)$ is negligible if, for all polynomials $p(n)$, $\varepsilon(n) < 1/p(n)$ for large enough n. With $x \overset{\$}{\leftarrow} X$, we denote sampling an element x uniformly at random from set X. With 0^m we denote the all zero bitstring of length m.

Quantum computing. We assume the reader is familiar with the usual notations in quantum computation, but we give a very short introduction here. A quantum system A is a complex Hilbert space $\mathcal{H}$, together with an inner product $\langle\cdot|\cdot\rangle$.The state of a quantum system is given by a vector $|\psi\rangle$ of unit norm ($\langle\psi|\psi\rangle = 1$). A joint system of $\mathcal{H}_1$ and $\mathcal{H}_2$ is denoted by $\mathcal{H} = \mathcal{H}_1 \otimes \mathcal{H}_2$, with elements $|\psi\rangle = |\psi_1\rangle|\psi_2\rangle$ for $|\psi_1\rangle \in \mathcal{H}_1, |\psi_2\rangle \in \mathcal{H}_2$. A unitary transformation $\mathbf{U}$ over a d-dimensional Hilbert space $\mathcal{H}$ is a $d \times d$ matrix $\mathbf{U}$ such that $\mathbf{U}\mathbf{U}^\dagger = \mathbf{I}_d$, where $\mathbf{U}^\dagger$ represents the conjugate transpose. In this thesis, we assume quantum-accessible oracles, i.e., we will implement an oracle $\mathcal{O} : X \to Y$ by a unitary transformation $\mathbf{O}$ such that: $\mathbf{O}|x, y\rangle = |x, y + \mathcal{O}(x)\rangle$, where $+ : Y \times Y \to Y$ is some group operation on Y and in this chapter this is simply the XOR operation $\oplus$. Any quantum algorithm making q queries can then be written as a combined transformation $\mathbf{U}_q\mathbf{O}_q \ldots \mathbf{U}_1\mathbf{O}_1\mathbf{U}_0$ for unitaries $\mathbf{U}_i$ applied between queries and oracle queries $\mathbf{O}_i$.

For a quantum register M we denote by $m \leftarrow \mathcal{M}_{\mathrm{comp}}(M)$ the measurement of M in the computational basis with outcome m. For a function f, we denote by $\mathcal{M}_f(M)$ the projective measurement of register M with projectors $P_y = \sum_{\mathbf{m}:f(\mathbf{m})=y} |\mathbf{m}\rangle\langle\mathbf{m}|$. In other words, applying $y \leftarrow \mathcal{M}_f(M)$ causes M to collapse to a superposition of values $\mathbf{m}$ which all map to the same image y under f. We will recall some notations and a definition from [Zha12]. Given two sets X and Y, define Y^X as the set of functions $f : X \to Y$. If a function f maps X to $Y \times Z$, we can think of f as two functions: one that maps X to Y and one that maps X to Z. We will define quantum indistinguishability from random in the following sense for function families $\mathcal{F}$.

Definition 2.1 (Quantum-Indistinguishability). *We call a family of functions $\mathcal{F} \subseteq Y^X$ quantum-indistinguishable from random if no efficient quantum adversary $\mathcal{A}$ making quantum queries can distinguish between a function drawn at random from $\mathcal{F}$ and a truly random function. That is, for every security parameter n and quantum adversary $\mathcal{A}$, there exists a negligible function $\varepsilon = \varepsilon(n)$ such that:*

$$\mathrm{Adv}^{\mathrm{QI}}_{Y^X}(\mathcal{F}; \mathcal{A}) := |\underset{f \overset{\$}{\leftarrow} \mathcal{F}}{\mathbb{P}}[A^f(1^n) = 1] - \underset{\mathcal{O} \overset{\$}{\leftarrow} Y^X}{\mathbb{P}}[A^{\mathcal{O}}(1^n) = 1]| < \varepsilon$$

We call $\mathrm{Adv}^{\mathrm{QI}}$ *the* quantum-oracle-distinguishing advantage.

The sponge construction. The concept of the cryptographic sponge construction was introduced in [BDPV07]. A sponge is a function $\mathcal{S}[f, \mathrm{pad}, r] : \{0,1\}^* \to \{0,1\}^\infty$ that uses a length-preserving transformation (a function or a permutation) f (also called internal function in this chapter), a sponge-compliant padding rule pad (defined below) and a parameter r called the bitrate. Padding is a procedure used to prepare input in a form suitable for the sponge construction. It increases the length of the input message $\mathbf{M}$ so that the length of $\mathbf{M}\|\mathrm{pad}[r](|\mathbf{M}|)$ is a multiple of r.

Definition 2.2. *A padding rule is* sponge-compliant *if it never results in the empty string and if it satisfies the following criterion:*

$$\forall n \geqslant 0\ \forall \boldsymbol{M}, \boldsymbol{M}' \in \{0,1\}^* : \boldsymbol{M} \neq \boldsymbol{M}' \Rightarrow \boldsymbol{M}\|\mathrm{pad}[r](|\boldsymbol{M}|) \neq \boldsymbol{M}'\|\mathrm{pad}[r](|\boldsymbol{M}'|)\|0^{nr}.$$

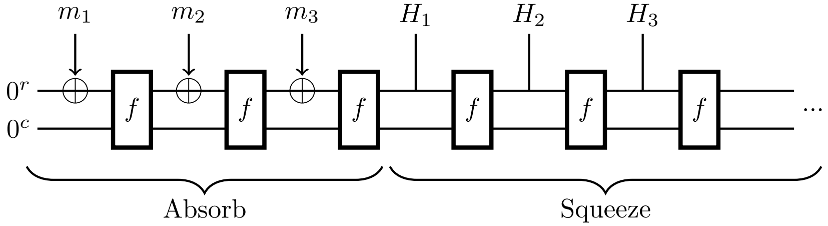


Figure 2.1: Sponge construction with three block input $m_1||m_2||m_3$ and three block output $H_1||H_2||H_3$. The application of the padding function is not shown. We call the output of the sponge construction $Z = \mathcal{S}[f, \text{pad}, r](m_1||m_2||m_3, \ell)$ if it outputs the first ℓ bits returned from the squeeze phase.

An example of such padding rule is pad10*, which appends a single bit 1 followed by the minimum number of bits 0 in order to get a multiple of r. A finite length output can be obtained by truncating the theoretically infinite output to its first ℓ bits.

The transformation f operates on $r + c$ bits, which we call the state. We call the first r bits of the state the *outer part* and the last c bits the *inner part*. One of the security parameters of the sponge construction is this inner part c of the state, which is hidden from the attacker. The size of this inner part c is called the capacity. For a state $s \in \{0,1\}^{r+c}$, we denote its outer part with an overline and the inner part with a hat: $s = (\overline{s}, \widehat{s})$ with $\overline{s} \in \{0,1\}^r$ and $\widehat{s} \in \{0,1\}^c$.

First, the state is initialized to zero. The input message is padded using pad and cut into r-bit blocks. Then the sponge construction proceeds in two phases. In the *absorbing phase*, the first r bits of the state (the outer part) are XORed with the r-bit message blocks, interleaved with applications of transformation f. When all message blocks are processed, the sponge construction switches to the *squeezing phase*. In this phase, the outer part of the state is iteratively returned as output blocks, again, interleaved with applications of f. The number of iterations is determined by the requested number of output bits ℓ. In particular, this means for $\ell \leqslant r$ that no additional applications of f are needed after the absorbing phase. For any message $\mathbf{M} \in \{0,1\}^*$, we denote the ℓ bit truncated output of the sponge construction by $Z = \mathcal{S}[f, \text{pad}, r](\mathbf{M}, \ell)$. A graphical description is shown in Figure 2.1.

In the following we will assume oracle access to the sponge, i.e. queries of (potentially a) superposition of messages $|\mathbf{m}, \ell\rangle$ to the sponge construction $\mathcal{S}[f, \text{pad}, r]$. It is important to specify the cost of each query. Different queries may not always be equally costly, since there might be both varying length input and varying length output. The cost of applying $\mathcal{S}$ is the number of evaluations of the internal function f. So evaluating $\mathcal{S}[f, \text{pad10}^*, r](\mathbf{M}, \ell)$ costs $\left\lceil \frac{|\mathbf{M}|+1}{r} \right\rceil + \left\lceil \frac{\ell}{r} \right\rceil$ queries. This varies slightly if a different padding rule is used.

Collapsing hash functions. At EUROCRYPT 2016 [Unr16b], Unruh introduced the notion of collapsing. This is a purely quantum notion, which is defined for a function H as follows:

Definition 2.3. *[Unr16b, Definition 23] For a function* H *and algorithms* $\mathcal{A}, \mathcal{B}$*, consider* *the following games:*

$$\begin{aligned} \mathsf{Game}_1 &: (S, M, h) \leftarrow \mathcal{A}(1^n), & m \leftarrow \mathcal{M}_{comp}(M), b \leftarrow \mathcal{B}(1^n, S, M) \\ \mathsf{Game}_2 &: (S, M, h) \leftarrow \mathcal{A}(1^n), & b \leftarrow \mathcal{B}(1^n, S, M) \end{aligned}$$

Here, S and M are quantum registers and $\mathcal{M}_{comp}(M)$ is a measurement of M in the computational basis. We call an adversary $(\mathcal{A}, \mathcal{B})$ valid if $\mathbb{P}[H(m) = h] = 1$ when we run $(S, M, h) \leftarrow \mathcal{A}(1^n)$ and measure M in the computational basis as m.

A function H is collapsing iff for any valid quantum-polynomial-time adversary $(\mathcal{A}, \mathcal{B})$, the difference $\mathrm{Adv}_H^{coll}(A, B) := |\mathbb{P}[b = 1 : \mathsf{Game}_1] - \mathbb{P}[b = 1 : \mathsf{Game}_2]|$ is negligible. We call Adv_H^{coll} the collapsing advantage *of H.*

Informally, the collapsing definition means the following. Any adversary in the notion of collapsing runs in two phases, given by algorithms $(\mathcal{A}, \mathcal{B})$. A valid algorithm $\mathcal{A}$ outputs a register of messages $M = |\phi\rangle = \sum_{\mathbf{m}} \alpha_{\mathbf{m}} |\mathbf{m}\rangle$ and a classical output h, such that $H(\mathbf{m}) = h$ for all $\mathbf{m}$ in register M. It also outputs an additional quantum register S (that may contain anything) that can potentially be used by algorithm $\mathcal{B}$. Now there are two possibilities before $\mathcal{B}$ receives the registers (S, M). In Game_2, there is no measurement on register M, so $\mathcal{B}$ receives $M = |\phi\rangle = \sum_{\mathbf{m}} \alpha_{\mathbf{m}} |\mathbf{m}\rangle$ as in the output of $\mathcal{A}$. In Game_1 however, register M is measured before $\mathcal{B}$ receives it, so in this case $\mathcal{B}$ receives $M = |\mathbf{m}\rangle$ with probability $|\alpha_{\mathbf{m}}|^2$. Algorithm $\mathcal{B}$ is also allowed to do anything, including the measurement of register M. We call function H collapsing if no adversary $(\mathcal{A}, \mathcal{B})$ can distinguish between these two games, i.e. can tell whether the measurement on register M happened or not.

Two related results from [Unr16b], which we use are:

Theorem 2.4. *[Unr16b, Theorem 38] Let Y be finite, $X \subseteq \{0,1\}^*$ (finite or infinite) and let q be the numbers of (quantum) queries any attacker makes. Then $H \xleftarrow{\$} Y^X$ is collapsing with advantage $O(\sqrt{q^3/|Y|})$.*

Lemma 2.5. *[Unr16b, Lemma 25] A collapsing function is collision resistant.*

Proof sketch: A tight reduction is given in [Unr16b]. The basic idea is to use two colliding messages (m, m') with $H(m) = H(m') = h$, and initialize register M with $|\psi_{m,m'}\rangle = \frac{1}{\sqrt{2}}(|m\rangle + |m'\rangle)$. Algorithm $\mathcal{B}$ is able to detect a measurement of register M with non-negligible probability, using the projective measurement $|\psi_{m,m'}\rangle \langle \psi_{m,m'}|$. Details are given in the original paper.

In another recent paper by Unruh [Unr16a], it is proven that the Merkle-Dåmgard construction is collapsing when the compression function is. From this result and Lemma 2.5, it follows that the Merkle-Dåmgard construction is collision resistant in the post-quantum setting.

2.3 — The post-quantum security of sponges

In this section, we provide necessary prerequisites for the main proof that the sponge construction is collapsing. We also show sample consequences of the theorems, before we show the main proof in the next section.

Collapsing internal functions. The sponge construction uses an internal function $f : \{0,1\}^{r+c} \to \{0,1\}^{r+c}$, which is either a random function f^R or a random permutation f^π. Since the function maps values in $X = \{0,1\}^{r+c}$ into two spaces $Y_0 = \{0,1\}^r$ and $Y_1 = \{0,1\}^c$ with $|X| = |Y_0| \cdot |Y_1|$, we can think of this function as two functions f_0 and f_1, mapping X to Y_0 and Y_1, respectively. In the following, let $P_i(f(x))$ be the projection of $f(x)$ onto Y_i (this means $P_i(f(x)) = f_i(x)$) for $i \in \{0,1\}$.

Lemma 2.6. *Let $\mathcal{F} \subseteq (Y_0 \times Y_1)^X$ be a family of quantum-indistinguishable functions, i.e., for any efficient quantum adversary $\mathcal{A}$ and security parameter n, there exists a negligible function $\varepsilon = \varepsilon(n)$ with*

$$\mathrm{Adv}^{\mathrm{QI}}_{(Y_0 \times Y_1)^X}(\mathcal{F}; A) := |\underset{f \xleftarrow{\$} \mathcal{F}}{\mathbb{P}}[\mathcal{A}^f(1^n) = 1] - \underset{\mathcal{O} \xleftarrow{\$} (Y_0 \times Y_1)^X}{\mathbb{P}}[\mathcal{A}^{\mathcal{O}}(1^n) = 1]| < \varepsilon$$

Let $\mathcal{F}_i \overset{\mathrm{def}}{=} \{P_i(f) | f \in \mathcal{F}\}$, i.e., the family of the projections of all elements of $\mathcal{F}$ onto Y_i, for $i \in \{0,1\}$. Then for all efficient quantum adversaries $\mathcal{A}_i$ and all $i \in \{0,1\}$:

$$\mathrm{Adv}^{\mathrm{QI}}_{Y_i^X}(\mathcal{F}; \mathcal{A}_i) := |\underset{f_i \xleftarrow{\$} \mathcal{F}_i}{\mathbb{P}}[\mathcal{A}_i^{f_i}(1^n) = 1] - \underset{\mathcal{O} \xleftarrow{\$} Y_i^X}{\mathbb{P}}[\mathcal{A}_i^{\mathcal{O}}(1^n) = 1]| < \varepsilon$$

Proof. Suppose there exists an adversary $\mathcal{A}_i$ such that $\mathrm{Adv}^{\mathrm{QI}}_{Y_i^X}(\mathcal{F}; \mathcal{A}_i) = \mu \geqslant \varepsilon$ for arbitrary but fixed $i \in \{0,1\}$. We will now construct an oracle machine $\mathcal{V}^{\mathcal{A}_i}$ to distinguish $f \xleftarrow{\$} \mathcal{F}_i$ from a random function with $\mathrm{Adv}^{\mathrm{QI}}_{(Y_0 \times Y_1)^X}(\mathcal{F}; \mathcal{V}^{\mathcal{A}_i}) \geqslant \varepsilon$.

The oracle machine $\mathcal{V}^{\mathcal{A}_i}$ is given black box access to $\mathcal{O}$, which is either randomly drawn from $\mathcal{F}$ or Y^X, and distinguishes the two cases as follows:

1. Construct function $g : X \to Y_i$ by $g(x) = P_i(\mathcal{O}(x))$.
2. Run $\mathcal{A}_i$ with function g, and output whatever $\mathcal{A}_i$ outputs.

When $\mathcal{V}^{\mathcal{A}_i}$ is given access to an element of $\mathcal{F}$, $\mathcal{A}_i$ will be given access to $g \in \mathcal{F}_i$. When $\mathcal{V}^{\mathcal{A}_i}$ is given a random $\mathcal{O} \xleftarrow{\$} Y^X$, $\mathcal{A}_i$ will be given $g = P_i(\mathcal{O})$ which is randomly distributed in Y_i^X. Hence, the advantage of $\mathcal{V}^{\mathcal{A}_i}$ will be exactly μ. Consequently, we got $\mathrm{Adv}^{\mathrm{QI}}_{(Y_0 \times Y_1)^X}(\mathcal{F}; \mathcal{V}^{\mathcal{A}_i}) = \mu \geqslant \varepsilon$ as claimed. □

Let $\mathcal{F}^R \subset (Y_0 \times Y_1)^X$ denote the family of random functions and let $\mathcal{F}^\pi \subset (Y_0 \times Y_1)^X$ denote the family of random permutations. When a random function f^R is used as the internal function, we have that $\mathrm{Adv}^{\mathrm{QI}}_{(Y_0 \times Y_1)^X}(\mathcal{F}^R; \mathcal{A}) = 0$ for any quantum adversary $\mathcal{A}$, since the output distributions for these functions are the same. This means that also for f_0^R and f_1^R as defined above we have $\mathrm{Adv}^{\mathrm{QI}}_{Y_i^X}(\mathcal{F}^R; \mathcal{A}_i) = 0$ for any quantum adversary $\mathcal{A}_i$.

In the case that the internal function is a random permutation $f^\pi \in \mathcal{F}^\pi$, it is shown in [Zha15, Section 3.1] that the distinguishing advantage between a random permutation and a random function is $\mathrm{Adv}^{\mathrm{QI}}_{(Y_0 \times Y_1)^X}(\mathcal{F}^\pi; \mathcal{A}) \leqslant \varepsilon$ with $\varepsilon \in O(q^3/|X|)$ for any $\mathcal{A}$ making q quantum queries. Using Lemma 2.6 we also have that the parts f_0^π, f_1^π have $\mathrm{Adv}^{\mathrm{QI}}_{Y_i^X}(\mathcal{F}^\pi; \mathcal{A}_i) = \varepsilon$ with $\varepsilon \in O(q^3/|X|)$ for any $\mathcal{A}_i$ making q quantum queries and $i \in \{0,1\}$.

In [Unr16b], it was shown that a random oracle $\mathcal{O} : \{0,1\}^* \to Y$ is collapsing with advantage $\mathrm{Adv}^{\mathrm{coll}}_{\mathcal{O}}(A, B) = |\mathbb{P}[\mathrm{Game}_1 : b = 1] - \mathbb{P}[\mathrm{Game}_2 : b = 1]| \leqslant \delta$ with $\delta \in O(\sqrt{q^3/|Y|})$, where the adversary $(\mathcal{A}, \mathcal{B})$ makes at most q queries. From this result, the following lemma follows immediately:

Lemma 2.7. *Let* $f : X \to Y_0 \times Y_1$ *either be a random function or a random permutation with* $|X| = |Y_0| \cdot |Y_1|$*. Let* $f_i : X \to Y_i$ *be given by* $P_i(f(x))$*, where* P_i *is the projection onto* Y_i*, for* $i \in \{0, 1\}$*. Then* f_i *is collapsing for* $i \in \{0, 1\}$ *with advantage* $O(\sqrt{q^3/|Y_i|})$ *for any adversary* $(\mathcal{A}, \mathcal{B})$ *making at most* q *queries.*

Proof. In the following, let $(\mathcal{A}, \mathcal{B})$ be any adversary making at most q queries. From the fact that random oracles are collapsing (Theorem 2.4) and Lemma 2.6, it follows straightforwardly that random functions f_0^R, f_1^R are collapsing with advantage $\mathrm{Adv}^{\mathrm{coll}}_{f_i^R}(\mathcal{A}, \mathcal{B}) = |\mathbb{P}[b = 1 : \mathsf{Game}_1] - \mathbb{P}[b = 1 : \mathsf{Game}_2]| \leqslant \delta_i^R$ with $\delta_i^R \in O(\sqrt{q^3/|Y_i|})$ for $i \in \{0, 1\}$. The random permutations f_0^π, f_1^π are indistinguishable from random functions with advantage $\varepsilon \in O(q^3/|X|)$, so we have collapsing advantage $\mathrm{Adv}^{\mathrm{coll}}_{f^\pi}(\mathcal{A}, \mathcal{B}) \leqslant \delta_i^R + \varepsilon =: \delta_i^\pi$ with $\delta_i^\pi \in O(\sqrt{q^3/|Y_i|})$ as well. □

Note that, as $|Y_i| \leqslant |Y|$, the collapsing advantage is higher for these projections, but that would also be the case for the success probability of finding collisions.

Collapsing step function. For the proof of Theorem 2.13 given below, we show that the sponge construction is collapsing by using a hybrid argument. Each step in the hybrid argument exploits the fact that the projections of the internal function on the inner and outer state used in the sponge are collapsing. However, for the flow of the proof it is easier to look at a different function, which we call step_f. To make this more formal: for $f(x) = (\overline{f(x)}, \widehat{f(x)})$, we define $\mathsf{step}_f(x, y) := f(x) \oplus (y \| 0^c) = (\overline{f(x)} \oplus y, \widehat{f(x)})$. In the proof below, we only use the second way of writing $\mathsf{step}_f(x, y)$, since this underlines the fact that we have two functions $\overline{f(x)}$ and $\widehat{f(x)}$. The next lemma shows that $\mathsf{step}_f(x, y)$ is collapsing if $\widehat{f(x)}$ is collapsing.

Lemma 2.8. *Let* $X = \overline{Y} \times \widehat{Y}$*,* $f : X \to X$ *be given by two functions* $f(x) = (\overline{f(x)}, \widehat{f(x)})$*, with* $\overline{f} : X \to \overline{Y}$*, and* $\widehat{f} : X \to \widehat{Y}$*. Define* $\mathsf{step}_f : X \times \overline{Y} \to X$ *as* $\mathsf{step}_f(x, y) := f(x) \oplus (y \| 0^c) = (\overline{f(x)} \oplus y, \widehat{f(x)})$*. Then if* $\widehat{f}$ *is collapsing with advantage* ε*, then also* step_f *is collapsing with advantage* ε*.*

Proof. Let $(\mathcal{A}, \mathcal{B})$ be a valid quantum-polynomial-time adversary for the collapsing games for function step_f with advantage $\mathrm{Adv}^{\mathrm{coll}}_{\mathsf{step}_f}(\mathcal{A}, \mathcal{B}) = \mu > \varepsilon$. We now construct a quantum-polynomial-time oracle machine $(\mathcal{V}^{\mathcal{A}}, \mathcal{W}^{\mathcal{B}})$ for function $\widehat{f}$ that has collapsing advantage $\mathrm{Adv}^{\mathrm{coll}}_{\widehat{f}}(\mathcal{V}^{\mathcal{A}}, \mathcal{W}^{\mathcal{B}}) = \mu$.

Let $\mathcal{V}^{\mathcal{A}}$ be the following quantum algorithm: it runs $(S^*, M^*, h^*) \leftarrow \mathcal{A}(1^n)$ to obtain quantum registers, such that $\mathcal{B}$ can distinguish between the collapsing games for step_f with advantage $\mu > \varepsilon$. In particular, M^* is a quantum register consisting of two registers $M^* = (X^*, Y^*)$ which contain superpositions of basis states $|x, y\rangle$ that fulfill $\mathsf{step}_f(x, y) = h^*$. Denoting $h^* = \overline{h} \| \widehat{h}$, $\mathcal{V}^{\mathcal{A}}$ outputs (S, M, h) with $S = (S^*, Y^*, h^*)$, $M = X^*$ and $h = \widehat{h}$. In other words, algorithm $\mathcal{V}^{\mathcal{A}}$ simply rearranges the registers from $\mathcal{A}$. The algorithm $\mathcal{W}^{\mathcal{B}}$ retrieves (S, M, h) and simply reverses the rearrangement from $\mathcal{V}^{\mathcal{A}}$: it sets M^* to be (X^*, Y^*) and sends (S^*, M^*) to $\mathcal{B}$. $\mathcal{W}^{\mathcal{B}}$ will output whatever $\mathcal{B}$ outputs.

We need to show two things now: $(\mathcal{V}^{\mathcal{A}}, \mathcal{W}^{\mathcal{B}})$ is a valid adversary for the collapsing games of $\widehat{f}$ and $\mathrm{Adv}^{\mathrm{coll}}_{\widehat{f}}(\mathcal{V}^{\mathcal{A}}, \mathcal{W}^{\mathcal{B}}) = \mu$. Validity of $(\mathcal{V}^{\mathcal{A}}, \mathcal{W}^{\mathcal{B}})$ follows from validity of

$(\mathcal{A}, \mathcal{B})$, because register M (which contains X^*) must yield messages x for which $\widehat{f(x)} = \widehat{h} = h$. Otherwise, $(\mathcal{A}, \mathcal{B})$ was invalid.

For the collapsing advantage of $(\mathcal{V}^{\mathcal{A}}, \mathcal{W}^{\mathcal{B}})$, we note that in Game_1, register M will be measured and thereby collapses to a single x such that $\widehat{f(x)} = \widehat{h}$. However, the crucial point is that also $y = \overline{f(x)} \oplus \overline{h}$ collapses, since by measuring x, the part $\overline{f(x)}$ cannot be in a superposition anymore and $\overline{h}$ was already classical. In Game_2, no such measurement occurs on register M, so x is left untouched and hence (possibly) in superposition. Therefore, the collapsing advantage of $(\mathcal{V}^{\mathcal{A}}, \mathcal{W}^{\mathcal{B}})$ for $\widehat{f}$ is equal to the one of $(\mathcal{A}, \mathcal{B})$ for step_f.

□

Note that the above Lemma does not hold for $\overline{f(x)}$ instead of $\widehat{f(x)}$. To be a valid adversary, $\mathcal{V}^{\mathcal{A}}$ has to construct a register M and a register c, where register M contains messages x such that $\overline{f(x)} = h$ with probability 1. However, from adversary $\mathcal{A}$ it receives messages $|x, y\rangle$ from register $M^* = (X^*, Y^*)$, such that $\overline{f(x)} \oplus y = \overline{h}$. In particular, this could mean that $|x, y\rangle$ is in a superposition satisfying $\overline{f(x)} = \overline{h} \oplus y$. If $|x, y\rangle$ is in a superposition, then possibly $\overline{f(x)}$ is also in a superposition of states. But $\mathcal{V}^{\mathcal{A}}$ should give a fixed outcome h (with probability 1) for $\overline{f(x)}$. The only possible way is to measure register Y^*, but this measurement (possibly) violates any collapsing advantage of $(\mathcal{A}, \mathcal{B})$ for step_f. Hence, it is not possible to construct a valid adversary $(\mathcal{V}^{\mathcal{A}}, \mathcal{W}^{\mathcal{B}})$ for $\overline{f(x)}$, that communicates with $(\mathcal{A}, \mathcal{B})$.

Collapsing sponges. Recall that a sponge is denoted by $\mathcal{S}[f, \mathrm{pad}, r](\mathbf{M}, \ell)$ with capacity c, where $f : \{0,1\}^{r+c} \to \{0,1\}^{r+c}$ is the internal permutation (or function), pad is the sponge-compliant padding rule, r is the bitrate, $\mathbf{M} \in \{0,1\}^*$ is the message and ℓ is the fixed output length. For simplicity we take $\ell = r$ in Theorem 2.13. We handle cases $\ell < r$ and $\ell > r$ separately, afterwards. This simplified sponge construction will be denoted by $\mathcal{S}_f$ and is defined as follows:

Definition 2.9. *We define the internal iteration function of the sponge as* $\mathcal{I}_f : (\{0,1\}^r)^* \to (\{0,1\}^{r+c})$*, with message length divisible by* r*,* $\mathcal{I}_f(\lambda) := 0^{r+c}$ *for the empty word* λ*, and* $\mathcal{I}_f(\boldsymbol{m} \| m') := f(\mathcal{I}_f(\boldsymbol{m}) \oplus (m' \| 0^c))$*, for* $m' \in \{0,1\}^r$*. The simplified sponge function is then defined by* $\mathcal{S}_f : (\{0,1\}^r)^* \to (\{0,1\}^r)$*, with* $\mathcal{S}_f(\boldsymbol{m}) = \overline{\mathcal{I}_f(\boldsymbol{m})}$*, where* $\overline{\mathcal{I}_f(\boldsymbol{m})}$ *denotes the outer part of* $\mathcal{I}_f(\boldsymbol{m})$*. We assume* $\boldsymbol{m} = \mathrm{pad}(\boldsymbol{M})$ *is the output of a sponge-compliant padding function.*

We omit the term "simplified" below. Our notation and proof technique follows Unruh's proof [Unr16a, Section 4] for the collapsing property of the Merkle-Damgård (MD) construction. However, the proof for the sponge construction turns out to be more complicated. The main reason being the necessity to consider the two parts of f: both $\overline{f(x)}$ and $\widehat{f(x)}$ need to be collapsing. There are two reasons for this. First, the output does not allow to compute the full final state of the function: only the outer part of the final state is given as output of the sponge. The second reason is that in contrast to MD, in a sponge, message blocks are not used as direct input to the internal function f. Instead, these message blocks are XORed with the outer part, which makes it more difficult to analyze. This even requires a second property of f, which is naturally fulfilled by random functions (or permutations):

Definition 2.10. *A family of functions $\mathcal{F} = \{f : X_n \to \{0,1\}^n\}$ is zero-preimage resistant if for any efficient quantum adversary $\mathcal{A}$ there exists a negligible $\varepsilon(n)$ such that*

$$\mathsf{Succ}^{\mathrm{Qzpre}}_{\mathcal{A}}(\mathcal{F}) := \mathbb{P}\left[f \xleftarrow{\$} \mathcal{F}, x \leftarrow \mathcal{A}^f(1^n) : f(x) = 0\right] < \varepsilon(n).$$

In the definition, we are giving the adversary quantum-oracle access to f as we are concerned with zero-preimage resistance of perfectly random functions. In a definition for efficient function families (i.e., with polynomial-size descriptions), $\mathcal{A}$ would just be given a description of f.

Lemma 2.11. *Given a security parameter $n \in \mathbb{N}$, $r, c \in \mathrm{poly}(n)$, the success probability of any q-query quantum adversary against the zero-preimage resistance of the family of random functions $\mathcal{F} = \{f : \{0,1\}^{r+c} \to \{0,1\}^c\}$ is upper bounded by*

$$\mathsf{Succ}^{\mathrm{Qzpre}}_{\mathcal{A}}(\mathcal{F}) \leqslant O(q^2/2^c)\,.$$

We omit a detailed proof of Lemma 2.11 as it can be obtained by following the proof for quantum preimage resistance of random functions in [HRS16].

We need this property later in the proof, because using a $\hat{f}$-preimage of zero, one can easily construct a collision for the sponge: given an input $x \in \{0,1\}^r$ such that $f(\widehat{x\|0^c}) = 0^c$, a collision of the sponge is given by two messages $m_1 = (x\|y \oplus x')$ and $m_2 = x'$ for any random $x' \in \{0,1\}^r$ and $y = \overline{f(x)}$. Note that the above construction does not give a collision for the internal function f, but it does give a collision for $\mathcal{S}_f$.

We will also make use of the following technical lemma:

Lemma 2.12. *Let ρ be a quantum state (density operator of trace 1). Let $\mathcal{M}$ be a projective measurement consisting of projectors $P_1, \dots, P_n$. Assume that applying $\mathcal{M}$ to ρ gives outcome 1 with probability $\geqslant 1 - \varepsilon$.*

Let ρ' be the result of applying $\mathcal{M}$ to ρ (and discarding the result).

Then the trace distance between ρ and ρ' is $\leqslant \sqrt{\varepsilon}$. (I.e., no algorithm can distinguish those states better than with probability $\sqrt{\varepsilon}$.)

Proof. Without loss of generality, we can assume that ρ is a pure state $\rho = |\Psi\rangle\langle\Psi|$. (The general case of the lemma is then obtained by considering a purification $|\Psi\rangle\langle\Psi|$ of the mixed state ρ.)

Then $\rho' = \sum_i P_i |\Psi\rangle\langle\Psi| P_i$. Let F denote the fidelity and let TD denote the trace distance. By [NC10, (9.60)], we have $F(|\Psi\rangle\langle\Psi|, \rho') = \sqrt{\langle\Psi|\rho'|\Psi\rangle}$. Thus,

$$F(\rho,\rho')^2 = \langle\Psi|\rho'|\Psi\rangle = \sum_i \langle\Psi|P_i|\Psi\rangle\langle\Psi|P_i|\Psi\rangle = \sum_i |\langle\Psi|P_i|\Psi\rangle|^2 \geqslant |\langle\Psi|P_1|\Psi\rangle|^2$$
$$\geqslant 1 - \varepsilon.$$

Thus

$$\mathrm{TD}(\rho,\rho') \leqslant^* \sqrt{1 - F(\rho,\rho')^2} \leqslant \sqrt{1 - (1-\varepsilon)} = \sqrt{\varepsilon}.$$

Here $(*)$ uses [NC10, (9.101)]. □

In the following we only assume padded messages, i.e. the padded message space $\mathcal{P} \subset (\{0,1\}^r)^*$ is consisting of r-bit block messages. As the padded message space $\mathcal{P}$ is the output of a sponge-compliant padding, it cannot contain the empty word. For a multi-block message $\mathbf{m} \in (\{0,1\}^r)^*$, let from now on $|\mathbf{m}|$ denote the number of r-bit blocks in $\mathbf{m}$ (so the bit-length of $\mathbf{m}$ is $r|\mathbf{m}|$), overloading notation.

Theorem 2.13. *Let $\mathcal{P} \subset (\{0,1\}^r)^*$ be the set of padded messages for some sponge-compliant padding with padded messages $\mathbf{m} \in \mathcal{P}$ such that $T \geqslant |\mathbf{m}| \geqslant 2$ for some upper bound T, that is defined as the maximum length of the messages used by the adversary. Let the internal function of the sponge $f : \{0,1\}^{r+c} \to \{0,1\}^{r+c}$ be $f(x) = (\overline{f(x)}, \widehat{f(x)})$ where both outer part $\overline{f(x)} : \{0,1\}^{r+c} \to \{0,1\}^r$ and inner part $\widehat{f(x)} : \{0,1\}^{r+c} \to \{0,1\}^c$ are collapsing with advantages $\overline{\varepsilon}, \widehat{\varepsilon}$ respectively, and let furthermore $\widehat{f}$ be zero-preimage resistant with advantage ε_z and let the output length of $\mathcal{S}_f$ be fixed to $\ell = r$ bits.Then $\mathcal{S}_f$ is collapsing on $\mathcal{P}$ with advantage*

$$\mathrm{Adv}^{\mathrm{coll}}_{\mathcal{S}_f} = \overline{\varepsilon} + (T-1)(\sqrt{\varepsilon_z} + \widehat{\varepsilon}).$$

Proof sketch: We have to show that if an adversary $(\mathcal{A}, \mathcal{B})$ outputs classical $h = \mathcal{S}_f(\mathbf{m})$, we can measure register M without the adversary noticing. We show this developing hybrids that successively measure more and more of the message register M. Afterwards we upper bound the advantage of $(\mathcal{A}, \mathcal{B})$ in the collapsing game by upper bounding the success probability of $(\mathcal{A}, \mathcal{B})$ in distinguishing any two consecutive hybrids.

The output of the sponge is simply the outer part of $\mathcal{I}_f(\mathbf{m})$. Hence, we can upper bound the advantage of detecting the measurement of the last state of the sponge using the collapsing property of $\overline{f}$. For all the remaining hybrids, we can upper bound the distinguishing advantage using the collapsing property of step_f and the zero-preimage resistance of $\widehat{f}$. The intuition is that a successful distinguisher either uses a superposition of messages where the i-th blocks of some of the messages are different (and this would allow to break the collapsing property of step_f) or of some messages of length i and some other messages that are longer, but which agree on the first i blocks (and this would allow to extract a zero-preimage of $\widehat{f}$).

We delay the full proof to Section 2.4 and first give implications of the proof. In the following we use Theorem 2.13 to derive proofs for the desired properties of hash functions. But first we still have to show that $\mathcal{S}[f, \mathrm{pad}, r](\mathbf{M}, \ell)$ is also collapsing for $r \neq \ell$.

Theorem 2.14. *Let $\mathcal{S}[f, \mathrm{pad}, r](\boldsymbol{M}, \ell)$ be a sponge construction with capacity c and internal function f having the desired properties as described in Theorem 2.13. Then, the collapsing advantage of any valid, efficient quantum adversary $(\mathcal{A}, \mathcal{B})$ against $\mathcal{S}$, making no more than q queries, is upper bounded by $\mathrm{Adv}^{\mathrm{coll}}_{\mathcal{S}}(\mathcal{A}, \mathcal{B}) \in O(\sqrt{q^3 \cdot \max(2^{-\ell}, 2^{-r}, 2^{-c})})$.*

Proof. In the following, let $(\mathcal{A}, \mathcal{B})$ be any adversary making at most q queries. We have handled the case where $\ell = r$ in Theorem 2.13. By plugging values $\overline{\epsilon} \in O(\sqrt{q^3/2^r})$ and $\widehat{\epsilon} \in O(\sqrt{q^3/2^c})$ for the collapsing of $\overline{f}, \widehat{f}$ respectively (using Lemma 2.7), and $\epsilon_z \in O(q^2/2^c)$ (using Lemma 2.11) into the bound of Theorem 2.13, we derive at a collapsing advantage of $O(\sqrt{q^3/2^r} + \sqrt{q^3/2^c} + \sqrt{q^2/2^c}) = O(\sqrt{q^3/2^r} + \sqrt{q^3/2^c})$ for a sponge construction $\mathcal{S}[f, \mathrm{pad}, r](\mathbf{M}, r)$ with capacity c. We denote $Z = \mathcal{S}[f, \mathrm{pad}, r](\mathbf{M}, \ell)$ to be the output of the sponge construction with $\ell > r$. Since the $\ell > r$ output bits are truncated after the squeezing phase, we have that the first r output bits of Z are equal to $\mathcal{S}_f(\mathbf{m})$

where $\mathbf{m} = \mathrm{pad}(\mathbf{M})$ (see Figure 2.1). Thus, we can simply ignore the remaining $\ell - r$ output bits: recall that in the collapsing notion the output register h is classical, hence when h is shorter the collapsing advantage should grow. Using the same arguments as in Theorem 2.13 we obtain the bound $O(\sqrt{q^3/2^r} + \sqrt{q^3/2^c})$.

When $\ell < r$, we only have a difference with the proof of Theorem 2.13 in the first step of the hybrid argument. Defining $\overline{f_\ell} : \{0,1\}^{r+c} \to \{0,1\}^\ell$ by $\overline{f_\ell}(x) = P_\ell(\overline{f(x)})$, where P_ℓ is the projection onto the first ℓ bits, we can apply Lemma 2.7 to get a collapsing advantage of $\overline{\varepsilon_\ell} := \mathrm{Adv}^{\mathrm{coll}}_{\overline{f_\ell}}(\mathcal{A}, \mathcal{B}) = O(\sqrt{q^3/2^\ell})$ for $\overline{f_\ell}$. The remaining part of the proof of Theorem 2.13 remains the same, but in the final result changing $\overline{\varepsilon}$ to $\overline{\varepsilon_\ell}$. This means the collapsing advantage in this case will be upper-bounded by $\mathrm{Adv}^{\mathrm{coll}}_{\mathcal{S}}(\mathcal{A}, \mathcal{B}) \in O(\sqrt{q^3/2^\ell} + \sqrt{q^3/2^c})$.

Lastly, note that in Lemma 4 we required $|\mathbf{m}| \geqslant 2$, but we drop this restriction here. The collapsing case for $|\mathbf{m}| = 1$ is trivial, as it follows from the collapsing property of $\overline{f}$ and $\mathcal{S}_f(\mathbf{m}) = \overline{f}(\mathbf{m}||\mathbf{0})$ where $\mathcal{S}_f$ is defined as in Theorem 2.13.

Putting things together we get the result in the corollary.

□

We are now ready to prove the following corollaries:

Corollary 2.15. *Let* $\mathsf{Sponge}[f, \mathrm{pad}, r](\boldsymbol{M}, \ell)$ *be a sponge construction with capacity* c *and internal function* f *having the desired properties as described in Theorem 2.13. Any quantum collision finder* $\mathcal{A}$ *making at most* q *queries, has a success probability of at most* $O(\sqrt{q^3 \cdot \max(2^{-\ell}, 2^{-r}, 2^{-c})})$.

Proof. The proof follows immediately from Theorem 2.14 and the tight reduction of collision resistance to collapsing(Lemma 2.5). □

Given Corollary 2.15, we immediately obtain the following corollary on quantum second-preimage resistance.

Corollary 2.16. *Let* $\mathcal{S}[f, \mathrm{pad}, r](\boldsymbol{M}, \ell)$ *be a sponge construction with capacity* c *and internal function* f *having the desired properties as described in Theorem 2.13. Any quantum second-preimage finder* $\mathcal{A}$ *making at most* q *queries, has a success probability of at most* $O(\sqrt{q^3 \cdot \max(2^{-\ell}, 2^{-r}, 2^{-c})})$.

The corollary follows from Corollary 2.15 as any second-preimage finder $\mathcal{A}$ can be used as collision finder. The reduction just runs $\mathcal{A}$ on a random domain element x and returns x together with $\mathcal{A}$'s output.

With slightly more effort, we also obtain the following corollary on quantum preimage resistance.

Corollary 2.17. *Let* $\mathcal{S}[f, \mathrm{pad}, r](\boldsymbol{M}, \ell)$ *be a sponge construction with capacity* c *and internal function* f *having the desired properties as described in Theorem 2.13. If* $|\boldsymbol{m}| \geqslant 2$ *for all* $\boldsymbol{m} \in \boldsymbol{M}$*, then any quantum preimage finder* $\mathcal{A}$ *making at most* q *queries, has a success probability of at most* $O(\sqrt{q^3 \cdot \max(2^{-\ell}, 2^{-r}, 2^{-c})})$.

The corollary follows from Corollary 2.15 as any preimage finder $\mathcal{A}$ can be used as collision finder. The reduction takes a random domain element x, runs $\mathcal{A}$ on $\mathcal{S}_f(x)$ and returns x together with $\mathcal{A}$'s output. A sponge function compresses inputs of length up to

$T \cdot r$ bits to an intermediate state of $r+c$ bits. For all practical parameters, $T \cdot r \gg (r+c)$ and a sponge is classically indistinguishable from a random function. Hence, every image has an exponential number of preimages. Therefore, the probability that $\mathcal{A}$ returns x is negligible.

On invertible permutations Note that Theorem 2.13 tells us that the sponge construction is collapsing if the internal function f is a random function. Furthermore, recall from the beginning of this section that a quantum computer, making at most q queries, successfully distinguishes a random function from a random permutation with probability $O(q^3/|X|)$, where in the case for the internal function we have $|X| = 2^{r+c}$. In other words, one could hope to use Theorem 2.13 with the internal block function being a random permutation, as the random function versus random permutation distinguishing advantage is lower than the collapsing advantage. However, this only works if the internal function f is a random permutation *that is not efficiently invertible*. If f is a permutation that we can efficiently invert, the theorem does not tell us anything. More specifically, if f is a random permutation, and the adversary has access to f, f^{-1}, then Theorem 2.13 does not apply.

In fact, if f is efficiently invertible, we cannot apply our main result Theorem 2.13 at all: In that case $\widehat{f(x)}$ is not zero-preimage-resistant (to find a zero-preimage, we simply invoke $f^{-1}(y\|0^c)$ for an arbitrary $y \in \{0,1\}^r$).

And $\widehat{f(x)}$ is also not collapsing: We get a collision of $\widehat{f(x)}$ by computing $x := f^{-1}(y\|z)$ and $x' := f^{-1}(y'\|z)$ for arbitrary y, y', z with $y \neq y'$. So $\widehat{f(x)}$ is not collision-resistant, hence not collapsing so neither is the step function. Similarly, $\overline{f(x)}$ is not collapsing, either.

So, none of the preconditions of Theorem 2.13 are satisfied, and we cannot derive that the sponge construction is collapsing (for efficiently invertible f).

That does not mean that the sponge construction is not collapsing in that setting: at least it is not obvious how one would use f^{-1} to break the collapsing property. For example, if we try to find a two-block collision $(m_1\|m_2, m_1'\|m_2')$ for $\mathcal{S}$, then we need that $f(m_1\|0^c) \oplus m_2\|0^c = f(m_1'\|0^c) \oplus m_2'\|0^c$. It remains unclear how can we solve this equation by using f^{-1}. For other kinds of collisions that we could think of, we fail similarly.

2.4 — Sponges are collapsing

Here we will formally prove Theorem 2.13, using a hybrid argument.

Theorem 2.13. *Let $\mathcal{P} \subset (\{0,1\}^r)^*$ be the set of padded messages for some sponge-compliant padding with padded messages $\boldsymbol{m} \in \mathcal{P}$ such that $T \geqslant |\boldsymbol{m}| \geqslant 2$ for some upper bound T, that is defined as the maximum length of the messages used by the adversary. Let the internal function of the sponge $f : \{0,1\}^{r+c} \to \{0,1\}^{r+c}$ be $f(x) = (\overline{f(x)}, \widehat{f(x)})$ where both outer part $\overline{f(x)} : \{0,1\}^{r+c} \to \{0,1\}^r$ and inner part $\widehat{f(x)} : \{0,1\}^{r+c} \to \{0,1\}^c$ are collapsing with advantages $\overline{\varepsilon}, \widehat{\varepsilon}$ respectively, and let furthermore $\widehat{f}$ be zero-preimage resistant with advantage ε_z and let the output length of $\mathcal{S}_f$ be fixed to $\ell = r$ bits.Then $\mathcal{S}_f$ is collapsing on $\mathcal{P}$ with advantage*

$$\mathrm{Adv}^{\mathrm{coll}}_{\mathcal{S}_f} = \overline{\varepsilon} + (T-1)(\sqrt{\varepsilon_z} + \widehat{\varepsilon}).$$

Proof. Assume a quantum adversary $(\mathcal{A}, \mathcal{B})$ that is valid on $\mathcal{P}$ for the sponge function $\mathcal{S}_f$. Let Game_1 and Game_2 be the collapsing games from Definition 2.3 for adversary $(\mathcal{A}, \mathcal{B})$.

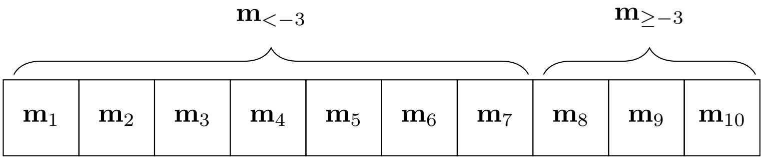


Figure 2.2: Example of 10-block message $\mathbf{m}$, showing the meaning of $\mathbf{m}_{<-3}$ and $\mathbf{m}_{\geqslant -3}$

Let

$$\varepsilon := \mathsf{Adv}^{\mathtt{coll}}_{\mathcal{S}_f}(\mathcal{A},\mathcal{B}) = \Big|\mathbb{P}[b = 1 : \mathsf{Game}_1] - \mathbb{P}[b = 1 : \mathsf{Game}_2]\Big|.$$

In the following we upper bound ε by bounding the success probability in several hybrid games. Each hybrid game represents a partial measurement of the message register. We find the bound for ε by bounding the success probability for distinguishing two consecutive hybrid games.

Recall that $|\mathbf{m}|$ denotes the number of r-bit blocks in $\mathbf{m}$. We will index the r-bit blocks by $\mathbf{m}_i$ with $1 \leqslant i \leqslant |\mathbf{m}|$. Furthermore, let $\mathbf{m}_{-i}$ denote the i-th block from the end (so $\mathbf{m}_{-1}$ is the last message block). Let $\mathbf{m}_{\geqslant -i}$ denote all the blocks in $\mathbf{m}$ starting from $\mathbf{m}_{-i}$ (so $\mathbf{m}_{\geqslant -i}$ consists of the last i blocks of $\mathbf{m}$). Let $\mathbf{m}_{<-i}$ denote the blocks before $\mathbf{m}_{-i}$ and let $\mathbf{m}_{>-i}$ denote the blocks after $\mathbf{m}_{-i}$. This means $\mathbf{m} = \mathbf{m}_{<-i}||\mathbf{m}_{\geqslant -i}$ for $i \leqslant |\mathbf{m}|$. See Figure 2.2 for an example, showing the meaning of these terms.

Let T be a (polynomial) upper bound on the number of blocks for $|\mathbf{m}|$ (so $|\mathbf{m}| \leqslant T$). For functions $\mathrm{partial}_i$ and input_i defined below, we need access to the message length $|\mathbf{m}|$ to determine the output. Notice that the functions are purely classical, hence we can use the length of the messages that are given as input to the functions and behave accordingly to this length. A quantum computer running these functions on qubits, possibly in superposition, will not change this functionality. Note that it can have a superposition of messages with different lengths.

We first define the $\mathrm{partial}_i$ function below: this function basically takes a snapshot of the processing of $\mathbf{m}$ in the sponge. For $i = -1$, all message blocks have been processed and the output of $\mathrm{partial}_i$ (the first entry) is the output of $\mathcal{S}_f$ (and recall this is fixed to $\ell = r$ bits). For $0 \leqslant i < |\mathbf{m}|$, the first entry of $\mathrm{partial}_i$ is the exact input to the $(m - i)$'th application of the internal function f. The second entry of $\mathrm{partial}_i$ is the message block that will be XORed with the output of the $(m-i)$'th application of the internal function f. Hence, this is nothing (i.e. $\perp$) when $i \leqslant 0$ or $i \geqslant |\mathbf{m}|$ and is the next message block to be processed by $\mathcal{J}_f$ otherwise, i.e. $\mathbf{m}_{-i}$ for $0 < i < |\mathbf{m}|$. The third entry of $\mathrm{partial}_i$ contains all message blocks to be processed by the sponge, except for the next block that will be XORed with the output of the $(m - i)$'th application of the internal function f (as this is the second entry of $\mathrm{partial}_i$). Hence, this will be $\perp$ for $i \leqslant 1$ and $\mathbf{m}_{>-i}$ for $i \geqslant |\mathbf{m}|$. Note that $\mathrm{partial}_i(\mathbf{m})$ always contains enough information to compute $\mathcal{J}_f(\mathbf{m})$ and therefore also $\mathcal{S}_f(\mathbf{m})$. Recall that $\mathcal{J}_f(\lambda) = 0^{r+c}$ for the empty world λ, hence $\mathcal{J}_f(\mathbf{m}_{-|\mathbf{m}|}) = \mathcal{J}_f(\lambda) = 0^{r+c}$.

To move from $\mathrm{partial}_0$ to $\mathrm{partial}_{-1}$ requires to apply $\overline{f}$ to the first element of $\mathrm{partial}_0$, and the output will be the first element of $\mathrm{partial}_{-1}$ which is the output of the sponge $\mathcal{S}_f$ when the output length is fixed to $\ell = r$ bits. To move from $\mathrm{partial}_{i+1}$ to $\mathrm{partial}_i$ for $0 \leqslant i < |\mathbf{m}|$ requires applying step_f to the first element of $\mathrm{partial}_{i+1}$ and rearranging the second and last elements of partial_{i+1} into the correct elements for partial_i (i.e. partial_{i+1} has all necessary information to construct partial_i).

Hence, for a message $\mathbf{m}$ and $-1 \leqslant i \leqslant T$, we define:

$$\text{partial}_i(\mathbf{m}) := \begin{cases} (\mathcal{S}_f(\mathbf{m}), \bot, \bot) & (\text{if } i = -1) \\ (\mathcal{I}_f(\mathbf{m}_{<-1}) \oplus (\mathbf{m}_{-1} \| 0^c), \bot, \bot) & (\text{if } i = 0) \\ (\mathcal{I}_f(\mathbf{m}_{<-(i+1)}) \oplus (\mathbf{m}_{-(i+1)} \| 0^c), \mathbf{m}_{-i}, \mathbf{m}_{>-i}) & (\text{if } 0 < i < |\mathbf{m}|) \\ (\bot, \bot, \mathbf{m}) & (\text{if } |\mathbf{m}| \leqslant i) \end{cases}$$

We need one more function $\text{input}_i(\mathbf{m})$, which is basically the first two elements of partial_i, and thus contains the necessary information to compute the first element in partial_{i-1} for $0 \leqslant i \leqslant |\mathbf{m}|$:

$$\text{input}_i(\mathbf{m}) := \begin{cases} (\mathcal{I}_f(\mathbf{m}_{<-1}) \oplus (\mathbf{m}_{-1} \| 0^c), \bot) & (\text{if } i = 0) \\ (\mathcal{I}_f(\mathbf{m}_{<-(i+1)}) \oplus (\mathbf{m}_{-(i+1)} \| 0^c), \mathbf{m}_{-i}) & (\text{if } 0 < i < |\mathbf{m}|) \\ (\bot, \bot) & (\text{if } |\mathbf{m}| \leqslant i) \end{cases}$$

The idea is that $\text{input}_i(\mathbf{m})$ will be used as input to collapsing functions (in particular $\overline{f}$ and step_f, where step_f as defined in Lemma 2.8), which will then allow us to (basically) map the elements in $\text{partial}_i(\mathbf{m})$ to the elements in $\text{partial}_{i+1}(\mathbf{m})$. Note also that the first entry of both partial_i and input_i is not the state *after* an application of internal function f, but rather the state XORed with message block m_i. Hence, this is the state *before* an application of f (or rather the input to $\overline{f}$ or the first entry of the input to step_f as we will see later).

We will now make this formal by deriving the following facts for input_i and partial_i from their definition:

Fact 1. $\text{input}_{|\boldsymbol{m}|-1}(\boldsymbol{m}) = (\boldsymbol{m}_1 \| 0^c, \boldsymbol{m}_2)$

From $\mathcal{I}_f(\mathbf{m}_{<-|\mathbf{m}|}) = \mathcal{I}_f(\lambda) = 0^c$, the fact follows straightforward. What this Fact will mean to our proof, is that in the ($|\mathbf{m}| - 1$)'th hybrid argument (defined below), we will measure the last two message blocks at the same time.

We denote the output entries of partial_i by $\text{partial}_i(\mathbf{m}) = (x_i, y_i, z_i)$, where $x_i \in \{0,1\}^{r+c}$, $y_i \in \{0,1\}^r$ and $z_i \subseteq \mathbf{m}$ is a set of message blocks. We explicitly write $\bot$ for the entry if applicable by definition of partial_i, input_i (and also use $\bot$ when $z_i = \emptyset$).

Fact 2. *Let* $\text{partial}_{-1}(\boldsymbol{m}) = (x_{-1}, \bot, \bot)$ *and* $\text{partial}_0(\boldsymbol{m}) = (x_0, \bot, \bot)$. *Then* $\overline{f}(\text{input}_0(\boldsymbol{m})) = \overline{f}(x_0) = \mathcal{S}_f(\boldsymbol{m}) = x_{-1}$

Note that we slightly abused notation here, as we ignored the second element $\bot$ of $\text{input}_0(\mathbf{m})$, i.e., $\overline{f}$ (and below also f) only acts on the first element of $\text{input}_0(\mathbf{m})$. By definition of partial_{-1}, $\mathbf{m}$ must be a message such that the output of the sponge is $\mathcal{S}_f(\mathbf{m}) = x_{-1}$. Also, $\text{input}_0(\mathbf{m}) \in \{0,1\}^{r+c}$ must be such that $f(\text{input}_0(\mathbf{m})) = \mathcal{I}_f(\mathbf{m})$. But this means that if we would apply $\overline{f}$ to $\text{input}_0(\mathbf{m})$, we get the sponge outcome: $\overline{f}(\text{input}_0(\mathbf{m})) = x_{-1} = \mathcal{S}_f(\mathbf{m})$. Lastly, as also mentioned above, we defined $\text{input}_0(\mathbf{m})$ in such a way that $\text{input}_0(\mathbf{m}) = x_0$.

Using $\text{step}_f(x, y) := f(x) \oplus (y \| 0^c) = (\overline{f(x)} \oplus y, \widehat{f(x)})$ as defined in Lemma 2.8, we now have the following facts for partial_i.

Fact 3. *For* $0 \leqslant i < |\boldsymbol{m}| - 1$*, let* $\mathrm{partial}_i(\boldsymbol{m}) = (x_i, y_i, z_i)$*. Then* $\mathrm{step}_f(\mathrm{input}_{i+1}(\boldsymbol{m})) = \mathrm{step}_f(x_{i+1}, y_{i+1}) = x_i$

Since $1 \leqslant i+1 < |\mathbf{m}|$, we have

$$\begin{aligned}\mathrm{step}_f(\mathrm{input}_{i+1}(\mathbf{m})) &= \mathrm{step}_f(\mathcal{I}_f(\mathbf{m}_{<-(i+2)}) \oplus (\mathbf{m}_{-(i+2)}\|0^c), \mathbf{m}_{-(i+1)}) \\ &= \mathrm{step}_f(x_{i+1}, y_{i+1}) \\ &= f\left(\mathcal{I}_f(\mathbf{m}_{<-(i+2)}) \oplus (\mathbf{m}_{-(i+2)}\|0^c)\right) \oplus (\mathbf{m}_{-(i+1)}\|0^c) \\ &= \mathcal{I}_f(\mathbf{m}_{<-(i+1)}) \oplus (\mathbf{m}_{-(i+1)}\|0^c) = x_i\end{aligned}$$

Fact 4. *From* $(\mathrm{partial}_i(\boldsymbol{m}), \mathrm{input}_{i+1}(\boldsymbol{m}))$*, one can compute* $\mathrm{partial}_{i+1}(\boldsymbol{m})$ *and vice versa.*

Note that element z_i is part of the elements in z_{i+1} and element y_{i+1} combined with the elements in z_i equals the elements in z_{i+1}. Hence, the missing element for $\mathrm{partial}_i$ is contained in $\mathrm{input}_{i+1}(\mathbf{m})$. This means that applying $\mathcal{M}_{\mathrm{partial}_i}$ and $\mathcal{M}_{\mathrm{input}_{i+1}}$ to message register M has the same effect as applying $\mathcal{M}_{\mathrm{partial}_{i+1}}$: they both measure the same elements.

So basically, the function partial interpolates between knowledge of only $\mathcal{S}_f(\mathbf{m})$ (case $i = -1$), and full knowledge of $\mathbf{m}$ (case $i = T-1$).

Recall that $(\mathcal{A}, \mathcal{B})$ is any adversary for the collapsing games of $\mathcal{S}_f$, i.e.

$$\mathsf{Adv}^{\mathrm{coll}}_{\mathcal{S}_f}(\mathcal{A}, \mathcal{B}) = \left|\mathbb{P}[b = 1 : \mathsf{Game}_1] - \mathbb{P}[b = 1 : \mathsf{Game}_2]\right|.$$

We define the following hybrid games, for $i = -1, \ldots, T-1$:

$$\mathsf{Hyb}_i : (S, M, h) \leftarrow \mathcal{A}(1^n) \tag{2.1}$$

$$(x_i, y_i, z_i) \leftarrow \mathcal{M}_{\mathrm{partial}_i}(M) \tag{2.2}$$

$$b \leftarrow \mathcal{B}(S, M) \tag{2.3}$$

Here, $\mathcal{M}_{\mathrm{partial}_i}$ is M_f with $f(\mathbf{m}) = \mathrm{partial}_i(\mathbf{m})$, as defined in Section 2.2. We make explicit that the hybrid games are defined for adversary $(\mathcal{A}, \mathcal{B})$, by denoting this with $\mathsf{Hyb}_i^{\mathcal{A},\mathcal{B}}$. By construction of function $\mathrm{partial}_i$, we have

$$\mathbb{P}[b = 1 : \mathsf{Hyb}_{-1}^{\mathcal{A},\mathcal{B}}] = \mathbb{P}[b = 1 : \mathsf{Game}_2]$$

since in $\mathsf{Hyb}_{-1}^{\mathcal{A},\mathcal{B}}$, the measurement $(x_{-1}, \perp, \perp) \leftarrow \mathcal{M}_{\mathrm{partial}_{-1}}(M)$ does not have any influence: in this case, x_{-1} has a determined outcome, namely $x_{-1} = \mathcal{S}_f(\mathbf{m}) = h$ for all $\mathbf{m} \in M$.

We also have

$$\mathbb{P}[b = 1 : \mathsf{Hyb}_{T-1}^{\mathcal{A},\mathcal{B}}] = \mathbb{P}[b = 1 : \mathsf{Game}_1]$$

since $(\mathbf{m}_1\|0^c, \mathbf{m}_2, \mathbf{m}_{>2}) \leftarrow \mathcal{M}_{\mathrm{partial}_{T-1}}(M)$ fully measures register M in the computational basis[5], which would also be the case in Game_1. So this means

$$\varepsilon = \left|\mathbb{P}[b = 1 : \mathsf{Hyb}_{T-1}^{\mathcal{A},\mathcal{B}}] - \mathbb{P}[b = 1 : \mathsf{Hyb}_{-1}^{\mathcal{A},\mathcal{B}}]\right| \tag{2.4}$$

[5] This is the measurement outcome for messages of length T, for shorter messages the whole measured message is in the last register or last two registers.

A standard hybrid argument shows that we can now bound ε by bounding the success probability of $(\mathcal{A}, \mathcal{B})$ in distinguishing any two consecutive hybrids. Towards bounding these distinguishing advantages, we now define oracle machines $(\mathcal{V}_i^{\mathcal{A}}, \mathcal{W}^{\mathcal{B}})$ for $i = -1, \ldots, T-1$ that use $(\mathcal{A}, \mathcal{B})$'s distinguishing advantage to either win some collapsing game or find a preimage of zero. Let the output of $\mathcal{A}$ be given by $(S_{\mathcal{A}}, M_{\mathcal{A}}, c_{\mathcal{A}})$ (i.e. these are valid quantum registers for the collapsing game for $\mathcal{S}_f$) and let the output bit of $\mathcal{B}$ be $b_{\mathcal{B}}$.

Let U_{input_i} refer to the unitary transformation $|\mathbf{x}\rangle\,|\mathbf{y}\rangle \to |\mathbf{x}\rangle\,|\mathbf{y} \oplus \text{input}_i(\mathbf{x})\rangle$. We define the oracle machines $(\mathcal{V}_i^{\mathcal{A}}, \mathcal{W}^{\mathcal{B}})$ for $i = -1, \ldots, T-1$ in Algorithm 2.1 and Algorithm 2.2.

Algorithm 2.1 Algorithm $\mathcal{V}_i^{\mathcal{A}}$ with $-1 \leqslant i \leqslant T-1$

Input: Security parameter n, index $-1 \leqslant i \leqslant T-1$.
Output: Valid quantum registers (S, M, h) or zero-preimage x_{zpre}.

```
$(S_{\mathcal{A}}, M_{\mathcal{A}}, h_{\mathcal{A}}) \leftarrow \mathcal{A}(1^n)$
$(x_i, y_i, z_i) \leftarrow \mathcal{M}_{\text{partial}_i}(M_{\mathcal{A}})$
If $x_i = \perp$, abort
If $\widehat{x_i} = 0^c$:
    $(x_{i+1}, y_{i+1}) \leftarrow \mathcal{M}_{\text{input}_{i+1}}(M_{\mathcal{A}})$
    If $x_{i+1} = \perp$, abort
    Else output $x_{zpre} := x_{i+1}$
Initialize $M$ with $|0^{r+c}\rangle|0^r\rangle$
Apply $U_{\text{input}_{i+1}}$ to $M_{\mathcal{A}}, M$.
Set $h := x_i$
Let $S = \{S_{\mathcal{A}}, M_{\mathcal{A}}, h = x_i, i\}$
Return $(S, M, h)$
```

Algorithm 2.2 Algorithm $\mathcal{W}^{\mathcal{B}}$

Input: Security parameter n, quantum registers (S, M)
Output: Bit b.

```
Unpack $S$, giving $S_{\mathcal{A}}, M_{\mathcal{A}}, h = x_i, i$
Apply $U_{\text{input}_{i+1}}$ to $M_{\mathcal{A}}, M$
Run $b_{\mathcal{B}} \leftarrow \mathcal{B}(S_{\mathcal{A}}, M_{\mathcal{A}})$
Return $b_{\mathcal{B}}$.
```

Notice that adversary $(\mathcal{V}_i^{\mathcal{A}}, \mathcal{W}^{\mathcal{B}})$, conditioned on $\widehat{x_i} \neq 0^c$ and $x_i \neq \perp$, outputs quantum registers (S, M, h). Register S contains among other $S_{\mathcal{A}}$ (which is possibly needed by $\mathcal{B}$) plus some additional helper-information for $\mathcal{W}^{\mathcal{B}}$. Message register M is either measured (as in $\mathsf{Game}_1^{\mathcal{V}_i^{\mathcal{A}}, \mathcal{W}^{\mathcal{B}}}$) or not (as in $\mathsf{Game}_2^{\mathcal{V}_i^{\mathcal{A}}, \mathcal{W}^{\mathcal{B}}}$). Finally, register h is a fixed classical output value of some function. The claims below contain proofs for the validity of the quantum registers (S, M, h).

We now have the following claims for adversary $(\mathcal{V}_i^{\mathcal{A}}, \mathcal{W}^{\mathcal{B}})$. We begin with showing that the advantage of $(\mathcal{A}, \mathcal{B})$ in distinguishing Hyb_{-1} from Hyb_0 is bounded by the collapsing advantage of any efficient quantum adversary against $\bar{f}$. This is done in the first four claims using the properties of $(\mathcal{V}_{-1}^{\mathcal{A}}, \mathcal{W}^{\mathcal{B}})$.

Claim 1. $(\mathcal{V}^{\mathcal{A}}_{-1}, \mathcal{W}^{\mathcal{B}})$ *is a valid adversary for* $\bar{f}$ *if* $(\mathcal{A}, \mathcal{B})$ *is a valid adversary for* $\mathcal{S}_f$.

We show this claim: after the measurement $(x_{-1}, \bot, \bot) \leftarrow \mathcal{M}_{\mathsf{partial}_{-1}}(M_{\mathcal{A}})$, we have that $M_{\mathcal{A}}$ contains a superposition of messages $|\mathbf{m}\rangle$ with $\mathsf{partial}_{-1}(\mathbf{m}) = (x_{-1}, \bot, \bot)$. So by Fact 2, $M_{\mathcal{A}}$ contains a superposition of messages $|\mathbf{m}\rangle$ such that $\bar{f}(\mathsf{input}_0(\mathbf{m})) = x_{-1} = h_{\mathcal{A}} = \mathcal{S}_f(\mathbf{m})$ as $\mathcal{A}$ is valid adversary for $\mathcal{S}_f$. Now algorithm $\mathcal{V}^{\mathcal{A}}_{-1}$ initializes M with $|0^{r+c}\rangle|0^r\rangle$ and applies U_{input_0} to $M_{\mathcal{A}}, M$ (this means ignoring the second entry of input_0 which is $\bot$ here). Thus after that, M is in a superposition of messages $|m\rangle$ such that $\bar{f}(m) = x_{-1} = h_{\mathcal{A}} =: h$. Concluding, $(\mathcal{V}^{\mathcal{A}}_{-1}, \mathcal{W}^{\mathcal{B}})$ is a valid adversary for $\bar{f}$ with output registers (S, M, h).

Let $\mathsf{Game}_1^{\mathcal{V}^{\mathcal{A}}_{-1}, \mathcal{W}^{\mathcal{B}}}$ denote Game_1 (i.e. with measurement) from Definition 2.3, but with adversary $(\mathcal{V}^{\mathcal{A}}_{-1}, \mathcal{W}^{\mathcal{B}})$ and function $\bar{f}$, analogous for $\mathsf{Game}_2^{\mathcal{V}^{\mathcal{A}}_{-1}, \mathcal{W}^{\mathcal{B}}}$ (i.e. without measurement).

Claim 2. $\mathbb{P}[b = 1 : \mathsf{Game}_2^{\mathcal{V}^{\mathcal{A}}_{-1}, \mathcal{W}^{\mathcal{B}}}] = \mathbb{P}[b = 1 : \mathsf{Hyb}_{-1}^{\mathcal{A},\mathcal{B}}]$.

We show this claim: in $\mathsf{Game}_2^{\mathcal{V}^{\mathcal{A}}_{i}, \mathcal{W}^{\mathcal{B}}}$ no measurement occurs between the invocation of U_{input_0} by $\mathcal{V}^{\mathcal{A}}_{-1}$ and the invocation of U_{input_0} by $\mathcal{W}^{\mathcal{B}}$.Thus, these two invocations cancel each other out. So only the invocations of $\mathcal{A}, \mathcal{M}_{\mathsf{partial}_{-1}}(M_{\mathcal{A}})$ and $\mathcal{B}$ remain, where $\mathcal{M}_{\mathsf{partial}_{-1}}(M_{\mathcal{A}})$ has no effect as the outcome is $(h_{\mathcal{A}}, \bot, \bot)$. This is exactly $\mathsf{Hyb}_{-1}^{\mathcal{A},\mathcal{B}}$.

Claim 3. $\mathbb{P}[b = 1 : \mathsf{Game}_1^{\mathcal{V}^{\mathcal{A}}_{-1}, \mathcal{W}^{\mathcal{B}}}] = \mathbb{P}[b = 1 : \mathsf{Hyb}_{0}^{\mathcal{A},\mathcal{B}}]$.

In $\mathsf{Game}_1^{\mathcal{V}^{\mathcal{A}}_{-1}, \mathcal{W}^{\mathcal{B}}}$, M is initialized with $|0^{r+c}\rangle|0^r\rangle$. U_{input_0} is applied to $M_{\mathcal{A}}, M$. M is measured in the computational basis (with outcome m). U_{input_0} is applied to $M_{\mathcal{A}}, M$. Then M is discarded. This is equivalent to executing $m \leftarrow \mathcal{M}_{\mathsf{input}_0}(M_{\mathcal{A}})$. Thus, in $\mathsf{Game}_1^{\mathcal{V}^{\mathcal{A}}_{-1}, \mathcal{W}^{\mathcal{B}}}$, both $(x_{-1}, y_{-1}, z_{-1}) \leftarrow \mathcal{M}_{\mathsf{partial}_{-1}}(M_{\mathcal{A}})$ and $m \leftarrow \mathcal{M}_{\mathsf{input}_0}(M_{\mathcal{A}})$ were executed. By Fact 4, this is equivalent to executing $(x_0, y_0, z_0) \leftarrow \mathcal{M}_{\mathsf{partial}_0}(M_{\mathcal{A}})$. This is exactly $\mathsf{Hyb}_0^{\mathcal{A},\mathcal{B}}$.

From Claim 2 and Claim 3 and by assumption of this Lemma, we get:

Claim 4. $\left|\mathbb{P}[b = 1 : \mathsf{Hyb}_0^{\mathcal{A},\mathcal{B}}] - \mathbb{P}[b = 1 : \mathsf{Hyb}_{-1}^{\mathcal{A},\mathcal{B}}]\right| = \mathsf{Adv}_{\bar{f}}^{\mathrm{coll}}(\mathcal{V}^{\mathcal{A}}_{-1}, \mathcal{W}^{\mathcal{B}}) \leqslant \bar{\varepsilon}$*, where* $\bar{\varepsilon}$ *is the collapsing advantage of* $\bar{f}$.

For the remaining hybrids, we will bound the success probability by using the collapsingness of the step function and the zero-preimage resistance of $\hat{f}$. Let

$$\mu_i := \left|\mathbb{P}[b = 1 : \mathsf{Hyb}_{i+1}^{\mathcal{A},\mathcal{B}}] - \mathbb{P}[b = 1 : \mathsf{Hyb}_{i}^{\mathcal{A},\mathcal{B}}]\right|$$

for $i \geqslant 0$ be the advantage of adversary $(\mathcal{A}, \mathcal{B})$ in distinguishing games Hyb_i and Hyb_{i+1}. Next, we will upper bound this advantage for $0 \leqslant i < T$. This is done analyzing the success probability of $(\mathcal{V}^{\mathcal{A}}_i, \mathcal{W}^{\mathcal{B}})$ for $0 \leqslant i < T$. Note that whenever the case $x_i = \bot$ in line 3 of $\mathcal{V}^{\mathcal{A}}_i$ occurs, the two hybrids $\mathsf{Hyb}_i^{\mathcal{A},\mathcal{B}}$ and $\mathsf{Hyb}_{i+1}^{\mathcal{A},\mathcal{B}}$ are perfectly indistinguishable as applying $\mathcal{M}_{\mathsf{partial}_{i+1}}(M_{\mathcal{A}})$ has no effect at all. Hence, these cases cannot contribute to $(\mathcal{A}, \mathcal{B})$'s success probability and we can abort. Therefore, the distinguishing advantage μ_i must come from cases where $x_i \neq \bot$. We can split these cases into two depending on the value of $\widehat{x_i}$. In the following, we denote by μ'_i the advantage of $(\mathcal{A}, \mathcal{B})$ in distinguishing Hyb_i and Hyb_{i+1} conditioned on $\widehat{x_i} = 0^c$ and by μ''_i the distinguishing advantage conditioned on $\widehat{x_i} \neq 0^c$. These two advantages are related by the following claim.

Claim 5. *There exists a* $p_i \in [0, 1]$ *such that*

$$\mu_i = p_i \mu'_i + (1 - p_i)\mu''_i.$$

As the conditioning is on a binary event such a p_i must exist.

Now, we first develop a bound on μ''_i in the next four claims.

Claim 6. $(\mathcal{V}^{\mathcal{A}}_i, \mathcal{W}^{\mathcal{B}})$ *is a valid adversary for function* step_f *for* $0 \leqslant i \leqslant T - 1$*, conditioned on* $\widehat{x_i} \neq 0^c$*.*

After measurement $(x_i, y_i, z_i) \leftarrow \mathcal{M}_{\mathsf{partial}_i}(M_{\mathcal{A}})$, we have that $M_{\mathcal{A}}$ contains a superposition of messages $|\mathbf{m}\rangle$ with $\mathsf{partial}_i(\mathbf{m}) = (x_i, y_i, z_i)$. Per assumption we got $\widehat{x_i} \neq 0^c$. So by Fact 3, $M_{\mathcal{A}}$ contains a superposition of messages $|\mathbf{m}\rangle$ with the property $\mathsf{step}_f(\mathsf{input}_{i+1}(\mathbf{m})) = x_i =: c$. Now $\mathcal{V}^{\mathcal{A}}_i$ initializes M with $|0^{r+c}\rangle|0^r\rangle$ and applies $U_{\mathsf{input}_{i+1}}$ to $M_{\mathcal{A}}, M$. Thus after that, M is in a superposition of $|x, y\rangle$ such that $\mathsf{step}_f(x, y) = h$. This means $(\mathcal{V}^{\mathcal{A}}_i, \mathcal{W}^{\mathcal{B}})$ is a valid adversary for step_f.

For $0 \leqslant i \leqslant T - 1$, let $\mathsf{Game}_1^{\mathcal{V}^{\mathcal{A}}_i, \mathcal{W}^{\mathcal{B}}}$ denote Game_1 (i.e. with measurement) of Definition 2.3, but with adversary $(\mathcal{V}^{\mathcal{A}}_i, \mathcal{W}^{\mathcal{B}})$ and function step_f. Analogous we define $\mathsf{Game}_2^{\mathcal{V}^{\mathcal{A}}_i, \mathcal{W}^{\mathcal{B}}}$ as the game with measurement.

Claim 7. $\mathbb{P}[b = 1 : \mathsf{Game}_2^{\mathcal{V}^{\mathcal{A}}_i, \mathcal{W}^{\mathcal{B}}}] = \mathbb{P}[b = 1 : \mathsf{Hyb}_i^{\mathcal{A},\mathcal{B}}]$*, conditioned on* $\widehat{x_i} \neq 0^c$*.*

In $\mathsf{Game}_2^{\mathcal{V}^{\mathcal{A}}_i, \mathcal{W}^{\mathcal{B}}}$, no measurement occurs between the two invocations of $U_{\mathsf{input}_{i+1}}$ by $\mathcal{V}^{\mathcal{A}}_i$ and $\mathcal{W}^{\mathcal{B}}$, so these two invocations cancel out. Thus only the invocations of $\mathcal{A}$, $\mathcal{M}_{\mathsf{partial}_i}$ and $\mathcal{B}$ remain. This is exactly $\mathsf{Hyb}_i^{\mathcal{A},\mathcal{B}}$.

Claim 8. $\mathbb{P}[\mathsf{Game}_1^{\mathcal{V}^{\mathcal{A}}_i, \mathcal{W}^{\mathcal{B}}}] = \mathbb{P}[b = 1 : \mathsf{Hyb}_{i+1}^{\mathcal{A},\mathcal{B}}]$*, conditioned on* $\widehat{x_i} \neq 0^c$*.*

Note that in $\mathsf{Game}_1^{\mathcal{V}^{\mathcal{A}}_i, \mathcal{W}^{\mathcal{B}}}$, after the measurement $\mathcal{M}_{\mathsf{partial}_i}$, on the registers $M_{\mathcal{A}}, M$ we have the following sequence of operations. As $\widehat{x_i} \neq 0^c$ per assumption, M is initialized with $|0^{r+c}\rangle|0^r\rangle$. $U_{\mathsf{input}_{i+1}}$ is applied to $M_{\mathcal{A}}, M$. M is measured in the computational basis (with outcome m). $U_{\mathsf{input}_{i+1}}$ is applied to $M_{\mathcal{A}}, M$. M is discarded. This is equivalent to executing $m \leftarrow \mathcal{M}_{\mathsf{input}_{i+1}}(M_{\mathcal{A}})$.

So $\mathcal{M}_{\mathsf{partial}_i}(M_{\mathcal{A}})$ and $\mathcal{M}_{\mathsf{input}_{i+1}}(M_{\mathcal{A}})$ were executed. By Fact 4, this is the same as executing $\mathcal{M}_{\mathsf{partial}_{i+1}}(M_{\mathcal{A}})$. This means $\mathsf{Game}_1^{\mathcal{V}^{\mathcal{A}}_i, \mathcal{W}^{\mathcal{B}}}$ is equivalent to $\mathsf{Hyb}_{i+1}^{\mathcal{A},\mathcal{B}}$, which is the claim.

From Claims 6, 7, and 8, Lemma 2.8 and the assumptions of this Lemma, we get:

Claim 9. $\mu''_i \leqslant \mathsf{Adv}^{\mathtt{coll}}_{\mathsf{step}_f}(\mathcal{V}^{\mathcal{A}}_i, \mathcal{W}^{\mathcal{B}}) \leqslant \widehat{\varepsilon}$*, where* $\widehat{\varepsilon}$ *is the collapsing advantage of* $\widehat{f}$*.*

Before we can put things together, we still have to upper bound μ'_i. This is done in the next claim. To upper bound μ'_i we have to analyze the case where $\widehat{x_i} = 0^c$. From the definition of input_{i+1} we know that applying $U_{\mathsf{input}_{i+1}}$ to $M_{\mathcal{A}}, |0^{r+c}\rangle|0^r\rangle$ leads to a superposition of messages in the second register of the following form: Every message in the superposition is either $\perp$ (if the message only consisted of i blocks) or some m_j such that $\widehat{f}(m_j) = 0^c$, i.e., a preimage of zero.

Note that there can only be at most a single message in $M_{\mathcal{A}}$ in the case that $x_{i+1} = \perp$: the adversary already applied $\mathcal{M}_{\text{partial}_i}(M_{\mathcal{A}})$ and thereby measured the whole message m_s which consisted of i blocks (i.e. the fact that $\widehat{x_i} = 0^c$ stems from the initial internal state of 0^c). Hence, $\mathsf{Hyb}_i^{\mathcal{A},\mathcal{B}}$ and $\mathsf{Hyb}_{i+1}^{\mathcal{A},\mathcal{B}}$ are perfectly indistinguishable as applying $\mathcal{M}_{\text{partial}_{i+1}}(M_{\mathcal{A}})$ has no effect at all.

It remains to show that executing $\mathcal{M}_{\text{input}_{i+1}}(M_{\mathcal{A}})$ outputs $\perp$ and not one of the m_j with $\widehat{f}(m_j) = 0^c$. This is the part where we need the zero-preimage resistance of $\widehat{f}$.

Claim 10. $\mu_i' \leqslant \sqrt{\varepsilon_z}$.

Conditioned on $\widehat{x_i} = 0$, we can see the success probability μ_i' of the adversary $(\mathcal{A}, \mathcal{B})$ in distinguishing $\mathsf{Hyb}_i^{\mathcal{A},\mathcal{B}}$ from $\mathsf{Hyb}_{i+1}^{\mathcal{A},\mathcal{B}}$ as the quantum state-discrimination problem between the two states ρ_0 and ρ_1, where ρ_0 is the superposition of messages $|\mathbf{m}\rangle$ in $M_{\mathcal{A}}$ after measuring $\mathcal{M}_{\text{partial}_i}(M_{\mathcal{A}})$ and ρ_1 is the superposition in $M_{\mathcal{A}}$ after measuring $\mathcal{M}_{\text{partial}_{i+1}}(M_{\mathcal{A}})$. But note that by Fact 4, the state ρ_1 is exactly the state after measuring $\mathcal{M}_{\text{partial}_i}(M_{\mathcal{A}})$ and $\mathcal{M}_{\text{input}_{i+1}}(M_{\mathcal{A}})$ conditioned on $\widehat{x_i} = 0$, i.e. the operations of Algorithm $\mathcal{V}_i^{\mathcal{A}}$. The probability that $\mathcal{V}_i^{\mathcal{A}}$ outputs a zero-preimage of $\widehat{f}$ is bounded by ε_z by assumption. Hence, the output of Algorithm $\mathcal{V}_i^{\mathcal{A}}$, conditioned on $\widehat{x_i} = 0^c$, will be "1"(i.e. abort) with probability $\mathbb{P}[(\perp, \perp) \leftarrow \mathcal{M}_{\text{input}_{i+1}}(M_{\mathcal{A}})] \geqslant 1 - \varepsilon_z$. By Lemma 2.12, the trace distance between ρ_0 and ρ_1 is therefore bounded by $\sqrt{\varepsilon_z}$, hence the distinguishing advantage in the two games can differ at most the trace distance between ρ_0 and ρ_1:

$$\mu_i' \leqslant \sqrt{\varepsilon_z}$$

Putting Claims 9 and 10 together with Claim 5 we obtain

Claim 11. *There exists a* $p_i \in [0, 1]$ *such that*

$$\mu_i \leqslant p_i \sqrt{\varepsilon_z} + (1 - p_i)\widehat{\varepsilon} \leqslant \sqrt{\varepsilon_z} + \widehat{\varepsilon}.$$

Thus, using Claims 4 and 11, we get:

$$\begin{aligned}
\varepsilon &\overset{(2.4)}{=} \left| \mathbb{P}[b = 1 : \mathsf{Hyb}_{-1}^{\mathcal{A},\mathcal{B}}] - \mathbb{P}[b = 1 : \mathsf{Hyb}_{T-1}^{\mathcal{A},\mathcal{B}}] \right| \\
&= \left| \sum_{i=-1}^{T-2} \mathbb{P}[b = 1 : \mathsf{Hyb}_i^{\mathcal{A},\mathcal{B}}] - \sum_{i=-1}^{T-2} \mathbb{P}[b = 1 : \mathsf{Hyb}_{i+1}^{\mathcal{A},\mathcal{B}}] \right| \\
&\leqslant \left| \mathbb{P}[b = 1 : \mathsf{Hyb}_{-1}^{\mathcal{A},\mathcal{B}}] - \mathbb{P}[b = 1 : \mathsf{Hyb}_0^{\mathcal{A},\mathcal{B}}] \right| + \sum_{i=0}^{T-2} \left| \mathbb{P}[b = 1 : \mathsf{Hyb}_i^{\mathcal{A},\mathcal{B}}] - \mathbb{P}[b = 1 : \mathsf{Hyb}_{i+1}^{\mathcal{A},\mathcal{B}}] \right| \\
&\leqslant \overline{\varepsilon} + (T - 1)(\sqrt{\varepsilon_z} + \widehat{\varepsilon})
\end{aligned}$$

This is exactly the claimed bound and thereby concludes the proof. □

2.5 — Comparison with published version

Concurrent to the work that this chapter is based on [CGHS17], there was another publication [Unr17] on the same subject. After the decision to merge the two works, many changes were applied that improved readability and even some results. Despite of this, the chapter is still largely based on the work from the unpublished version [CGHS17], as the author of this thesis had a lot more influence on that work. The main differences between this chapter (in the next comparison referred to as SP1) and the published version [CGH$^+$18] (referred to as SP2) are:

- Improved readability and simplified proof. The published version SP2 has been greatly improved on readability. Theorem 2.13 in SP1 is quite technical, and in SP2 a better idea of the proof is sketched. For example, in SP1, we directly prove the collapsing property of the sponge construction $\mathcal{S}$ for the absorb and squeeze phase. However, one can view the sponge construction as the composition of the padding function pad, the absorb function $\mathcal{S}^{AB}$ and the squeeze function $\mathcal{S}^{SQ}$, i.e. $\mathcal{S} = \mathcal{S}^{SQ} \circ \mathcal{S}^{AB} \circ \text{pad}$. In SP2, this breakdown is used by showing that all components are collapsing, i.e. showing the collapsing property for the pad, $\mathcal{S}^{AB}$ and $\mathcal{S}^{SQ}$. This fact and earlier results [Unr16b] prove that the composition of these three functions, i.e. the full sponge function, is collapsing. A second improvement in SP2 is the removal of functions such as step, partial and input by factoring the problem differently: for each game, it is simply stated which register is measured and at what cost. Instead of linking the proof together using the step function, only the internal state registers, i.e. $\widehat{s}_i$ in each step i, are measured. Using the collapsingness of $\widehat{f}$, this implies that all internal registers s_i can be measured with negligible success probability for the adversary to notice. When all internal registers s are measured, a simple lemma shows that this implies that all message registers $\mathbf{m}$ are collapsed to basis states, completing the proof.
- Improved results for collapsing bound. In SP2, the main results are slightly better. This is mainly due to the fact that in SP2, there is just one measurement that measures whether there are preimages of zero in the message register, i.e. messages where the zero-preimage resistance of $\widehat{f}$ breaks. This removes the factor $T - 1$ in front of $\sqrt{\varepsilon_z}$ in the main result of this chapter. As this also removes the conditioning on zero pre-images, the factors p_i also disappear, although this does not influence the tightness of the result.
- More results. In addition to the proof for the collapsingness of sponges, also a direct proof for collision resistance is given in SP2, which is tighter than our proof in Corollary 2.15. In addition, a quantum search algorithm for finding collisions in any random function, including the sponge construction, is given to provide an upper bound.

CHAPTER 3

Oops I did it again

3.1 — Overview

Context. The first post-quantum signature schemes considered for standardization are hash-based Merkle Signature Schemes [MCF19, HBG$^+$18]. These schemes form the most confidence-inspiring post-quantum solution for digital signatures as their security only relies on some mild assumptions about properties of cryptographic hash-functions [HRS16]. Hash-based signature schemes can be split into stateful (e.g. [Mer89, BGD$^+$06, BDK$^+$07, HRB13] and two recent RFCs [HRS16, LM95]) and stateless (e.g. [BHH$^+$15, ABD$^+$17]) proposals. In this context, statefulness means that the secret key changes after every signature. In case a 'secret key state' is used twice, all security guarantees vanish. In practice it turns out that in many scenarios keeping a state becomes a complicated issue [MKF$^+$16]. However, currently stateful schemes are the ones considered for standardization as these schemes are far more efficient in terms of signature size and signing speed than the stateless alternatives.

The reason these schemes are stateful is that their core building block are so-called one-time signature schemes (OTS). A one-time signature scheme allows to use a key pair to sign a single (arbitrary) message. If a key pair is used to sign a second, different message, no security guarantees are given. The security reductions only apply as long as just a single message is signed. While this is commonly interpreted as the schemes are entirely broken if a key pair is used to sign twice, this is not necessarily the case. It is known that if an adversary has full control of the messages to be signed, the schemes are fully broken after two signatures, i.e. the secret key can be extracted without any effort. However, in practice the OTS causing statefulness are used to sign the digest of an adversarially chosen message. Moreover, in both recent RFC's [MCF19, HBG$^+$18] these message digests are randomized. Hence, the actually signed message (digest) is unpredictable for an adversary.

Taking the message digest into account is one of the crucial steps in the construction of hash-based few-time signature schemes like HORS [RR02] that allow to use a key pair to sign a small number of messages before security drops below the acceptable limit. This opens up the question if classical hash-based OTS are still one-time when we take the message digest into account or if a similar argument applies as for HORS. In practice, this question translates to the question if reuse of a secret key state leads to a hard fail or if one is "only" facing graceful degradation of security.

Summary. In this chapter we analyze the security of hash-based one-time signature-schemes under different kinds of two-message-attacks. We carry out the analysis for the most prominent proposals: Lamport's scheme [Lam79], the optimized version of Lamport's scheme [Mer89], and the Winternitz OTS (WOTS) [Mer89]. It turns out that actually, all three schemes are still secure under two-message attacks if we take into account that a message digest is signed, at least if we consider the attacks asymptotically (see Table 3.1).

The general working of these schemes is as follows. We consider the case that a message $\mathbf{m}$ is first compressed using a cryptographic hash function H to obtain a fixed length message digest $\mathbf{m}^* = H(\mathbf{m})$, as well as the case that the message is under full control of the adversary (in this case it means H is the identity function and thus $\mathbf{m}^* = \mathbf{m}$). A mapping function G is used to map $\mathbf{m}^*$ to some index set $B = (B_1, \ldots, B_\ell) = G(\mathbf{m}^*)$. Finally, secret values indicated by the index set B are published as signature. Generally, the secret values are the preimages of public key values under a cryptographic hash function F. Verification works by applying F to the given values and comparing the results to the respective public key values. In case of WOTS secrets are arranged in hash chains. The end nodes of the chains are the public key values. In this case, there exists some dependency, i.e., if a value from a chain is part of the signature, all later values of that chain can be derived applying F.

After seeing two signatures under the same key, there exist two possible ways to forge a signature. First, an adversary can try to find a message that is mapped to an index set which is covered by the union of the index sets of the two seen signatures. In this case, all the required secret values are contained in the two signatures. Second, an adversary can try to compute the secret values not covered by the union of the index sets for a signature from the respective public key values. However, this requires to break one of the security properties of F and would also allow to forge signatures after seeing just the public key. Parameters in practice are chosen such that this is infeasible. Consequently, we just consider the first kind of attacks in this chapter. The possibility and complexity of attacks of this type depends on the properties of hash function H, the message-mapping function G, and possible dependencies of secret values (as in the case of WOTS). In our analysis we focus on the latter two. For H we assume that it behaves like a random oracle. This decision follows the same reasoning as above. Vulnerabilities of H would already allow for forgeries under one-message attacks. For WOTS this implies that the obtained results also apply to the recent variants of WOTS that minimize security assumptions [BDE⁺11, Hül13, HRS16] as the mapping function and the arrangement of secret values for these variants is the same as in the original scheme.

For Lamport's scheme, we obtain exact complexities for two-message attacks. For the optimized Lamport scheme and WOTS analysis becomes extremely complex when looking at the actual mapping functions. This is caused by a checksum which is added to the message. This checksum introduces a lot of dependencies between probabilities, eventually leading to sums with an exponential number of summands. Therefore, we decided to analyze a simplified variant where we assume that the checksums are independent and uniformly distributed. For this simplified message mapping, we obtain exact complexities. We experimentally verified the results obtained for the simplified mapping function.

We analyze security of the OTS without initial message hash H in terms of full break resistance, universal, selective, and existential unforgeability under random and adaptively chosen message attacks. Please note that as we assume H to be a random oracle, existential unforgeability under an adaptively chosen message attack (EU-CMA) of a scheme with initial randomized message hashing is equivalent to existential unforgeability under

Table 3.1: Complexity for an existential forgery under a random message attack for the given signature scheme with typical parameters (see text).

Signature scheme	Attack Complexity
Lamport	$O((1.34)^{m})$
Optimized Lamport	$O((1.14)^{m+\log m})$
Winternitz	$O((1.09)^{m+\log m})$

a random message attack (EU-RMA) of the scheme without initial message hash. Accordingly, the crucial case for practice is EU-RMA security of the scheme without initial message hash. It covers the case of accidental reuse of an OTS key pair when using one of the recent RFC's for stateful hash-based signatures [MCF19, HBG+18]. While all three schemes turn out to be EU-RMA-secure under two-message attacks in the asymptotic setting, we get different results for typical parameter choices. For Lamport's scheme with a message digest size of 256 bits, the complexity to produce existential forgeries under two-random-message attacks is still 2^{106} hash function calls, ignoring the costs for pairwise comparison of all message digests. Hence, in this setting a signer is still on the safe side even after using a one-time key pair twice. For the optimized Lamport OTS with 256 bit message digests, the complexity to produce existential forgeries under two-random-message attacks is already down to 2^{51}. Which means attacks are not for free, but they are possible. For WOTS in the same setting, using the parameters from [HBG+18], we are left with an attack complexity of 2^{34} hash function computations. This can be done on a modern computer within few days if not hours. These parameters use a Winternitz parameter of $w = 16$, i.e. hash chains of length 16. For bigger values of w, the attack complexity goes down even further. These results show that Lamport's scheme is still somewhat forgiving but especially for WOTS, measures have to be taken that prevent OTS key reuse in any case. However, as soon as we are considering attacks on quantum-computers, complexities drop at least by a square-root factor. In this case even Lamport's scheme has to be considered broken after two-random-message attacks for typical parameters.

Organization. In Section 3.2 we discuss the models we use as well as required notation. We start our analysis in Section 3.3 with Lamport's scheme. We continue in Section 3.4 with the optimized Lamport scheme and in Section 3.5 with WOTS. In Section 3.6, we experimentally verify our results.

3.2 — The model

Security of one-time signature schemes (OTS) can be analyzed with regard to all traditional security definitions for general signature schemes. The difference is that the number of adversarial signature queries is limited to $q = 1$. Formally, any signature scheme that achieves EU-CMA-security (see definition below) when the adversary may only make a single signature query is a OTS. To understand the security of a OTS under two-message attacks in any of the models, we simply investigate the security for $q = 2$. We first discuss the traditional definitions and afterwards we discuss how to analyze security within these models.

3.2.1 – Digital signature schemes. First, what exactly are we talking about? From a formal perspective the objects we are talking about are digital signature schemes, defined as follows:

Definition 3.1 (Digital Signature Scheme). *Let $\mathcal{M}$ be the message space. A digital signature scheme* $\mathsf{Dss} = (\mathsf{kg}, \mathsf{sign}, \mathsf{vf})$ *is a triple of probabilistic polynomial time algorithms:*

- $\mathsf{kg}(1^n)$ *on input of a security parameter* 1^n *outputs a private signing key* sk *and a public verification key* pk*;*
- $\mathsf{sign}(\mathsf{sk}, \boldsymbol{m})$ *outputs a signature* σ *under* sk *for message* $\boldsymbol{m}$*, if* $\boldsymbol{m} \in \mathcal{M}$*;*
- $\mathsf{vf}(\mathsf{pk}, \sigma, \boldsymbol{m})$ *outputs 1 iff* σ *is a valid signature on* $\boldsymbol{m}$ *under* pk*;*

such that the following correctness condition is fulfilled:

$$\forall(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{kg}(1^n), \forall(\boldsymbol{m} \in \mathcal{M}) : \mathsf{vf}(\mathsf{pk}, \mathsf{sign}(\mathsf{sk}, \boldsymbol{m}), \boldsymbol{m}) = 1.$$

Throughout this thesis *signature scheme* always refers to a digital signature scheme.

3.2.2 – Security of signature schemes. The definition above is only a functional definition of the object at hand that says nothing about security. It leaves the question of how to define security for a signature scheme. In general we can split security notions into the goals an adversary $\mathcal{A}$ has to achieve (e.g., a valid signature on any new message for existential unforgeability) and the attack capabilities given to $\mathcal{A}$ (e.g., adaptively learning signatures on messages of its choice after seeing the public key). For the goals, the relevant notions[1] are:

Full break (FB): $\mathcal{A}$ can compute the secret key.

Universal forgery (UU): $\mathcal{A}$ can forge a signature for any given message. $\mathcal{A}$ can efficiently answer any signing query.

Selective forgery (SU): $\mathcal{A}$ can forge a signature for some message of its choice. In this case $\mathcal{A}$ commits itself to a message before the attack starts.

Existential forgery (EU): $\mathcal{A}$ can forge a signature for one arbitrary message. $\mathcal{A}$ might output a forgery for any message for which it did not learn the signature from an oracle during the attack.

On the other hand, for the attacks we got[2]:

Random message attack (RMA): $\mathcal{A}$ learns the public key and the signatures on a set of random messages.

Adaptively chosen message attack (CMA): $\mathcal{A}$ learns the public key and is allowed to adaptively ask for the signatures on messages of its choice[3].

These two attacks are parameterized by the number of signature queries q the adversary is allowed to ask. For one-time schemes we only require that a notion is fulfilled for $q = 1$.

Any combination of a goal and an attack from the above sets gives a meaningful notion of security. The strength of the notion increases going down each list. Accordingly, a scheme that is only secure against a full break under a random message attack offers the weakest kind of security while a scheme that offers existential unforgeability under adaptively chosen message attacks offers the strongest security guarantees.

3.2.3 – Formal definitions. We now give formal definitions for the notions above.
EU-CMA. The standard security notion for digital signature schemes is existential un-

[1] We omit strong unforgeability here as it is irrelevant for this context

[2] We omit key-only attacks as these allow for no signature queries at all

[3] We omit the non-adaptive setting as it turns out that there is no difference in the given setting.

forgeability under adaptive chosen message attacks (EU-CMA) which is defined using the following experiment. By $\mathsf{Dss}(1^n)$ we denote a signature scheme with security parameter n.

```
Experiment $\mathsf{Exp}^{\mathsf{EU\text{-}CMA}}_{\mathsf{Dss}(1^n)}(\mathcal{A})$
    $(\mathsf{sk}, \mathsf{pk}) \leftarrow \mathsf{kg}(1^n)$
    $(M^\star, \sigma^\star) \leftarrow \mathcal{A}^{\mathsf{sign}(\mathsf{sk},\cdot)}(\mathsf{pk})$
    Let $\{(\mathbf{m}_i, \sigma_i)\}_1^q$ be the query-answer pairs of $\mathsf{sign}(\mathsf{sk}, \cdot)$.
    Return 1 iff $\mathsf{vf}(\mathsf{pk}, M^\star, \sigma^\star) = 1$ and $M^\star \notin \{\mathbf{m}_i\}_1^q$.
```

For the success probability of an adversary $\mathcal{A}$ in the above experiment we write

$$\mathrm{Succ}^{\text{EU-CMA}}_{\mathsf{Dss}(1^n)}(\mathcal{A}) = \mathbb{P}\left[\mathsf{Exp}^{\text{EU-CMA}}_{\mathsf{Dss}(1^n)}(\mathcal{A}) = 1\right].$$

Definition 3.2 (EU-CMA). *Let $n \in \mathbb{N}$, Dss a digital signature scheme as defined above. We call Dss $(t, \epsilon(t), q)$-EU-CMA-secure if $\mathrm{InSec}^{\text{EU-CMA}}(\mathsf{Dss}(1^n); t, q)$, the maximum success probability of all possibly probabilistic adversaries $\mathcal{A}$ running in time $\leqslant t$, making at most q queries to sign in the above experiment, is bounded by $\epsilon(t)$:*

$$\mathrm{InSec}^{\text{EU-CMA}}(\mathsf{Dss}(1^n); t, q) \overset{\mathrm{def}}{=} \max_{\mathcal{A}}\{\mathrm{Succ}^{\text{EU-CMA}}_{\mathsf{Dss}(1^n)}(\mathcal{A})\} \leqslant \epsilon(t).$$

A $(t, \epsilon(t))$-EU-CMA-secure one-time signature scheme is a Dss that is $(t, \epsilon(t), 1)$-EU-CMA secure, i.e. the number of signing oracle queries of the adversary is limited to one.

We can give similar definitions for the remaining notions. The difference between the different notions is described by a modified experiment. The definition of success probability and what it means for a scheme to fulfill the notion can be obtained replacing the experiment in the above definitions (and, of course, tracing the resulting changes through the definition).

SU-CMA. Selective unforgeability is formally described by the following experiment. In this experiment $\mathcal{A}$ consists of two independent algorithms $(\mathcal{A}_1, \mathcal{A}_2)$. The first of which, $\mathcal{A}_1$, outputs the target message and some temporary state $\mathcal{S}$ that is forwarded to $\mathcal{A}_2$.

```
Experiment $\mathsf{Exp}^{\mathsf{SU\text{-}CMA}}_{\mathsf{Dss}(1^n)}(\mathcal{A} = (\mathcal{A}_1, \mathcal{A}_2))$
    $(M_{\mathcal{A}}, \mathcal{S}) \leftarrow \mathcal{A}_1(1^n)$
    $(\mathsf{sk}, \mathsf{pk}) \leftarrow \mathsf{kg}(1^n)$
    $\sigma^\star \leftarrow \mathcal{A}^{\mathsf{Sign}(\mathsf{sk},\cdot)}(\mathsf{pk}, M_{\mathcal{A}}, \mathcal{S})$
    Let $\{(\mathbf{m}_i, \sigma_i)\}_1^q$ be the query-answer pairs of $\mathsf{sign}(\mathsf{sk}, \cdot)$.
    Return 1 iff $\mathsf{vf}(\mathsf{pk}, M_{\mathcal{A}}, \sigma^\star) = 1$ and $M_{\mathcal{A}} \notin \{\mathbf{m}_i\}_1^q$.
```

UU-CMA. Universal unforgeability is formally described by the following experiment. The difference to the SU notion is that the target message $M_{\mathcal{A}}$ is now selected by the experiment.

```
Experiment $\mathsf{Exp}^{\mathsf{UU\text{-}CMA}}_{\mathsf{Dss}(1^n)}(\mathcal{A} = (\mathcal{A}_1, \mathcal{A}_2))$
    $(\mathsf{sk}, \mathsf{pk}) \leftarrow \mathsf{kg}(1^n)$
    $\mathcal{S} \leftarrow \mathcal{A}_1^{\mathsf{Sign}(\mathsf{sk},\cdot)}(\mathsf{pk})$
    $M_{\mathcal{A}} \xleftarrow{\$} \mathcal{M}$
    $\sigma^\star \leftarrow \mathcal{A}_2(\mathcal{S}, M_{\mathcal{A}})$
    Return 1 iff $\mathsf{vf}(\mathsf{pk}, M_{\mathcal{A}}, \sigma^\star) = 1$.
```

EU-RMA. Existential unforgeability under random message attacks (EU-RMA) is defined using the following experiment. Instead of giving the adversary oracle access as in the EU-CMA game, the experiment generates signatures on q random messages and hands these to the adversary.

```
Experiment $\mathsf{Exp}^{\mathsf{EU\text{-}RMA}}_{\mathsf{Dss}(1^n)}(\mathcal{A})$
    $(\mathsf{sk}, \mathsf{pk}) \leftarrow \mathsf{kg}(1^n)$
    Let $\{(\mathbf{m}_i, \sigma_i)\}_1^q$ be the set of q message signature pairs, obtained by
        sampling $\mathbf{m}_i \xleftarrow{\$} \mathcal{M}$ and computing $\sigma_i = \mathsf{sign}(\mathsf{sk}, \mathbf{m}_i)$.
    $(M^\star, \sigma^\star) \leftarrow \mathcal{A}(\mathsf{pk}, \{(\mathbf{m}_i, \sigma_i)\}_1^q)$
    Return 1 iff $\mathsf{vf}(\mathsf{pk}, M^\star, \sigma^\star) = 1$ and $M^\star \not\in \{\mathbf{m}_i\}_1^q$.
```

SU-RMA. Similarly to the previous notion, SU-RMA is defined be the experiment

```
Experiment $\mathsf{Exp}^{\mathsf{SU\text{-}RMA}}_{\mathsf{Dss}(1^n)}(\mathcal{A} = (\mathcal{A}_1, \mathcal{A}_2))$
    $(M_{\mathcal{A}}, \mathcal{S}) \leftarrow \mathcal{A}_1(1^n)$
    $(\mathsf{sk}, \mathsf{pk}) \leftarrow \mathsf{kg}(1^n)$
    Let $\{(\mathbf{m}_i, \sigma_i)\}_1^q$ be the set of q message signature pairs, obtained by
        sampling $\mathbf{m}_i \xleftarrow{\$} \mathcal{M}$ and computing $\sigma_i = \mathsf{sign}(\mathsf{sk}, \mathbf{m}_i)$.
    $\sigma^\star \leftarrow \mathcal{A}(\mathsf{pk}, \{(\mathbf{m}_i, \sigma_i)\}_1^q, M_{\mathcal{A}}, \mathcal{S})$
    Return 1 iff $\mathsf{vf}(\mathsf{pk}, M_{\mathcal{A}}, \sigma^\star) = 1$.
```

UU-RMA. Finally, universal unforgeability under random message attacks is formally described by the following experiment.

```
Experiment $\mathsf{Exp}^{\mathsf{UU\text{-}RMA}}_{\mathsf{Dss}(1^n)}(\mathcal{A} = (\mathcal{A}_1, \mathcal{A}_2))$
    $(\mathsf{sk}, \mathsf{pk}) \leftarrow \mathsf{kg}(1^n)$
    Let $\{(\mathbf{m}_i, \sigma_i)\}_1^q$ be the set of q message signature pairs, obtained by
        sampling $\mathbf{m}_i \xleftarrow{\$} \mathcal{M}$ and computing $\sigma_i = \mathsf{sign}(\mathsf{sk}, \mathbf{m}_i)$.
    $\mathcal{S} \leftarrow \mathcal{A}_1(\mathsf{pk}, \{(\mathbf{m}_i, \sigma_i)\}_1^q)$
    $M_{\mathcal{A}} \xleftarrow{\$} \mathcal{M}$
    $\sigma^\star \leftarrow \mathcal{A}_2(\mathcal{S}, M_{\mathcal{A}})$
    Return 1 iff $\mathsf{vf}(\mathsf{pk}, M_{\mathcal{A}}, \sigma^\star) = 1$.
```

Attack complexity. For a $(t, \epsilon(t))$-secure scheme, we define the attack complexity as $2t^*$ for $t^* = \min_t\{\epsilon(t) \geqslant \frac{1}{2}\}$. As we will show in the next chapters, the most costly operations of all attacks are calls to the message digest function H. We measure the attack complexity as the number of calls to H and denote its (constant) cost for one call with C_H.

Further model decisions. For our analysis we made several decisions on how we are analyzing the security in the above models. We are not interested in attacks that exploit weaknesses of the used hash-functions as these already apply in the one-message attack setting. Therefore, we model all used hash functions as random oracles. Due to this decision, RMA-attacks model the setting where randomized hashing is used for the initial message digest. Hence, we do not do a separate analysis for variants of the schemes that use randomized hashing.

3.3 — Lamport's scheme

We start with analyzing Lamport's scheme which was the first proposal for a hash-based signature scheme. For $q = 1$ it achieves the strongest security notion EU-CMA-security when the used function is one-way; this holds even without hashing the message first. Now let us look at the two-message attack case.

3.3.1 – Scheme description. The first and most intuitive proposal for an OTS is Lamport's scheme (sometimes called Lamport-Diffie OTS) [Lam79]. The scheme uses a one-way function $F : \{0,1\}^n \to \{0,1\}^n$, and signs m bit strings. The secret key consists of $2m$ random bit strings

$$\mathsf{sk} = (\mathsf{sk}_{1,0}, \mathsf{sk}_{1,1}, \ldots, \mathsf{sk}_{m,0}, \mathsf{sk}_{m,1})$$

of length n. The public key consists of the $2m$ outputs of the one-way function

$$\mathsf{pk} = (\mathsf{pk}_{1,0}, \mathsf{pk}_{1,1}, \ldots, \mathsf{pk}_{m,0}, \mathsf{pk}_{m,1}) = (F(\mathsf{sk}_{1,0}), F(\mathsf{sk}_{1,1}), \ldots, F(\mathsf{sk}_{m,0}), F(\mathsf{sk}_{m,1}))$$

when evaluated on the elements of the secret key. Signing a message (digest) $\mathbf{m}^* \in \{0,1\}^m$ corresponds to publishing the corresponding elements of the secret key:

$$\sigma = (\sigma_1, \ldots, \sigma_m) = (\mathsf{sk}_{1,\mathbf{m}_1^*}, \ldots, \mathsf{sk}_{m,\mathbf{m}_m^*}).$$

To verify a signature the verifier checks whether the elements of the signature are mapped to the right elements of the public key using F:

$$(F(\sigma_1), \ldots, F(\sigma_m)) \stackrel{?}{=} (\mathsf{pk}_{1,\mathbf{m}_1^*}, \ldots, \mathsf{pk}_{m,\mathbf{m}_m^*}).$$

For Lamport's scheme, the message mapping G can be considered the identity.

3.3.2 – Security under two-message attacks. Considering a CMA setting, we cannot achieve any security without an initial message hash. Otherwise, an adversary $\mathcal{A}$ can choose any pair of messages $(\mathbf{m}_1^*, \mathbf{m}_2^*)$ such that $\mathbf{m}_1^* = \neg\mathbf{m}_2^*$, where $\neg$ denotes bitwise negation, and will learn the full secret key. In the following we assume a message $\mathbf{m}$ is first hashed using a hash function $H : \{0,1\}^* \to \{0,1\}^m$, i.e., an m-bit message digest $\mathbf{m}^*$ is used to select the secret key elements. Our results are summarized in Table 3.2.

FB-CMA. A full break requires $\mathcal{A}$ to find a pair of messages $(\mathbf{m}_1, \mathbf{m}_2)$ such that $H(\mathbf{m}_1) = \neg H(\mathbf{m}_2)$. This task has the same complexity as collision finding for H. The only difference between the two tasks is that the equality condition is replaced by equality after negation. Sadly, this does not mean that we get a reduction from collision resistance as the counter example of the identity function shows: The identity function is collision resistant as no collisions exist but it is trivial to find a pair such that one message is the negation of the other. However, assuming H behaves like a random function a birthday bound argument shows that the complexity of finding such a pair is $C_B \cdot 2^{m/2}$ where $C_B = \sqrt{2\log(\frac{1}{1-p})}$

Table 3.2: Overview of the computational complexity for two-message attacks against Lamport's scheme. If the success probability of an attack is not constant in terms complexity, we give the attack complexity to achieve a success probability of $1/2$. C_H is the cost for one hash operation and C_B is the birthday-bound constant.

Security Goal	Attack Complexity	$\mathbb{P}$[Success]
EU-CMA	$C_H \cdot (4/3)^{m/3}$	$\frac{1}{2}$
SU-CMA	$C_H \cdot (4/3)^{m/3}$	$\frac{1}{2}$
UU-CMA	$C_H \cdot 2^{m/2}$	$\frac{1}{2}$
FB-CMA	$C_H \cdot C_B \cdot 2^{m/2}$	$\frac{1}{2}$
EU-RMA	$C_H \cdot (4/3)^{m}$	$\frac{1}{2}$
SU-RMA	-	$(3/4)^m$
UU-RMA	-	$(3/4)^m$
FB-RMA	-	$(1/2)^{m/2}$

is the birthday-bound constant for success probability p, which can be carried out as pre-computation as long as H is known. For $p = 1/2$ this constant is $C_B \approx 1.1774$.

EU-CMA. To produce a valid forgery in a chosen message setting, an adversary $\mathcal{A}$ has to find a triple of messages $\mathbf{m}_1, \mathbf{m}_2, \mathbf{m}_3$ such that

$$\text{break}(\mathbf{m}_1, \mathbf{m}_2, \mathbf{m}_3) = (\forall i \in [0, m-1]) : H(\mathbf{m}_1)_i = H(\mathbf{m}_2)_i \vee H(\mathbf{m}_1)_i = H(\mathbf{m}_3)_i)$$

where $H(\cdot)_i$ denotes the i-th bit of the message digest. In this case, we say that $\mathbf{m}_2, \mathbf{m}_3$ form a cover for $\mathbf{m}_1$.

For random messages $\mathbf{m}_1, \mathbf{m}_2, \mathbf{m}_3$, the probability that $\mathbf{m}_2, \mathbf{m}_3$ cover $\mathbf{m}_1$ is the inverse probability of each bit of $\mathbf{m}_1^*$ not being covered by $\mathbf{m}_2^*, \mathbf{m}_3^*$:

$$\mathbb{P}_{\mathbf{m}_1}[\text{break}(\mathbf{m}_1, \mathbf{m}_2, \mathbf{m}_3) = 1] = (1 - (1/2)^2)^m = (3/4)^m$$

For an existential forgery, $\mathcal{A}$ can start by hashing $\tau > 2$ random messages, pick a random set of two hashed message and check if these cover a hashed third message. There are $\binom{\tau}{2}$ such pairs of hashed messages, and $\tau - 2$ hashed messages that are potentially covered, leaving a total of $\binom{\tau}{2}(\tau - 2)$ possibilities. We can bound the success probability of an existential forgery by the union bound:

$$\mathbb{P}_{\{\mathbf{m}_1,\ldots,\mathbf{m}_\tau\}}[\exists(\mathbf{m}_a, \mathbf{m}_b, \mathbf{m}_c) \in \{\mathbf{m}_1, \ldots, \mathbf{m}_\tau\} : \text{break}(\mathbf{m}_a, \mathbf{m}_b, \mathbf{m}_c) = 1]$$
$$\leqslant \binom{\tau}{2}(\tau - 2)(3/4)^m \leqslant \frac{1}{2}\tau^3(3/4)^m$$

We want to know for which τ this upper bound reaches $1/2$, which is $\tau = (4/3)^{m/3}$. Hence, the attack complexity is lower bounded by $(4/3)^{m/3}$. As an example, if we consider $m = 256$ then $2^{36} > (4/3)^{m/3}$. It has to be noted that this is all pre-computation, which can be done before choosing a victim: no knowledge of the public key is required.

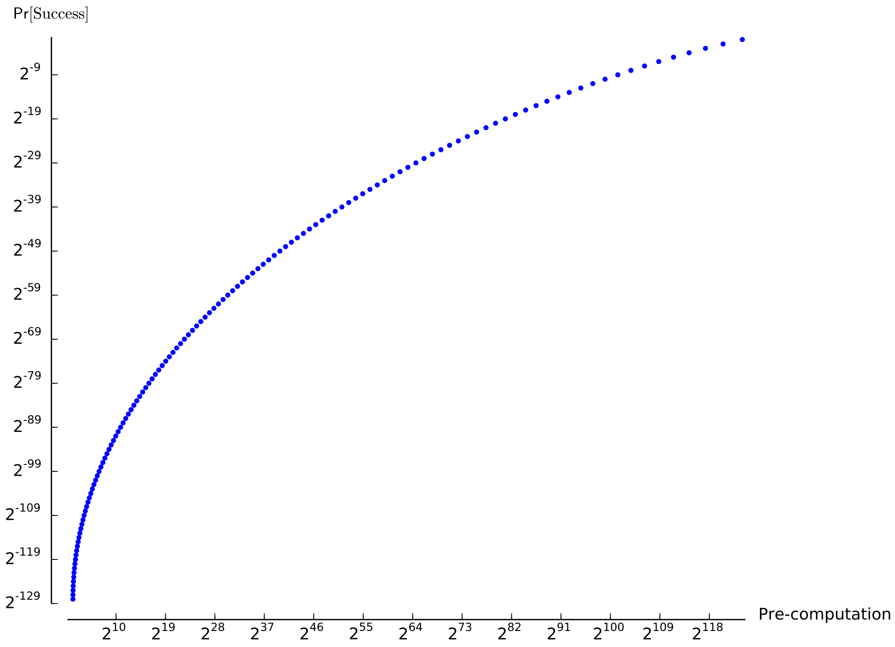


Figure 3.1: This plot shows the relation between the amount of pre-computation and the success probability of a universal forgery in a chosen message attack on Lamport's One-Time Signature Scheme.

It remains to show how tight our upper bound is. In Section 3.6, we experimentally verify that it is tight for the case of optimized Lamport and Winternitz.

SU-CMA. For selective forgeries, $\mathcal{A}$ can pick a message $\mathbf{m}$ for which it needs to find a cover before receiving signatures. However, since no knowledge of the public key is needed to start an attack, there is no difference between a selective forgery and an existential forgery. $\mathcal{A}$ can simply search for three messages $(\mathbf{m}_1, \mathbf{m}_2, \mathbf{m}_3)$ satisfying the break condition before the attack starts using the correct hash function. It can then commit to $\mathbf{m}_1$ before learning pk, and use the signatures of $\mathbf{m}_2, \mathbf{m}_3$ to sign $\mathbf{m}_1$. This means, the complexity of a selective forgery can again be lower bounded by $(4/3)^{m/3}$.

UU-CMA. For universal forgeries, $\mathcal{A}$ can try to find two messages $\mathbf{m}_1, \mathbf{m}_2$ such that they have non-overlapping message digests in r indices. After the experiment, $\mathcal{A}$ can forge any message with probability $(1/2)^{m-r}$, since a message digest has to overlap with the digests of $\mathbf{m}_1, \mathbf{m}_2$ in $m - r$ indices. The probability that any two messages $\mathbf{m}_1, \mathbf{m}_2$ have non-overlapping message digests in r indices is $\binom{m}{r}(1/2)^r(1/2)^{m-r} = \binom{m}{r}(1/2)^m$. Using similar arguments as in the EU-CMA case after τ calls to H, the probability that two messages have r non-overlapping indices is bounded by at least $1/2$ if $\binom{\tau}{2} \geqslant 1/2 \cdot 2^m \binom{m}{r}^{-1}$, where we can estimate that $\tau = 2^{m/2}\binom{m}{r}^{-1/2}$. It is easy to see that the more pre-computation an attacker is doing, the higher the success probability becomes. Figure 3.1 shows the success probability as a function of the pre-computation carried out. For $m = 256$, a pre-computation of 2^{136} calls to H is required to reach a probability of $1/2$.

EU-RMA. In this case, the adversary gets a signature of two random messages $(\mathbf{m}_1, \mathbf{m}_2)$ and has to find a third message $\mathbf{m}_3$ that is covered by $\mathbf{m}_1, \mathbf{m}_2$. The difference to the CMA case is that $\mathcal{A}$ cannot optimize the choice of $\mathbf{m}_1, \mathbf{m}_2$. This means each index should be covered, which happens with probability $(3/4)^m$. In consequence, $\mathcal{A}$ has to compute $\tau = (4/3)^m$ message digests before it finds a forgery with probability $\geqslant 1/2$. For $m = 256$, this means the attacker has to compute about 2^{106} message digests. However, for $m = 128$ bit message digests, this would mean a computational cost of 2^{53}, which should be pretty easy for strong attackers.

SU-RMA. For SU-RMA, the adversary selects a message before it receives two signatures of two random messages. There is no way for $\mathcal{A}$ to optimize the selection of this message, as $\mathcal{A}$ does not know (or has influence on) the two random messages for which it learns the signatures. The probability that $\mathcal{A}$ can afterwards sign the selected message is $(3/4)^m$. This is also the success probability of the attack. Note that this probability is constant for fixed parameters, i.e., there is nothing the adversary can do.

UU-RMA. For random message attacks, there is no difference between universal and selective forgery attacks since the adversary has no power over the signed messages and cannot affect his success probability by choice of a target message. This means also in this case, the probability of a forgery is $(3/4)^m$.

FB-RMA. The probability of a full break under a random message attack, is simply the probability that two messages are each-others negated version. This happens with probability $(1/2)^m$.

3.4 — Optimized Lamport

The optimized Lamport scheme is very similar to Lamport's scheme and first appeared in [Mer89]. While it is interesting on its own, it is also of interest as it can be viewed as a special, simplified version of the Winternitz OTS discussed in the next section.

3.4.1 – Scheme description. The optimized Lamport scheme uses a one-way function $F : \{0,1\}^n \to \{0,1\}^n$, and signs m bit messages. The secret key consists of $\ell = m + \log m + 1$ random bit strings

$$\mathsf{sk} = (\mathsf{sk}_1, \ldots, \mathsf{sk}_\ell)$$

of length n. The public key consists of the ℓ outputs of the one-way function

$$\mathsf{pk} = (\mathsf{pk}_1, \ldots, \mathsf{pk}_\ell) = (F(\mathsf{sk}_1), \ldots, F(\mathsf{sk}_\ell))$$

when evaluated on the elements of the secret key. Signing a message $\mathbf{m}^* \in \{0,1\}^m$ corresponds to first computing and appending a checksum to $\mathbf{m}^*$ to obtain the message mapping $G(\mathbf{m}^*) = B = \mathbf{m}^* \| C$ where $C = \sum_{i=1}^{m} \neg \mathbf{m}_i^*$. The signature consists of the secret key element if the corresponding bit in B is 1, and the public key element otherwise:

$$\sigma = (\sigma_1, \ldots, \sigma_m) \text{ with } \sigma_i = \begin{cases} \mathsf{sk}_i & \text{, if } B_i = 1, \\ \mathsf{pk}_i & \text{, if } B_i = 0. \end{cases}$$

To verify a signature the verifier checks whether the full public key is obtained by hashing the elements of the signature that correspond to 1 bits in B:

$$\text{Return 1, iff } (\forall i \in [1, \ell]) : \mathsf{pk}_i = \begin{cases} F(\sigma_i) & \text{, if } B_i = 1, \\ \sigma_i & \text{, if } B_i = 0. \end{cases}$$

3.4.2 – Security under two-message attacks. As with the non-optimized Lamport scheme, we cannot achieve any security without initial message hash. While it is impossible to learn the whole secret key from a two-message attack for typical parameters (this is the case as for m being a power of two the most significant bit of the checksum is only 1 for the all zero message, and it is impossible to learn the remaining secret key values from the signature of a single message), it is trivial to obtain all secret key elements but the one that corresponds to the most significant bit of the checksum. This allows to sign any message but the all 0 message. An adversary can for example use the all 1 message (to learn the secret key values for the message part of B) and any message with a single one (to learn the secret key values of the checksum part of B, besides the one at the most significant position).

In the following we assume a message $\mathbf{m}$ is first hashed using a hash function $H : \{0,1\}^* \to \{0,1\}^m$ to obtain a message digest $\mathbf{m}^*$ – making attacks significantly harder. It is easy to see that checksum C follows a binomial distribution. However, the analysis of the scheme as described above turned out too complex to be carried out exactly due to the dependency between C and $\mathbf{m}^*$. The problem is that it would be possible to condition on two checksums to cover a third one in the existential forgery setting. These conditions would give an exact Hamming weight for the message parts. However, there would be exponentially many possibilities, each with a specific probability, rendering a very complex analysis. For that reason, we simplified the analysis assuming that C is uniformly random and thereby that digest $\mathbf{m}^*$ and checksum C are independent of each other. Note that the neglected dependency, and the neglected distribution of C, can make the attack both easier and harder, depending on whether the higher order bits of C are covered. Our theoretical results are summarized in Table 3.3. For an experimental verification of our results see Section 3.6.

Remark: It is important to note that for extreme cases our analysis is not good enough. In the FB-CMA, UU-CMA, SU-RMA and FB-RMA settings for the optimized Lamport (and also for Winternitz in Chapter 3.5), we are trying to push the message mappings to extreme cases to allow for forgeries. However, due to the inverse nature of the checksum, our analysis leads to impossible message mappings. For example, a high weight message part means a low weight checksum part for optimized Lamport, but in our analysis we are trying to push both message and checksum part to high weights. Therefore we expect the complexity to be much higher for these extreme cases (i.e. when r is very low or very high, with the meaning of r as described in optimized Lamport and Winternitz for the UU-CMA and SU-RMA case). So although this is a non-tight approximation in theory, we will show with experiments (Section 3.6) that this seems good enough in the average case.

FB-CMA. As mentioned above for m being a power of two (which is the typical setting), it is impossible to learn the whole secret key from a two-message attack. For other choices of m, an adversary $\mathcal{A}$ has to find two messages $\mathbf{m}_1, \mathbf{m}_2$ such that $(B_1)_i = 1$ or $(B_2)_i = 1$ for all $i \in \{0, \ldots, \ell - 1\}$.

As H is modeled as random oracle and we assume the checksum is uniformly random and independent of the message, every random input message $\mathbf{m}$ leads to a random message mapping B of length ℓ. For two random input messages $\mathbf{m}_1, \mathbf{m}_2$, the probability that at least one of the two corresponding message mappings B_1, B_2 is 1 at each position is:

$$\mathbb{P}[\mathrm{FB}(\mathbf{m}_1, \mathbf{m}_2)] = (3/4)^\ell.$$

Table 3.3: Overview of the computational complexity for two-message attacks against the optimized Lamport scheme. If the success probability of an attack is not constant in terms of complexity, we give the attack complexity to achieve a success probability of 1/2 (aside from SU-RMA as the best we can achieve is a success probability of $\frac{3}{8}$). C_H is the cost for one hash operation.

Security Goal	Attack Complexity	$\mathbb{P}$[Success]
EU-CMA	$C_H \cdot (8/7)^{\ell/3}$	$\frac{1}{2}$
SU-CMA	$C_H \cdot (8/7)^{\ell/3}$	$\frac{1}{2}$
UU-CMA	$C_H \cdot (4/3)^{\ell/2}$	$\frac{1}{2}$
FB-CMA	$C_H \cdot (4/3)^{\ell/2}$	$\frac{1}{2}$
EU-RMA	$C_H \cdot (8/7)^{\ell}$	$\frac{1}{2}$
SU-RMA	$C_H \cdot 2^{\ell}$	$\frac{3}{8}$
UU-RMA	-	$(7/8)^{\ell}$
FB-RMA	-	$(3/4)^{\ell}$

Similar to the strategy of the existential forgery in Lamport's scheme, we can hash τ messages and check all pairs for a full break. The probability of a full break is bounded by $\binom{\tau}{2}(3/4)^{\ell}$. We can therefore lower bound the attack complexity of a full break by $(4/3)^{\ell/2}$ calls to H. For $m = 256$, this complexity equals 2^{54}.

EU-CMA. We will now explore forgeries for a third message, given the signatures for two messages. We define the condition for a break for three messages $\mathbf{m}_1, \mathbf{m}_2, \mathbf{m}_3$ with message mappings B_1, B_2, B_3 as:

$$\text{break}(\mathbf{m}_1, \mathbf{m}_2, \mathbf{m}_3) := (\forall i \in [0, \ell - 1]) : (B_1)_i = 1 \Rightarrow (B_2)_i = 1 \vee (B_3)_i = 1 \qquad (3.1)$$

where $(B_j)_i$ denotes the i-th bit of the mapping of message $\mathbf{m}_j$. If the condition is fulfilled, we say that $\mathbf{m}_2, \mathbf{m}_3$ form a cover of $\mathbf{m}_1$.

In other words: we only need the secret values for those bits of the first message mapping that are 1, so the probability for a break is higher for target messages with a low weight message mapping. Recall that we assume that $\mathbf{m}_j^*$ and C_j are independent, meaning we assume we have three independent random bit strings.

To get the probability that we cover a bit of B_1, we can condition on the value of that bit $b \in \{0, 1\}$:

$$\begin{aligned}
&\mathbb{P}[(B_1)_i \leqslant (B_2)_i \vee (B_1)_i \leqslant (B_3)_i] \\
&= \sum_{b \in \{0,1\}} \mathbb{P}[(B_1)_i \leqslant (B_2)_i \vee (B_1)_i \leqslant (B_3)_i \,|(B_1)_i = b]\mathbb{P}[(B_1)_i = b] \\
&= \frac{1}{2} \cdot \mathbb{P}[0 \leqslant (B_2)_i \vee 0 \leqslant (B_3)_i \,|(B_1)_i = 0] \\
&\quad + \frac{1}{2} \cdot \mathbb{P}[1 \leqslant (B_2)_i \vee 1 \leqslant (B_3)_i \,|(B_1)_i = 1] \\
&= \frac{1}{2} \cdot 1 + \frac{1}{2} \cdot \frac{3}{4} = \frac{7}{8}
\end{aligned}$$

This means that the probability that the break condition is fulfilled for three random messages is $\left(\frac{7}{8}\right)^{\ell}$.

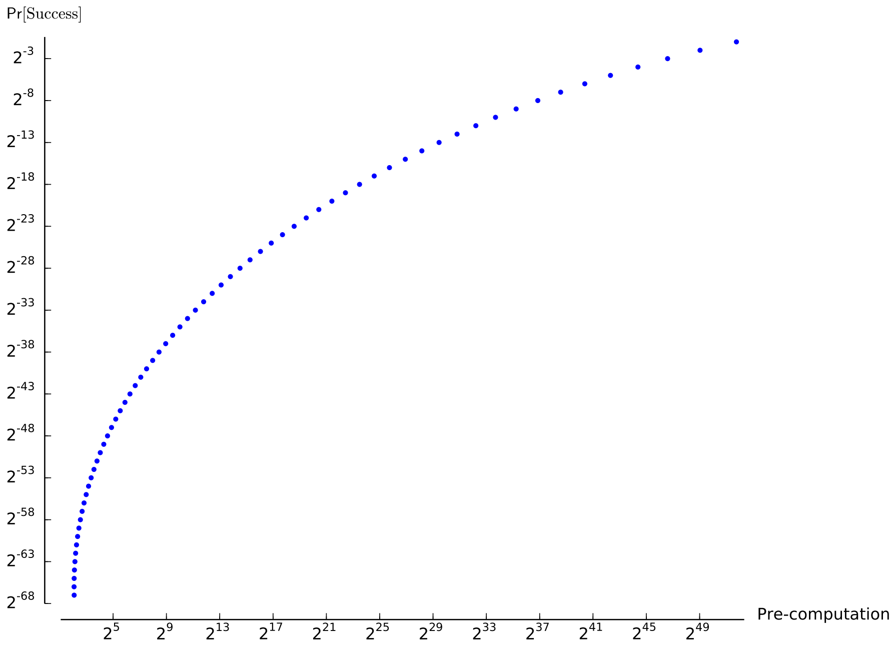


Figure 3.2: This plot shows the relation between the amount of pre-computation and the success probability of a universal forgery in a chosen message attack on the optimized Lamport scheme.

As with the original Lamport scheme, we can precompute τ message mappings, and calculate the upper bound for the success probability. This time, for the bound to reach $1/2$ we need to compute $\tau = (8/7)^{\ell/3}$ message mappings, using similar arguments as in the EU-CMA case for Lamport. For $m = 256$, this means the adversary needs to precompute $\tau = 2^{17}$ hash digests. For $m = 128$, this would mean $\tau = 2^9$ hash digests.

SU-CMA. As with the original Lamport scheme, the adversary does not need knowledge of the public key to compute three messages that satisfy the break condition. This means that also for the optimized Lamport scheme, a selective forgery has the same complexity as an existential forgery under chosen message attacks.

UU-CMA. The goal of the adversary is to find two messages $\mathbf{m}_1, \mathbf{m}_2$ such that their combined mappings have the highest weight possible. The probability that any two messages have weight r is equal to $\binom{\ell}{r}(3/4)^r(1/4)^{\ell-r}$, where we again assume that $\mathbf{m}^*$ and C are independent. Note that the mean of this distribution is at $\ell \cdot (3/4)$, which means $\mathcal{A}$ should not take any r below $\ell \cdot (3/4)$. After τ calls to H, the probability that two of the messages $\mathbf{m}_1, \mathbf{m}_2$ have a combined weight of r is bounded by at least $1/2$ if $\binom{\tau}{2} \geqslant 1/2 \cdot \left(\binom{\ell}{r}(3/4)^r(1/4)^{\ell-r}\right)^{-1}$. We can estimate the pre-computation complexity as square-root of the right part of this inequality. After the online phase of the attack, $\mathcal{A}$ can sign a new message with probability $(1/2)^{\ell-r}$, since for the positions that are not covered by B_1 or B_2, the bit of the new message must be 0. The relation between the pre-computation and the success probability is given in Figure 3.2 for $m = 256$.

EU-RMA. According to Eqn. 3.1, two messages $\mathbf{m}_2, \mathbf{m}_3$ have a probability of $(7/8)^\ell$ to cover a random third message $\mathbf{m}_1$. This means that after receiving the signature of two random messages, the adversary has to search $\tau = (8/7)^\ell$ messages to forge a third signature (again using arguments described in earlier analyses), since it only needs the secret values for the bits of $\mathbf{m}_1$ that are 1. For $m = 256$, this means a computational cost of about 2^{51}, which is in reach for a strong attacker. For $m = 128$, this would mean a computational cost of 2^{26}, which can be done within minutes on today's CPUs.

SU-RMA. Unlike with the original Lamport scheme, for the optimized Lamport scheme an adversary can optimize his selection of the target message in a random message attack. In our simplified analysis messages that have low-weight message mappings are more likely to be covered by the mappings of two random messages. However, note that we can only select a single target message instead of a whole cover, which makes the pre-computation more costly. The probability to find a message mapping B with weight r is equal to $\binom{\ell}{r}(1/2)^\ell$, which is again symmetric around $\ell/2$. An attacker should therefore always pick a message with weight $r \leqslant \ell/2$. This message can be signed, after receiving the signatures of two random messages, with probability $(3/4)^r$, since all positions of B that are 1 have to be covered by the mappings of the two random messages. If we again estimate the pre-computation as $\tau = \left(\binom{\ell}{r}(1/2)^\ell\right)^{-1}$ to find a message mapping with weight r with probability bounded by $1/2$, we get the relation between pre-computation and success probability for a selective forgery in Figure 3.3 for $m = 256$. Note that this figure looks similar to Figure 3.2 but a far more pre-computation is required to achieve the same bound on the success probability. Even for strong attackers, it should be infeasible to get a high success probability.

UU-RMA. For a universal forgery under a random message attack, the attacker cannot influence anything in the experiment. This means the success probability for this forgery is simply the success probability of the conditional break: $(7/8)^\ell$.

FB-RMA. The probability of a full break under a random message attack, is simply the probability that all bits are covered. This would be 0 in the real scenario (as described in the beginning of Section 3.4.2), but happens with probability $(3/4)^\ell$ in our simplified analysis, which is 2^{-54} when $m = 256$.

3.5 — Winternitz OTS

The Winternitz one-time signature scheme (WOTS) is a further improvement of the optimized Lamport scheme. Instead of using the hash of each secret key value as public key, the public key values are obtained by hashing more than once, i.e. w times. That way, more than one bit can be encoded per selection of a hash value. The basic idea for the Winternitz OTS (WOTS) was proposed in [Mer89]. What we know as WOTS today is a generalization that was proposed by Even, Goldreich, and Micali [EGM96]. There exist several variants that reduce the assumptions made about the used hash function [BDE+11, Hül13, HRS16]. Recent standardization proposals for hash-based signatures [MCF19, HBG+18] as well as a recent proposal for stateless hash-based signatures [BHH+15] use WOTS as one-time signature scheme.

3.5.1 – Scheme description. WOTS uses a length-preserving (cryptographic hash) function $F : \{0,1\}^n \to \{0,1\}^n$. It is parameterized by the message length m and the

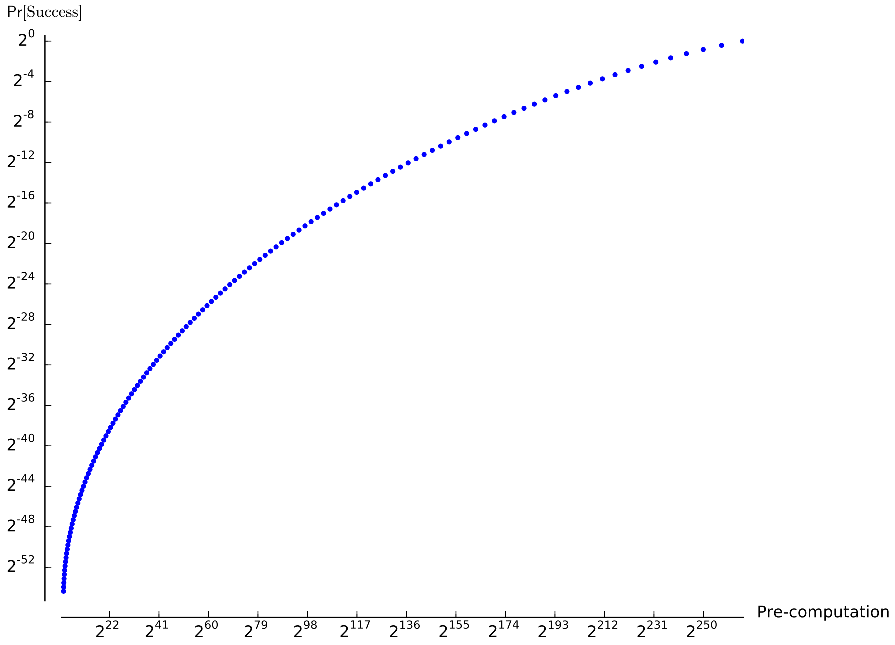


Figure 3.3: This plot shows the relation between the amount of pre-computation and the success probability of a selective forgery in a chosen message attack on the optimized Lamport's One-Time Signature Scheme.

Winternitz parameter $w \in \mathbb{N}, w > 1$, which determines the time-memory trade-off. The two parameters are used to compute

$$\ell_1 = \left\lceil \frac{m}{\log(w)} \right\rceil, \quad \ell_2 = \left\lfloor \frac{\log(\ell_1(w-1))}{\log(w)} \right\rfloor + 1, \quad \ell = \ell_1 + \ell_2.$$

The scheme uses $w - 1$ iterations of F on a random input. We define them as

$$F^a(x) = F(F^{a-1}(x))$$

and $F^0(x) = x$.

Now we describe the three algorithms of the scheme:

Key generation algorithm $(\mathsf{kg}(1^n))$**:** On input of security parameter 1^n the key generation algorithm choses ℓ n-bit strings uniformly at random. The secret key $\mathsf{sk} = (\mathsf{sk}_1, \ldots, \mathsf{sk}_\ell)$ consists of these ℓ random bit strings. The public verification key pk is computed as

$$\mathsf{pk} = (\mathsf{pk}_1, \ldots, \mathsf{pk}_\ell) = (F^{w-1}(\mathsf{sk}_1), \ldots, F^{w-1}(\mathsf{sk}_\ell))$$

Signature algorithm $(\mathsf{sign}(1^n, \mathbf{m}^*, \mathsf{sk}))$**:** A message (digest) $\mathbf{m}^*$ of length m and the secret signing key sk, the signature algorithm first computes a base w representation of

$\mathbf{m}^*$: $\mathbf{m}^* = (\mathbf{m}^*_1 \ldots \mathbf{m}^*_{\ell_1})$, $\mathbf{m}^*_i \in \{0, \ldots, w-1\}$. Next it computes the checksum

$$C = \sum_{i=1}^{\ell_1} (w - 1 - \mathbf{m}^*_i)$$

and computes its base w representation $C = (C_1, \ldots, C_{\ell_2})$. The length of the base-w representation of C is at most ℓ_2 since $C \leqslant \ell_1(w-1)$. We set $B = (B_1, \ldots, B_\ell) = \mathbf{m}^* \parallel C$. The signature is computed as

$$\sigma = (\sigma_1, \ldots, \sigma_\ell) = (F^{B_1}(\mathsf{sk}_1), \ldots, F^{B_\ell}(\mathsf{sk}_\ell)).$$

Verification algorithm $(\mathsf{vf}(1^n, \mathbf{m}^*, \sigma, \mathsf{pk}))$**:** A message (digest) $\mathbf{m}^*$ of length m, a signature σ and the public verification key pk, the verification algorithm first computes the B_i, $1 \leqslant i \leqslant \ell$ as described above. Then it does the following comparison:

$$\mathsf{pk} = (\mathsf{pk}_1, \ldots, \mathsf{pk}_\ell) \stackrel{?}{=} (F^{w-1-B_1}(\sigma_1), \ldots, F^{w-1-B_\ell}(\sigma_\ell)).$$

If the comparison holds, it returns **true** and **false** otherwise.

Remark: the difference between the basic WOTS as described above and the variants proposed in [BDE+11, Hül13, HRS16] is how F is iterated. As all the attacks below are independent of this choice, our results apply to all those variants, too.

3.5.2 – Two-message attacks. Without hashing the message, the scheme does not offer any security once an attacker can choose two messages to be signed. As always, the adversary simply chooses the all zero and the all one message to be signed, and afterwards knows all secret values (for some parameter choices it will actually be impossible to extract the whole secret key for the same reason as for optimized Lamport. However, in that case, as for the optimized Lamport scheme, it is possible to select two messages that allow learn all but one secret key element).

In the following we assume a message $\mathbf{m}$ is first hashed using a hash function $H : \{0,1\}^* \to \{0,1\}^m$ to obtain a message digest $\mathbf{m}^*$ – making attacks significantly harder. As for the optimized Lamport scheme, the analysis of the scheme as described above turned out too complex to be carried out exactly due to the dependency between C and $\mathbf{m}^*$. We simplified the analysis assuming that C is uniformly random and thereby that digest $\mathbf{m}^*$ and checksum C are independent of each other. It applies again that the neglected dependency can make the attack both easier and harder, depending on the setting. Our theoretical results are summarized in Table 3.4.

Remark: It is important to note that for extreme cases this analysis is not good enough, as was the case for the analysis in Section 3.4 optimized Lamport. However, in experiments we verify that this analysis seems good enough, see Section 3.6 for the results of this.

FB-CMA. The adversary has to find messages $\mathbf{m}_1, \mathbf{m}_2$ with mappings B_1, B_2 such that for all $0 \leqslant i \leqslant \ell$: either $(B_1)_i = 0$ or $(B_2)_i = 0$. The probability to cover an index of the secret key equals $(1 - (\frac{w-1}{w})^2)$ for each i, which means the probability that this is true for all i equals: $(1 - (\frac{w-1}{w})^2)^\ell$. After hashing τ messages, the probability to find two messages satisfying the condition of a full break will be upper bounded by at least $1/2$ if $\binom{\tau}{2} \geqslant 1/2 \cdot (1 - (\frac{w-1}{w})^2)^{-\ell}$, which means we can lower bound the attack

Table 3.4: Overview of the computational complexity for two-message attacks against the Winternitz OTS. If the success probability of an attack is not constant in terms of complexity, we give the attack complexity to achieve a success probability of 1/2. C_H is the cost for one hash operation.

Security Goal	Attack Complexity	$\mathbb{P}$[Success]
EU-CMA	$C_H \cdot \left(\frac{(w+1)(4w+1)}{6w^2}\right)^{\ell/3}$	$\frac{1}{2}$
SU-CMA	$C_H \cdot \left(\frac{(w+1)(4w+1)}{6w^2}\right)^{\ell/3}$	$\frac{1}{2}$
UU-CMA	$C_H \cdot \left(1-(\frac{w-1}{w})^2\right)^{\ell/2}$	$\frac{1}{2}$
FB-CMA	$C_H \cdot \left(1-(\frac{w-1}{w})^2\right)^{\ell/2}$	$\frac{1}{2}$
EU-RMA	$C_H \cdot \left(\frac{(w+1)(4w+1)}{6w^2}\right)^{\ell}$	$\frac{1}{2}$
SU-RMA	$C_H \cdot \left(\frac{1}{w}\right)^{\ell}$	$\frac{1}{2}$
UU-RMA	-	$\left(\frac{(w+1)(4w+1)}{6w^2}\right)^{\ell}$
FB-RMA	-	$\left(1-(\frac{w-1}{w})^2\right)^{\ell}$

complexity by $\tau \geqslant (1-(\frac{w-1}{w})^2)^{-\ell/2}$. As a sanity check, we see that for $w = 2$ we get $\tau = (4/3)^{\ell/2}$, which is the complexity of a full break for the optimized Lamport scheme. Typical parameters for applications are $w = 16$ and $m = 256$, which leads to $\ell = 67$ and $\tau = 2^{102}$.

EU-CMA. For an existential forgery, we first define the condition for a break for WOTS:

$$\mathrm{break}(\mathbf{m}_1, \mathbf{m}_2, \mathbf{m}_3) := (\forall i \in [0, \ell-1]) : (B_1)_i \geqslant (B_2)_i \vee (B_1)_i \geqslant (B_3)_i \tag{3.2}$$

where $(B_j)_i$ denotes the i-th digit of the base-w values of the message mapping B_j for message $\mathbf{m}_j$; $j \in \{1, 2, 3\}$. If the condition is true, we say $\mathbf{m}_2, \mathbf{m}_3$ form a cover of $\mathbf{m}_1$.

We will first see what the probability is to cover one index of B_1. If we condition on the value of $(B_1)_i$, we get:

$$\begin{aligned}
\mathbb{P}[(B_1)_i \geqslant (B_2)_i \vee (B_1)_i \geqslant (B_3)_i] &= \\
\sum_{x=0}^{w-1} \mathbb{P}[(B_1)_i \geqslant (B_2)_i \vee (B_1)_i \geqslant (B_3)_i | (B_1)_i = x] \mathbb{P}[(B_1)_i = x] &= \\
\sum_{x=0}^{w-1} \frac{1}{w}\left(1-\left(\frac{w-(x+1)}{w}\right)^2\right) &= \\
\frac{1}{w^3}\left(\sum_{x=0}^{w-1} w^2 - \sum_{x=0}^{w-1}(w-(x+1))^2\right) &= \\
1 - \frac{1}{w^3}\sum_{i=0}^{w-1} i^2 &= \\
1 - \frac{w(w-1)(2w-1)}{6w^3} &= \\
\frac{(w+1)(4w-1)}{6w^2} &
\end{aligned}$$

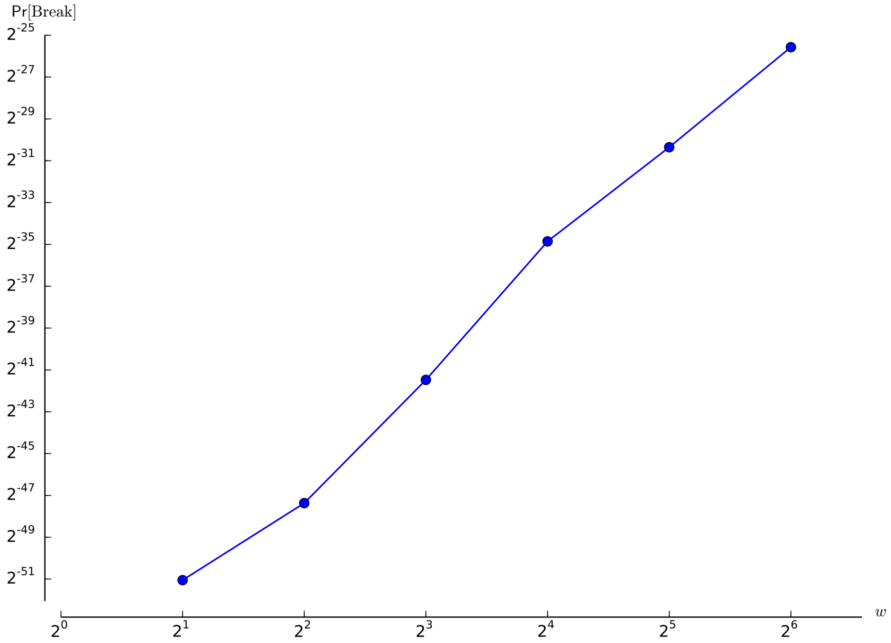


Figure 3.4: This plot shows the logarithmic relation between w and Pr[break] for $w \in \{2, 4, 8, 16, 32, 64\}$. The logarithmic decrease of the exponent in Pr[break] is clearly making the probability grow faster for larger w.

Again as a sanity check, we see that for $w = 2$, this probability equals $(7/8)$, which we already concluded for the optimized Lamport scheme.

In total we see that the probability for a conditional break is:

$$\begin{aligned}\Pr[\text{break}(\mathbf{m}_1, \mathbf{m}_2, \mathbf{m}_3) = 1] &= \left(\frac{(w+1)(4w-1)}{6w^2}\right)^{\ell} \\ &\approx \left(\frac{(w+1)(4w-1)}{6w^2}\right)^{\frac{m+\log m}{\log w}}\end{aligned}$$

We see that for bigger w, the probability that one of the indices is not covered grows, but the number of indices shrinks. The logarithmic decrease of the exponent is in this case more important, which means the bigger the w, the bigger the probability of the conditional break (which means less computational power required for forgeries).

Similar to the arguments for the EU-CMA cases for Lamport and optimized Lamport scheme, an adversary needs to pre-compute about $\tau = \left(\left(\frac{(w+1)(4w-1)}{6w^2}\right)^{-\frac{m+\log m}{\log w}}\right)^{1/3}$ message mappings for the bound on the probability to find a cover in the list of τ message mappings to reach $1/2$. As an example, if we set $m = 256$ and $w = 16$, we have $\tau = 2^{12}$. Note that, unlike the FB-CMA setting, it is much easier to forge a third signature for bigger w: while it becomes harder to get $B_i = 0$, the probability for a message cover grows.

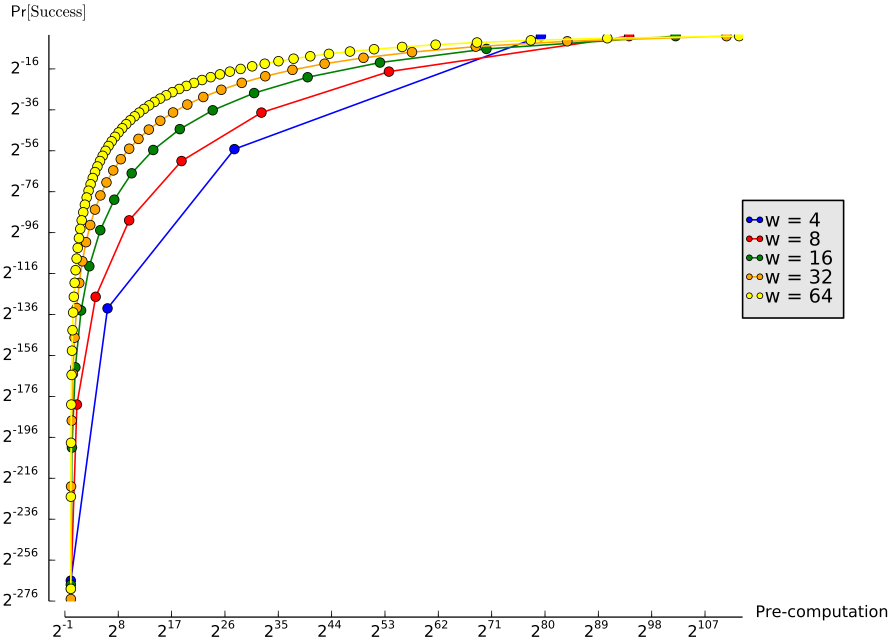


Figure 3.5: This plot shows the relation between the amount of pre-computation and the lower bound for the success probability for a universal forgery under a chosen message attack on WOTS for different values of w and for each $r \in \{0, \dots, w-1\}$.

SU-CMA. As with Lamport's scheme and the optimized Lamport scheme, $\mathcal{A}$ does not need knowledge of the public key to start any pre-computation. This means we obtain the same complexity for a selective forgery as for an existential forgery under CMA.

UU-CMA. For a universal forgery, $\mathcal{A}$ can try to compute two message mappings B_1, B_2 such that either $(B_1)_i \leqslant r$ or $(B_2)_i \leqslant r$ for all $i \in \{0, \dots, \ell-1\}$, where $r \in \{0, \dots, w-1\}$. The probability that any two messages satisfy these rules equals $\left(1-\left(\frac{w-(r+1)}{w}\right)^2\right)^{\ell}$, which means the probability that there exist two such messages in a list of τ messages is bounded by at least 1/2 if $\binom{\tau}{2} \geqslant 1/2 \cdot \left(1-\left(\frac{w-(r+1)}{w}\right)^2\right)^{-\ell}$, using again the same arguments as for Lamport and optimized Lamport. Now $\mathcal{A}$ obtains a successful forgery for M_3 with probability *at least* $\left(\frac{w-r}{w}\right)^{\ell}$, since we ignored the cases where $(B_3)_i$ is smaller than r, but still bigger than $(B_1)_i$ or $(B_2)_i$. The pre-computation τ and corresponding success probability for different values of w and $r \in \{0, \dots, w-1\}$ are given in Figure 3.5.

EU-RMA. For WOTS, two messages cover a third one with probability:

$$\Pr[\text{break}(\mathbf{m}_1, \mathbf{m}_2, \mathbf{m}_3) = 1] \approx \left(\frac{(w+1)(4w-1)}{6w^2}\right)^{\frac{m+\log m}{\log w}}.$$

This means that when an attacker receives two signatures of two random messages, it has

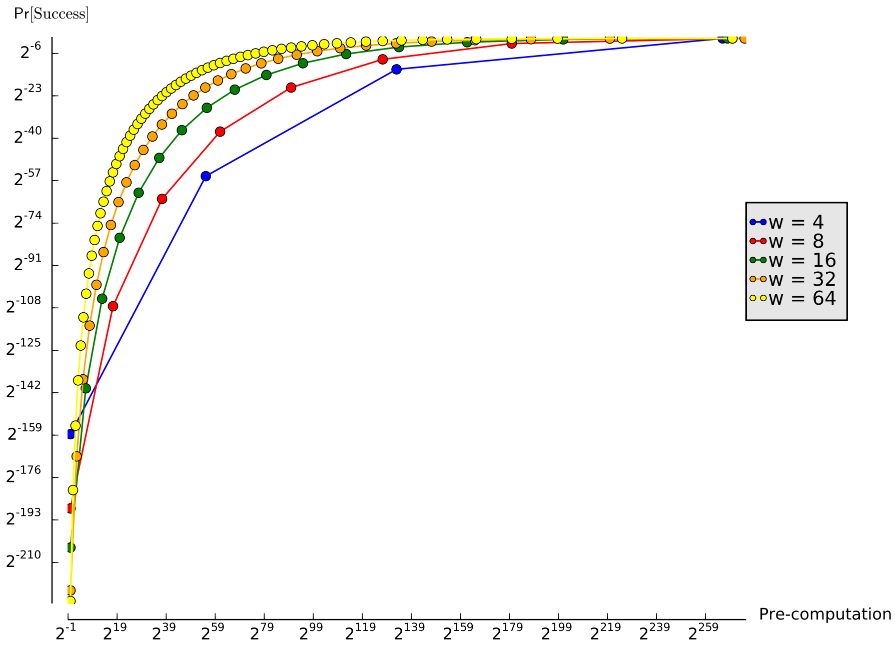


Figure 3.6: This plot shows the relation between the amount of pre-computation and the success probability of a selective forgery under random message attacks on WOTS for different values of w and for each $r \in \{0, \dots, w-1\}$

to compute about $\tau = \left(\frac{(w+1)(4w-1)}{6w^2}\right)^{-\frac{m+\log m}{\log w}}$ messages to find a covered third message. For $m = 256$ and $w = 16$, this equals 2^{34}, which can easily be done on today's CPUs.

SU-RMA. For the selective forgery, an attacker can select an optimal message with a mapping that contains as high values as possible. For the analysis, we will use the same strategy as for the universal forgery, but in this case we want $(B_1)_i \geqslant r$ for all $i \in \{0, \dots, \ell-1\}$, which happens with probability $\left(\frac{w-r}{w}\right)^{\ell}$. Hence, the pre-computation can again be bound by $\tau \geqslant \left(\frac{w-r}{w}\right)^{-\ell}$ to upper bound the probability of finding such a message in a list of τ messages by at least $1/2$. The probability that the adversary can sign his selected message after he received two signatures on random messages equals $\left(1 - \left(\frac{w-(r+1)}{w}\right)^2\right)^{\ell}$ in this case. A plot of the computational costs with corresponding success probability is given in Figure 3.6. As for the optimized Lamport scheme, it looks similar to the graph of the universal forgery under chosen message attacks, but with lower success probabilities since $\mathcal{A}$ only has control over the selected message.

UU-RMA. The probability of a successful universal forgery under a random message attack equals the probability that three random messages fulfill the break condition:

$$\Pr[\text{break}(\mathbf{m}_1, \mathbf{m}_2, \mathbf{m}_3) = 1] \approx \left(\frac{(w+1)(4w-1)}{6w^2}\right)^{\frac{m+\log m}{\log w}}$$

The attacker has no influence on the process and cannot use any computational power before or after the online phase of the attack to increase his success probability. Recall that for specific cases this probability can also be 0 (e.g. for $w = 2$ it is equal to optimized Lamport).

FB-RMA. Similar to Lamport and optimized Lamport, a full break occurs exactly when all secret values are exposed. For Winternitz with parameter w, this happens with probability $(1 - (\frac{w-1}{w})^2)^\ell$, which is a negligible probability for any w.

3.6 — Experimental verifications

In sections 3.3, 3.4, and 3.5 we discussed the attack complexity of several different attacks. For the optimized Lamport scheme and WOTS, we assumed that the checksum is uniformly random and hence the message digest and its checksum behave as independent bit strings. However, as already mentioned there, the actual situation is that the checksum is dependent of the message digest. To verify the obtained results we carried out experiments for the EU-CMA case for optimized Lamport and WOTS.

We determined a lower bound for the number of calls τ to the message digest function H, such that a list of size τ of message digests allows to find an existential forgery with probability upper bounded by at least 1/2. We performed several experiments for different values of τ, to see how realistic our assumption matches the real situation and how tight our bound is. We checked how many times a list of τ message mappings contained a cover for optimized Lamport with digest length of $m = 128$ bits and for WOTS, with $m = 256$ and $w = 16$ (which are the parameters suggested in [HBG$^+$18]). We performed 100 experiments per value of τ. As can be seen from the results in Table 3.5 on the next page, the experiments closely match the theoretical results using the checksum simplification. The theoretical analysis predicts that $\tau = 2^9$ is required for the bound on the probability of an existential forgery to reach 1/2 for the optimized Lamport scheme with $m = 128$. For WOTS, the analysis suggests $\tau = 2^{12}$ when $m = 256$ and $w = 16$. From the results of the experiments, we can conclude that the simplifying assumption of independent message digests and checksums is not causing a significant difference to the real setting in the case of EU-CMA.

Table 3.5: Experimental results for the success probability of an EU-CMA adversary, using a list of τ message mappings for optimized Lamport (left table) with digest length $m = 128$ and for WOTS (right table) with $w = 16$ and digest length $m = 256$

τ	$\mathbb{P}$[Success]
2^8	0.02
2^9	0.13
2^{10}	0.77
2^{11}	1.0
2^{12}	1.0

τ	$\mathbb{P}$[Succes]
2^{11}	0.1
2^{12}	0.49
2^{13}	0.94
2^{14}	1.0
2^{15}	1.0

CHAPTER 4

HILA5 pindakaas

4.1 — Overview

Context. HILA5 [Saa17b] is a public-key lattice-based encryption scheme designed by Saarinen and published at SAC 2017. We will introduce the background of lattice-based cryptography (and terms as e.g. RLWE and NTRU) in Section 6.2, as this is not required to understand this chapter. HILA5 was submitted as a "Key Encapsulation Mechanism and Public Key Encryption Algorithm" [Saa17a] to NIST's call [NIS16] for post-quantum proposals. A Key Encapsulation Mechanism (KEM) is a public-key encryption technique that fixes a random symmetric session key, that can be used to encrypt data using symmetric encryption in the Data Encapsulation Mechanism (DEM). This approach is called a KEM-DEM technique for hybrid encryption. A DEM can both encrypt and authenticate data when using authenticated encryption with associated data(AEAD) [Rog02].

HILA5's design is based on Ring Learning With Errors (RLWE) over NTRU NTT rings. HILA5 has a similar design as other lattice-based public-key encryption schemes such as New Hope [ADPS16], but changes the reconciliation method by which Alice and Bob achieve the same key. New Hope [ADPS16] is a lattice-based KEM, presented as a key-exchange protocol. In these protocols, two parties perform an interactive "noisy Diffie-Hellman", i.e. first Alice sends her public key and then Bob sends a ciphertext. Using her private key, Alice decrypts to approximately the same shared secret that Bob computed using Alice's public key. Additional reconciliation is applied to reduce the probability of decryption failures.

Recall the two main attacker models for public-key encryption schemes in Section 1.2, i.e. the IND-CPA (a possible way of using such schemes is using keys one-time) and IND-CCA setting (keys can be reused/cached). The HILA5 submission [Saa17a] states

> *This design also provides IND-CCA secure KEM-DEM [CS03] public key encryption if used in conjunction with an appropriate AEAD [Rog02] such as NIST approved AES256-GCM [FIP01, Dwo07].*

Ajtai–Dwork [AD97] and NTRU [HPS98] are the oldest lattice-based encryption systems. In lattice-based public-key encryption schemes there is a probability that decryption fails, i.e. it does not output the correct plaintext from the ciphertext. In 1999 Hall, Goldberg, and Schneier [HGS99] developed a reaction attack which recovers the Ajtai–Dwork private key by observing decryption failures for suitably crafted encryptions to the public key. These decryption failures can be observed by e.g. applications that use the public-key encryption scheme to generate a shared secret key between two parties. After a decryption failure, these parties are not able to communicate. Hence, these failures are detectable in certain scenarios. Recall from Section 1.2 that the attacker capabilities in the IND-CCA setting specifically allow for decryption queries and thus allow for detection of decryption failures. This is not allowed in the IND-CPA setting.

Hall, Goldberg, and Schneier wrote "We feel that the existence of these attacks effectively limits these ciphers to theoretical considerations only. That is, any implementation of the ciphers will be subject to the attacks we present and hence not safe." In other words: it highlights the importance to match applications that use these public-key encryption schemes with the correct attacker model.

Hoffstein and Silverman [HS00] adapted the attack to NTRU. As a defense, they suggested modifying NTRU to use the Fujisaki–Okamoto transform [FO99]. For a system without decryption failures, this transform turns a CPA-secure system into a CCA-secure one. At the same time this complicates and slows down the cryptosystem. For NTRU, the transform turns out to still allow attacks that exploit occasional decryption failures induced by *valid* ciphertexts; see [HNP$^+$03].

New Hope [ADPS16] allows occasional decryption failures for valid ciphertexts, and explicitly avoids the "changes" that would be required for the Fujisaki–Okamoto transform. To prevent any model-mismatch attacks (e.g. reaction attacks or other chosen-ciphertext attacks) by a malicious Bob, New Hope requires using keys one-time only (i.e. IND-CPA setting), meaning keys that change with every execution of the protocol. The New Hope paper warns that reusing a public key in multiple protocol runs ("key caching") would be "disastrous for security", although it does not describe an attack.

Fluhrer [Flu16] showed the details of how to attack the key reuse in a similar key-exchange protocol. Followup work [DAS$^+$17] extended the attack to more key-exchange protocols.

HILA5 is similar to New Hope, and still does not use the Fujisaki–Okamoto transform. HILA5 includes an error-correction method that practically eliminates decryption failures for valid ciphertexts. HILA5 does not warn against key caching: on the contrary, the most natural interpretation of the HILA5 security claims is that HILA5 is secure against chosen-ciphertext attacks (i.e. IND-CCA).

Summary. In this chapter we show a reaction attack on HILA5: We compute Alice's private key by sending her multiple encapsulation messages and using her answers to determine whether her decapsulated shared secret matches a certain guess or not. Our attack works independently of whether an AEAD is used or not and despite the error correcting code introduced in HILA5. This shows that the correct claim for the attacker model in HILA5 should have been IND-CPA at best.

We have fully implemented our attack and experimentally verified that it works with high probability. We use the HILA5 reference implementation for Alice's part and also to verify that the retrieved private key works for decryption. We use a slightly modified version of the same software for computations on the attacker's side; of course the attacker need not follow the computations an honest party would.

Organization. In Section 4.2 we describe the necessary preliminaries for this chapter. Specifically, we introduce the relevant parts of the HILA5 scheme and give more details on Fluhrer's attack. In Section 4.3, we describe how we circumvent the error-correcting code and how to adapt Fluhrer's attack to the HILA5 case. Finally, in Section 4.4 we discuss candidate countermeasures.

4.2 — Preliminaries

Although HILA5 is a lattice-based encryption scheme, we do not formally introduce lattices (other than defining the polynomial ring R below) yet as this is not relevant to understand the attack in this chapter. For an introduction to lattices, we defer to Section 6.2. This section describes the HILA5 scheme and Fluhrer's attack on RLWE schemes.

4.2.1 – The HILA5 scheme. We describe the scheme as given in [Saa17a, Section 4.9] but leave out formatting and NTT conversions. These are used in the attack implementation to interface with the reference implementation but do not contribute to the security and hamper readability.

The major computations take place in the polynomial ring $R = \mathbb{Z}_q[x]/(x^n + 1)$, where $n = 1024$ and $q = 12289$. Alice's private key is a small, random polynomial $a \in R$, where small (here and in the following) means that the coefficients are chosen from a narrow distribution around zero, more precisely the discrete binomial distribution Ψ_{16} which has integer values in $[-16, 16]$. To compute the public key she picks another small random polynomial $e \in R$ and a random $g \in R$ and computes $A = ga + e$. She publishes (g, A) and keeps a as her private key.

An honest Bob picks two random small polynomials $b, e' \in R$ and computes $B = gb + e'$ and $y = Ab$. Bob sends B to Alice. The second value

$$y = Ab = (ga + e)b = gab + eb \approx gab$$

is very close to what Alice can compute using her secret and B:

$$x = aB = a(gb + e') = gab + e'a \approx gab,$$

because a, b, e, e' are all small.

A simple rounding operation to achieve a shared secret, such as taking the top bits of each coefficient, will induce differences between Alice's and Bob's version with too high probability. For example, Bob could take $k[i] = \lfloor 2y[i]/q \rceil$ and Alice could take $k'[i] = \lfloor 2x[i]/q \rceil$, where we use $t[i]$ to denote the ith coefficient of polynomial or vector t, but for indices with $(gab)[i] \approx 0$ (or $q/2$) the error-terms can cause the values to flip to a different bit, i.e., $k[i] \neq k'[i]$. For this rounding operation, we call elements of $\{0, q/2\}$ the "edges", as these are the values for which it is probable that errors occur.

This is why Bob sends a second vector, a binary reconciliation vector c, to help Alice recover the same k as Bob. Basically, this means that the scheme uses two pairs of edges. If $y[i]$ was close to one edge of a certain pair, Bob will choose the other pair of edges, so that Alice can still successfully recover the shared secret. In previous work [Pei14], the reconciliation vector achieves a successful shared secret with high probability, as long as $|x[i] - y[i]| < q/8$.

HILA5 differs in how these reconciliation bits are computed. For each coefficient $y[i]$ of y Bob computes $k[i] = \lfloor 2y[i]/q \rceil$, $c[i] \equiv \lfloor 4y[i]/q \rceil \bmod 2$. Additionally Bob computes another reconciliation vector d that should further reduce the probability of decryption failures:

$$d[i] = \begin{cases} 1 & \text{if } |(y[i] \bmod \lfloor q/4 \rceil) - \lfloor q/8 \rfloor| \leqslant \beta \\ 0 & \text{otherwise,} \end{cases}$$

where $\beta = 799$. Positions with $d[i] = 1$ are those for which it is likely that Alice and Bob recover the same value. In other words, for these indices the value $(gab)[i]$ is likely to

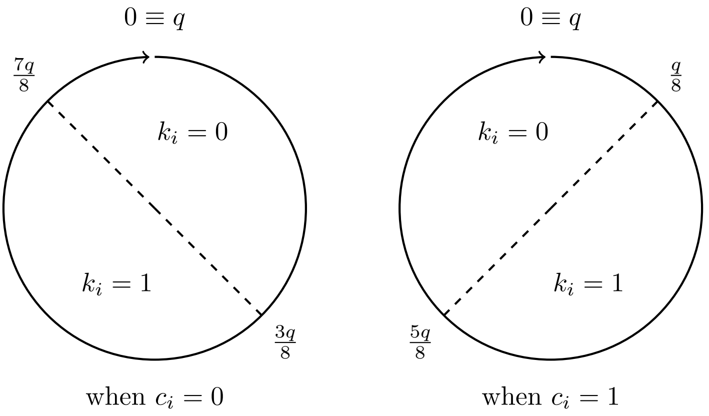


Figure 4.1: Two different mappings in HILA5 from values in $[0, q)$ to bits $\{0, 1\}$: the left mapping is used by Alice when entry $c_i = 0$ and the right map is used when $c_i = 1$. The dashed lines are called the "edges".

be far away from an edge, thus further reducing the probability of errors in the shared secret. He then selects the first 496 positions i for which $d[i] = 1$ and restarts with fresh b and e' if there are fewer. (Note that the description suggests to discard some positions if there are more than 496 such positions while the code deterministically discards the later ones by setting $d[j] = 0$ for them.)

The encapsulation consists of B, d, c, and an extra part r described below; here d covers the full n positions while c can be compressed to those positions i where $d[i] = 1$.

Alice recovers the $k[i]$ at the selected 496 positions by computing

$$k'[i] = \lfloor 2\,(x[i] - c[i] \cdot \lfloor q/4 \rceil + \lfloor q/8 \rceil \bmod q)/q \rfloor.$$

In Figure 4.1 a visualization of this mapping by Alice is given.

The HILA5 submission shows that $k'[i] = k[i]$ with probability $1 - 2^{-36}$. Let k (resp. k') be the 496-bit string given by the concatenation of the $k[i]$ (resp. $k'[i]$).

The role of r is not well described but the HILA5 design overview says that is an encrypted encoding of a part of k. It is computed by splitting k as $k = m \| z$, where m gets the first 256 bits and z the remaining 240 bits. HILA5 uses a custom-designed error-correcting code XE5 that corrects at least 5 errors to compute a 240-bit checksum s of m and then computes $r = s \oplus z$, where $\oplus$ denotes bitwise addition (XOR).

Alice computes $k' = m' \| z'$, the checksum s' on m', and applies the XE5 error correction to m', s', z' and r to correct m' to m.

4.2.2 – Fluhrer's attack. The chosen-ciphertext attack on HILA5 that we are going to present is a variant of the following attack against key reuse in RLWE-based key exchange protocols presented by Fluhrer in 2016 [Flu16]. This section assumes that Bob computes the $c[i]$ and $k[i]$ in a way similar to the previous section. The $d[i]$ were added in HILA5 and will be considered in the next section.

Recall that Alice's version of the shared secret key is

$$gab + e'a,$$

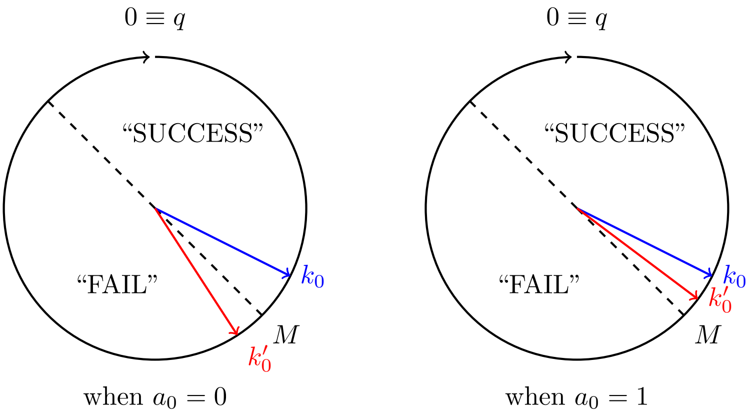


Figure 4.2: Visualization of Fluhrer's attack on HILA5[1]. Evil Bob artificially forces the first coefficient of $(gab)[0]$ to be close to edge M. Depending on the private key value a_0 of Alice, Alice and evil Bob either have the same shared secret (i.e. $k_0 = k'_0$: "SUCCESS") or they do not (i.e. $k_0 \neq k'_0$: "FAIL").

where g is some large public generator element, a and b are Alice's and Bob's small private keys, and e' is a small noise vector chosen by Bob. This version of the shared secret differs from Bob's by some small error, hence they need to employ a reconciliation mechanism to arrive at the same secret bit string.

The general strategy of an evil Bob is to artificially force one (say, the first) coefficient of $ga b$ to be close to the edge M between the intervals that are mapped to bits 0 and 1 during reconciliation. An honest user would set the reconciliation bit $c[0]$ in that case, so Alice would use another mapping that is less likely to produce an error; but evil Bob does not. Since evil Bob proceeds honestly except for the first bit, he knows two possibilities for Alice's private key, hence he can query Alice with one of these guesses and distinguish between 0 and 1 based on her reaction (i.e. a "FAIL" versus "SUCCESS" as depicted in Figure 4.2). If we assume for the moment that evil Bob can choose, hence knows, $(gab)[0]$, this reaction tells him that $(e'a)[0]$ lies in a certain interval.

After a few queries using binary search with varying values for $(gab)[0]$, evil Bob knows the exact distance of $(e'a)[0]$ from the edge, and if he sets $e' = 1$, this distance is nothing but the first coefficient of Alice's private key a. Note that in Fluhrer's setting the edge M is at zero and he uses b with $(gab)[0] = 1$, hence evil Bob can just multiply that b by small distances to obtain a prescribed $(gab)[0]$ when searching for $(e'a)[0]$. In our adaptation of the attack to HILA5, this step is more involved; see Section 4.3.2.

One could apply this method individually to each coefficient to extract Alice's full private key. However, being able to recover the coefficient at one position is enough: due to the structure of the underlying polynomial ring R, evil Bob can shift the ith coefficient of a into the constant term of $e'a$ by setting e' to $-x^{n-i}$, i.e., a vector with one entry of

[1] The original attack of Fluhrer [Flu16] targeted a lattice-based encryption scheme that uses a different rounding mechanism (i.e. the edges are different). For consistency and to avoid confusion, we already visualize this step in our attack for HILA5. However, to get to this part in our attack, Evil Bob has to perform more involved steps, see Section 4.3.2.

-1 and 0 elsewhere. This means evil Bob can apply the method for each coefficient of Alice's private key once.

We now come back to the assumption made above. Notice that evil Bob does not a priori know a vector $b \in R$ such that $(gab)[0] = 1$, but he can still reasonably guess one: Alice's public key is $ga+e$ for small vectors a and e, hence if b is a small low-weight vector such that $(b \cdot (ga + e))[0]$ is close to 1, there is a good chance that in fact $(gab)[0] = 1$. Thus, while evil Bob does not have a deterministic method to find an "evil" b, he can still just make educated guesses based on Alice's public key until he finds one that works. Finding $b \in R$ with $(b \cdot (ga + e))[0]$ close to 1 is an offline computation using only Alice's public key; testing for $(gab)[0] = 1$ requires interaction with Alice.

There are several follow-ups to Fluhrer's paper, e.g. [DAS+17], but a small and new generalization of Fluhrer's attack is sufficient to attack HILA5.

4.3 — Chosen-ciphertext attack on HILA5

In this section, we describe how we circumvent the error-correction code and how to adapt Fluhrer's attack to the HILA5 case.

4.3.1 – Working around error correction. The HILA5 construction includes XE5 as an error-correcting code that is applied to the shared secret after decapsulation. Both Alice and Bob compute their version of a redundancy check, which will help Alice to correct up to 5 errors in the shared secret. The redundancy part r is divided into ten subcodewords $r = r_0, \ldots, r_9$ of variable sizes. For the purpose of the attack, these sizes do not matter, but we use the same notation L_i for the size, as in the HILA5 paper. This means we can index each $r_i = r_{(i,0)} \ldots r_{(i,L_i-1)}$ for $i \in \{0, \ldots, 9\}$.

Bob first computes his part of the HILA5 encapsulation, i.e., he computes his version of the shared secret, selects the indices that are safe to use by Alice and computes the reconciliation vector. The last 240 bits of Bob's shared secret are used in XE5 error-correction. From these bits, Bob constructs his redundancy check r', and sends this as part of the ciphertext.

Upon receiving Bob's ciphertext, Alice first computes her part of the HILA5 decapsulation, i.e., she computes her version of the shared secret. Then she computes her own redundancy check r and computes the distance r^{Δ} with Bob's r' from the ciphertext:

$$r^{\Delta} = r' \oplus r$$

To determine which bits in the shared secret are erroneous, Alice determines a weight $w_\ell^{\Delta} \in [0, 10]$ for each of the 256 bits by the following formula:

$$w_\ell^{\Delta} = r^{\Delta}_{0,\lfloor \ell/16 \rfloor} + \sum_{j=1}^{9} r^{\Delta}_{j,\ell \bmod L_j}$$

Now, if a single bit ℓ of Alice's shared secret is flipped, it means $w_\ell^{\Delta} = 10$ [Saa17a, Lemma 2], and it is therefore detectable and correctable by Alice. Moreover, it is shown that XE5 corrects bit ℓ as long as $w_\ell^{\Delta} \geqslant 6$ [Saa17a, Theorem 1], which means XE5 can correct at least 5 bits in the shared secret. This means that applying Fluhrer's original attack directly to HILA5 will not work, as Fluhrer's original attack depends crucially on the attacker's ability to detect single-bit errors in Alice's version of the shared secret. Thus, to apply Fluhrer's attack, we have to work around these error-correction abilities.

In the attack described in the next section, we focus on inducing errors only in the first bit $\ell = 0$ of the shared secret. This means the attacker evil Bob needs to force w_0^Δ to be less than 6, as this means XE5 is no longer capable of correcting the first bit. However, evil Bob needs to leave the remaining error-correction in place, otherwise he still does not know if the first bit was the only flipped bit. In order to do that, evil Bob needs to change his redundancy check r' to do exactly that. As w_0^Δ is obtained by summing up the first bits of the subcodeword distances r_i^Δ, he can flip any 5 of the bits labeled $r'_{(0,0)}$ through $r'_{(9,0)}$ to force $w_0^\Delta < 6$. Our attack flips the first 5 of these bits. This means in the following section we consider the issue of error-correction solved and can directly apply a modification of Fluhrer's attack.

4.3.2 – Details of the attack. This section elaborates on evil Bob's approach to recover Alice's private key. As mentioned before, the general procedure mimics Fluhrer's attack (Section 4.2.2). The major steps are:

1. Guess a small low-weight secret b_0 such that $(gab_0)[0]$ is at the edge M.
2. For each $\delta \in \{-16, \dots, 16\}$, compute b_δ such that $(gab_\delta)[0] = M + \delta$.
3. For each target coefficient of Alice's private key:
    a) Choose e' such that $(e'a)[0]$ is the target coefficient.
    b) Perform a binary search using the b_δ to recover the target coefficient. (Alice's coefficient $(gab_\delta + e'a)[0]$ maps to a 1 bit iff $(-e'a)[0] > \delta$.)
4. If the results look "bad" after recovering a few coefficients in this way, the guess for b_0 was probably wrong and evil Bob should start over at step 1.

Note that for each oracle query, i.e., for every interaction with Alice, Bob proceeds honestly except for using specially crafted b_δ and e', setting $d_0 = c_0 = 1$, and flipping a few bits in the error correction as described in Section 4.3.1. We now explain and analyze the steps above in more detail.

Forcing coefficients near the edge. In HILA5's reconciliation mechanism, there is no edge at zero for any choice of reconciliation bit, hence Fluhrer's attack does not apply without modifications. We chose to set the reconciliation bit c_0 to 1 and attack the edge at

$$M = \lfloor q/8 \rceil = 1536.$$

To perform the binary search for Alice's private key coefficients in the attack, we need to find small low-weight vectors b_δ such that

$$(gab_\delta)[0] = M + \delta$$

for all δ with $|\delta| \leqslant 16$. (As mentioned in Section 4.2.2, Fluhrer's evil Bob attacked $M = 0$, thus he could guess b_1 based on Alice's public key and set $b_\delta = \delta \cdot b_1$.) One could of course try to guess each b_δ individually based on Alice's public key, but as we want to get all b_δ right at the same time, this has exponentially low success probability. Instead, we make use of a special property of the M used in HILA5: The inverse

$$M^{-1} \bmod q = -8$$

is small.[2] Hence, as soon as evil Bob successfully guessed b_0, he may simply set

$$b_\delta = (1 + \delta M^{-1} \bmod q) \cdot b_0.$$

In our case, we choose b_0 with only two non-zero coefficients from $\{\pm 1\}$, thus b_δ will have only two non-zero coefficients bounded by $1 + 8\delta$. This property is necessary to make sure evil Bob can actually know what Alice's version of the shared secret will be (except for the target bit that leaks information): If the coefficients of b_δ are too large, the error $eb - e'a$ between Alice's and Bob's shared secrets becomes too large to recover from and their shared secrets will mismatch no matter what the value of the attacked bit is. In theory, with these parameters we still expect a tiny possibility of unintended errors, but this happens so rarely that it is not an issue in practice. If it ever does occur, Bob can detect that his recovered shared secret key is wrong and simply start over with a new b_0.

When evil Bob chooses a random b_0 with two non-zero coefficients in $\{\pm 1\}$ and with $(Ab_0)[0] = M$, the probability that in fact $(gab_0)[0] = M$ holds is just the probability that two Ψ_{16}-distributed values sum to zero:

$$\sum_{i=0}^{32} \binom{32}{i}^2 / 2^{64} \approx 9.9\%,$$

hence he can expect to find a good b_0 after about 10 tries. Since A can be approximated by a uniformly distributed sequence over $\mathbb{Z}_q$, the expected number of ± 1-combinations of two coefficients of A which equal M is

$$\binom{1024}{2} \cdot 4/q \approx 170.$$

Hence, the probability that evil Bob exhausts this pool of choices without finding a good b_0 is roughly 2^{-25}.

(If this ever happens, then evil Bob can still try a larger interval, i.e., search for b_0 with $|(Ab_0)[0] - M| \leqslant T$ for some small T. This would in theory work for a wider range of keys, but the expected number of wrong guesses grows slightly. One could also choose three non-zero coefficients in b_0, although this increases the chance of unintended errors in Alice's shared secret. We have not had any problems with $T = 0$ in practice.)

Detecting bad guesses. After choosing b_0 based on Alice's public key as described above, evil Bob may just go ahead and try to recover Alice's private key using that b_0. If it is correct, he will of course find a sequence that looks like it was sampled from the Ψ_{16} distribution. If b_0 is bad, say, $(gab_0)[0] = M + \gamma$ for some small $\gamma \neq 0$, then

$$(gab_\delta)[0] = M + \delta + \gamma - 8\delta\gamma,$$

hence typically $(gab_\delta)[0]$ is considerably smaller than M if $\delta > 0$ and considerably larger if $\delta < 0$; in both cases Alice's part $(e'a)[0]$ is dominated by $\delta + \gamma - 8\delta\gamma$, which means the oracle output does not depend on the secret. This implies the binary search will always converge to 0 or -1 when b_0 is bad. (For $\delta = 0$, the behavior *does* depend on $(e'a)[0]$

[2] Note that this also holds for some other "natural" choices of M as rounded fractions of q, but it is not automatically true for any conceivable M.

since γ is small, so both cases really occur.) Evil Bob can detect this failure mode by determining a few coefficients and checking whether all of them are in $\{0, -1\}$. If this is the case, evil Bob simply starts over with a new b_0. The probability that an actual secret key starts with a sequence of u coefficients from $\{0, -1\}$ is about 0.27^u, hence setting $u = 8$ reduces the probability of a false negative to roughly 2^{-15}. There is a small probability of false positives if evil Bob uses only this heuristic (e. g., when $|\gamma| = 1$), but this can easily be detected using statistical methods (the recovered sequence will not be Ψ_{16}-distributed) or by simply testing the obtained secret key in the end and running the attack again if it failed. In practice the heuristic works fine.

The number of queries. Assuming we already have a good b_0, the binary search needs an expected $5 + \varepsilon$ queries to the oracle to recover one coefficient.[3] Since evil Bob decides whether he has a good b_0 based on the first few coefficients that he obtains using that b_0, he usually wastes a few hundred queries on guesses for b_0 that turn out to be useless: If he looks at the first 8 coefficients obtained from each b_0 as suggested above, this adds expected ≈ 400 queries to the 5120 needed to recover all the coefficients. In summary, evil Bob will with overwhelming probability recover Alice's secret key in less than 6000 queries.

Evil Bob can trade computation for a smaller number of queries: retrieve some coefficients, and reduce the original lattice problem to low enough dimension to solve by computation.

4.3.3 – Implementation. We implemented a proof of concept of the attack in Python, reusing portions of the HILA5 reference implementation via the `ctypes` library. The only modifications we made to the reference implementation were making some functions non-`static` to be able to call them from within Python, and adding extra parameters to the encapsulation function (not used by Alice) such that evil Bob can override his private values b and e'. The complete attack script can be found at `https://helaas.org/hila5-20171218.tar.gz`. As expected, we have never observed the attack script failing to recover Alice's private key. The empirical number of queries matches the theoretical prediction made above.

4.4 — Discussion of candidate countermeasures

A KEM is defined by three algorithms. Key generation produces a private key and a public key. Encapsulation produces a ciphertext and a shared secret key, given a public key. Decapsulation produces a shared secret key or failure, given a ciphertext and a private key. The HILA5 submission document [Saa17a] gives details and reference code for a particular KEM, the "HILA5 KEM".

Our attack is a key-recovery attack against the HILA5 KEM: the attacker, evil Bob, ends up computing the private key of a target Alice. This private key gives the attacker the ability to run the decapsulation algorithm using Alice's private key, and thus the ability to immediately decrypt legitimate ciphertexts sent by other users to Alice.

Our attack is a chosen-ciphertext attack: evil Bob chooses ciphertexts to provide to Alice (different from the legitimate ciphertexts), and learns something from observing

[3] The ε arises from the fact that Ψ_{16} samples from $33 > 2^5$ distinct values, but the extremal values occur so rarely that $\varepsilon \approx 2^{-27}$.

the outputs of Alice decapsulating those ciphertexts. Formally, the attack shows that the HILA5 KEM does not provide IND-CCA2 security.

There are two important ways that the attack does not need the full power of a CCA2 decapsulation oracle. First, the attack is what is called a "reaction attack" in [HGS99] or a "sloppy Alice attack" in [VDvT02]: evil Bob has a guess for the output of each decapsulation, and learns whether Alice's actual decapsulation output matches this guess. Evil Bob does not need any further information.

Second, evil Bob chooses all of his ciphertexts, and learns the private key from Alice's reactions, before seeing the legitimate ciphertexts to decrypt. Formally, the attack shows not only that the HILA5 KEM does not provide IND-CCA2 security, but also that it does not provide IND-CCA1 security.

4.4.1 – Hashing the secret key does not stop the attack. One can easily stop key-recovery attacks by defining HILA5Hash as follows. HILA5Hash key generation computes a uniform random 32-byte string s, and then runs HILA5 key generation to obtain a public key, hashing s to generate all randomness used in HILA5 key generation. The HILA5Hash secret key is s. HILA5Hash encapsulation is the same as HILA5 encapsulation. HILA5Hash decapsulation reconstructs the HILA5 secret key from s (again running the HILA5 key-generation algorithm; alternatively, the HILA5 secret key can be cached), and then runs the HILA5 decapsulation algorithm.

Unless the hash function is easy to invert, a key-recovery attack against HILA5 does not produce a key-recovery attack against HILA5Hash. However, this hashing does not prevent the attacker from decrypting legitimate ciphertexts sent by other users to Alice.

4.4.2 – AEAD does not stop the attack. A PKE is defined by three algorithms. Key generation produces a private key and a public key, as in a KEM. Encryption produces a ciphertext, given a plaintext and a public key. Decryption produces a plaintext or failure, given a ciphertext and a private key.

The subtitle of the HILA5 submission is "Key Encapsulation Mechanism (KEM) and Public Key Encryption Algorithm". The submission document does not include a definition of a PKE, but NIST had already stated before submission that it would automatically convert each submitted KEM to a PKE using the following "standard conversion technique": "appending to the KEM ciphertext, an AES-GCM ciphertext of the plaintext message" where the AES-GCM key is "the symmetric key output by the encapsulate function". This is the standard Cramer–Shoup "KEM-DEM" construction, using AES-GCM as the DEM. We write "HILA5 PKE" for the PKE that NIST will automatically produce in this way from the HILA5 KEM.[4]

Breaking the IND-CCA2 security of a KEM does not necessarily imply breaking the IND-CCA2 security of a PKE obtained in this way. IND-CCA2 attacks against the KEM can see shared keys produced by decapsulation, whereas IND-CCA2 attacks against the PKE are merely able to see the result of AES-GCM decryption using those keys.

However, our attack against the HILA5 KEM is also a key-recovery attack against the HILA5 PKE. It is important here that the attack is a reaction attack: what evil Bob needs

[4] NIST actually deviates slightly from the KEM-DEM construction: it specifies a "randomly generated IV" for AES-GCM, while Cramer and Shoup use a deterministic DEM. For consistency with the ciphertext sizes mentioned in [Saa17a], we actually define "HILA5 PKE" to be the Cramer–Shoup construction using AES-GCM with an all-zero IV. Switching to NIST's construction would expand ciphertext sizes by 12 bytes using the default IV sizes for AES-GCM, and would not affect our attack.

to know is merely whether a guessed shared key is correct. Starting from this guessed shared key, evil Bob produces a valid AES-GCM ciphertext using this guess as an AES key. If decapsulation in fact produces this shared key then AES-GCM decryption succeeds and produces the plaintext that evil Bob started with. If decapsulation produces a different shared key then AES-GCM decryption is practically guaranteed to fail (anything else would be a surprising security flaw in AES-GCM), so evil Bob sees a decryption failure from the PKE.

To summarize, evil Bob sees decryption failures from the PKE, and learns from this which guesses were correct, which is the same information that evil Bob obtains from the KEM. Evil Bob then computes the secret key from this information. Consequently, the HILA5 PKE does not provide IND-CCA2 security, and does not even provide IND-CCA1 security.

4.4.3 – Black holes would stop the attack. Like other chosen-ciphertext attacks, our attack is inapplicable to scenarios where the results of decapsulation and decryption are hidden from the attacker. For example, if ciphertexts are sent to NSA's public key, and if NSA hides the results of applying its secret key to those ciphertexts, then an attacker outside NSA cannot use our attack to compute NSA's secret key. However, if NSA reacts to those results in a way that leaks to the attacker which ciphertexts were valid, then the attacker can compute NSA's secret key.

4.4.4 – The Fujisaki–Okamoto transform would stop the attack. We briefly outline a more radical change to HILA5, which we call "HILA5FO". HILA5FO ciphertexts are slightly larger than HILA5 ciphertexts, encapsulation and decapsulation are more complicated, and decapsulation is extrapolated (from reported HILA5 benchmarks) to be several times slower, but HILA5FO would stop our attack.

The idea of the HILA5FO KEM is to reapply the encapsulation algorithm as part of decapsulation, and check whether the resulting ciphertext is identical to the received ciphertext. This is not a new idea: it is used in many other submissions to NIST (with various differences in details), typically with credit to Fujisaki and Okamoto [FO99].

HILA5 does not provide any easy way to reconstruct the randomness used in encapsulation (most importantly Bob's b), so the HILA5FO KEM computes this randomness as a hash of a plaintext recovered as part of decapsulation. The HILA5 KEM does not transmit a plaintext, so the HILA5FO KEM is instead built from the HILA5 PKE.

Encapsulation in the HILA5FO KEM thus chooses a random plaintext, and encrypts this plaintext using the HILA5 PKE (the HILA5 KEM producing a shared key for AES-GCM) using a hash of the plaintext to compute all randomness used inside the PKE. Decapsulation applies HILA5 PKE decryption (HILA5 KEM decapsulation producing a shared key for AES-GCM decryption), and checks that the resulting plaintext produces the same ciphertext.

Deriving a PKE from the HILA5FO KEM would involve two layers of AES-GCM, which can be compressed to one layer as follows: place 32 bytes of randomness at the beginning of the user-supplied plaintext, and then encrypt this plaintext using the HILA5 PKE, again using a hash of the plaintext to compute all randomness used inside the PKE. The overall ciphertext size is the original plaintext size, plus 32 bytes (the randomness), plus the HILA5 KEM ciphertext size, plus 16 bytes (the AES-GCM authenticator), i.e., 32 bytes more than the HILA5 PKE.

CHAPTER 5

Conclusions and future work

We now revisit the research questions posed in Chapters 2 to 4 of this thesis.

Q1: Is the sponge construction (and thus hash functions like SHA3) collapsing?

We have shown in Chapter 2 that the sponge construction is indeed collapsing, when the internal function is a random function or a random permutation. We have shown that the required properties of the internal functions are (left- and right-)collapsing and zero-preimage resistance. We first showed that a certain step function is collapsing. This step function is then used in the hybrid argument, which ultimately proves that the whole construction is collapsing. Based on the complexity of attacks on the internal function, we can give bounds on attacks on the sponge construction, thereby finishing the security reduction. The results of this chapter strengthen the confidence in the sponge construction, as this is another result that shows that quantum attackers do not outperform classical attacks by much more than a square-root speedup.

A very recently published follow-up of this work [CHS19] proves that the sponge construction, under the same setting as in the chapter, is actually indistinguishable from a random oracle. This work also proves that a sponge, with a non-trivial inner part (i.e. instantiated with a symmetric key), can be used to build a quantum-secure CBC-MAC.

Q2: What can we say about the security of hash-based one-time signatures, when a user accidentally signs twice?

In Chapter 3 we analyzed the security of the most prominent hash-based OTS – Lamport's scheme, its optimized variant, and WOTS – under different kinds of two-message attacks. For some attacks we could make a tradeoff between pre-computation and online computation. In fact, for most attacks there is no requirement to know the public key, meaning that once an attacker found two messages with the right properties, he can do an attack on any hash-based OTS key-pair. Interestingly, it turns out that the schemes are still secure under two message attacks, at least asymptotically. However, this does not imply anything for typical parameters. Our results show that for Lamport's scheme, security only slowly degrades in the relevant attack scenarios and typical parameters are still somewhat secure, even in case of a two-message attack. As we move on to optimized Lamport and its generalization WOTS, security degrades faster and faster, and typical parameters do not provide any reasonable level of security under two-message attacks.

We do not advocate signing two messages with any OTS, despite the asymptotic results in this chapter.

A direct application of this work is the analysis of fault attacks on hash-based signatures [CMP18]: the introduced faults force a key-reuse of an OTS key-pair. Our work is also cited on a Request for Comments page by NIST [NIS19], where stateful hash-based signature schemes are considered for standardization.

Q3: Can error-correction prevent the reaction attack?

In Chapter 4 we analyzed whether the error-correction technique in HILA5 can stop reaction attacks. We answered this negatively by performing a modified version of Fluhrer's attack on HILA5. As an evil Bob can modify his share of the key-exchange, he can circumvent the error-correcting capabilities of Alice by injecting certain errors in the error-correction code. Once this error-correcting code has been neutralized, the reaction attack can continue. Evil Bob sends Alice multiple encapsulated messages and using her answers can determine whether her shared secrets match a certain guess or not. By means of a binary search, these guesses turn into the recovery of the whole secret key, coefficient by coefficient. This work shows that it is very important to determine the appropriate security model for a public-key encryption scheme like HILA5 that matches the use in practice (i.e. one-time keys versus cached keys).

Shortly after publication of these results, the author of HILA5 implied that he should indeed have been claiming only IND-CPA security instead of IND-CCA[1]. The HILA5 submission to NIST was later merged into Round5 [BBF$^+$17], which specifies several IND-CPA and IND-CCA schemes.

Open problems

We end Part I on this thesis by posing some open problems that remain after the questions answered in Chapters 2 to 4.

I: Can we prove that the sponge construction is collapsing with an invertible permutation as internal function?

A big open question in the context of Chapter 2 is whether we can prove that the sponge construction is collapsing when an efficiently invertible permutation is used as an internal function. This is an important question, as this will show that the hash function SHA3 is collapsing. Unfortunately, our results (and the follow-up work [CHS19]) does not imply this quite yet. Proving the property of collapsing with invertible permutations most definitely requires a different strategy than in our proof, as we have already concluded that the internal function does not have the required properties when it is efficiently invertible. However, it could still be that a different property of the internal function does suffice to show that the sponge construction is collapsing.

[1] Discussed on the PQC-forum: `https://groups.google.com/a/list.nist.gov/d/msg/pqc-forum/_3ZyCahlBJo/AwYySk-mBQAJ`

II: Can we improve the analysis for the two-message attacks on WOTS by removing the independence assumption on message and checksum part?

In Chapter 3, we made an important assumption in the analysis of two-message attacks on WOTS: we assumed that both message and checksum are independently chosen at random. However, the checksum is completely determined by the message, which immediately falsifies our assumption. Or, to turn it around, when a checksum is given, there is only a specific subset of messages that have that checksum. The good thing is that the attacks should only get harder when this restriction is handled, which means that our work can provide upper bounds for attacks. Still, it remains an open question whether this assumption can be removed or replaced by some other analysis. An additional question is whether such an improved analysis can help improve the EU-CMA attack. If the attacks would focus on covering one part with two messages, be it message part or checksum part, it might be that this also improves the running time to find forgeries. Another improvement might be in the analysis of the checksum itself: the higher order bits of the checksum are usually mostly zero, which means our assumption that all bits of the checksum are uniformly random is also not matching reality. We have not taken this into account.

III: Is it possible to build a tight IND-CCA encryption scheme based on lattices?

In Chapter 4, we showed several fixes to the reaction attack on HILA5. The most obvious choice would be to use the FO transform [FO99], that turns any IND-CPA scheme into a IND-CCA version. However, the downside in this method is that this adds a non-tight layer: there is a security loss in this transform when this is done in the QROM, i.e. dealing with quantum attackers [TU16]. This means the keys and ciphertext need to be bigger than in the initial scheme to deal with this loss. There have been several works [SXY18, JZC$^+$18, BHHP19] trying to remove this security loss, but it comes with additional assumptions on the scheme (i.e. the tight versions work only with deterministic schemes that are OW-CPA). It is therefore an open question how to build a tight IND-CCA encryption scheme based on an IND-CPA lattice-based encryption scheme. As we can learn from the HILA5 case, it might be best practice to only use IND-CCA encryption schemes in deployed cryptography, as these model-mismatch attacks (e.g. reaction attacks) can easily occur by software mistakes.

PART II

SIDE-CHANNEL ATTACKS

CHAPTER 6

Implementations in post-quantum cryptography

6.1 — Problem description

In Part I of this thesis, we have looked at several research problems concerning model-mismatch attacks in post-quantum cryptography: attacks that can occur in the case of e.g. accidental wrong usage of a primitive by applications. However, there is another large category of attacks possible when applications use cryptography; and, in the digital world of today, everyone uses cryptography: to secure online banking, emails, messaging apps, log in to social media, and many more. The fact that there exist many applications, also means many different devices use cryptography: laptops, phones, smartcards, and many "new connected devices" will follow. All these devices and applications are potentially vulnerable to so-called side-channel attacks: attacks that make use of the fact that devices leak physical information, potentially including cryptographic values that are supposed to be kept secret. These threats do not disappear when everyone starts using post-quantum cryptography.

A particularly interesting area in post-quantum cryptography is lattice-based cryptography. We have already seen HILA5 in Chapter 4, but there exist many more efficient lattice-based proposals for signatures, encryption, and key-exchange. Modern lattice-based cryptography has also already seen (limited) real-world evaluation, e.g., the experiments with the NewHope [ADPS16] key-exchange and more recently the NTRU-HRSS [HRSS17] Key Encapsulation Mechanism by Google [Bra16, Lan16, Lan18], the implementations of NTRU [HPS98] and BLISS [DDLL13b] in strongSwan [str15] and the addition of NTRU Prime [BCLvV17] in OpenSSH [Ayi19]. While the theoretical and practical security of these schemes is under active research[1], security of implementations is still a largely unexplored area. This is the main focus of Part II of this thesis.

6.2 — Lattice-based cryptography

6.2.1 – Lattices. We define a *lattice* Λ as a discrete subgroup of $\mathbb{R}^n$: given $m \leqslant n$ linearly independent vectors $\mathbf{b}_1, \ldots, \mathbf{b}_m \in \mathbb{R}^n$, the lattice Λ is given by the set $\Lambda(\mathbf{b}_1, \ldots, \mathbf{b}_m)$ of all integer linear combinations of the $\mathbf{b}_i$'s:

$$\Lambda(\mathbf{b}_1, \ldots, \mathbf{b}_m) = \left\{ \sum_{i=1}^{m} x_i \mathbf{b}_i \mid x_i \in \mathbb{Z} \right\}.$$

[1] Official discussion forum of the NIST 'competition' at `https://groups.google.com/a/list.nist.gov/forum/#!forum/pqc-forum`

We call $\{\mathbf{b}_1, \dots, \mathbf{b}_m\}$ a basis of Λ and define m as the rank. We represent the basis as a matrix $\mathbf{B} = (\mathbf{b}_1, \dots, \mathbf{b}_m)$, which contains the vectors $\mathbf{b}_i$ as column vectors. In this thesis, we mostly consider full-rank lattices, i.e. $m = n$, unless stated otherwise. A basis for a lattice is not unique: given a basis $\mathbf{B} \in \mathbb{R}^{n \times n}$ of a full-rank lattice Λ, we can apply any unimodular transformation matrix $\mathbf{U} \in \mathbb{Z}^{n \times n}$ and $\mathbf{UB}$ will also be a basis of Λ. The characteristics of a basis are important in lattice-based cryptography, see Section 6.2.2. We define a "good basis" and a "bad basis" for a lattice Λ: a good basis is a basis that has relatively short and orthogonal vectors, whilst a bad basis has long vectors and describes a very thin parallelepiped. Here, short and long are defined using some norm $||\cdot||$, where we usually use the Euclidean norm denoted with $||\cdot||_2$.

The LLL algorithm [LLL82], and the BKZ algorithm and its improved versions [CN11], transform a basis $\mathbf{B}$ to its LLL/BKZ-reduced basis $\mathbf{B}'$ in polynomial time[2]: a reduced basis has relatively short and orthogonal vectors. The LLL and BKZ algorithms can therefore be seen as algorithms that improve the lattice basis, although there is usually still a gap between the LLL/BKZ-reduced basis and a "good basis" (especially for random high-dimensional lattices). In particular, in an LLL/BKZ-reduced basis the shortest vector $\mathbf{v}$ of $\mathbf{B}'$ satisfies $||\mathbf{v}||_2 \leqslant 2^{\frac{n-1}{4}}(|\det(\mathbf{B})|)^{1/n}$ and there are looser bounds for the other basis vectors. Besides the LLL/BKZ-reduced basis, NTL's [Sho15] implementation of LLL and BKZ also returns the unimodular transformation matrix $\mathbf{U}$, satisfying $\mathbf{UB} = \mathbf{B}'$.

An *integer lattice* is a lattice for which the basis vectors are in $\mathbb{Z}^n$. For integer lattices it makes sense to consider elements modulo q, so coefficients are taken from $\mathbb{Z}_q$ and thus vectors from $\mathbb{Z}_q^n$. On top of that, more efficiency can be gained to use, e.g., cyclic lattices: these lattices have a basis, whose vectors are rotations of one single vector. A generalization of a cyclic lattice is called an ideal lattice: they are lattices corresponding to ideals in polynomial rings of the form $\mathcal{R} = \mathbb{Z}[x]/\langle h \rangle$ for some irreducible polynomial h of degree n (see e.g. [LPR10] for more background on ideal-lattice-based cryptography). Note that being cyclic adds additional structure to these lattices which normal lattices do not have. In cryptography often the ring $\mathcal{R}_q = \mathbb{Z}_q[x]/(x^n \pm 1)$ is used, which has the additional benefit that it inherits efficient polynomial arithmetic via the Number Theoretic Transform (NTT). The elements in $\mathcal{R} = \mathbb{Z}[x]/(x^n + 1)$ can be represented as polynomials of degree less than n. For each polynomial $f(x) \in \mathcal{R}$ we define the corresponding vector of coefficients as $\mathbf{f} = (f_0, f_1, \dots, f_{n-1})$. Addition of polynomials $f(x) + g(x)$ corresponds to addition of their coefficient vectors $\mathbf{f} + \mathbf{g}$. Multiplication of $f(x) \cdot g(x) \bmod (x^n + 1)$ defines a multiplication operation on the vectors $\mathbf{f} \cdot \mathbf{g} = \mathbf{gF} = \mathbf{fG}$, where $\mathbf{F}, \mathbf{G} \in \mathbb{Z}^{n \times n}$ are matrices, whose columns are the rotations of (the coefficient vectors of) $\mathbf{f}, \mathbf{g}$, with signs matching the reduction modulo $x^n + 1$. Lattices using polynomials modulo $x^n + 1$ are often called *NTRU lattices* after the NTRU encryption scheme [HPS98]. When we work in $\mathcal{R}_q = \mathbb{Z}_q[x]/(x^n + 1)$ (or $\mathcal{R}_{2q}$), we assume n is a power of 2 and q is a prime bigger than 2.

6.2.2 – Hard lattice problems. Two fundamental problems on lattices are the shortest vector problem and the closest vector problem. The shortest vector problem (SVP) is given by the following definition: *given a basis $\boldsymbol{B}$ of a lattice Λ, find an $\mathbf{s} \in \Lambda$ such that* $||\mathbf{s}|| = \lambda_1(\Lambda) = \min_{\boldsymbol{v} \in \Lambda, \boldsymbol{v} \neq \mathbf{0}} ||\boldsymbol{v}||$. The closest vector problem (CVP) is defined by the following definition: *given a basis $\boldsymbol{B}$ of a lattice Λ and a target vector $\mathbf{t} \in \mathbb{R}^n$, find a $\mathbf{y} \in \Lambda$ such*

[2] BKZ has a subroutine that does not run in polynomial time, i.e. depending on the so-called blocksize BKZ's subroutine has exponential running time.

that $\|\mathbf{y} - \mathbf{t}\| = \min_{\boldsymbol{v} \in \Lambda} \|\boldsymbol{v} - \mathbf{t}\|$. In cryptography, often the approximate versions of these problems (i.e. SVP_γ and CVP_γ) are used with a so-called approximation factor $\gamma > 1$: solutions to the above problems within a factor γ suffice as a solution (e.g. a short vector $\tilde{\mathbf{s}} \in \Lambda, \tilde{\mathbf{s}} \neq \mathbf{0}$ with $\|\tilde{\mathbf{s}}\| \leqslant \gamma\lambda_1(\Lambda)$ for SVP_γ). In general these problems are easy to solve given a "good basis" $\mathbf{B}$ of Λ, but are hard when a "bad basis" $\mathbf{B}'$ is given of the same lattice. Although LLL and BKZ can improve the bad basis, the approximation factor that is left after applying these algorithms is too large to break lattice-based cryptography. Hence, these lattice problems could be used as the trapdoor one-way function that allows to build public-key cryptography: a bad basis of the lattice is given as the public key and a good basis is the accompanying secret key.

However, the more modern lattice-based cryptosystems do not use the lattice problems straight-away as the trapdoor. Instead, they use problems that are related to SVP and CVP. Most notably, Regev [Reg09] introduced the Learning-with-Errors problem: given multiple noisy inner-products $(\mathbf{a}, \langle \mathbf{a}, \mathbf{s} \rangle + e \bmod q)$, find $\mathbf{s}$. Here, $\mathbf{a} \in \mathbb{Z}_q^n$ is random and $e \in \mathbb{Z}$ (and possibly also $\mathbf{s} \in \mathbb{Z}_q^n$) is randomly generated according to some narrow error distribution D defined over $\mathbb{Z}$. When $D = D_\sigma$ is the discrete Gaussian distribution (we give a definition of this distribution in Chapter 8) centered around 0 with parameter $\sigma > 0$, then specific LWE instances have a (quantum) reduction from a closest vector problem [Reg09]. Although the originally proposed primitives based on LWE had quite large outputs (e.g. keys and ciphertexts), improved versions exist that are based on rings (ring-LWE), i.e. instead of defining LWE over $\mathbb{Z}_q$ it is defined over, e.g., the polynomial ring $\mathcal{R}_q$ as introduced in the previous section. Recall that such polynomial rings have two benefits: the underlying lattice is an ideal (e.g. cyclic) lattice and there are efficient algorithms for polynomial multiplication using the NTT. This leads to both smaller and more efficient schemes than those based on plain LWE. A problem related to LWE is called the Short Integer Solution (SIS) problem: given $\mathbf{A} \in \mathbb{Z}_q^{m \times n}$ and a target vector $\mathbf{t} \in \mathbb{Z}_q^m$ (can be the all-zero vector), find a small vector $\mathbf{s} \in \mathbb{Z}_q^n$ such that $\mathbf{A}\mathbf{s} \equiv \mathbf{t} \bmod q$. Many lattice-based signature schemes use a combination of (ring-)LWE and (ring-)SIS. In Part II of this thesis we focus on several implementation problems for lattice-based signature schemes.

6.3 — Side-channel attacks

6.3.1 – Introduction. To break (public-key) cryptography, it is not always necessary to break the underlying hard mathematical problem (mathematical cryptanalysis): there is a gap between the mathematical theories on the white board and the security of their cryptographic implementations. Side-channel attacks use the fact that real-world devices potentially leak physical information: in the form of EM-radiation, memory-access patterns or in general the execution time of an algorithm. If such a device was performing a cryptographic operation and these physical leakages depend on secret values, the side-channel information might allow to break the cryptographic primitive. Side-channel attacks have shown to be very effective in breaking real-world security, such as the widely used internet protocol SSL/TLS [AP13] or email-encryption software like GnuPG [GST14]. Hence, these attacks must always be considered when cryptography is implemented in practice. In this thesis, we focus on two types of side-channel attacks: cache attacks and fault attacks, introduced in the following sections.

6.3.2 – Cache attacks. The cache is a small bank of memory which exploits the temporal and the spatial locality of memory access to bridge the speed gap between the faster processor and the slower memory. The cache consists of *cache lines*, which, on modern architectures like the Intel architecture we tested, can store an aligned *block* of memory of size 64 bytes. In a typical processor there are several cache *levels*. At the top level, closest to the execution core, is the *L1 cache*, which is the smallest and the fastest of the hierarchy. Each successive level (L2, L3, etc.) is bigger and slower than the preceding level: this hierarchy is caused by the steeply growing cost for fast memory. When the processor accesses a memory address it looks for the block containing the address in the L1 cache. In a *cache hit*, the block is found in the cache and the data is accessed. Otherwise, in a *cache miss*, the search continues on lower levels, eventually retrieving the memory block from the lower levels or from the memory. The cache then *evicts* a cache line and replaces its contents with the retrieved block, allowing faster future access to the block. Because cache misses require searches in lower cache levels, they are slower than cache hits. Cache timing attacks exploit this timing difference to gain information [Ber05, Per05, OST06, GBK11, LYG$^+$15]. In a nutshell, when an attacker uses the same cache as a victim, victim memory accesses change the state of the cache. The attacker can then use the timing variations to check which memory blocks are cached and from that deduce which memory addresses the victim has accessed. Ultimately, the attacker learns the cache line of the victim's table access: a range of possible values for the index of the access.

In this thesis we focus on the FLUSH+RELOAD attack [YF14, GBK11]. A FLUSH+RELOAD attack uses the `clflush` instruction of the x86-64 architecture to evict a memory block from the cache. The attacker then lets the victim execute some operation including a load before measuring the time to access some memory block. If during its execution the victim has accessed an address within the block, the block will be cached and the attacker's access will be fast. If, however, the victim has not accessed the block, the attacker will cause the processor to reload the block from memory, and the access will take much longer. Thus, the attacker learns whether the victim accessed the memory block during that operation. The FLUSH+RELOAD attack has been used to attack implementations of RSA [YF14], AES [GBK11, AIES14], ECDSA [YB14, vdPSY15] and other software [ZJRR14, GSM15], before.

6.3.3 – Fault attacks. Fault attacks are active attacks, where an adversary injects faults during the execution of cryptographic algorithms and caused a different execution; in contrast, cache attacks do not change the (outcome of the) victim's execution itself. The injected faults may lead to unintended behaviour of the execution of the algorithm, potentially changing the outcome of the primitive. These faults can be injected in the device by, e.g., clock glitching or any type of rare sequence of events. There are several targets to inject a fault in a calculation, leading to various attack scenarios. For example, a loop-abort fault attack simply aborts a procedure with a loop before its intended end. If this would happen during a loop that ensures the security of secret values, these values are potentially output in plain because of the abort. Another type is a differential fault attack, that requires two (or more) executions of the cryptographic algorithm. By injecting a fault during one of the executions, the attack tries to gather information from the difference of the two outputs which could allow to break the primitive. These attacks have been applied to various implementations of cryptography, but up to recently did require physical access to the device to inject the fault. The so-called Rowhammer attack [GMM16] is a remote fault-attack, thereby increasing the applicability and impact of fault attacks tremendously.

6.4 — Challenges and research questions

In Part II of the thesis, we will look at several implementation attacks, with a focus on lattice-based schemes. We now present the research questions that will be answered in this part, accompanying the following four chapters.

Q1: How much does left-to-right sliding window exponentiation hurt security?

As an intermezzo, we will first look at possible side-channel attacks on deployed cryptography based on RSA. It is well known that constant-time implementations of modular exponentiation in RSA cannot use sliding windows. However, software libraries such as Libgcrypt, used by GnuPG, continue to use sliding windows. The reason is that it is widely believed that, even if the complete pattern of squarings and multiplications is observed through a side-channel attack, the number of exponent bits leaked is not sufficient to carry out a full key-recovery attack against RSA. Specifically, a 4-bit sliding windows implementation leaks only 40% of the bits, and 5-bit sliding windows leak only 33% of the bits. But is this really the case or can we actually retrieve more bits? We investigate this possibility in Chapter 7.

Q2: Are side-channel attacks possible on the discrete Gaussian sampler?

It is not always trivial to perform a side-channel attack, even in the presence of information leakage. Many lattice-based schemes, especially digital signature schemes, use noise sampled from a discrete Gaussian distribution to achieve a tight security reduction and high efficiency. This is often coupled with rejection sampling: occasionally rejecting signatures to remove the dependency between signatures and the secret key. A very promising signature scheme implementing these techniques is called BLISS [DDLL13b]. However, it is not straightforward to securely implement samplers for the discrete Gaussian distribution. This makes it a good target for a side-channel attack. In Chapter 8, we investigate whether side-channel attacks are possible on the discrete Gaussian sampler.

Q3: Is BLISS-B a free countermeasure against side-channel attacks?

In Q2, that is answered in Chapter 8, we investigate the applicability of side-channel attacks on BLISS, targeting samplers for the discrete Gaussian distribution. However, there is an upgraded version called BLISS-B, which in particular does not seem vulnerable to the attacks on BLISS developed in Chapter 8. This is due to the fact that BLISS-B signatures hide more information than the original BLISS, including information that could be crucial for an attacker. Hence, any attack on BLISS is not naturally extended to BLISS-B, on the contrary: this could mean that using BLISS-B instead of BLISS is an easy countermeasure against side-channel attacks. As it is superior to BLISS, the upgraded BLISS-B has been implemented in e.g. strongSwan [str15]. Another important problem in the applicability of cache-attacks in real-life scenarios is the issue of synchronization: what if the victim is not synchronized with the attacker? It is not known how to solve that problem for the side-channel attacks on lattice-based schemes. In Chapter 9 we investigate whether we can a perform practical asynchronous cache attack on BLISS-B.

Q4: Do deterministic versions of lattice-based signature schemes hurt security?

As it is not straightforward to sample from a discrete Gaussian distribution, it might be a better option to remove such distributions entirely from lattice-based signature schemes: noise from the uniform distribution can also be used, at the expense of bigger signature sizes. On top of that the lattice-based scheme can be made completely deterministic: i.e. given a secret key and a message, there is a unique signature output by the signature algorithm. While this is a good strategy against devices that do not have access to good randomness generators, it opens the door to another category of attacks: differential fault attacks. These attacks have been shown to be successful against deterministic elliptic-curve based signature schemes, but are these attacks also possible on deterministic lattice-based schemes? This is investigated in Chapter 10 of the thesis.

CHAPTER 7

Sliding right into disaster

7.1 — Overview

Context. In this intermezzo chapter we take a brief step out of the realm of post-quantum cryptography and look at deployed implementations of RSA, motivating the importance of the topic of side-channel attacks. Modular exponentiation in cryptosystems such as RSA is typically performed using a square-and-multiply sequence that traverses over the bits of the exponent. There are two ways to do this: either starting from the most significant bit (MSB) in a left-to-right manner or starting from the least significant bit (LSB) in a right-to-left manner. To increase performance, most practical RSA implementations use the Chinese remainder theorem (CRT) to speed up the decryption operation, as this halves both the size of the exponent as well as the modulus. As multiplication is the most time-consuming operation, implementations can furthermore use precomputed values to decrease the number of multiplications in the exponentiation: instead of traversing over the key bit-by-bit, multiple bits are grouped together in a so-called window and one multiplication is done for all the bits covered in the window. There are two options for the window methods. The fixed window method simply takes a fixed number w of consecutive bits in one window and puts the next window adjacent to it. In the exponentiation, it performs one multiplication every w bits with the value depending on the bits in the window. The sliding window method permits gaps consisting of 0-bits between windows and performs a squaring for these bits instead. On average this takes fewer multiplications. Typically these sliding window methods are described in a right-to-left manner, starting with the encoding of the exponent from the least significant bit, leading to the potential disadvantage that the exponent has to be parsed twice: once for the encoding and once for the exponentiation.

This motivated researchers to develop left-to-right analogues of the integer encoding methods that can be integrated directly with left-to-right exponentiation methods. For example, the only method for sliding-window exponentiation in the Handbook of Applied Cryptography [MvOV96, Chap 14.6] is the left-to-right version of the algorithm. Doche [Doc05] writes "To enable 'on the fly' recoding, which is particularly interesting for hardware applications" in reference to Joye and Yen's [JY00] left-to-right algorithm.

Given these endorsements, it is no surprise that many implementations chose a left-to-right method of encoding of the exponent. For example, Libgcrypt implements a left-to-right exponentiation with integrated encoding. Libgcrypt is part of the GnuPG code base [GNUa], and is used in particular by GnuPG 2.x, which is a very popular implementation of the OpenPGP standard [CDF$^+$07] for applications such as encrypted email and files. Libgcrypt is also used by various other applications; see [Gnub] for a list of frontends.

The attack target. In order to compute an RSA signature (or perform an RSA decryption), a modular exponentiation must be performed: given inputs base b, secret exponent d and modulus p, compute $b^d \bmod p$. This is typically performed using a square-and-multiply operation that traverses over the bits of $d = d_{n-1} \dots d_0$. Typically about half of the bits of d are 1, leading to about $n/2$ time-consuming multiplications.

To speed up the modular exponentiation, the number of multiplications is reduced by using a small table of pre-computed values: instead of performing the multiplications bit-by-bit, a group of w bits of d are grouped together to perform one “big” multiplication using the pre-computed values (we explain what these pre-computed values are in Section 7.2). To do this, first the sliding window form of d has to be computed: this is an encoding of the bits using windows of size w. Such a method can be done left-to-right or right-to-left, but the left-to-right method is the only version that can do encoding and multiplication “on the fly”: it traverses the bits to find the correct windows and performs the appropriate multiplications and squarings immediately. A right-to-left method should first group the bits of d into windows and then traverse the windows a second time to do the squares and multiplications. The exponentiation requires on average $n/(w + 1)$ multiplications.

Since a multiplication is the most time-consuming operation, a side-channel attack could recover the sequence of squares and multiplies, independent of the direction of left-to-right versus right-to-left. For example, a cache-attack like the FLUSH+RELOAD attack as discussed in Section 6.3 is able to recover this sequence. An additional benefit for sliding-window methods is that a side-channel attack targeting the square-and-multiply sequence recovers less information for bigger w. It might recover that there was a multiplication for a certain window, but cannot recover the individual value contained in the window. On average only 1 out of w bits can be recovered from a window, which is a certain $d_i = 1$ depending on the location of the multiplication. Such a side-channel attack tracing the positions of the multiplications could actually recover a bit more: the lack of multiplications during the processing of some bits also tells that certain bits $d_i = 0$. This leads to the common belief of “security under leakage” for these sliding-window methods. For window width w only about a fraction $2/(w+1)$ bits would leak: each window has 1 bit known to be 1, and each gap (i.e. a number of processed bits without multiplication) has on average 1 bit known to be 0, compared to $w + 1$ bits occupied on average by the window and the gap. Typical implementations (including Libgcrypt) use $w = 4$ or $w = 5$ depending on the size of the RSA modulus, leading to 40% and 33% recoverable bits by a side-channel attack. This is too few to use the key-recovery attack [HS09] by Heninger and Shacham: this method requires at least 50% of the bits known and its running time grows exponentially for each additional unknown bit.

Libgcrypt 1.7.6, the latest version at the time of doing this research, resists the attacks of [LYG$^+$15, GPPT15], because the Libgcrypt maintainers accepted patches to protect against chosen-ciphertext attacks and to hide timings obtained from loading precomputed elements. However, the maintainers refused a patch to switch from sliding windows to fixed windows; they said that this was unnecessary to stop the attacks. RSA-1024 in Libgcrypt uses the CRT method and $w = 4$, too few to use the key-recovery attack [HS09] by Heninger and Shacham. RSA-2048 uses CRT and $w = 5$. In this chapter, we investigate whether this common belief is correct and analyze the information that can be gained from side-channel attacks on the sliding window exponentiation.

Summary. In this chapter we show that the common belief is incorrect for the left-to-right encoding: this encoding actually leaks many more bits. An attacker learning the location of multiplications in the left-to-right square-and-multiply sequence can recover the key for RSA-1024 with CRT and $w = 4$ in a search through fewer than 10000 candidates for most keys, and fewer than 1000000 candidates for practically all keys. Note that RSA-1024 and RSA-1280 remain widely deployed in some applications, such as DNSSEC. Scaling up to RSA-2048 does not stop our attack: we show that 13% of all RSA-2048 keys with CRT and $w = 5$ are vulnerable to our method after a search through 2000000 candidates.

We analyze the reasons that left-to-right leaks more bits than right-to-left and extensive experiments show the effectiveness of this attack. We will give a short intuition here why there is a difference between these two methods. Recall that in the sliding-window methods, w bits are processed together for one multiplication using a table with pre-computed values. To reduce the size of these tables, the windows are limited to *odd* values: i.e. the rightmost bit in any window should be 1 (independent of the direction of left-to-right versus right-to-left). This is easily achieved in the right-to-left method: whenever a bit $d_i = 1$ the window extends from there to als include the next $w - 1$ values. This means the subsequent $w - 1$ bits to the left of d_i (i.e. $d_{i+w-1}, \dots, d_{i+1}$) are grouped together with d_i in a window. Since the processing of the window started with $d_i = 1$, which is the least significant bit in the window, it is ensured that the value in the window is odd. This ensures that whenever a multiplication occurs, at least w squarings follow. However, when this is turned around in the left-to-right method, the processing starts with the most-significant bit in such a window: in this case when $d_i = 1$ the window extends from there. This means the subsequent $w - 1$ bits to the *right* of d_i (i.e. $d_{i-1}, \dots, d_{i-w+1}$) are (potentially) grouped together with d_i in a window. However, since each window must contain an odd value, there is some additional processing: the method needs to check whether $d_{i-w+1} = 1$ as that is the least-significant bit for the window starting at d_i. If this is the case, the w bits can be processed with one multiplication as usual, as the window contains an odd value. But if $d_{i-w+1} = 0$, the window needs to be *smaller*: the method needs to check the bit to the left of d_{i-w+1} and repeat these checks up to the point where the least significant bit in the window is 1 (which can be d_i). This means that after a multiplication in the left-to-right method there is not a fixed number of squarings, but a number that depends on the bits $d_{i-1}, \dots, d_{i-w+1}$. In the extreme case that $d_{i-1} = \dots = d_{i-w+1} = 0$, two multiplications could be as close as adjacent with no squarings in between: this is something a side-channel attack could detect and thereby determine several bits to be zero. This event is one attack avenue we cover in Section 7.3, but more of these *known bits* are possible.

We further improve the algorithm by Heninger and Shacham to make use of less readily available information to attack RSA-2048, and prove that our extended algorithm efficiently recovers the full key when the side channel leaks data with a *self-information* rate greater than 1/2. Figure 7.1 visualizes the distribution of information recovered (for $w = 4$) in previous analysis (i.e. the common belief) for the right-to-left method and the two new analyses for the left-to-right method described in this chapter.

To illustrate the real-world applicability of this attack, we demonstrate how to obtain the required side-channel data (the pattern of squarings and multiplications) from the

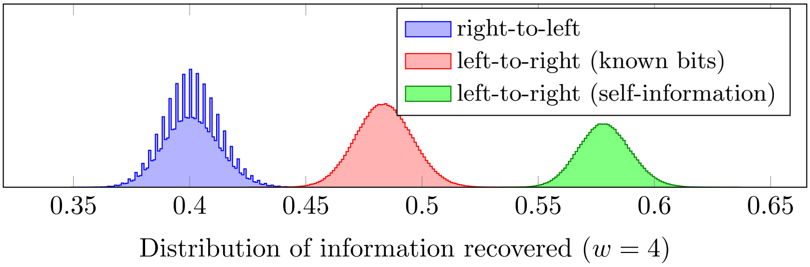


Figure 7.1: This graph shows the probability (y-axis) to recover a percentage of bits (x-axis) from the square-and-multiply sequence using the described analysis. The sequence of squares and multiplies of left-to-right windowed exponentiation contains much more information about the exponent than from exponentiation in the other direction, both in the form of known bits (red, Section 7.3-7.5) and information-theoretic bits (green, Section 7.6). Recovering close to 50% of the information about the key allows an efficient full key recovery attack.

modular-exponentiation routine in Libgcrypt version 1.7.6 using a FLUSH+RELOAD [YF14, YB14] cache-timing attack that monitors the target's cache-access patterns. The attack combines a small number of traces (at most 20) using the same secret RSA key, and does not depend on further front end details.

Targeted Software. We target Libgcrypt version 1.7.6, which is the latest version at the time of doing this research [BBG$^+$17b] that this chapter is based on. We compiled Libgcrypt using GCC version 4.4.7 and the -O2 optimization level. We performed the attack on an HP-Elite 8300 desktop machine, running Centos 6.8 with kernel version 3.18.41-20. The machine has a 4-core Intel i5-3470 processor, running at 3.2 GHz, with 8 GiB of DDR3-1600 CL-11 memory.

Organization. In Section 7.2, we discuss the algorithms of RSA, in particular the sliding window exponentiation method. In Section 7.3, we analyze the left-to-right versus right-to-left sliding windows, and show how to gain as many secret bits as possible from the square-and-multiply sequence. In Section 7.4, we analyze in theory how many secret bits we can get from the bit recovery rules. We also verify the analysis with experiments. In Section 7.5, we complete the bit analysis and show how to do a full RSA key recovery using Heninger-Shacham's algorithm. In Section 7.6, we show how to retrieve more information from the left-to-right method, that are not immediately translatable to known bits. Lastly, we verify the found vulnerabilities in Section 7.7 by attacking the implementation in Libgcrypt.

7.2 — Preliminaries

7.2.1 – RSA-CRT. RSA key generation is done by generating two random primes p, q. The public key is then set to be (e, N) where e is a (fixed) public exponent and $N = pq$. The private key is set to be (d, p, q) where $ed \equiv 1 \pmod{\phi(n)}$ and $\phi(n) = (p-1)(q-1)$. An RSA signature of a message m is done by computing $s = h(m)^d \bmod N$ where h is a padded cryptographically secure hash function. Signature verification is done by computing $z = s^e \bmod N$ and verifying that z equals $h(m)$. A common optimization for RSA signatures is based on the Chinese Remainder Theorem (CRT). Instead of computing $s = h(m)^d \bmod N$ directly, the signer computes $s_p = h(m)^{d_p} \bmod p$,

Algorithm 7.1 Sliding window modular exponentiation.

```
Require: Three integers $b$, $d$ and $p$ where $d_n \cdots d_1$ is a window form of $d$.
Ensure: $a \equiv b^d \pmod{p}$.
1: procedure MOD_EXP($b, d, p$)
2:     $b_1 \leftarrow b$, $b_2 \leftarrow b^2 \bmod p$, $a \leftarrow 1$
3:     for $i \leftarrow 1$ to $2^{w-1} - 1$ do          ▷ precompute table of small powers of $b$
4:         $b_{2i+1} \leftarrow b_{2i-1} \cdot b_2 \bmod p$
5:     for $i \leftarrow n$ to $1$ do
6:         $a \leftarrow a \cdot a \bmod p$
7:         if $d_i \neq 0$ then
8:             $a \leftarrow a \cdot b_{d_i} \bmod p$
9:     return $a$
```

$s_q = h(m)^{d_q} \bmod q$ (where d_p and d_q are derived from the secret key) and then combines s_p and s_q into s using the CRT. The computations of s_p and s_q work with half-size operands and have half-length exponents, leading to a speedup of a factor 2–4.

7.2.2 – Sliding Window Modular Exponentiation. In order to compute an RSA signature (more specifically the values of s_p and s_q defined above), two modular exponentiation operations must be performed. A modular exponentiation operation gets as inputs base b, exponent d, and modulus p and outputs $b^d \bmod p$. A common method used for efficient implementations is the sliding window method, which assumes that the exponent d is given in a special representation, the window form. For a window size w, the window form of d is a sequence of digits $d_{n-1} \cdots d_0$ such that $d = \sum_{i=0}^{n-1} d_i 2^i$ and d_i is either 0 or an odd number between 1 and $2^w - 1$.

Algorithm 7.1 performs the sliding window exponentiation method, assuming that the exponent is given in a window form, in two steps: It first precomputes the values of $b^1 \bmod p, b^3 \bmod p, \cdots, b^{2^w-1} \bmod p$ for odd powers of b. Then, the algorithm scans the digits of d from the most significant bit (MSB) to the least significant bit (LSB). For every digit, the algorithm performs a squaring operation (Line 6) on the accumulator variable a. Finally, for every non-zero digit of d, the algorithm performs a multiplication (Line 8).

7.2.3 – Sliding Window Conversion. The representation of a number d as (sliding) windows is not unique, even for a fixed value of w. In particular, the binary representation of d is a valid window form. However, since each non-zero digit requires a costly multiplication operation, it is desirable to reduce the number of non-zero digits in d's sliding window representation.

Right-to-Left Sliding Windows. One approach to computing d's sliding windows (with fewer non-zero digits) scans d's binary representation from the least significant bit (LSB) to the most significant bit (MSB) and generates d's sliding windows from the least significant digit (right) to the most significant digit (left). For every clear bit (i.e. a bit that is 0 and does not belong to a window), a zero digit is appended to the left of the current position. For each set bit (i.e. a bit that is 1), the following steps are performed: a non-zero digit is appended whose value is the w-bit integer ending at the current bit. Additionally, the next $w-1$ digits in the expansion are set to be zero digits. The scan resumes from the leftmost bit unused so far. Finally, any leading zeroes in the window form are truncated.

For example, let $w = 3$, and $d = 181$, which is $\underline{1}\ \underline{0\ 1\ 1}\ 0\ \underline{1\ 0\ 1}$ in binary. The windows are underlined. This yields the sliding window form 10030005.

Left-to-Right Sliding Windows. An alternative approach is the left-to-right window form, which scans the bits of d from the most to least significant bit and the window form is generated from the most significant digit to the least significant one. Similar to the right-to-left form, for every scanned clear bit a zero digit is appended to the right of the current position. Since we require digits to be odd, when a set bit is encountered, the algorithm cannot simply set the digit to be the w-bit integer starting at the current bit. Instead, it looks for the the longest integer u that has its most significant bit at the current bit, terminates in a set bit, and has its number of bits k is at most w. The algorithm sets the next $k-1$ digits in the expansion to be zero, sets the subsequent digit to be u and resumes the scan from the next bit unused so far. As before, leading zeroes in the sliding window form are truncated.

Using the $d = 181$ and $w = 3$ example, the left-to-right sliding windows are in this case $\underline{1\ 0\ 1}\ \underline{1\ 0\ 1}\ 0\ \underline{1}$ and the corresponding window form is 500501 (and 00500501 before truncation of leading zeroes).

Left-to-Right vs. Right-to-Left. While both methods produce an expansion whose average density (the ratio between the non-zero digits and the total length) is about $1/(w+1)$, generating the window form using the right-to-left method guarantees that every non-zero digit is followed by at least $w-1$ zero digits. This is different from the left-to-right method, where two non-zero digits can be as close as adjacent. As explained in Section 7.4, such consecutive non-zero digits can be observed by the attacker, aiding key recovery for sliding window exponentiations using the left-to-right direction.

7.2.4 – GnuPG's Sliding Window Exponentiation. While producing the right-to-left sliding window form requires a dedicated procedure, the left-to-right form can be generated "on-the-fly" during the exponentiation algorithm, combining the generation of the expansion and the exponentiation itself in one go. The left-to-right sliding window form [MvOV96, Algorithm 14.85], shown in Algorithm 7.2, is the prevalent method used by many implementations, including GnuPG.

Every iteration of the main loop (Line 6) constructs the next non-zero digit u of the window form by locating the location i of the leftmost set bit of d which was not previously handled (Line 8) and then removing the trailing zeroes from $d_i \cdots d_{i-w+1}$. It appends the squaring operations needed in order to handle the zero digits preceding u (Line 13) before performing the multiplication operation using u as the index to the precomputation table (thus handling u), and keeping track of trailing zeroes in z.

7.3 — Sliding right versus sliding left analysis

In this section, we show how to recover some bits of the secret exponents, assuming that the attacker has access to the square-and-multiply sequence performed by Algorithm 7.2. We show that more bits can be found by applying this approach to the square-and-multiply sequence of the left-to-right method compared to that of the right-to-left method. At a high level, our approach consists of two main steps. In the first step, we show how to directly recover some of the bits of the exponent by analyzing the

Algorithm 7.2 Left-to-right sliding window modular exponentiation.

```
Require: Three integers $b$, $d$ and $p$ where $d_n \cdots d_1$ is the binary representation of $d$.
Ensure: $a \equiv b^d \pmod{p}$.
 1: procedure MOD_EXP($b, d, p$)
 2:     $b_1 \leftarrow b$, $b_2 \leftarrow b^2$, $a \leftarrow 1$, $z \leftarrow 0$
 3:     for $i \leftarrow 1$ to $2^{w-1} - 1$ do          ▷ precompute table of small odd powers of b
 4:         $b_{2i+1} \leftarrow b_{2i-1} \cdot b_2 \bmod p$
 5:     $i \leftarrow n$
 6:     while $i \neq 1$ do                              ▷ main loop for computing $b^d \bmod p$
 7:         $z \leftarrow z +$ COUNT_LEADING_ZEROS($d_i \cdots d_1$)
 8:         $i \leftarrow i -$ COUNT_LEADING_ZEROS($d_i \cdots d_1$)
                                                     ▷ $i$ is the leftmost unscanned set bit of $d$
 9:         $l \leftarrow \min(i, w)$
10:         $u \leftarrow d_i \cdots d_{i-l+1}$
11:         $t \leftarrow$ COUNT_TRAILING_ZEROS($u$)
12:         $u \leftarrow$ SHIFT_RIGHT($u, t$)     ▷ remove trailing zeroes by shifting $u$ to the right
13:         for $j \leftarrow 1$ to $z + l - t$ do
14:             $a \leftarrow a \cdot a \bmod p$
15:         $a \leftarrow a \cdot b_u \bmod p$                 ▷ notice that $u$ is always odd
16:         $i \leftarrow i - l$
17:         $z \leftarrow t$
18:     return $a$
```

Rule 0: $\underline{x} \to \underline{1}$

Rule 1: $\underline{1}x^i\underline{1}x^{w-i-1} \to \underline{1x}^i\underline{1}0^{w-i-1}$ for $i = 0, \ldots, w-2$

Rule 2: $xxx\underline{11} \to 1xx\underline{11}$

Rule 3: $\underline{1}x^i x^{w-1}\underline{1} \to \underline{1}0^i x^{w-1}\underline{1}$ for $i > 0$

Figure 7.2: Rules to deduce known bits from a square-and-multiply sequence

sequence of squaring and multiplication operations performed by Algorithm 7.2. This step shows that we are capable of directly recovering an average of 48% of the bits of d_p and d_q for 1024-bit RSA with $w = 4$, the window size used by Libgcrypt for 1024-bit RSA. However, the number of remaining unknown bits required for a full key recovery attack is still too large to recover them with a brute-force attack. In Section 7.5.2 we show that applying a modified version of the techniques of [HS09] allows us to recover the remaining exponent bits and obtain the full private key, if at least 50% of the bits are recovered.

7.3.1 – Analyzing the square and multiply sequence. Assume the attacker has access to the sequence $S \in \{s, m\}^*$ corresponding to the sequence of square and multiply operations performed by Algorithm 7.2 executed on some exponent d. Notice that the squaring operation (Line 13) is performed once per bit of d, while the multiplication operation is performed only for some exponent bits. Thus, we can represent the attacker's knowledge about S as a sequence $s \in \{0, 1, \underline{1}, x, \underline{x}\}^*$ where $0, 1$ indicate known bits of d, x denotes an unknown bit and the positions of multiplications are underlined. For $y \in \{0, 1, \underline{1}, x, \underline{x}\}$ we denote by y^i the i-fold repetition of y.

Since at the start of the analysis all the bits are unknown, we convert S to the sequence s as follows: every sm turns into a $\underline{x}$, all remaining s into x. As a running example, the sequence of squares and multiplies S = smssssssmsmssssssm is converted into $D_1 = \underline{x}xxxxxx\underline{x}\underline{x}xxxx\underline{x}$. We assume that the attacker knows $w = 4$.

To obtain bits of d from D_1, the attacker applies the rewrite rules in Figure 7.2.

Rule 0: Multiplication bits. Because every digit in the window form is odd, a multiplication always happens at bits that are set. We denote such a bit by $\underline{1}$.

Applied to D_1 we obtain $D_2 = \underline{1}xxxxxx\underline{1}\underline{1}xxxx\underline{1}$.

Rule 1: Trailing zeros. Algorithm 7.2 tries to include as many set bits as possible in one digit of the window form. So when two multiplications are fewer than w bits apart, we learn that there were no further set bits available to include in the digit corresponding to the rightmost multiplication. Rule 1 sets the following bits to zero accordingly and we denote this by 0.

Applied to D_2 we obtain $D_3 = \underline{1}xxxxxx\underline{1}\underline{1}000x\underline{1}$.

Rule 2: Leading one. If we find two immediately consecutive multiplications, it is clear that as 7.2 was building the left digit, there were no trailing zeroes in $u = d_i \cdots d_{i-l+1}$, i.e. $t = 0$ in Line 11. This tells us that the bit $w - 1$ positions to the left of $\underline{1}$ is set. We denote such a leading one bit by 1.

Applied to D_3 we obtain $D_4 = \underline{1}xxx1xx\underline{1}\underline{1}000x\underline{1}$.

Rule 3: Leading zeroes. Every set bit of d is included in a non-zero digit of the window form, so it is at most $w - 1$ bits to the left of a multiplication. If two consecutive multiplications are more than w bits apart, we know that there are zeroes in between, denoting this by 0.

Applied to D_4 we obtain $D_5 = \underline{1}0001xx\underline{1}\underline{1}000x\underline{1}$.

Larger example. Consider the bit string

01000001111100101001100110101001100001100011111100011100100001001.

The corresponding sequence of square and multiply operations (using $w = 4$) evolves as follows as we apply the rules:

$x\underline{x}xxxxxxx\underline{x}xxx\underline{x}xxxx\underline{x}xxx\underline{x}xx\underline{x}xxxx\underline{x}\underline{x}xxxxx\underline{x}xxxxxx\underline{x}x\underline{x}xxxxx\underline{x}xx\underline{x}xxxxxxx\underline{x}$

$x\underline{1}xxxxxxx\underline{1}xxx\underline{1}xxxx\underline{1}xxx\underline{1}xx\underline{1}xxxx\underline{1}\underline{1}xxxxx\underline{1}xxxxxx\underline{1}x\underline{1}xxxxx\underline{1}xx\underline{1}xxxxxxx\underline{1}$

$x\underline{1}xxxxxxx\underline{1}xxx\underline{1}xxxx\underline{1}xxx\underline{1}xx\underline{1}0xxx\underline{1}\underline{1}000xx\underline{1}xxxxxx\underline{1}x\underline{1}00xxx\underline{1}xx\underline{1}0xxxxxx\underline{1}$

$x\underline{1}xxxxxxx\underline{1}xxx\underline{1}xxxx\underline{1}xxx\underline{1}xx\underline{1}01xx\underline{1}\underline{1}000xx\underline{1}xxxxxx\underline{1}x\underline{1}00xxx\underline{1}xx\underline{1}0xxxxxx\underline{1}$

$x\underline{1}0000xxx\underline{1}xxx\underline{1}0xxx\underline{1}xxx\underline{1}xx\underline{1}01xx\underline{1}\underline{1}000xx\underline{1}000xxx\underline{1}x\underline{1}00xxx\underline{1}xx\underline{1}0000xxx\underline{1}$.

Out of the 64 bits, 34 become known through this analysis.

Iterative application. The previous examples show that by applying rules iteratively, we can discover a few more bits. In particular, for a window where a leading one is recovered (Rule 2), one may learn the leading bit of the preceding window. Reasoning back from an application of Rule 2 in the example above gives 3 more known leading bits:

$$\mathtt{x\underline{1}0000xxx\underline{1}xxx\underline{1}01xx\underline{1}1xx\underline{1}1x\underline{1}01xx\underline{1}\underline{1}000xx\underline{1}000xxx\underline{1}x\underline{1}00xxx\underline{1}xx\underline{1}00000xxx\underline{1}}.$$

This iterative behavior is hard to analyze and occurs rarely in practice. Therefore the following analysis disregards it. Note that the algorithm of Section 7.5.2 does use the additional bits.

Completeness. The iterative application of these rules recovers almost all knowable bits, which is sufficient for our application. We present an alternative approach that is less intuitive and direct, but is complete in the sense that it recovers all knowable bits, in Section 7.5.

7.4 — Analyzing bit recovery rules

In this section we analyze the number of bits we are theoretically expected to recover using Rules 0–3 described in the previous section. The analysis applies to general window size w and the bit string length n. There are a number of techniques that can be used here. O'Connor [O'C99] modeled recoding rules using regular languages in order to study the resulting weight distribution.

Renewal processes with rewards. We model the number of bits recovered as a renewal reward process [Ros83]. A renewal process is associated with interarrival times $\underline{X} = (X_1, X_2, \dots)$ where the X_i are independent, identically distributed and non-negative variables with a common distribution function F and mean μ. Let

$$S_n = \sum_{i=1}^{n} X_i, \quad n \in \mathbb{N},$$

where $\underline{S} = (0, S_1, S_2, \dots)$ is the sequence of arrival times and

$$N_t = \sum_{n=1}^{\infty} I(S_n \leqslant t), \quad t \in \mathbb{R}^+$$

is the associated counting process. Now let $\underline{Y} = (Y_1, Y_2, \dots)$ be an i.i.d. sequence associated with $\underline{X}$ in the sense that Y_i is the reward for the interarrival X_i. Note that even though both $\underline{X}$ and $\underline{Y}$ are i.i.d., X_i and Y_i can be dependent. Then the stochastic process

$$R_t = \sum_{i=1}^{N_t} Y_i, \quad t \in \mathbb{R}^+,$$

is a renewal reward process. The function $r(t) = \mathbb{E}(R_t)$ is the renewal reward function. We can now state the *renewal reward theorem* [ME78]. Since $\mu_X < \infty$ and $\mu_Y < \infty$ we have for the renewal reward process

$$\begin{aligned} R_t/t &\to \mu_Y/\mu_X \text{ as } t \to \infty \text{ with probability } 1, \\ r(t)/t &\to \mu_Y/\mu_X \text{ as } t \to \infty. \end{aligned}$$

This is related to our attack in the following way. The n bit locations of the bit string form an interval of integers $[1, n]$, labeling the leftmost bit as 1. We set $X_1 = b + w - 1$, where b is the location of the first bit set to 1, that is, the left boundary of the first window.

Then the left boundary of the next window is independent of the first $b+w-1$ bits. The renewal process examines each window independently. For each window X_i we gain information about *at least* the multiplication bit. This is the reward Y_i associated with X_i. The renewal reward theorem now implies that for bit strings of length n, the expected number of recovered bits will converge to $\frac{n\mu_Y}{\mu_X}$.

7.4.1 – Recovered bit probabilities. In the following we analyze the expected number of bits that are recovered (the reward) in some number of bits (the renewal length) by the rules of Section 7.3.1. Then by calculating the probability of each of these rules' occurrence, we can compute the overall number of recovered bits by using the renewal reward theorem. Note that Rule 0 (the bits set to $\underline{1}$) can be incorporated into the other rules by increasing their recovered bits by one.

Some observations. We begin with some intuition on bit string probabilities. The first observation is that in a uniformly random bit string, each bit follows a Bernoulli distribution and is independent of all other bits. This means also that at every point the probability that the next x bits are zeroes (followed by a 1) is $\mathbb{P}[x] = 1/2^{x+1}$. The average reward R_t is the average number of windows N_t times average reward μ_y per window. At the left boundary of a new window, before sliding back, the first bit is a one, while the others are uniformly distributed. The effective window has length w with probability $1/2$ (last bit is a 1), $w-1$ with probability $1/2^2$ of length $w-1$ (last bits are 10) and so on. For each of the rules in the previous section we will analyze the probability that the rule applies, the average renewal length, and the average reward gained.

Rule 1: Trailing zeroes. The first rule applies to short windows. Recall that we call a window a "short window" whenever the length between between two multiplications is less than $w-1$.

Let $0 \leqslant j \leqslant w-2$ denote the length between two multiplications. (A length of $w-1$ is a full-size window.) The probability of a short window depends on these j bits, as well as $w-1$ bits after the multiplication: the multiplication bit should be the right-most 1-bit in the window. The following theorem gives the probability of a short window.

Theorem 7.1. *Let X be an interarrival time. Then the probability that $X = w$ and we have a short window with reward $Y = w - j$, $0 \leqslant j \leqslant w-2$ is*

$$p_j = \frac{1 + \sum_{i=1}^{j} 2^{2i-1}}{2^{j+w}}.$$

Proof. To prove this theorem, we first make two observations. The first observation is that given a substring of the multiplication chain, the attack recovers a bit independently from bits that are more than $w-1$ positions away. Given a set bit in the bit string, its position in the window that contains it determines whether or not it can be recovered. This only depends on the values of the $w-1$ bits before it and after it. This means that after a renewal, we can calculate the probability of seeing a string of $j > 0$ zeroes followed by a short window by looking at the set of size $j+w$ bitstrings.

The second observation is that this probability p_j is equal to $\frac{|C_j|}{2^{j+w}}$ where $|C_j|$ is equal to the number of elements in

$$C_j = \{x_0, \ldots, x_{j-1}\underline{1}y_0 \ldots y_{w-2} | x_k, y_\ell \in \{0,1\}; 0 \leqslant k \leqslant j-1, 0 \leqslant \ell \leqslant w-2\},$$

that is, the set of $j+w$ bitstrings which after the side-channel attack will show an $\underline{1}$ at position j. We will now prove our theorem by induction. For the base case, let $j=0$, then the $w-1$ bits after the multiplication should be the all-zero bitstring. This means $|C_0 = 1|$ and $p_0 = \frac{1}{2^w}$.

Suppose now we have for $j > 0$

$$|C_j| = 1 + \sum_{i=1}^{j} 2^{2i-1},$$

and therefore we have the following recursive formula

$$|C_j| = |C_{j-1}| + 2^{2j-1}.$$

Then by conditioning on the first bit x_0 we get

$$\begin{aligned} C_{j+1} =& \{x_0 \dots x_j \underline{1} y_0 \dots y_{w-2} | x_k, y_\ell \in \{0,1\}\} \\ =& \{0|x_1 \dots x_j \underline{1} y_0 \dots y_{w-2} | x_k, y_\ell \in \{0,1\}\} \\ & \cup \{1|x_1 \dots x_j \underline{1} y_0 \dots y_{w-2} | x_k, y_\ell \in \{0,1\}\}. \end{aligned}$$

Since when $x_0 = 0$, we get the same conditions as in the case of a window of size j, we have

$$|\{0|x_1 \dots x_j \underline{1} y_0 \dots y_{w-2} | x_k, y_\ell \in \{0,1\}\}| = |C_j|.$$

In the case that $x_0 = 1$, we have to count the set

$$\{1|x_1 \dots x_j \underline{1} y_0 \dots y_{w-2} | x_k, y_\ell \in \{0,1\}\}$$

and this will always have multiplication bit $\underline{1}$ at position $j+1$ if $y_0, \dots, y_{w-j-2}$ are an all-zero string, which leaves $2^j \cdot 2^{j+1} = 2^{2j+1}$ possibilities for the remaining bits x_i and y_i.

This means that $|C_{j+1}| = |C_j| + 2^{2j+1} = 1 + \sum_{i=1}^{j+1} 2^{2i-1}$, from which our theorem follows. □

We see in the proof that the bits $y_{w-j-1}, \dots, y_{w-2}$ can take any values. Also since bit $y_{w-j-2} = 0$ is known, we have a renewal at this point where future bits are independent.

Rule 2: Leading one. As explained in Section 7.3.1, this rule means that when after renewal an ultra-short window occurs (a 1 followed by $w-1$ zeroes) we get an extra bit of information about the previous window. The exception to this rule is if the previous window was also an ultra-short window. In this case the $\underline{1}$ of the window is at the location of the multiplication bit we would have learned anyways and therefore we do not get extra information. As seen in the previous section, an ultra-short window occurs with probability $p_0 = 1/2^w$. If an ultra-short window occurs after the current window with window-size $1 \leqslant j \leqslant w-1$, we therefore recover $(w-j)+1$ bits (all bits of the current window plus 1 for the leading bit) with probability $p_j p_0$ and $(w-j)$ with probability $p_j(1-p_0)$.

Rule 3: Leading zeroes. The last way in which extra bits can be recovered is the leading zeroes. If a window of size $w-d$ is preceded by more than d zeroes, then we can recover the excess zeroes. Let X_0 be a random variable of the length of a bit string of zeros until the first 1 is encountered. Then X_0 is geometrically distributed with $p = 1/2$. So $\mathbb{P}[X_0 = k] = (1/2)^k \cdot (1/2) = (1/2)^{k+1}$. This distribution has mean $\mu_X = 1$.

Let X_w be a random variable representing the length of the bit string from the first 1 that was encountered until the multiplication bit. For general window length of w, we have

$$\mathbb{P}[X_w = k] = \begin{cases} \frac{1}{2^{w-1}} & k = 1 \\ \frac{1}{2^{w-k+1}} & k > 1. \end{cases}$$

Now the distribution of the full bit string is the sum of the variables X_0 and X_w. We have that $\mathbb{P}[X_0 + X_w = k] = \sum_{i=1}^{\min(k,w)} \mathbb{P}[X_w = i] \cdot \mathbb{P}[X_0 = k - i]$.
Notice that this rule only recovers bits if the gap between two multiplications is at least $w - 1$. This means that these cases are independent of Rule 1.

There is a small caveat in this analysis: the renewal length is unclear. In the case that we have a sequence of zeroes followed by a short window of size $j < w$, we are implicitly conditioning on the $w - j$ bits that follow. This means we cannot simply renew after the the $\underline{1}$ and since we also cannot distinguish between a short and regular window size, we also cannot know how much information we have on the bits that follow.

We solve this by introducing an upper and lower bound. For the upper bound the number of recovered bits remains as above and the renewal at $X_0 + w$. This is an obvious upper bound. This means that for a sequence of zeroes followed by a short window of size j, we assume a probability of 1 of recovering information on the $w - j$ bits that follow the sequence. We get an average number of recovered bits of

$$\overline{R} = \sum_{k=w}^{\infty} \sum_{i=1}^{\min(k,w)} (k - i + 1) \cdot \mathbb{P}[X_w = i] \cdot \mathbb{P}[X_0 = k - i],$$

and a renewal length of

$$\overline{N} = \sum_{k=w}^{\infty} \sum_{i=1}^{\min(k,w)} (k + w - i) \cdot \mathbb{P}[X_w = i] \cdot \mathbb{P}[X_0 = k - i].$$

For the lower bound we could instead assume a probability of 0 of recovering information on the $w - j$ bits. We can however get a tighter bound by observing that the bits that follow this rule are more likely a 0 than a 1 and we are more likely to recover a 1 at the start of a new window then we are a 0. Therefore we bound the renewal at $X_0 + X_w$ and throw away the extra information.

For a lower bound on the number of recovered bits this gives the following expectation

$$\underline{R} = \sum_{k=w}^{\infty} (k - w + 1) \cdot \mathbb{P}[X_0 + X_w = k],$$

and a renewal length of

$$\underline{N} = \sum_{k=w}^{\infty} \sum_{i=1}^{\min(k,w)} k \cdot \mathbb{P}[X_w = i] \cdot \mathbb{P}[X_0 = k - i].$$

Table 7.1: Summary of bit information recovered from different patterns

Rule	Pattern	Probability	Renewal length	Recovered bits
1	Window size $1 < j < w$	$p_j(1 - p_0)$	w	$(w - j)$
		$p_j p_0$	w	$(w - j) + 1$
2	Window size $j = 1$	p_0	w	w
3	ub window size $> w$	$\mathbb{P}[X + Y = k]$	$\overline{N}$	$\overline{R}$
	lb window size $> w$	$\mathbb{P}[X + Y = k]$	$\underline{N}$	$\underline{R}$

We summarize the results in Table 7.1.

From this, we can calculate the expected renewal length for fixed w, by summing over all possible renewal lengths with corresponding probabilities. We can do the same for the expected number of recovered bits per renewal. Finally, we are interested in the expected total number of recovered bits in an n-bit string. We calculate this by taking the average number of renewals (by dividing n by the expected renewal length) and multiplying this with the number of recovered bits per window. Since we have upper and lower bounds for both the renewal length and recovered bits for Rule 3, we also get lower and upper bounds for the expected total number of recovered bits.

Recovered bits for right-to-left. The analysis of bit recovery for right-to-left exponentiation is simpler. The bit string is an alternation of X_0 and X_w (see Rule 3), where $X_w = w$ and X_0 is geometrically distributed with $p = 1/2$. Therefore the expected renewal length N and the expected reward R are

$$N = \sum_{i=0}^{\infty} (w + i) \cdot \mathbb{P}[X_0 = i] = w + 1 \quad \text{and} \quad R = \sum_{i=0}^{\infty} (1 + i) \cdot \mathbb{P}[X_0 = i] = 2.$$

Then by the renewal reward theorem, we expect to recover $\frac{2n}{w+1}$ bits.

7.4.2 – Experimental verification with perfect side-channel. To validate the previous analyses with experiments, we sampled n-bit binary strings uniformly at random and used Algorithm 7.2 to derive the square and multiply sequence. We then applied Rules 0–3 from Section 7.3.1 to extract known bits.

Case $n = 512, w = 4$. Figure 7.1 shows the total fraction of bits learned for right-to-left exponentiation compared to left-to-right exponentiation, for $w = 4$, over 1,000,000 experiments with $w = 4$ and $n = 512$, corresponding to the our target Libgcrypt's implementation for 1024-bit RSA. On average we learned 251 bits, or 49%, for left-to-right exponentiation with 512-bit exponents. This is between our computed lower bound of $\beta_L = 245$ (from a renewal length of $\underline{N} = 4.67$ bits and reward of 2.24 bits on average per renewal) and upper bound $\beta_U = 258$ (from a renewal length of $\overline{N} = 4.90$ bits and reward of 2.47 bits per renewal). The average number of recovered bits for right-to-left exponentiation is $204 \approx \frac{2n}{w+1}$ bits, or 40%, as expected.

Figure 7.3 shows the distribution of the number of new bits learned for each rule with left-to-right exponentiation by successively applying Rules 0–3 for 100,000 exponents. Both Rule 0 and Rule 3 contribute about $205 \approx \frac{2n}{w+1}$ bits, which is equal both to our theoretical analysis and is also the number of bits learned from the right-to-left exponentiation. The spikes visible in Rule 3 are due to the fact that we know that any least significant bits occurring after the last window must be 0, and we credit these bits

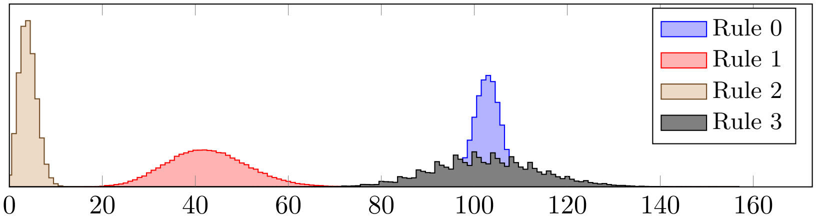


Figure 7.3: We generated 100,000 random 512-bit strings and generated the square and multiply sequence with $w = 4$. We then applied Rules 0–3 successively to recover bits of the original string. We plot the distribution of the number of recovered bits in our experiments.

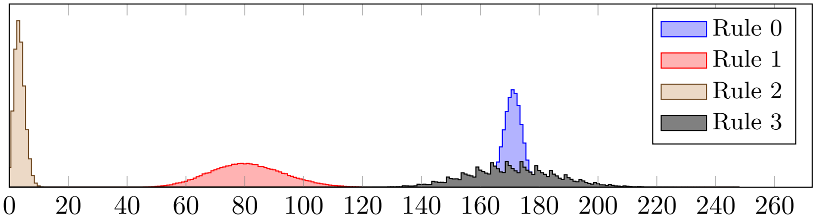


Figure 7.4: We generated 100,000 random 1024-bit strings and generated the square and multiply sequence with $w = 5$. We then applied Rules 0–3 successively to recover bits of the original string and plot the distribution of the number of new bits recovered for each rule. Compared to the $w = 4$ case, we learn a lower fraction of bits in this case.

learned to Rule 3. The number of bits learned from this final step is equal to $n \bmod w$, leading to small spikes at intervals of w bits.

Case $n = 1024, w = 5$. For $n = 1024$ and $w = 5$, corresponding to Libgcrypt's implementation of 2048-bit RSA, we recover 41.5% of bits, or 425, on average using Rules 0–3. This is between our lower bound of $\beta_L = 412$ (from a lower bound average renewal length of $\underline{N} = 5.67$ bits, and expected 2.29 bits on average per renewal) and upper bound of $\beta_U = 436$ (from an average renewal length of $\overline{N} = 5.89$ bits with an average reward of 2.51 bits per renewal). Note that the reward per renewal is about the same as in the first case ($n = 512, w = 4$), but the average renewal length is higher. This means that we win fewer bits for this case.

In Figure 7.4 we plot the frequency of the number of recovered bits per rule in 100,000 experiments, applied in order. Note again that both Rule 0 and Rule 3 contribute about $\frac{2n}{w+1}$ bits, which is what we would expect in theory.

7.5 — Full RSA Key Recovery from Known Bits

7.5.1 – Learning all learnable bits. Section 7.3 describes how to learn some bits of the secret exponent, given the sequence of square and multiply operations performed by Algorithm 7.2, using four simple rewrite rules. It turns out that one can learn further bits using a stateful algorithm. To see why there are more bits to be learned, consider for $w = 4$ the bit string

$$1001101010101010101010101010000010001,$$

producing the sequence

$$\text{ssssmsssmssssmssssmssssmssssmssmssssssmssssm}$$

which is first converted to

$$\mathtt{xxx\underline{x}xx\underline{x}xxx\underline{x}xxx\underline{x}xxx\underline{x}xxx\underline{x}x\underline{x}xxxxx\underline{x}xxx\underline{x}}$$

and then rules 0–3 recover these bits:

$$\mathtt{xxx\underline{1}xx\underline{1}0xx}\underset{a}{\underline{\mathtt{1}}}\mathtt{x}\underset{\uparrow}{\mathtt{x}}\mathtt{x}\underset{b}{\underline{\mathtt{1}}}\mathtt{xxx}\underset{c}{\underline{\mathtt{1}}}\mathtt{xxx}\underset{d}{\underline{\mathtt{1}}}\mathtt{x}\underset{e}{\underline{\mathtt{1}}}\mathtt{00xxx\underline{1}xxx\underline{1}}$$

But every bit sequence leading to the above sequence of squares and multiplies must have a 1 in the position marked with an arrow and a 0 to the left of it.

To see why the most significant bit of the multiplier for multiplication b must be there, consider the alternatives. It cannot be one bit earlier, as then it would be included in the window for a. It also cannot be later, i.e. a multiplication with 3 or 1, as then the same would hold for c, and hence d, which is not possible, as otherwise the multiplication at d would include the 1 at e.

These missed opportunities are rare (although this leads to 9 bits in the degenerate example chosen above) and do not significantly affect the efficiency of the pruning, but in the interest of completeness, we provide an algorithm to recover these additional bits and prove that it finds all bits whose values can be known with certainty.

7.5.2 – Possible window widths. We can find out all knowable bits if we keep track of the possible widths of the multiplier of each observed multiplication. Consider a multiplication at position $i = 5$, i.e. $\mathtt{xxxx\underline{1}xxxxx}$. For $w = 4$, this corresponds to one of four cases:

Case 1: $\mathtt{x1xx\underline{1}xxxxx}$, multiplier width: 4
Case 2: $\mathtt{xx1x\underline{1}0xxxx}$, multiplier width: 3
Case 3: $\mathtt{xxx1\underline{1}00xxx}$, multiplier width: 2
Case 4: $\mathtt{xxxx\underline{1}000xx}$, multiplier width: 1

The key idea is now that **multipliers do not overlap**. More precisely, if the multiplier of the multiplication at position i has width m_i and the multiplier of the multiplication at position j has width m_j (with $i > j$), then

$$i + m_i - w \geqslant j + m_j.$$

From this, we can derive a simple linear-time algorithm that calculates the possible multiplier widths for each multiplication. We define
m_i^- to be the smallest possible width of the multiplication at position i, and
m_i^+ to be the largest possible width of the multiplication at position i.
Let $M = \{k_0, k_1, \ldots, k_n\}$ be the positions of the multiplications, with $k_0 > \cdots > k_n$. Then

$$m_{k_i}^- = \begin{cases} 1, & \text{if } i = n \\ \max(1, k_{i+1} + m_{k_{i+1}}^- + w - k_i), & \text{otherwise, and} \end{cases}$$

$$m_{k_i}^+ = \begin{cases} w, & \text{if } i = 0 \\ \min(w, k_{i-1} + m_{k_{i-1}}^+ - w - k_i), & \text{otherwise.} \end{cases}$$

Note that m^- can be calculated going from right to left, while m^+ can be calculated going from left to right.

Given m^- and m^+, we can calculate all the knowable bits of the input sequence. Let us represent this by $b_i \in \{0, 1, x\}$, defined as follows:

$$b_i = \begin{cases} 1, & \text{if } i \in M \\ 1, & \text{else if } j + m_j^- - 1 = i = j + m_j^+ - 1 \text{ for some } j \in M \\ x, & \text{else if } j + m_j^- - 1 \leqslant i \leqslant j + m_j^+ - 1 \text{ for some } j \in M \\ 0 & \text{otherwise.} \end{cases}$$

The first case includes the bits recovered by Rule 0 in Section 7.3 (last bit), the second includes Rule 2 (first bit) and the last case includes Rules 1 and 3 (trailing and leading zeroes).

Theorem 7.2. *The Algorithm in Section 7.5.2 is correct and complete.*

This means not only that the input sequence has bit i set according to b_i, but also that if $b_i = x$, then there are two input sequences that produce the given sequence of squares and multiplies and differ at bit i. In other words: All knowable bits are known.

Sketch. Correctness follows by construction of the algorithm. The algorithm is complete because for every unknown bit (x) we can construct two sequences, one with a 1 and one with a 0 in that spot, that map to the same square-and-multiply sequence. □

Example. For the input above, with $w = 4$, we learn

$$\begin{array}{ll}
\text{Input:} & \texttt{xxx\underline{x}xx\underline{x}xxx\underline{x}xxx\underline{x}xxx\underline{x}xxx\underline{x}x\underline{x}xxxxx\underline{x}xxx\underline{x}} \\
m^-: & \texttt{\ \ \ 4\ \ 3\ \ \ 3\ \ \ 3\ \ \ 3\ \ \ 3\ 1\ \ \ \ \ 1\ \ \ 1} \\
m^+: & \texttt{\ \ \ 4\ \ 3\ \ \ 3\ \ \ 3\ \ \ 3\ \ \ 3\ 1\ \ \ \ \ 3\ \ \ 3} \\
b_i: & \texttt{1xx11x101x101x101x101x101000xx10xx1.}
\end{array}$$

Self-information. Calculating m^+ and m^- not only allows us to read off all knowable bits, it also allows us to calculate the number of input sequences that produce the given sequence of squares and multiplies. We define the function $f(i, z)$ to be the number of colliding sequences for the output including the window at position k_i, under the additional constraint that the earliest 1 occurs at position z. This function is recursively defined (and admits to a straight-forward efficient dynamic programming implementation). The base case is $f(i, z) = 1$ for $i > n$, otherwise we have

$$f(i, z) = \sum_{m=m_{k_i}^-}^{m_{k_i}^+} [k_i + m - 1 \leqslant z] \cdot 2^{\max(0, m-2)} \cdot f(i + 1, k_i + m - 1 - w),$$

where

- the sum iterates over all possible multiplier widths at this position,
- the characteristic function selects those where the first 1 of the window is after z,
- the power of two takes into account the unknown bits in this window,

- and the recursive call goes to the next window, updating the constraint on the next allowed 1.

The total number of colliding sequences is then $f(0, l)$, where l is the length of the input, and the self-information of the sequence is $I_s = -\log p_s = -\log \frac{f(0,l)}{2^l}$.

Example. For the example above, this yields 1408 possible input sequences and a self-information of 24.5, and hence 0.70 bits of self-information per bit of input. Note that 1408 is not simply two to the number of x's: not all assignments to the unknown bits yield this sequence of squares and multiplies.

7.5.3 – Heninger-Shacham's algorithm. Once we have used the recovered sequence of squares and multiplies to derive some information about the bits of the Chinese remainder theorem coefficients $d_p = d \bmod (p-1)$ and $d_q = d \bmod (q-1)$, we can use a modified version of the branch and prune algorithm of Heninger and Shacham [HS09] to recover the remaining unknown bits of these exponents to recover the full private key. The algorithm will recover the values d_p and d_q from partial information. In order to do so, we use the integer forms of the RSA equations

$$\begin{aligned} ed_p &= 1 + k_p(p-1) \\ ed_q &= 1 + k_q(q-1) \end{aligned}$$

which these values satisfy for positive integers $k_p, k_q < e$.

RSA Coefficient Recovery. As described in [IGI+16, YGH16], k_p and k_q are initially unknown, but are related via the equation $(k_p - 1)(k_q - 1) \equiv k_p k_q N \bmod e$. Thus we need to try at most e pairs of k_p, k_q. In the most common case, $e = 65537$. As described in [YGH16], incorrect values of k_p, k_q quickly result in no solutions.

LSB-Side Branch and Prune Algorithm. At the beginning of the algorithm, we have deduced some bits of d_p and d_q using Rules 0–3. Given candidate values for k_p and k_q, we can then apply the approach of [HS09] to recover successive bits of the key starting from the least significant bits. Our algorithm does a depth-first search over the unknown bits of d_p, d_q, p, and q. At the ith least significant bit, we have generated a candidate solution for bits $0 \dots i-1$ of each of our unknown values. We then verify the equations

$$\begin{aligned} ed_p &= 1 + k_p(p-1) \bmod 2^i \\ ed_q &= 1 + k_q(q-1) \bmod 2^i \\ pq &= N \bmod 2^i \end{aligned} \tag{7.1}$$

and prune a candidate solution if any of these equations is not satisfied.

Analysis. Heuristically, we expect this approach to be efficient when we know more than 50% of bits for d_p and d_q, distributed uniformly at random [HS09, PPS12]. We also expect the running time to grow exponentially in the number of unknown bits when we know many fewer than 50% of the bits. From the analysis of Rules 0–3 above, we expect to recover 48% of the bits. While the sequence of recovered bits is not, strictly speaking, uniformly random since it is derived using deterministic rules, the experimental performance of the algorithm matched that of a random sequence.

7.6 — RSA Key Recovery from Squares and Multiplies

The sequence of squares and multiplies encodes additional information about the secret exponent that does not translate directly into knowledge of particular bits. A first example of this kind of conditional information is a variant of the trailing zeros rule. Consider again $w = 4$ and we have learned the following substring

$$xxx\underline{1}x_1x_2x_3\underline{1}x_5,$$

and suppose $x_1 = 0$. Then the only way the parsing could have occurred such that $x_4 = \underline{1}$ is if this sequence was followed by an unset bit ($x_5 = 0$). If however $x_1 = 1$, then no information is gained on x_5. A similar reasoning follows for any $\underline{1}$ followed by a short window.

A second type is a variant of the leading one rule. Consider the following substring

$$x_1x_2x_3\underline{1}x_5\underline{1}xx,$$

and suppose $x_5 = 1$. Then the only way the $\underline{1}$ is in position 4 and thus not $x_5 = \underline{1}$, is if the window's left boundary is at x_1. Therefore x_1 must be 1. We can gain information on the options for the left boundary of the first window if it is followed by any short window.

In this section, we give a new algorithm that exploits this additional information by recovering RSA keys directly from the square-and-multiply sequence. This gives a significant speed improvement over the key recovery algorithm described in Section 7.5, and brings an attack against $w = 5$ within feasible range.

7.6.1 – Pruning from Squares and Multiplies. Our new algorithm generates a depth-first tree of candidate solutions for the secret exponents, and prunes a candidate solution if it could not have produced the ground-truth square-and-multiply sequence obtained by the side-channel attack. Let $\mathrm{SM}(d) = s$ be the deterministic function that maps a bit string d to a sequence of squares and multiplies $s \in \{s, m\}^*$.

In the beginning of the algorithm, we assume we have ground truth square-and-multiply sequences s_p and s_q corresponding to the unknown CRT coefficients d_p and d_q. We begin by recovering the coefficients k_p and k_q using brute force as described in Section 7.5.2. We will then iteratively produce candidate solutions for the bits of d_p and d_q by generating a depth-first search tree of candidates satisfying Equations (7.1) starting at the least significant bits. We will attempt to prune candidate solutions for d_p or d_q at bit locations i for which we know the precise state of Algorithm 7.2 from the corresponding square and multiply sequence s, namely when element i of s is a multiplication or begins a sequence of w squares. To test an i-bit candidate exponent d_i, we compare $s' = \mathrm{SM}(d_i)$ to positions 0 through $i - 1$ of s, and prune d_i if the sequences do not match exactly.

7.6.2 – Algorithm Analysis. We analyze the performance of this algorithm by computing the expected number of candidate solutions examined by the algorithm before it recovers a full key. Our analysis was inspired by the information-theoretic analysis of [PPS12], but we had to develop a new approach to capture the present scenario. Let $p_s = \mathbb{P}[\mathrm{SM}(d_i) = s]$ be the probability that a fixed square-and-multiply sequence s is observed for a uniformly random i-bit sequence d_i. This defines the probability distribution D_i of square-and-multiply sequences for i-bit inputs. In order to understand how much information a sequence s leaks about an exponent, we will use the *self-information*, defined as $I_s = -\log p_s$. This is the analogue of the number of bits known for the algorithm given in Section 7.5.2. As with the bit count, we can express the number of candidate solutions that generate s in terms of I_s: $\#\{d \mid \mathrm{SM}(d) = s\} = 2^i p_s = 2^i 2^{-I_s}$. For a given sequence s, let I_i denote the self-information of the least significant i bits.

Theorem 7.3 (Heuristic). *If for the square-and-multiply sequences s_{p_i} and s_{q_i}, we have $I_i > i/2$ for almost all i, then the algorithm described in Section 7.6.1 runs in expected linear time in the number of bits of the exponent.*

Sketch. In addition to pruning based on s, the algorithm also prunes by verifying the RSA equations up to bit position i. Let $\mathrm{RSA}_i(d_p, d_q) = 1$ if $(ed_p - 1 + k_p)(ed_q - 1 + k_q) = k_p k_q N \bmod 2^i$ and 0 otherwise. For random (incorrect) candidates d_{p_i} and d_{q_i}, $\Pr[\mathrm{RSA}_i(d_{p_i}, d_{q_i}) = 1] = 1/2^i$.

As in [HS09], we heuristically assume that, once a bit has been guessed incorrectly, the set of satisfying candidates for d_{p_i} and d_{q_i} behave randomly and independently with respect to the RSA equation at bit position i.

Consider an incorrect guess at the first bit. We wish to bound the candidates examined before the decision is pruned. The number of incorrect candidates satisfying the square-and-multiply constraints and the RSA equation at bit i is

$$\begin{aligned}
\#\{d_{p_i}, d_{q_i}\} &\leqslant \#\{d_{p_i} \mid \mathrm{SM}(d_{p_i}) = s_{p_i}\} \cdot \#\{d_{q_i} \mid \mathrm{SM}(d_{q_i}) = s_{q_i}\} \cdot \mathbb{P}[\mathrm{RSA}_i(d_{p_i}, d_{q_i}) = 1] \\
&= 2^i 2^{-I_i} \cdot 2^i 2^{-I_i} \cdot 2^{-i} \\
&= 2^{i - 2I_i} \\
&\leqslant 2^{i \cdot (1-2c)}
\end{aligned}$$

with $I_i/i \geqslant c$ for some $c > 1/2$. In total, there are $\sum_i \#\{d_{p_i}, d_{q_i}\} \leqslant \sum_i 2^{i\cdot(1-2c)} \leqslant 1/(1-2^{1-2c})$ candidates. But *any* of the n bits can be guessed incorrectly, each producing a branch of that size. Therefore, the total search tree has at most $n \cdot (1 + \frac{1}{1-2^{1-2c}})$ nodes. □

A similar argument also tells us about the expected size of the search tree, which depends on the *collision entropy* [Ren61]

$$H_i = -\log \sum_{s \in \{s,m\}^i} p_s^2$$

of the distribution D_i of distinct square-and-multiply sequences. This is the log of the probability that two i-bit sequences chosen according to D_i are identical.

For our distribution D_i, the H_i are approximately linear in i. We can define the *collision entropy rate* $H = H_i/i$ and obtain an upper bound for the expected number of examined solutions. The proof is outside the scope of this thesis, but can be found in the full version of this work [BBG$^+$17a].

Theorem 7.4. *The expected total number of candidate solutions examined by the Algorithm in Section 7.6.1 for n-bit d_p an d_q is*

$$E\left[\sum_i \#\{d_{p_i}, d_{q_i}\}\right] \leqslant n \left(1 + \frac{1 - 2^{n\cdot(1-2H)}}{1 - 2^{1-2H}}\right).$$

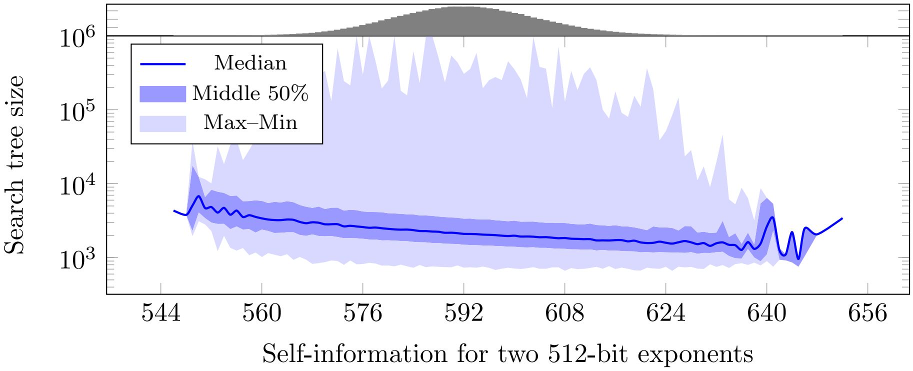


Figure 7.5: We attempted 500,000 key recovery trials for randomly generated 1024-bit RSA keys with $w = 4$. We plot the distribution of the number of candidates tested by the algorithm against the self-information of the observed square-and-multiply sequences calculated using the algorithm in Theorem 7.5.2 and measured in bits. The histogram above the plot shows the distribution of self-information across all the trials.

Entropy calculations. We calculated the collision entropy rate by modeling the leak as a Markov chain. For $w = 4$, $\mathrm{H} = 0.545$, and thus we expect the Algorithm in Section 7.6.1 to comfortably run in expected linear time. For $w = 5$, $\mathrm{H} = 0.461$, and thus we expect the algorithm to successfully terminate on some fraction of inputs. We give more details on this computation in the full version of this work.

7.6.3 – Experimental Evaluation for $w = 4$. We ran 500,000 trials of our sequence-pruning algorithm for randomly generated d_p and d_q with 1024-bit RSA and plot the distribution of running times in Figure 7.5. For a given trial, if the branching process passed 1,000,000 candidates examined without finding a solution, we abandoned the attempt. For each trial square-and-multiply sequence s, we computed the number of bit sequences that could have generated it. From the average of this quantity over the 1 million exponents generated in our trial, the collision entropy rate in our experiments is $\mathrm{H} = 0.547$, in line with our analytic computation above. The median self-information of the exponents generated in our trials was 295 bits; at this level the median number of candidates examined by the algorithm was 2,174. This can be directly compared to the 251 bits recovered in Section 7.4.2, since the self-information in that case is exactly the number of known bits in the exponent.

7.6.4 – Experimental Evaluation for $w = 5$. We ran 500,000 trials of our sequence-pruning algorithm for 2048-bit RSA and $w = 5$ with randomly generated d_p and d_q and plot the distribution of running times in Figure 7.6. 8.6% of our experimental trials successfully recovered the key before hitting the panic threshold of 1 million tries. Increasing the allowed tree size to 2 million tries allowed us to recover the key in 13% of trials. We experimentally estimate a collision entropy rate $\mathrm{H} = 0.464$, in line with our analytic computation. The median self-information for the exponents generated in our trials is 507 bits, significantly higher than the 425 bits that can be directly recovered using the analysis in Section 7.4.2.

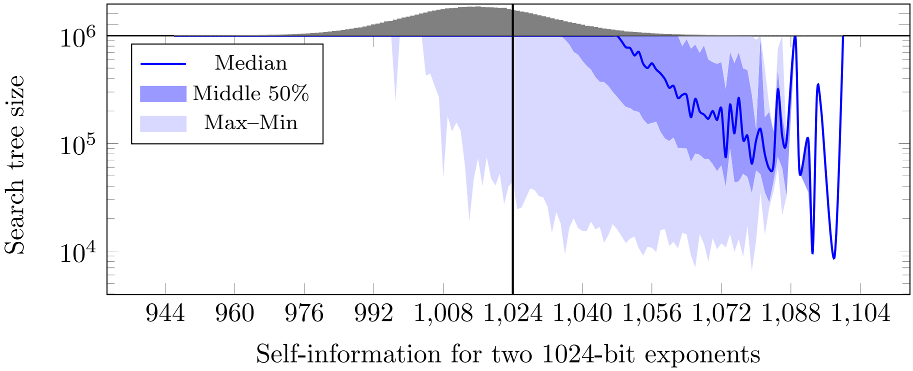


Figure 7.6: We attempted 500,000 key recovery trials for randomly generated 2048-bit RSA keys with $w = 5$, and plot the distribution of search tree size by the self-information. The vertical line marks the 50% rate at which we expect the algorithm to be efficient.

7.7 — Attacking Libgcrypts RSA

In the previous sections we showed how an attacker with access to the square-and-multiply sequence can recover the private RSA key. To complete the discussion we show how the attacker can obtain the square-and-multiply sequence and demonstrate the applicability of the attack on Libgcrypt.

7.7.1 – The Side-Channel Attack. To demonstrate the vulnerability in Libgcrypt, we use the FLUSH+RELOAD attack [YF14]. Mounting the FLUSH+RELOAD attack on Libgcrypt presents several challenges. First, as part of the defense against the attack of [YF14], Libgcrypt uses the multiplication code to perform the squaring operation. While this is less efficient than using dedicated code for squaring, the code reuse means that we cannot identify the multiply operations by probing a separate multiply routine. Instead we probe code locations that are used between the operations to identify the call site to the modular reduction. The second challenge is achieving a sufficiently high temporal resolution. Prior side-channel attacks on implementations of modular exponentiation use large (1024–4096 bits) moduli [YF14, ZJRR12, LYG$^+$15, GST14, GPPT15], which facilitate side-channel attacks [Wal03]. In this attack we target RSA-1024, which uses 512-bit moduli. The operations on these moduli are relatively fast, taking a total of less than 2500 cycles on average to compute a modular multiplication. To be able to distinguish events of this length, we must probe at least twice as fast, which is close to the limit of the FLUSH+RELOAD attack and would result in a high error rate [ABF$^+$16]. We use the amplification attack of [ABF$^+$16] to slow down the modular reduction. We target the code of the subtraction function used as part of the modular reduction. The attack increases the execution time of the modular reduction to over 30000 cycles. Our third challenge is that even with amplification, there is a chance of missing a probe [ABF$^+$16]. To reduce the probability of this happening, we probe two memory locations within the execution path of short code segments. The likelihood of missing both probes is small enough to allow high-quality traces.

Overall, we use the FLUSH+RELOAD attack to monitor seven victim code location. The monitored locations can be divided into three groups. To distinguish between the expo-

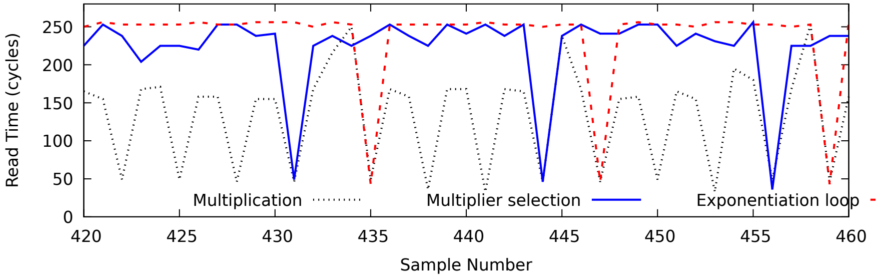


Figure 7.7: Libgcrypt Activity Trace.

nentiations Libgcrypt performs while signing, we monitor locations in the entry and exit of the exponentiation function. We also monitor a location in the loop that precomputes the multipliers to help identifying these multiplications. To trace individual modular multiplications, we monitor locations within the multiplication and the modular reduction functions. Finally, to identify the multiplications that are not squarings, we monitor a location in the code that conditionally copies the multiplier and in the entry to the main loop of the exponentiation. The former is accessed when Libgcrypt selects the multiplier before it performs the multiplication. The latter is accessed after the multiplication when the next iteration of the main loop starts. We repeatedly probe these locations once every 10000 cycles, allowing for 3–4 probes in each modular multiplication or squaring operation.

7.7.2 – Results. To mount the attack, we use the `FR-trace` software, included in the Mastik toolkit [Yar16]. `FR-trace` provides a command-line interface for performing the Flush+Reload and the amplification attacks we require for recovering the square-and-multiply sequences of the Libgcrypt exponentiation. `FR-trace` waits until there is activity in any of the monitored locations and collects a trace of the activity. Figure 7.7 shows a part of a collected activity trace.

Recall that the Flush+Reload attack identifies activity in a location by measuring the time it takes to read the contents of the location. Fast reads indicate activity. In the figure, monitored locations with read time below 100 cycles indicate that the location was active during the sample. Because multiplications take an average 800 cycles, whereas our sample rate is once in 10000 cycles, most of the time activity in the multiplication code is contained within a single sample. In Figure 7.7 we see the multiplication operations as “dips” in the multiplication trace (dotted black). Each multiplication operation is followed by a modular reduction. Our side-channel amplification attack “stretches” the execution of the modular reduction and it spans over more than 30000 cycles. Because none of the memory addresses traced in the figure is active during the modular reduction, we see gaps of 3–4 samples between periods of activity in any of the other monitored locations.

To distinguish between multiplications that use one of the precomputed multipliers and multiplications that square the accumulator by multiplying it with itself, we rely on activity in the multiplier selection and in the exponentiation loop locations. Before multiplying with a precomputed multiplier, the multiplier needs to be selected. Hence we would expect to see activity in the multiplier selection location just before starting the multiplication, and due to the temporal granularity of the attack we are likely to see both

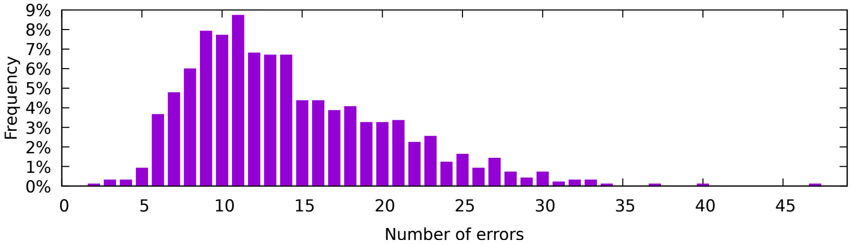


Figure 7.8: Distribution of the number of errors in captured traces.

events in the same sample. Similarly, after performing the multiplication and the modular reduction, we expect to see activity in the beginning of the main exponentiation loop. Again, due to attack granularity, this activity is likely to occur within the same sample as the following multiplication. Thus, because we see activity in the multiplier selection location during sample 431 and activity in the beginning of the exponentiation loop in the following multiplication (sample 435), we can conclude that the former multiplication is using one of the precomputed multipliers. In the absence of errors, this allows us to completely recover the sequence of square and multiplies performed and with it, the positions of the non-zero digits in the sliding window representation of the exponent.

However, capture errors do occur, as shown in Figure 7.8. To correct these, we capture multiple traces of signatures using the same private key. On average there are 14 errors in a captured trace. We find that in most cases, manually aligning traces and using a simple majority rule is sufficient to recover the complete square-and-multiply sequence. In all of the cases we have tried, combining twenty sequences yielded the complete sequence.

CHAPTER 8

Flush, Gauss, and reload

8.1 — Overview

Context. In Section 6.2 we gave a brief introduction to lattice-based cryptography and in Section 6.3 we discussed the threat of side-channel attacks, in particular cache attacks. In this chapter we will combine these two topics and make a first step towards understanding implementation security of lattice-based cryptography: we investigate whether samplers for the discrete Gaussian distribution are vulnerable to side-channel attacks. In particular, we investigate the applicability of cache-attacks on the Bimodal Lattice Signature Scheme (BLISS) by Ducas, Durmus, Lepoint, and Lyubashevsky from CRYPTO 2013 [DDLL13b]. The samplers for the discrete Gaussian distribution are a good target for such cache attacks, as it is unclear (at the time of writing this work) whether these samplers can be implemented securely in practice (i.e. constant-time). Additionally, note that more lattice-based schemes (i.e. not only BLISS) use noise sampled according to a discrete Gaussian distribution, so any attack on the discrete Gaussian sampler might be applicable to other lattice-based schemes and implementations. It is possible to avoid such attacks by using schemes which avoid discrete Gaussians, at the cost of more aggressive assumptions (e.g. [GLP12, LDK$^+$17, BAA$^+$17] and Chapter 10).

The attack target. At the time of writing, BLISS was the most recent piece in a line of work on identification-scheme-based lattice signatures, also known as signatures without trapdoors. An important step in the signature scheme is blinding a secret value in some way to make the signature statistically independent of the secret key. For this, a blinding (or noise) value $\mathbf{y}$ is sampled according to a discrete Gaussian distribution (see Section 8.2.2 for the definition). In the case of BLISS, $\mathbf{y}$ is an integer polynomial of degree less than some system parameter n and each coefficient is sampled separately. Essentially, $\mathbf{y}$ is used to hide the secret polynomial $\mathbf{s}$ in the signature equation $\mathbf{z} = \mathbf{y} + (-1)^b(\mathbf{s} \cdot \mathbf{c})$, where noise polynomial $\mathbf{y}$ and bit b are unknown to an attacker and $\mathbf{c}$ is the challenge polynomial from the identification scheme which is given as part of the signature $(\mathbf{z}, \mathbf{c})$.

If an attacker learns the complete noise polynomials $\mathbf{y}$ for just a few signatures, he can compute the secret key using linear algebra and guessing the bit b per signature. Actually, the attacker will only learn the secret key up to the sign but for BLISS $-\mathbf{s}$ is also a valid secret key. However, such an exhaustive leakage would be quite unlikely and probably not achievable in practice. Are other, more realistic attacks possible on BLISS?

Summary. In this chapter we present a FLUSH+RELOAD attack on BLISS. We implemented the attack for two different algorithms for Gaussian sampling. First we attack the *CDT* *sampler with guide table*, as described in [PDG14] and used in the attacked implementation as default sampler [DDLL13a]. CDT is the fastest way of sampling discrete Gaussians, but requires a large table stored in memory. Then we also attack a rejection sampler, specifically the Bernoulli-based sampler that was proposed in [DDLL13b], and also provided in [DDLL13a].

On a high level, our attacks exploit cache access patterns of the implementations to learn a few coefficients of $\mathbf{y}$ per observed signature. We then develop mathematical attacks to use this partial knowledge of different $\mathbf{y}_j$'s together with the public signature values $(\mathbf{z}_j, \mathbf{c}_j)$ to compute the secret key, given observations from sufficiently many signatures.

In detail, there is an interplay between requirements for the offline attack and restrictions on the sampling. First, restricting to cache access patterns that provide relatively precise information means that the online phase only allows to extract a few coefficients of $\mathbf{y}_j$ per signature. This means that trying all guesses for the bits b per signature becomes a bottleneck. We circumvent this issue by only collecting coefficients of $\mathbf{y}_j$ in situations where the respective coefficient of $\mathbf{s} \cdot \mathbf{c}_j$ is zero as in these cases the bit b_j has no effect.

Second, each such collected coefficient of $\mathbf{y}_j$ leads to an equation with some coefficients of $\mathbf{s}$ as unknowns. However, it turns out that for CDT sampling the cache patterns do not give exact equations. Instead, we learn equations which hold with high probability, but might be off by ± 1 with non-negligible probability. We managed to turn the computation of $\mathbf{s}$ into a lattice problem and show how to solve it using the LLL algorithm [LLL82]. For Bernoulli sampling we can obtain exact equations but at the expense of requiring more signatures. It turns out that the number of required signatures and the success probability of the attack largely depend on the chosen parameters for the discrete Gaussian distribution, in a way that is unrelated to the security parameter.

We first tweaked the BLISS implementation to provide us with the exact cache lines used, modeling a perfect side-channel. For BLISS-I, designed for 128 bits of security, the attack on CDT needs to observe on average 441 signatures during the online phase. Afterwards, the offline phase succeeds after 37.6 seconds with probability 0.66. This corresponds to running LLL once. If the attack does not succeed at first, a few more signatures (on average a total of 446) are sampled and LLL is run with some randomized selection of inputs. The combined attack succeeds with probability 0.96, taking a total of 85.8 seconds. Similar results hold for other BLISS versions. In the case of Bernoulli sampling, we are given exact equations and can use simple linear algebra to finalize the attack, given a success probability of 1.0 after observing 1671 signatures on average and taking 14.7 seconds in total.

To remove the assumption of a perfect side-channel we perform a proof-of-concept attack using the FLUSH+RELOAD technique on a modern laptop. This attack achieves similar success rates, albeit requiring 3438 signatures on average for BLISS-I with CDT sampling. For Bernoulli sampling, we now have to deal with measurement errors. We do this again by formulating a lattice problem and using LLL in the final step. The attack succeeds with a probability of 0.88 after observing an average of 3294 signatures.

Section 8.7 includes a discussion of candidate countermeasures against our specific attacks, including countermeasures presented in previous and concurrent work. We show that the standard countermeasures induce significant overhead.

As an extension of our work, we discuss two additional sampling techniques in Section 8.8: the Knuth-Yao [KY76,DG14] and the Discrete Ziggurat [BCG$^+$13] samplers. We argue that it is very likely that these sampling techniques are also vulnerable to cache attacks. We further discuss potential countermeasures for each technique. However, because there is no implementation of BLISS with any of these samplers, we do not perform any experiments (with or without perfect side-channels) to verify the vulnerabilities.

Organization This chapter contains the following. In Section 8.2, we give brief introductions to BLISS and the used methods for discrete Gaussian sampling. We then present two information leakages through cache-memory for CDT sampling and provide a strategy to exploit this information for secret key extraction in Section 8.3. In Section 8.4, we present an attack strategy for the case of Bernoulli sampling. To examine the performance of the attacks, we present experimental results for both strategies assuming a perfect side-channel in Section 8.5. In Section 8.6, we show that realistic experiments also succeed, using FLUSH+RELOAD attacks. Finally, in Section 8.7, we discuss candidate countermeasures against our specific attacks. As an extension, we briefly examine the applicability of the attacks on two other samplers in Section 8.8, with discussion of countermeasures.

8.2 — Preliminaries

This section describes the BLISS signature scheme and the used discrete Gaussian samplers. For the background on lattices see Section 6.2. In particular, the part about NTRU lattices and LLL/BKZ is important for this chapter.

8.2.1 – BLISS. We provide the basic algorithms of BLISS, as given in [DDLL13b]. Details of the motivation behind the construction and associated security proofs are given in the original work. All arithmetic for BLISS is performed in $R = \mathbb{Z}[x]/(x^n + 1)$ and possibly with each coefficient reduced modulo q or $2q$. We follow notation of BLISS and also use boldface notation for polynomials.

By D_σ we denote the discrete Gaussian distribution with parameter $\sigma > 0$. In the next subsection, we will discuss this distribution in more detail and how to sample from it in practice. The main parameters of BLISS are dimension n, modulus q and parameter σ. BLISS uses a cryptographic hash function H, which outputs binary vectors of length n and weight κ; density parameters δ_1 and δ_2, such that $d_1 = \lceil \delta_1 n \rceil$ and $d_2 = \lceil \delta_2 n \rceil$ determine the density of the polynomials forming the secret key; and d, determining the length of the second signature component.

Algorithm 8.1 BLISS Key Generation

```
Output: A BLISS key pair $(\mathbf{A}, \mathbf{S})$ with public key $\mathbf{A} = (\mathbf{a}_1, \mathbf{a}_2) \in R_{2q}^2$ and secret key
    $\mathbf{S} = (\mathbf{s}_1, \mathbf{s}_2) \in R_{2q}^2$ such that $\mathbf{AS} = \mathbf{a}_1 \cdot \mathbf{s}_1 + \mathbf{a}_2 \cdot \mathbf{s}_2 \equiv q \bmod 2q$
choose $\mathbf{f}, \mathbf{g} \in R_{2q}$ uniformly at random with exactly $d_1$ entries in $\{\pm 1\}$ and $d_2$ entries in $\{\pm 2\}$
$\mathbf{S} = (\mathbf{s}_1, \mathbf{s}_2) = (\mathbf{f}, 2\mathbf{g} + 1)$
if $\mathbf{S}$ violates certain bounds (details in [DDLL13b]), then restart
$\mathbf{a}_q = (2\mathbf{g} + 1)/\mathbf{f} \bmod q$ (restart if $\mathbf{f}$ is not invertible)
return $(\mathbf{A}, \mathbf{S})$ where $\mathbf{A} = (2\mathbf{a}_q, q - 2) \bmod 2q$
```

Algorithm 8.1 generates correct keys because

$$\mathbf{a}_1 \cdot \mathbf{s}_1 + \mathbf{a}_2 \cdot \mathbf{s}_2 = 2\mathbf{a}_q \cdot \mathbf{f} + (q-2) \cdot (2\mathbf{g}+1) \equiv 2(2\mathbf{g}+1) + (q-2)(2\mathbf{g}+1) \equiv q \bmod 2q.$$

Note that when an attacker has a candidate for key $\mathbf{s}_1 = \mathbf{f}$, he can validate correctness by checking the distributions of $\mathbf{f}$ and $\mathbf{a}_q \cdot \mathbf{f} \equiv 2\mathbf{g} + 1 \bmod 2q$, and lastly verify that $\mathbf{a}_1 \cdot \mathbf{f} + \mathbf{a}_2 \cdot (\mathbf{a}_q \cdot f) \equiv q \bmod 2q$, where $\mathbf{a}_q$ is obtained by halving $\mathbf{a}_1$.

Signature generation (Algorithm 8.2) uses $p = \lfloor 2q/2^d \rfloor$, which is the highest order bits of the modulus $2q$, and constant $\zeta = \frac{1}{q-2} \bmod 2q$. In general, with $\lfloor . \rceil_d$ we denote the d highest order bits of a number. In Step 1 of Algorithm 8.2, two integer vectors are sampled, where each coordinate is drawn independently and according to the discrete Gaussian distribution D_σ. This is denoted by $\mathbf{y} \leftarrow D_{\mathbb{Z}^n,\sigma}$.

Algorithm 8.2 BLISS Sign

```
Input: Message $\mu$, public key $\mathbf{A} = (\mathbf{a}_1, q-2)$, secret key $\mathbf{S} = (\mathbf{s}_1, \mathbf{s}_2)$
Output: A signature $(\mathbf{z}_1, \mathbf{z}_2^\dagger, \mathbf{c}) \in \mathbb{Z}_{2q}^n \times \mathbb{Z}_p^n \times \{0,1\}^n$ of the message $\mu$
1: $\mathbf{y}_1, \mathbf{y}_2 \leftarrow D_{\mathbb{Z}^n,\sigma}$
2: $\mathbf{u} = \zeta \cdot \mathbf{a}_1 \cdot \mathbf{y}_1 + \mathbf{y}_2 \bmod 2q$
3: $\mathbf{c} = H(\lfloor \mathbf{u} \rceil_d \bmod p, \mu)$
4: choose a random bit b
5: $\mathbf{z}_1 = \mathbf{y}_1 + (-1)^b \mathbf{s}_1 \cdot \mathbf{c} \bmod 2q$
6: $\mathbf{z}_2 = \mathbf{y}_2 + (-1)^b \mathbf{s}_2 \cdot \mathbf{c} \bmod 2q$
7: continue with a probability based on $\sigma, ||\mathbf{Sc}||, \langle \mathbf{z}, \mathbf{Sc} \rangle$ (details in [DDLL13b]), else
   restart
8: $\mathbf{z}_2^\dagger = (\lfloor \mathbf{u} \rceil_d - \lfloor \mathbf{u} - \mathbf{z}_2 \rceil_d) \bmod p$
9: return $(\mathbf{z}_1, \mathbf{z}_2^\dagger, \mathbf{c})$
```

In the attacks, we concentrate on the first signature vector $\mathbf{z}_1$, since $\mathbf{z}_2^\dagger$ only contains the d highest order bits and therefore lost information about $\mathbf{s}_2 \cdot \mathbf{c}$; furthermore, $\mathbf{A}$ and $\mathbf{f}$ determine $\mathbf{s}_2$ as shown above. So in the following, we only consider $\mathbf{z}_1, \mathbf{y}_1$ and $\mathbf{s}_1$, and thus will leave out the indices.

In lines 5 and 6 of Algorithm 8.2, we compute $\mathbf{s} \cdot \mathbf{c}$ over R_{2q}. However, since secret $\mathbf{s}$ is sparse with small coefficients (i.e. upper bounded by 2) and challenge $\mathbf{c}$ is sparse and binary (i.e. with weight κ), the absolute value of $||\mathbf{s} \cdot \mathbf{c}||_\infty \leqslant 2\kappa \ll 2q$, with $|| \cdot ||_\infty$ the ℓ_∞-norm. This means these computations are simply additions over $\mathbb{Z}$, and we can therefore model this computation as a vector-matrix multiplication over $\mathbb{Z}$:

$$\mathbf{s} \cdot \mathbf{c} = \mathbf{sC},$$

where $\mathbf{C} \in \{-1, 0, 1\}^{n \times n}$ is the matrix whose columns are the rotations of challenge $\mathbf{c}$ (with minus signs matching reduction modulo $x^n + 1$). In the attacks we access individual coefficients of $\mathbf{s} \cdot \mathbf{c}$; note that the jth coefficient equals $\langle \mathbf{s}, \mathbf{c}_j \rangle$, where $\mathbf{c}_j$ is the jth column of $\mathbf{C}$.

For completeness, we also show the verification procedure (Algorithm 8.3), although we do not use it further in this chapter. Note that reductions modulo $2q$ are done before truncating and reducing modulo p.

Algorithm 8.3 BLISS Verify

```
Input: Message $\mu$, public key $\mathbf{A} = (\mathbf{a}_1, q-2) \in R_{2q}^2$, signature $(\mathbf{z}_1, \mathbf{z}_2^\dagger, \mathbf{c})$
Output: Accept or reject the signature
1: if $\mathbf{z}_1, \mathbf{z}_2^\dagger$ violate certain bounds (details in [DDLL13b]), then reject
2: accept iff $\mathbf{c} = H(\lfloor \zeta \cdot \mathbf{a}_1 \cdot \mathbf{z}_1 + \zeta \cdot q \cdot \mathbf{c} \rceil_d + \mathbf{z}_2^\dagger \bmod p, \mu)$
```

Parameter Set	λ	n	q	σ	δ_1, δ_2	κ
BLISS-0 (Toy)	$\leqslant 60$	256	7681	100	0.55, 0.15	12
BLISS-I	128	512	12289	215	0.3, 0	23
BLISS-II	128	512	12289	107	0.3, 0	23
BLISS-III	160	512	12289	250	0.42, 0.03	30
BLISS-IV	192	512	12289	271	0.45, 0.06	39

Table 8.1: Parameter suggestions for different security levels of BLISS (from [DDLL13b])

The authors of BLISS [DDLL13b] proposed several parameter sets for the signature scheme for different levels of security. We give the important ones to understand this chapter in Table 8.1.

8.2.2 – Discrete Gaussian distribution. The probability distribution of a (centered) discrete Gaussian distribution is a distribution over $\mathbb{Z}$ with mean 0 and parameter $\sigma > 0$. A value $x \in \mathbb{Z}$ is sampled with probability:

$$\frac{\rho_\sigma(x)}{\sum_{y=-\infty}^{\infty} \rho_\sigma(y)},$$

where $\rho_\sigma(x) = \exp\left(\frac{-x^2}{2\sigma^2}\right)$ is the continuous Gaussian distribution with mean 0 and standard deviation σ. Note that the sum in the denominator ensures that this is actually a probability distribution (i.e. a proper probability density function (PDF)). We denote the denominator by $\rho_\sigma(\mathbb{Z})$.

To make sampling practical, most lattice-based schemes use three simplifications: First, a tail-cut τ is used, restricting the support of the Gaussian to a finite interval $[-\tau\sigma, \tau\sigma]$. The tail-cut τ is chosen such that the probability of a real discrete Gaussian sample landing outside this interval is negligible in the security parameter. Second, values are sampled from the positive half of the support and then a bit is flipped to determine the sign. For this the probability of obtaining zero in $[0, \tau\sigma]$ needs to be halved. The resulting distribution on the positive numbers is denoted by D_σ^+. Finally, the precision of the sampler is chosen such that the statistical distance between the output distribution and the exact distribution is negligible in the security parameter.

There are two generic ways to sample from a discrete Gaussian distribution: using the cumulative distribution function [Pei10] or via rejection sampling [GPV08]. Both these methods have some improvements which we describe next. These modified versions are implemented in [DDLL13a] and the main samplers under investigation.

We note that there are also other ways [DG14, RVV13, PG13, BCG$^+$13] of efficiently sampling discrete Gaussians. As an extension, we discuss two additional sampling techniques in Section 8.8: the Knuth-Yao [KY76, DG14] and the Discrete Ziggurat [BCG$^+$13] samplers. We investigate whether these sampling techniques are also vulnerable to cache attacks. Because there is no implementation of BLISS with any of these samplers, we do not perform any experiments (with or without perfect side-channels) to verify vulnerabilities.

CDT sampling. The basic idea of using the cumulative distribution function in the sampler, is to approximate the probabilities $p_y = \mathbb{P}[x \leqslant y \mid x \leftarrow D_\sigma]$, computed with λ bits of precision, and save them in a large table. At sampling time, one samples a uniformly random $r \in [0, 1)$, and performs a binary search through the table to locate $y \in [-\tau\sigma, \tau\sigma]$ such that $r \in [p_{y-1}, p_y)$. Restricting to the non-negative part $[0, \tau\sigma]$ corresponds to using the probabilities $p^*_y = \mathbb{P}[|x| \leqslant y \mid x \leftarrow D_\sigma]$, sampling $r \in [0, 1)$ and locating $y \in [0, \tau\sigma]$. While this is the most efficient approach, it requires a large table. We denote the method that uses the approximate cumulative distribution function with tail cut and the modifications described next, as the *CDT sampling* method.

One can speed up the binary search for the correct sample y in the table, by using an additional *guide table* I [PDG14, L'E11, CA74]. The BLISS implementation we attack uses I with 256 entries. The guide table stores for each $u \in \{0, \ldots, 255\}$ the smallest interval $I[u] = (a_u, b_u)$ such that $p^*_{a_u} \leqslant u/256$ and $p^*_{b_u} \geqslant (u + 1)/256$. The first byte of r is used to select $I[u]$ leading to a much smaller interval for the binary search. Effectively, r is picked byte-by-byte, stopping once a unique value for y is obtained. The CDT sampling algorithm with guide table is summarized in Algorithm 8.4.

Algorithm 8.4 CDT Sampling with Guide Table

```
Input: Big table $T[y]$ containing values $p^*_y$ of the cumulative distribution function of
    the discrete Gaussian distribution (using only non-negative values), omitting the first
    byte. Small table $I$ consisting of the 256 intervals
Output: Value $y \in [-\tau\sigma, \tau\sigma]$ sampled with probability according to $D_\sigma$
pick a random byte $r$
let $(I_{\min}, I_{\max}) = (a_r, b_r)$ be the left and right bounds of interval $I[r]$
if $(I_{\max} - I_{\min} = 1)$:
    generate a random sign bit $b \in \{0, 1\}$
    return $y = (-1)^b I_{\min}$
let $i = 1$ denote the index of the byte to look at
pick a new random byte $r$
while (1):
    $I_z = \lfloor \frac{I_{\min} + I_{\max}}{2} \rfloor$
    if $(r > (i\text{th byte of } T[I_z]))$:
        $I_{\min} = I_z$
    else if $(r < (i\text{th byte of } T[I_z]))$:
        $I_{\max} = I_z$
    else if $(I_{\max} - I_{\min} = 1)$:
        generate a random sign bit $b \in \{0, 1\}$
        return $y = (-1)^b I_{\min}$
    else:
        increase $i$ by 1
        pick new random byte $r$
```

Bernoulli sampling (Rejection sampling). The basic idea behind rejection sampling is to sample a uniformly random integer $y \in [-\tau\sigma, \tau\sigma]$ and accept this sample with probability $\rho_\sigma(y)/\rho_\sigma(\mathbb{Z})$. For this, a uniformly random value $r \in [0, 1)$ is sampled and y is accepted iff $r \leqslant \rho_\sigma(y)$. This method has two huge downsides: calculating the values of $\rho_\sigma(y)$ to high precision is expensive and the rejection rate can be quite high.

In the same paper introducing BLISS [DDLL13b], the authors also propose a more efficient Bernoulli-based sampling algorithm. We recall the algorithms used (Algorithms 8.5, 8.6, 8.7), more details are given in the original work. We denote this method as *Bernoulli sampling* in the remainder of this chapter.

Algorithm 8.5 Sampling from $D^+_{K\sigma}$ for $K \in \mathbb{Z}$

```
Input: Parameter $\sigma > 0$, integer $K = \lfloor \frac{\sigma}{\sigma_2} + 1 \rfloor$, where $\sigma_2 = \frac{1}{2\ln 2}$
Output: An integer $y \in \mathbb{Z}^+$ according to $D^+_{K\sigma_2}$
sample $x \in \mathbb{Z}$ according to $D^+_{\sigma_2}$
sample $z \in \mathbb{Z}$ uniformly in $\{0, \ldots, K-1\}$
$y \leftarrow Kx + z$
sample $b$ with probability $\exp\left(-z(z + 2Kx)/(2\sigma^2)\right)$
if $b = 0$ then restart
return $y$
```

Algorithm 8.6 Sampling from $D_{K\sigma}$

```
Output: An integer $y \in \mathbb{Z}$ according to $D_{K\sigma_2}$
sample integer $y \leftarrow D^+_{K\sigma}$ (using Algorithm 8.5)
if $y = 0$ then restart with probability 1/2
generate random bit $b$ and return $(-1)^b y$
```

The basic idea is to first sample a value x, according to (what the authors of BLISS describe as) the "binary discrete Gaussian distribution", which is D_{σ_2} where $\sigma_2 = \frac{1}{2\ln 2}$ (Step 1 of Algorithm 8.5). This can be done efficiently using uniformly random bits [DDLL13b]. The actual sample $y = Kx+z$, where $z \in \{0, \ldots, K-1\}$ is sampled uniformly at random and $K = \lfloor \frac{\sigma}{\sigma_2} + 1 \rfloor$, is then distributed according to the target discrete Gaussian distribution D_σ, by rejecting with a certain probability (Step 4 of Algorithm 8.5). The number of rejections in this case is much lower than in the original method. This step still requires computing a bit, whose probability is an exponential value. However, it can be done more efficiently using Algorithm 8.7, where ET is a small table.

8.3 — Attack 1: CDT sampling

In this section we explore the mathematical foundations of cache attacks on the CDT sampling. For a brief summary on cache attacks see Section 6.3 We first explain the phenomena we can observe from cache misses and hits in Algorithm 8.4 and then show how to exploit them to derive the secret signing key of BLISS using LLL. Sampling of the first noise polynomial $\mathbf{y} \in D_{\mathbb{Z}^n,\sigma}$ is done coefficientwise. Similarly the cache attack targets coefficients y_i for $i = 0, \ldots, n-1$ independently.

Algorithm 8.7 Sampling a bit with probability $\exp(-x/(2\sigma^2))$ for $x \in [0, 2^\ell)$

```
Input: $x \in [0, 2^\ell)$ an integer in binary form $x = x_{\ell-1} \dots x_0$. Table ET with precomputed
    values $ET[i] = \exp(-2^i/(2\sigma^2))$ for $0 \leqslant i \leqslant \ell - 1$
Output: A bit b with probability $\exp(-x/(2\sigma^2))$ of being 1
for $i = \ell - 1$ to 0:
    if $x_i = 1$ then
        sample $A_i$ with probability $ET[i]$.
        if $A_i = 0$ then return 0
return 1
```

8.3.1 – Weaknesses in cache. Sampling from a discrete Gaussian distribution using both an interval table I and a table with the actual values T, might leak information via cache memory. The best we can hope for is to learn the cache-lines of index r of the interval and of index I_z of the table lookup in T. Note that we cannot learn the sign of the sampled coefficient y_i. Also, the cache line of $T[I_z]$ always leaves a range of values for $|y_i|$. However, in some cases we can get more precise information combining cache-lines of table lookups in both tables. Here are two observations that narrow down the possibilities:

Intersection: We can intersect knowledge about the used index r in I with the knowledge of the access $T[I_z]$. Getting the cache-line of $I[r]$ gives a range of intervals, which is simply another (bigger) interval of possible values for sample $|y_i|$. If the values in the range of intervals are largely non-overlapping with the range of values learned from the access to $T[I_z]$, then the combination gives a much more precise estimate. For example: if the cache-line of $I[r]$ reveals that sample $|y_i|$ is in set $S_1 = \{0, 1, 2, 3, 4, 5, 7, 8\}$ and the cache-line of $T[I_z]$ reveals that sample $|y_i|$ must be in set $S_2 = \{7, 8, 9, 10, 11, 12, 13, 14, 15\}$, then by intersecting both sets we know that $|y_i| \in S_1 \cap S_2 = \{7, 8\}$, which is much more precise information.

Last-Jump: If the elements of an interval $I[r]$ in I are divided over two cache-lines of T, we can sometimes track the search for the element to sample. If a small part of $I[r]$ is in one cache-line, and the remaining part of $I[r]$ is in another, we are able to distinguish if this small part has been accessed. For example, interval $I[r] = \{5, 6, 7, 8, 9\}$ is divided over two cache-lines of T: cache-line $T_1 = \{0, 1, 2, 3, 4, 5, 6, 7\}$ and line $T_2 = \{8, 9, 10, 11, 12, 13, 14, 15\}$. The binary search starts in the middle of $I[r]$, at value 7, which means line T_1 is always accessed. However, only for values $\{8, 9\}$ also line T_2 is accessed. So if both lines T_1 *and* T_2 are accessed, we know that sample $|y_i| \in \{8, 9\}$.

We will restrict ourselves to only look for cache access patterns that give even more precision, at the expense of requiring more signatures:

1. The first restriction is to only look at cache weaknesses (of type Intersection or Last-Jump), in which the number of possible values for sample $|y_i|$ is two. Since we do a binary search within an interval, this is the most precision one can get (unless an interval is unique): after the last comparisons (table lookup in T), one of two values will be returned. This means that by picking either of these two values we limit the error of $|y_i|$ to at most 1.
2. The probabilities of sampling values using CDT sampling with guide table I are

known to match the following probability requirement :

$$\sum_{r=0}^{255} \mathbb{P}[X = x \mid X \in I[r]] = \frac{\rho_\sigma(x)}{\rho_\sigma(\mathbb{Z})}. \tag{8.1}$$

Due to the above condition, it is possible that adjacent intervals are partially overlapping. That is, for some r, s we have that $I[r] \cap I[s] \neq \emptyset$. In practice, this only happens for $r = s + 1$, meaning adjacent intervals might overlap. For example, if the probability of sampling x is greater than $1/256$, then x has to be an element in at least two intervals $I[r]$. Because of this, it is possible that for certain parts of an interval $I[r]$, there is a biased outcome of the sample.

The second restriction is to only consider cache weaknesses for which additionally one of the two values is significantly more likely to be sampled, i.e., if $|y_i| \in \{\gamma_1, \gamma_2\} \subset I[r]$ is the outcome of cache access patterns, then we further insist on

$$\mathbb{P}[|y_i| = \gamma_1 \mid |y_i| \in \{\gamma_1, \gamma_2\} \subset I[r]] \gg \mathbb{P}[|y_i| = \gamma_2 \mid |y_i| \in \{\gamma_1, \gamma_2\} \subset I[r]].$$

So we search for values γ_1 so that $\mathbb{P}[|y_i| = \gamma_1 \mid |y_i| \in \{\gamma_1, \gamma_2\} \subset I[r]] = 1 - \alpha$ for small α, which also matches access patterns for the first restriction. Then, if we observe a matching access pattern, it is safe to assume the outcome of the sample is γ_1.

3. The last restriction is to only look at cache-access patterns, which reveal that $|y_i|$ is larger than $\beta \cdot \mathbb{E}[\langle \mathbf{s}, \mathbf{c} \rangle]$, for some constant $\beta \geqslant 1$, which is an easy calculation using the distributions of $\mathbf{s}, \mathbf{c}$. If we use this restriction in our attack targeted at coefficient y_i of $\mathbf{y}$, we learn the sign of $|y_i|$ by looking at the sign of coefficient z_i of $\mathbf{z}$, since:

$$\mathrm{sign}(y_i) \neq \mathrm{sign}(z_i) \leftrightarrow \langle \mathbf{s}, \mathbf{c} \rangle > (y_i + z_i).$$

So by requiring that $|y_i|$ must be larger than the expected value of $\langle \mathbf{s}, \mathbf{c} \rangle$, we expect to learn the sign of y_i. We therefore omit the absolute value sign in $|y_i|$ and simply write that we learn $y_i \in \{\gamma_1, \gamma_2\}$, where the γ's took over the sign of y_i (which is the same as the sign of z_i).

There is some flexibility in these restrictions, in choosing parameters α, β. Choosing these parameters too restrictively, might lead to no remaining cache-access patterns, choosing them too loosely makes other parts fail.

In the last part of the attack described next, we use LLL to calculate short vectors of a certain (random) lattice we create using BLISS signatures. We noticed that LLL works very well on these lattices, probably because the basis used is sparse. This implies that the vectors are already relatively short and orthogonal. The parameter α determines the shortness of the vector we look for, and therefore influences if an algorithm like LLL finds our vector. For the experiments described in Section 8.5, we required $\alpha \leqslant 0.1$. This made it possible for every parameter set we used in the experiments to always have at least one cache-access pattern to use.

Parameter β influences the probability that one makes a huge mistake when comparing the values of y_i and z_i. However, for the parameters we used in the experiments, we did not find recognizable cache-access patterns which correspond to small y_i. This means, we did not need to use this last restriction to reject certain cache-access patterns.

8.3.2 – Exploitation. For simplicity, we assume we have one specific cache access pattern, which reveals if $y_i \in \{\gamma_1, \gamma_2\}$ for $i = 0, \dots, n-1$ of polynomial $\mathbf{y}$, and if this is the case, y_i has probability $(1-\alpha)$ to be value γ_1, with small α. In practice however, there might be more than one cache weakness, satisfying the above requirements. This would allow the attacker to search for more than one cache access pattern done by the victim. For the attack, we assume the victim is creating N signatures[1] $(\mathbf{z}_j, \mathbf{c}_j)$ for $j = 1, \dots, N$, and an attacker is gathering these signatures with associated cache information for noise polynomial $\mathbf{y}_j$. We assume the attacker can search for the specific cache access pattern, for which he can determine if $y_{ji} \in \{\gamma_1, \gamma_2\}$. For the cases revealed by cache access patterns, the attacker ends up with the following equation:

$$z_{ji} = y_{ji} + (-1)^{b_j} \langle \mathbf{s}, \mathbf{c}_{ji} \rangle, \tag{8.2}$$

where the attacker knows coefficient z_{ji} of $\mathbf{z}_j$, rotated coefficient vectors $\mathbf{c}_{ji}$ of challenge $\mathbf{c}_j$ (both from the signatures) and $y_{ji} \in \{\gamma_1, \gamma_2\}$ of noise polynomial $\mathbf{y}_j$ (from the side-channel attack). Unknowns to the attacker are bit b_j and $\mathbf{s}$.

If $z_{ji} = \gamma_1$, the attacker knows that $\langle \mathbf{s}, \mathbf{c}_{ji} \rangle \in \{0, 1, -1\}$. Moreover, with high probability $(1-\alpha)$ the value will be 0, as by the second restriction y_{ji} is biased to be value γ_1. So if $z_{ji} = \gamma_1$, the attacker adds $\xi_k = \mathbf{c}_{ji}$ to a list of *good* vectors. The restriction $z_{ji} = \gamma_1$ means that the attacker will in some cases not use the information in Equation (8.2), although he knows that $y_{ji} \in \{\gamma_1, \gamma_2\}$.

When the attacker collects enough of these vectors $\xi_k = \mathbf{c}_{ji}; 0 \leqslant i \leqslant n-1, 1 \leqslant j \leqslant N, 1 \leqslant k \leqslant n$, he can build a matrix $\mathbf{L} \in \{-1, 0, 1\}^{n \times n}$, whose columns are the ξ_k's. This matrix satisfies:

$$\mathbf{sL} = \mathbf{v} \tag{8.3}$$

for some unknown but short vector $\mathbf{v}$. The attacker does not know $\mathbf{v}$, so he cannot simply solve for $\mathbf{s}$, but he does know that $\mathbf{v}$ has norm about $\sqrt{\alpha n}$, and lies in the lattice spanned by the rows of $\mathbf{L}$. He can use a lattice reduction algorithm, like LLL, on $\mathbf{L}$ to search for $\mathbf{v}$. LLL also outputs the unimodular matrix $\mathbf{U}$ satisfying $\mathbf{UL} = \mathbf{L}'$. The attack tests for each row of $\mathbf{U}$ (and its rotations) whether it is sparse and could be a candidate for $\mathbf{s} = \mathbf{f}$. As stated before, correctness of a secret key guess can be verified using the public key.

This last step does not always succeed, but does with high probability. To make sure the attack succeeds, this process is randomized and repeated if necessary. Instead of collecting exactly n vectors $\xi_k = \mathbf{c}_{ji}$, we gather $m > n$ vectors, and pick a random subset of n vectors as input for LLL. While we do not have a formal analysis of the success probability, experiments (see Section 8.5) confirm that this method works and succeeds in finding the secret key (or its negative) in few rounds of randomized repetition.

A summary of the attack is given in Algorithm 8.8.

8.4 — Attack 2: Bernoulli sampling

In this section, we discuss the foundations and strategy of our second cache attack on the Bernoulli-based sampler (Algorithms 8.5, 8.6, and 8.7). We show how to exploit the fact that this method uses a small table ET, leaking very precise information about the sampled value.

[1] Here $\mathbf{z}_j$ refers to the first signature polynomial $\mathbf{z}_{j1}$ of the jth signature $(\mathbf{z}_{j1}, \mathbf{z}_{j2}^{\dagger}, \mathbf{c}_j)$.

Algorithm 8.8 Cache-attack on BLISS with CDT Sampling

```
Input: Access to cache memory of a victim with a key-pair $(\mathbf{A}, \mathbf{S})$. Input parameters
    $n, \sigma, q, \kappa$ of BLISS. Access to signature polynomials $(\mathbf{z}_1, \mathbf{z}_2^\dagger, \mathbf{c})$ produced using $\mathbf{S}$. Victim
    uses CDT sampling with tables $T, I$ for noise polynomials $\mathbf{y}$. Cache weakness that
    allows to determine if coefficient $y_i \in \{\gamma_1, \gamma_2\}$ of $\mathbf{y}$, and when this is the case, the
    value of $y_i$ is biased towards $\gamma_1$
Output: Secret key $\mathbf{S}$
let $k = 0$ be the number of vectors collected so far and let $M = []$ be an empty list of
    vectors
while $(k < m)$:        // collect $m$ vectors $\xi_k$ before randomizing LLL
    collect signature $(\mathbf{z}_1, \mathbf{z}_2^\dagger, \mathbf{c})$, together with cache information for each coefficient
        $y_i$ of noise polynomial $\mathbf{y}$
    for each $i = 0, \ldots, n-1$:
        if $y_i \in \{\gamma_1, \gamma_2\}$ (determined via cache information) and $z_{1i} = \gamma_1$:
            add vector $\xi_k = \mathbf{c}_i$ to $M$ and set $k = k + 1$
while (1):
    choose random subset of $n$ vectors from $M$ and construct matrix $\mathbf{L}$ whose
        columns are those vectors from $M$
    perform LLL basis reduction on $\mathbf{L}$ to get: $\mathbf{UL} = \mathbf{L}'$, where $\mathbf{U}$ is a unimodular
        transformation matrix and $\mathbf{L}'$ is LLL reduced
    for each $j = 1, \ldots, n$:
        check if row $\mathbf{u}_j$ of $\mathbf{U}$ has the same distribution as $\mathbf{f}$ and if $(\mathbf{a}_1/2) \cdot \mathbf{u}_j \bmod 2q$
            has the same distribution as $2\mathbf{g}+1$. Lastly verify if $\mathbf{a}_1 \cdot \mathbf{u}_j + \mathbf{a}_2 \cdot (\mathbf{a}_1/2) \cdot \mathbf{u}_j \equiv$
            $q \bmod 2q$
        return $\mathbf{S} = (\mathbf{u}_j, (\mathbf{a}_1/2) \cdot \mathbf{u}_j \bmod 2q)$ if this is the case
```

8.4.1 – Weaknesses in cache. The Bernoulli-sampling algorithm described in Section 8.2.2 uses a table with exponential values $\mathrm{ET}[i] = \exp(-2^i/(2\sigma^2))$ and inputs of bit-size $\ell = O(\log K)$, which means this table is quite small. Depending on bit i of input x, line 3 of Algorithm 8.7 is performed, requiring a table look-up for value $\mathrm{ET}[i]$. In particular when input $x = 0$, no table look-up is required. An attacker can detect this event by examining cache activity of the sampling process. If this is the case, it means that the sampled value z equals 0 in Step 2 of Algorithm 8.5. The possible values for the result of sampling are $y \in \{0, \pm K, \pm 2K, \ldots\}$. So for some cache access patterns, the attacker is able to determine if $y \in \{0, \pm K, \pm 2K, \ldots\}$.

8.4.2 – Exploitation. We will use the same methods as described in Section 8.3.2, but now we know that for a certain cache access pattern the coefficient y_i is in the set $\{0, \pm K, \pm 2K, \ldots\}$, $i = 0, \ldots, n-1$ of the noise polynomial $\mathbf{y}$. If $\max|\langle \mathbf{s}, \mathbf{c} \rangle| < K$, (which is something anyone can check using the public parameters and which holds for typical implementations, see Section 8.2.1), we can determine y_i completely using the knowledge of signature vector $\mathbf{z}$. When more signatures[2] $(\mathbf{z}_j, \mathbf{c}_j); j = 1, \ldots, N$ are created, the attacker can search for the specific access pattern and verify whether $y_{ji} \in \{0, \pm K, \pm 2K, \ldots\}$, where y_{ji} is the i'th coefficient of noise polynomial $\mathbf{y}_j$.

[2] Again, $\mathbf{z}_j$ refers to the first signature polynomial $\mathbf{z}_{j1}$ of the jth signature $(\mathbf{z}_{j1}, \mathbf{z}_{j2}^\dagger, \mathbf{c}_j)$.

If the attacker knows that $y_{ji} \in \{0, \pm K, \pm 2K, \ldots\}$ and it additionally holds that $z_{ji} = y_{ji}$, where z_{ji} is the i'th coefficient of signature polynomial $\mathbf{z}_j$, he knows that $\langle \mathbf{s}, \mathbf{c}_{ji} \rangle = 0$. If this is the case, the attacker includes coefficient vector $\zeta_k = \mathbf{c}_{ji}$ in the list of *good* vectors. Also for this attack the attacker will discard some known y_{ji} if it does not satisfy $z_{ji} = y_{ji}$.

Once the attacker has collected n of these vectors $\xi_k = \mathbf{c}_{ji}; 0 \leqslant i \leqslant n-1, 1 \leqslant j \leqslant N, 1 \leqslant k \leqslant n$, he can form a matrix $\mathbf{L} \in \{-1, 0, 1\}^{n \times n}$, whose columns are the ξ_k's, satisfying $\mathbf{sL} = 0$, where 0 is the all-zero vector. With very high probability, the ξ_k's have no dependency other than introduced by $\mathbf{s}$. This means $\mathbf{s}$ is the only kernel vector. Note the subtle difference with Equation (8.3): we do not need to randomize the process, because we know the right-hand side is the all-zero vector. The attack procedure is summarized in Algorithm 8.9.

Algorithm 8.9 Cache-attack on BLISS with Bernoulli sampling

```
Input: Access to cache memory of victim with a key-pair $(\mathbf{A}, \mathbf{S})$. Input parameters
    $n, \sigma, q, \kappa$ of BLISS, with $\kappa < K$. Access to signatures $(\mathbf{z}_1, \mathbf{z}_2^\dagger, \mathbf{c})$ produced using $\mathbf{S}$.
    Victim uses Bernoulli sampling with the small exponential table to sample noise poly-
    nomial $\mathbf{y}$
Output: Secret key $\mathbf{S}$
 1: let $k = 0$ be the number of vectors gained so far and let $M = []$ be an empty list of
    vectors
 2: while($k < n$):
 3:     collect signature $(\mathbf{z}_1, \mathbf{z}_2^\dagger, \mathbf{c})$ together with cache information for each coefficient
        $y_i$ of noise polynomial $\mathbf{y}$
 4:     for each $i = 1, \ldots, n$ do:
 5:         if $y_i \in \{0, \pm K, \pm 2K, ..\}$ (according to cache information), and $z_{1i} = y_i$
            then add coefficient vector $\xi_k = \mathbf{c}_i$ as a column to $M$ and set $k = k + 1$
 6: form a matrix $\mathbf{L}$ from the columns in $M$. Calculate kernel space of $\mathbf{L}$. This gives a
    matrix $\mathbf{U} \in \mathbb{Z}^{\ell \times n}$ such that $\mathbf{UL} = 0$, where 0 is the all-zero matrix
 7: for each $j = 1, \ldots, \ell$ do:        // we expect $\ell = 1$
 8:     check if row $\mathbf{u}_j$ of $\mathbf{U}$ has the same distribution as $\mathbf{f}$ and if $(\mathbf{a}_1/2) \cdot \mathbf{u}_j \bmod 2q$
        has the same distribution as $2\mathbf{g}+1$. Lastly verify if $\mathbf{a}_1 \cdot \mathbf{u}_j + \mathbf{a}_2 \cdot (\mathbf{a}_1/2) \cdot \mathbf{u}_j \equiv$
        $q \bmod 2q$
 9:     return $\mathbf{S} = (\mathbf{u}_j, (\mathbf{a}_1/2) \cdot \mathbf{u}_j \bmod 2q)$ if this is the case
10: remove a random entry from $M$, put $k = k - 1$, goto step 2
```

Possible extensions One might ask why we not always use the knowledge of y_{ji}, since we can completely determine its value, and work with a non-zero right-hand side. Unfortunately, bits b_j from Equation 8.2 of the signatures are unknown. This means an attacker has to use a linear solver 2^N times, where N is the number of required signatures (grouping columns appropriately if they come from the same signature). For large N this becomes infeasible and N is typically on the scale of lattice dimension n (which is related to the security parameter). By requiring that $z_{ji} = y_{ji}$, we remove the unknown bit b_j from the Equation (8.2).

Similar to the first attack, an attacker might also use vectors $\xi_k = \mathbf{c}_{ji}$, where $\langle \mathbf{s}, \mathbf{c}_{ji} \rangle \in \{-1, 0, 1\}$, in combination with LLL and possibly randomization. This approach might help if fewer signatures are available, but the easiest way is to require exact knowledge, which comes at the expense of needing more signatures, but has a very fast and efficient offline part. Section 6.3 deals with this approximate information.

8.5 — Results with a perfect side-channel

In this section we provide experimental results, where we assume the attacker has access to a perfect side-channel: no errors are made in measuring the table accesses of the victim. We apply the attack strategies discussed in the previous two sections and show how many signatures are required for each strategy.

Attack setting Sections 8.3 and 8.4 outline the basic ideas behind cache attacks against the two sampling methods for noise polynomials $\mathbf{y}$ used in the target implementation of BLISS. We now consider the following idealized situation: the victim is signing random messages and an attacker collects these signatures. The attacker knows the exact cache-lines of the table look-ups done by the victim while computing the noise vector $\mathbf{y}$. We assume cache-lines have size 64 bytes and each element is 8 bytes large (type LONG). To simplify exposition, we assume the cache-lines are divided such that element i of any table is in cache-line $\lfloor i/8 \rfloor$.

Our test machine is an AMD FX-8350 Eight-Core CPU running at 4.1 GHz. We use the *research oriented* C++ implementation of BLISS, made available by the authors on their webpage [DDLL13a]. Both of the analyzed sampling methods are provided by the implementation, where the tables T, I and ET are constructed dependent on σ. We use the NTL library [Sho15] for LLL reductions and kernel calculations.

CDT sampling When the signing algorithm uses CDT sampling as described in Algorithm 8.4, the perfect side-channel provides the values of $\lfloor r/8 \rfloor$ and $\lfloor I_z/8 \rfloor$ of the table accesses for r and I_z in tables I and T. We apply the attack strategy of Section 8.3.

We first need to find cache-line patterns, of type intersection or last-jump, which reveal that $|y_i| \in \{\gamma_1, \gamma_2\}$ and $\mathbb{P}[|y_i| = \gamma_1 | \, |y_i| \in \{\gamma_1, \gamma_2\}] = 1 - \alpha$ with $\alpha \leqslant 0.1$. One way to do that is to construct two tables: one table that lists elements $I[r]$, that belong to certain cache-lines of table I, and one table that lists the accessed elements I_z inside these intervals $I[r]$, that belong to certain cache-lines of table T. We can then brute-force search for all cache weaknesses of type intersection or last-jump. For example, in BLISS-I the first eight elements of I (meaning $I[0], \ldots, I[7]$) belong to the first cache-line of I, but for the elements in $I[7] = \{7, 8\}$, the sampler accesses element $I_z = 8$, which is part of the second cache-line of T. This is an intersection weakness: if the first cache-line of I is accessed and the second cache-line of T is accessed, we know $y_i \in \{7, 8\}$. Similarly, one can find last-jump weaknesses, by searching for intervals $I[r]$ that access multiple cache-lines of T. Once we have these weaknesses, we need to use the biased restriction with $\alpha \leqslant 0.1$. This can be done by looking at all bytes except the first of the entry $\mathsf{T}[I_z]$ (this is already used to determine interval $I[r]$). If we denote the integer value of these 7 bytes by $(\mathsf{T}[I_z])_{\mathrm{byte} \neq 1}$, then we need to check if $\mathsf{T}[I_z]$ has property

$$(\mathsf{T}[I_z])_{\mathrm{byte} \neq 1} / (2^{56} - 1) \leqslant \alpha$$

(or $(\mathsf{T}[I_z])_{\mathrm{byte} \neq 1} / (2^{56} - 1) \geqslant (1 - \alpha)$). If one of these properties holds, then we have $y_i \in \{I_z - 1, I_z\}$ and $\mathbb{P}[|y_i| = I_z | \, |y_i| \in \{I_z - 1, I_z\}] = 1 - \alpha$ (or with I_z and $I_z - 1$ swapped).

Parameter Set	Type	Cache line u	Cache line(s) I_z	$\{\gamma_1, \gamma_2\}$	γ_1	α
BLISS-0 (Toy)	Last-Jump	25	6,7	{127,128}	127	0.0447
BLISS-I	Intersection	0	1	{7, 8}	8	0.0998
	Last-Jump	6	6,7	{55,56}	55	0.0246
	Last-Jump	21	25,26	{207, 208}	207	0.0465
	Last-Jump	24	31,32	{254,255 }	255	0.0431
	Last-Jump	27	40,41	{327,328 }	327	0.0200
	Last-Jump	28	41,42	{334,335}	335	0.0226
	Last-Jump	29	48,49	{390,391}	391	0.0104
	Last-Jump	30	51,52	{414,415}	415	0.0018
BLISS-II	Last-Jump	26	17,18	{143,144}	143	0.0643
BLISS-III	Last-Jump	8	10,11	{87,88}	87	0.0903
	Last-Jump	10	13,14	{111,112}	111	0.0139
	Last-Jump	18	24,25	{199,200}	199	0.0272
	Last-Jump	20	28,29	{231,232}	231	0.0087
BLISS-IV	Last-Jump	9	12,13	{103,104}	103	0.0545
	Last-Jump	10	14,15	{119,120}	119	0.0015

Table 8.2: We found these weaknesses in cache, for the five suggested parameter sets, satisfying the size and biased requirement, as described in Section 8.3. For each weakness, we give the type (intersection or last-jump), the corresponding values of $\lfloor u/8 \rfloor$ and $\lfloor I_z/8 \rfloor$, the possible outcomes $\{\gamma_1, \gamma_2\}$, where the outcome has probability $(1 - \alpha)$ to be γ_1.

For each of the suggested parameter sets of BLISS we found at least one of these weaknesses using the above method. The weaknesses are given in Table 8.2.

Cache weaknesses in CDT sampling

We collect m (possibly rotated) coefficient vectors $\mathbf{c}_j$ and then run LLL at most $t = 2(m - n) + 1$ times, each time searching for $\mathbf{s}$ in the unimodular transformation matrix using the public key. We consider the experiment failed if the secret key is not found after this number of trials; the randomly constructed lattices have a lot of overlap in their basis vectors which means that increasing t further is not likely to help. We performed 1000 repetitions of each experiment (different parameters and sizes for m) and measured the success probability p_{succ}, the average number of required signatures $\overline{N}$ to retrieve m usable challenges, and the average length of $\mathbf{v}$ if it was found. The expected number of required signatures $\mathbb{E}[N]$ is also given, as well as the running time for the LLL trials. This expected number of required signatures can be computed as:

$$\mathbb{E}[N] = \frac{m}{n \cdot \mathbb{P}[\text{CP}] \cdot \mathbb{P}[\langle \mathbf{s}_1, \mathbf{c} \rangle = 0]},$$

where CP is the event of a usable cache-access pattern for a coordinate of $\mathbf{y}$.

From the results in Table 8.3 we see that, although BLISS-0 is a toy example (with security level $\lambda \leqslant 60$), it requires the largest average number $\overline{N}$ of signatures to collect m columns, i.e., before the LLL trials can begin. This illustrates that the cache-attack

Parameter Set	m	p_{succ}	$\|\|\mathbf{v}\|\|_2^2$	$\overline{N}$	$\mathbb{E}[N]$	Offline Time (in s)
BLISS-0 (Toy)	256	0.690	10	2537	2518	1.9
$(n, \sigma, \kappa) =$	257	0.841	10	2547	2528	2.9
$(256, 100, 12)$	258	0.886	10	2565	2538	3.5
	259	0.903	10	2571	2548	4.0
	260	0.943	10	2580	2558	4.5
	261	0.943	10	2596	2568	4.6
BLISS-I	512	0.655	29	441	450	37.6
$(n, \sigma, \kappa) =$	513	0.809	29	442	451	60.0
$(512, 215, 23)$	514	0.881	29	442	452	71.3
	515	0.925	30	443	453	73.9
	516	0.95	30	446	454	81.3
	517	0.961	30	446	455	85.8
BLISS-II	512	0.478	33	2021	2020	37.5
$(n, \sigma, \kappa) =$	513	0.675	34	2023	2024	72.1
$(512, 107, 23)$	514	0.772	34	2030	2028	95.6
	515	0.818	35	2033	2032	110.4
	516	0.870	35	2033	2036	117.5
	517	0.897	35	2041	2040	122.0
BLISS-III	512	0.855	23	945	930	42.2
$(n, \sigma, \kappa) =$	513	0.950	23	946	932	51.6
$(512, 250, 30)$	514	0.975	23	951	934	55.9
	515	0.987	24	954	935	55.3
	516	0.987	24	952	937	55.8
	517	0.996	24	957	939	54.4
BLISS-IV	512	0.617	35	1206	1189	46.2
$(n, \sigma, \kappa) =$	513	0.817	36	1209	1191	75.3
$(512, 271, 39)$	514	0.885	36	1211	1194	88.4
	515	0.932	36	1215	1196	93.7
	516	0.947	36	1216	1198	102.4
	517	0.955	36	1217	1201	104.4

Table 8.3: Experimental results with a perfect side-channel, when BLISS is used with CDT sampling (Algorithm 8.4). For each parameter set, we managed to gather m equations from $\overline{N}$ signatures. The running time of the offline part is given in seconds.

depends less on the dimension n, but mainly on σ. For BLISS-0 with $\sigma = 100$, there is only one usable cache weakness with the restrictions we made.

For all cases, we see that a small increase of m greatly increases the success probability p_{succ}. The experimental results suggest that picking $m \approx 2n$ suffices to get a success probability close to 1.0. This means that one only needs more signatures to always succeed in the offline part.

Bernoulli sampling When the signature algorithm uses Bernoulli sampling from Algo-

Parameter Set	m	p_{succ}	$\overline{N}$	$\mathbb{E}[N]$	Offline Time (in s)
BLISS-0 (Toy)	256	1.0	1105	1102	0.8
BLISS-I	512	1.0	1671	1694	14.7
BLISS-II	512	1.0	824	839	14.4
BLISS-III	512	1.0	3018	2970	16.0
BLISS-IV	512	1.0	4223	4154	18.1

Table 8.4: Experimental results with a perfect side-channel, when BLISS is used with Bernoulli sampling (Algorithms 8.5, 8.6, 8.7).

rithm 8.6, a perfect side-channel determines if there has been a table access in table ET. Thus, we can apply the attack strategy given in Section 8.4. We require $m = n$ (possibly rotated) challenges $\mathbf{c}_i$ to start the kernel calculation. We learn whether any element has been accessed in table ET, e.g., by checking the cache-lines belonging to the small part of the table. We performed only 100 experiments this time, since we noticed that $p_{\text{succ}} = 1.0$ for all parameter sets with a perfect side-channel. This means that the probability that n random challenges $\mathbf{c}$ are linearly independent is close to 1.0. We state the average number $\overline{N}$ of required signatures in Table 8.4. This time, the expected number is simply:

$$\mathbb{E}[N] = \left(\left(\frac{1}{\rho_\sigma(\mathbb{Z})} \sum_{x=-\lfloor \tau\sigma/K \rfloor}^{\lfloor \tau\sigma/K \rfloor} \rho_\sigma(xK) \right) \cdot \mathbb{P}[\langle \mathbf{s}_1, \mathbf{c} \rangle = 0] \right)^{-1}$$

for $K = \lfloor \frac{\sigma}{\sigma_2} + 1 \rfloor$ and tail-cut $\tau \geqslant 1$.

Note that the number of required signatures is smaller for BLISS-II than for BLISS-I. This might seem surprising as one might expect it to increase or be about the same as for BLISS-I because the dimensions and security level are the same for these two parameter sets. However, σ is chosen a lot smaller in BLISS-II, which means that also value K is smaller. This influences $\overline{N}$ significantly as the probability to sample values xK is larger for small σ.

8.6 — Proof-of-concept implementation

So far, the experimental results were based on the assumption of a perfect side-channel: we assumed that we would get the cache-line of every table look-up in the CDT sampling and Bernoulli sampling. In this section, we reduce the assumption and discuss the results of more realistic experiments using the FLUSH+RELOAD technique.

When moving to real hardware some of the assumptions made in Section 8.5 no longer hold. In particular, allocation does not always ensure that tables are aligned at the start of cache lines and processor optimizations may pre-load memory into the cache, resulting in false positives. One such optimization is the *spatial prefetcher*, which pairs adjacent cache lines into 128-byte chunks and prefetches a cache line if an access to its pair results in a cache miss [Int12].

FLUSH+RELOAD on CDT sampling Due to the spatial prefetcher, FLUSH+RELOAD cannot be used consistently to probe two paired cache lines. Consequently, to determine access to two consecutive CDT table elements, we must use a pair that spans two unpaired cache lines. In Table 8.5, we show that when the CDT table is aligned at 16 bytes, we can always find such a pair for BLISS-I. Although this is not a proof that our attack works in all scenarios, i.e. for all σ and all offsets, it would also not be a solid defence to pick exactly those scenarios for which our attack would not work, e.g., because α could be increased.

Offset within cache-pair	Cache lines I_z	$\{\gamma_1, \gamma_2\}$	γ_1
0	15,16	{207,208}	207
16	8,9	{141,142}	141
32	1,2	{27,28}	27
48	0,1	{9,10}	9
64	3,4	{55,56}	55
80	7,8	{117,118}	117
96	6,7	{99,100}	99
112	9,10	{145,146}	145

Table 8.5: Last-jump weaknesses for BLISS-I, in the case of different offsets (in bytes) within the same cache-pairs. For every offset, we found a last-jump weakness satisfying the size and biased requirement ($\alpha < 0.1$), allowing the attacks described in Section 8.6.

The attack was carried out on an HP Elite 8300 with an i5-3470 processor. running CentOS 6.6. Before sampling each coordinate y_i, for $i = 0, \ldots, n-1$, we flush the monitored cache lines using the `clflush` instruction. After sampling the coordinate, we reload the monitored cache lines and measure the response time. We compare the response times to a pre-defined threshold value to determine whether the cache lines were accessed by the sampling algorithm.

A visualization of the FLUSH+RELOAD measurements for CDT sampling is given in Figure 8.1. Using the intersection and last-jump weakness of the sampling method in cache-memory, we can determine which value is sampled by the victim by probing two locations in memory. To reduce the number of false positives, we focus on one of the weaknesses from Table 8.2 as a target for the FLUSH+RELOAD. This means that the other weaknesses are not detected and we need to observe more signatures than with a perfect side-channel, before we collect enough columns to start with the offline part of the attack.

We executed 50 repeated attacks against BLISS-I, probing the last-jump weakness for $\{\gamma_1, \gamma_2\} = \{55, 56\}$. We completely recovered the private key in 46 out of the 50 cases. On average we require 3438 signatures for the attack, to collect $m = 2n = 1024$ equations. We tried LLL five times after the collection and considered the experiment a failure if we did not find the secret key in these five times. We stress that this is not the optimal strategy to minimize the number of required signatures or to maximize the success probability. However, it is an indication that this proof-of-concept attack is feasible.

Other processors We also experimented with a newer processor (Intel core i7-5650U) and found that this processor has a more aggressive prefetcher. In particular, memory locations near the start and the end of the page are more likely to be prefetched. Consequently, the alignment of the tables within the page can affect the attack success rate. We find that in a third of the locations within a page the attack fails, whereas in the other two

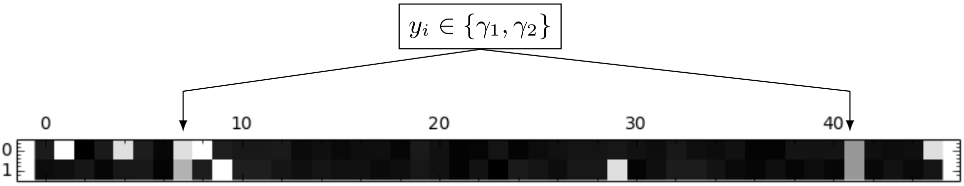


Figure 8.1: Visualization of FLUSH+RELOAD measurements of table look-ups for BLISS-I using CDT sampling with guide table I. Two locations in memory are probed, denoted in the vertical axis by 0, 1, and they represent two adjacent cache-lines. For interval $I[51] = [54, 57]$, there is a last-jump weakness for $\{\gamma_1, \gamma_2\} = \{55, 56\}$, where the outcome of $|y_i|$ is biased towards $\gamma_1 = 55$ with $\alpha = 0.0246$. For each coordinate (the horizontal axis), we get a response time for each location we probe: dark regions denote a long response time, while lighter regions denote a short response time. When both of the probed locations give a fast response, it means the victim accessed both cache-lines for sampling y_i. In this case the attacker knows that $|y_i| \in \{55, 56\}$; here for $i = 8$ and $i = 41$.

thirds it succeeds with probabilities similar to those on the older processor. We note that, as demonstrated in Table 8.2, there are often multiple weaknesses in the CDT. While some weaknesses may fall in unexploitable memory locations, others may still be exploitable.

FLUSH+RELOAD on Bernoulli sampling For attacking BLISS using Bernoulli sampling, we need to measure if table ET has been accessed at all. Due to the spatial prefetcher we are unable to probe all of the cache lines of the table. Instead, we flush all cache lines containing ET before sampling and reload only even cache lines after the sampling. Flushing even cache lines is required for the FLUSH+RELOAD attack. We flush the odd cache lines to trigger the spatial prefetcher, which will prefetch the paired even cache lines when the sampling accesses an odd cache line. Thus, flushing all of the cache lines gives us a complete coverage of the table even though we only reload half of the cache lines.

Since we do not get error-free side-channel information, we are likely to collect some $\mathbf{c}$ with $\langle \mathbf{s}, \mathbf{c}_i \rangle \neq 0$ as columns in $\mathbf{L}$. Instead of computing the kernel (as in the idealized setting) we used LLL (as in CDT) to handle small errors and we gathered more than n columns and randomized the selection of $\mathbf{L}$.

We tested the attack on a MacBook air with the newer processor (Intel core i7-5650U) running Mac OS X El Capitan. We executed 50 repeated attacks against BLISS-I, probing three out of the six cache lines that cover the ET table. We completely recovered the private key in 44 of these samples. On average we required 3294 signatures for the attack to collect $m = n + 100 = 612$ equations. The experiment is considered a failure if we did not find the secret key after trying LLL five times.

Conclusion Our proof-of-concept implementation demonstrates that in many cases we can overcome the limitations of processor optimizations and perform the attack on BLISS. The attack, however, requires a high degree of synchronization between the attacker and the victim, which we achieve by modifying the victim code. For a similar level of synchronization in a real attack scenario, the attacker will have to be able to find out when each coordinate is sampled. One possible approach for achieving this is to use the attack of Gullasch et al. [GBK11] against the Linux Completely Fair Scheduler. The combination of a cache attack with the attack on the scheduler allows the attacker to monitor each and every table access made by the victim, which is more than required for our attacks.

8.7 — Discussion of candidate countermeasures

In this chapter we presented cache attacks on two different discrete Gaussian samplers. In the following we discuss some candidate countermeasures against our specific attacks but note that other attacks might still be possible.

A standard countermeasure against cache-attacks are constant-time accesses. In this case constant-time table accesses means accessing *every element of the table for every coordinate* of the noise vector, and were also discussed (and implemented) by Bos et al. [BCNS15] for key exchange. This increased the number of table accesses by about two orders of magnitude. However, in the case of signatures the tables are much larger than for key exchange: a much larger parameter σ for the discrete Gaussian distribution is required. For 128 bits of security, $\sigma = 8/\sqrt{2\pi} \approx 3.19$ suffices for key exchange, resulting in a table size of 52 entries. In contrast, BLISS-I uses $\sigma = 215$, resulting in a table size of 2580 entries. It therefore seems that this countermeasure induces significant overhead for signatures: at least as much as for the key exchange. It might be the case that constant-time accesses to a certain part of the table is already sufficient as a countermeasure against our attack, but it is unclear how to do this precisely. One might think that constant-time accesses to table I in the CDT sampler is already sufficient as a countermeasure. In this case, the overhead is somewhat smaller, since I contains 256 entries. However, the last-jump weakness only uses the knowledge of accesses in the T table, which is still accessible in that case.

In the case of the Bernoulli-based sampler, doing constant-time table accesses does not induce that much overhead: the size of table ET is about $\ell \approx 2 \log K$. This means swapping line 2 and 3 of Algorithm 8.7 might prevent our attack as all elements of ET are always accessed. Note that removing line 4 of Algorithm 8.7 (and returning 0 or 1 at the end of the loop) does not help as a countermeasure. It does make the sampler constant-time, but we do not exploit that property. We exploit the fact that table accesses occur, depending on the input.

Concurrent work by Saarinen [Saa18] discusses another candidate countermeasure: the VectorBlindSample procedure. The VectorBlindSample procedure basically samples m vectors of discrete Gaussian values with a smaller σ, shuffles them in between, and adds the resulting vectors. The problem of directly applying our attack is that we need side-channel information of all summands for a coefficient. The chances for this are quite small. However, it does neither mean that other attacks are not possible nor that it is impossible to adapt our attack. In fact, recent work [Pes16] has shown (adapted) attacks that still work despite the usage of this shuffling.

More recently, another promising approach [MW17] uses several building blocks to generate discrete Gaussian samples in an efficient and simple manner. A sampler for a discrete Gaussian distribution with small standard deviation is used as core sampler, and multiple samples are combined to generate values with arbitrary standard deviation. As the core sampler for values with small standard deviation can be made constant-time more easily (as it only requires small tables), the whole sampler is made constant-time more easily.

Alternatives to Gaussian noise. This chapter shows that high-precision Gaussian samplers are a prime target for attacking lattice-based schemes. And while the above countermeasure can fix the exploited leak in this specific implementation, different attack techniques and side-channels might still allow for key recovery. Due to the complexity, implementing such samplers both correctly and efficiently is already challenging and error prone, even without considering side-channel attacks.

Some cryptographers seem to have noted this, as there exist lattice-based schemes that avoid Gaussians for these reasons. For instance, the NewHope key exchange [ADPS16] uses the centered binomial distribution (which is trivial to sample from) as a low-precision approximation to Gaussians. The lattice-based signature schemes TESLA [ABB$^+$16] and Dilithium [DLL$^+$17] also avoid discrete Gaussians and use uniform noise instead. Both proposals cite implementation concerns as a motivation for this design choice.

Another new approach is that of rounded Gaussians [HLS18]. Sampling from the continuous Gaussian distribution is easier than sampling from the discrete Gaussian distribution: it can be done with basic arithmetic, coupled with the sine and cosine functions. A discrete value could then be obtained by rounding this Gaussian value to the nearest integer. However, this rounded Gaussian distribution is not exactly the discrete Gaussian distribution. Hence in [HLS18] the authors adapt the proofs of BLISS to handle these rounded Gaussians.

8.8 — Other samplers

In the following section we look at two other methods of sampling from the discrete Gaussian distribution: Knuth-Yao [KY76, DG14] and the discrete Ziggurat [BCG$^+$13]. While we did not implement an attack against any specific implementations that use these samplers, we examine different ways to exploit specific ways to implement these samplers. The attack target remains to learn a BLISS secret key. This section only presents theoretical concepts and can be skipped as it is supplemental work. We leave an experiment-based evaluation of the described attacks for future work.

8.8.1 – Knuth-Yao. The basic idea of the Knuth-Yao sampling method [KY76, DG14] is to build a binary tree, known as the discrete distribution generating (DDG) tree, using binary representations of the probability distribution D_σ. At sampling time, one performs a random walk in this tree. A different representation of this tree is given using a binary matrix $P_{\text{mat}} \in \{1, 0\}^{\tau\sigma \times \lambda}$, where λ indicates the precision of the table. Each row of P_{mat} corresponds to the probability of sampling the index of the row. Each element of P_{mat} represents a node in the DDG tree, and each non-zero element corresponds to a terminal node in the tree. When such a terminal node is hit, the corresponding index of the row is output. One can again restrict to the non-negative part of the discrete Gaussian distribution, and draw a random sign at the end. Algorithm 8.10 describes the steps of Knuth-Yao, more details can be found for instance in [DG14].

Weaknesses in cache. The value to track in this method is row, which can be done by tracking the end of the for-loop, using knowledge of the accesses in P_{mat}. Notice that the 2-dimensional array P_{mat} is stored as a 1-dimensional array, storing $P_{\text{mat}}[row][col]$ at $P_{\text{mat}}[row \cdot MAXCOL + col]$ in memory. This means that it is possible, depending on e.g. storage type, size and precision, that $P_{\text{mat}}[i][col]$ and $P_{\text{mat}}[i-1][col]$ are stored in different cache-lines for *all* $i \in \{0, \ldots, MAXROW\}$. If, during sampling time, cache-line of $P_{\text{mat}}[i][col]$ is accessed and $P_{\text{mat}}[i-1][col]$ is not, it means that $|y| = i$. For instance,

Algorithm 8.10 Knuth-Yao Sampling

```
Input: Probability matrix $P_{mat} \in \{1,0\}^{\tau\sigma\times\lambda}$
Output: Sample $y$ with probability according to $D_\sigma$
set $d = 0$ and $col = 0$
while ($true$):
    sample bit $r$ uniformly
    $d = 2d + r$
    for $row = MAXROW$ down to 0 do
        $d = d - P_{mat}[row][col]$
        if $d = -1$ then
            set $x = row$
            sample bit $s$ uniform
            return $y = (-1)^s x$
    $col = col + 1$
    if $col = MAXCOL$ then restart
```

in [dCRVV15] the entries are stored as data-type INT, resulting in this weakness.

But even when multiple rows are stored in the same cache-line, which is more likely, there are possibilities of tracking the for-loop. Suppose $P_{mat}[j][col]$ up to $P_{mat}[k][col]$ are stored in the same cache-line. If the attacker is able to figure out the *number of* accesses ℓ in this cache-line, he knows that $|y| = k - \ell$. For instance, by using FLUSH+RELOAD *between* table accesses from the victim might leak this number ℓ. Note that the attacker has to be much faster than the victim in order to perform operations even between table accesses, but it might be possible to use degrading techniques such as done in [ABF$^+$16]. It is possible that all values between $P_{mat}[j][col]$ up to $P_{mat}[k][col]$ were accessed and the for-loop has not ended. There are two ways of overcoming this: he can also monitor the next cache-line, to measure if this line is not accessed. Or the attacker can require that $\ell < k - j$ before acting on it, since he then knows that the for-loop ended for sure.

For both of these exploits, the attacker does not make errors if all cache-line measurements were correct. This means he can apply the same attack strategy for BLISS with the Bernoulli-based sampler, as shown in Section 4, and use similar adaptations to real-life experiments, as shown in Section 6.

Countermeasures. To block our specific exploit, the end of the for-loop has to be hidden in terms of accesses in P_{mat}. One possible countermeasures is similar to that of the CDT sampler: perform the implementation in constant time. It means that the implementation always has to finish the for-loop in this case. However, that means that one has to access $\tau\sigma$ elements for each sample, which could mean a significant slow-down for the sampler, as stated earlier.

In [dCRVV15, RRVV14], a modification has been proposed for the Knuth-Yao sampler, that uses one (or two) additional look-up tables L, which represents the first few levels of the binary tree. At sampling time, one first uses L to see if a sample can be determined, and otherwise performs a loop as in Algorithm 8.10. Depending on the size of L, it only means the complexity of our exploit increases: only for samples that are not covered in L, the loop in P_{mat} is performed. This simply means requiring more signatures before the attack can be finished. [RRVV14] discusses the Knuth-Yao sampler for a lattice-based encryption scheme, where the authors noticed the possibility of a timing/power attack for this sampling method. The authors tried to mitigate this exploit, when the sampling did not succeed with the look-up table, by applying a random permutation at the end of sampling. It has the effect that we can only measure that a certain weakness has occurred in the sampling of the noise vector but we do not know the index. However, we can still check every index of **z** for a possible match since we want the Gaussian sample to be equal to the corresponding coordinate of **z**. A sample is discarded if more than one index is possible. On top of all, it might also be possible to spy on the look-up tables L themselves.

8.8.2 – Discrete Ziggurat. The discrete Ziggurat [BCG+13] is a sampler, that allows for a time-memory tradeoff to adapt to the needs of a use-case. The idea is to divide the area beneath the probability distribution function of D_σ into m rectangles R_i of equal volume. At sampling time, a random rectangle is picked and a sample inside this picked rectangle is output, depending on a further rejection step. These rectangles can be divided into two parts: one part where a sample is immediately accepted, and another part where rejection sampling is performed. Only for the rejection sampling one needs high-precision values of the probability distribution D_σ. In practice two tables are used to store the x- and y-coordinates of the rectangles. Picking more rectangles m means that fewer rejection steps are required on average, speeding up the sampler. However, naturally a higher m also means more storage space. More details are in [BCG+13], a simplified version of the sampler is given in Algorithm 8.11. Here, $(R_x[i], R_y[i])$ denotes the lower right corner of rectangle R_i for $1 \leqslant i \leqslant m$, where $R_x[i]$ is rounded down to the nearest integer.

Algorithm 8.11 Discrete Ziggurat Sampling

```
Input: Number of rectangles $m$, tables $R_x$ and $R_y$ of size $m$,
Output: Sample $y$ with probability according to $D_\sigma$
while (true):        // continue until success
    sample $i \leftarrow \{1, \dots, m\}$, $s \leftarrow \{0, 1\}$, $x \leftarrow \{0, ..., R_x[i]\}$ uniformly.
    if $0 < x \leqslant R_x[i-1]$ then return $y = (-1)^s x$
    else
        if $x = 0$ then
            sample bit $b$ uniformly
            if $b = 0$ then return $y = (-1)^s x$
            else continue
        else        // in rejection area of $R_i$
            sample bit $b$, using values $R_x[i-1], R_x[i], R_y[i-1], R_y[i]$ (details in [BCG+13])
            if $b = 0$ return $y = (-1)^s x$.
            else continue
```

Weaknesses in Cache. For a potential cache-attack, parameter m is of high influence: a small m means a small table, which means fewer possibilities of cache weaknesses. However, to compete with the CDT sampler in terms of speed, m has to be chosen quite large, for instance 256 or 512.

The weakness in this sampler is based on the fact that if a victim has to perform rejection steps, the outcome is bounded to a certain range: $|y| \in (R_x[i-1], R_x[i]]$ for some $i \in \{0, \dots, m\}$. Furthermore, only when rejection steps have to be performed, values $R_y[i-1]$ and $R_y[i]$ are required. Now in the case that we have a large m, it is possible that this range of values for $|y|$ is simply one unique value: when $R_x[i-1]+1 = R_x[i]$. If it happens to be that $R_x[i-1]$ and $R_x[i]$ are in different cache-lines, we can monitor for cache access patterns similar to the last-jump weakness for the CDT sampler. This means an attacker monitors two adjacent cache-lines, corresponding to the accesses of $R_x[i-1]$ and $R_x[i]$. In addition to this, also the cache-line corresponding to value $R_y[i-1]$ (and/or $R_y[i]$) has to be monitored. Only in the case of rejection steps, these values are accessed. When all these three cache-lines are accessed by the victim and the sampling stops, the attacker knows that $|y| = R_x[i]$.

This means an attacker can monitor certain cache-lines, and when these are accessed, he knows the value of $|y|$ without making errors. It means also for this sampler, the attack strategy of Section 4 can be applied.

Countermeasures. For our specific exploit, we make use of the fact that we know when rejection steps are performed, since only in that case R_y values are accessed. This means that in this case, one can overcome this issue by always access the corresponding R_y values, no matter if rejection steps are performed. This simply requires two more loads. However, it does not mean other exploits are not possible.

CHAPTER 9

To BLISS-B or not to be

9.1 — Overview

Context. In the previous chapter, we presented the first-of-its-kind side-channel attack on a lattice-based signature scheme called BLISS. The attack targets a noise vector that is sampled from the discrete Gaussian distribution and used to hide information of the secret key in the signature. Dedicated algorithms, among others those proposed by the authors of BLISS, are used to sample from this distribution. We showed how to retrieve estimations of some elements of the noise vector by carefully examining access-patterns to cache memory. Using signatures and recovered (estimations of) noise elements by the side-channel, we then recovered the secret key by means of a lattice-basis reduction.

However, the presented attacks have some shortcomings. First, in the proof-of-concept cache attack (Section 8.6) we only target the "research-oriented" reference implementation of BLISS [DDLL13b]. We also modified its code in order to achieve perfect synchronization of the attacker with the calls to the sampler. While this method demonstrates the existence and exploitability of the side-channel, it is not a realistic real-world setting.

Second, and maybe more importantly, the attacks do not apply to BLISS-B [Duc14], an improved version of BLISS that accelerates the signing operation by a factor of up to 2.8, depending on the used parameter set. Due to its better performance, this new variant is the only option for the adoption of BLISS in strongSwan, an IPsec-based VPN suite [str15].

The new attack target. Recall that the main operation in BLISS is to multiply the secret key $\mathbf{s}$ with a binary *challenge vector* $\mathbf{c}$ and add a *noise vector* $\mathbf{y}$ which is sampled at random from a discrete Gaussian distribution. The result $\mathbf{z} = \mathbf{y} + (-1)^b(\mathbf{s} \cdot \mathbf{c})$, where b is a random bit, together with the challenge vector $\mathbf{c}$ form the signature. In Chapter 8 we used the recovered values of $\mathbf{y}$ over many signatures, to construct a lattice from the challenge vectors such that $\mathbf{s}$ is part of the solution to the shortest vector problem in that lattice. This short vector is found using a lattice-basis reduction.

In BLISS-B, however, the secret $\mathbf{s}$ is multiplied with a ternary vector $\mathbf{c}' \in \{-1, 0, 1\}^n$ for which $\mathbf{c}' \equiv \mathbf{c} \bmod 2$. Still, only the binary version $\mathbf{c}$ is part of the signature and $\mathbf{c}'$ is undisclosed. Thus, the signs of the coefficients of the used challenge vectors are unknown and constructing the appropriate lattice to find $\mathbf{s}$ is infeasible for secure parameters. Note that this problem (or similar ones) are also present in other works on implementation attacks targeting the original BLISS, both for side-channel attacks [Pes16] as well as fault attacks [BBK16, EFGT16]. Hence, one might be tempted to think of BLISS-B as a "free" side-channel countermeasure.

Summary. In this chapter we show that BLISS-B is not a free side-channel countermeasure. First, we present a new key-recovery attack that can, given side-channel information on the Gaussian samples in $\mathbf{y}$, recover the secret key $\mathbf{s}$. Apart from being applicable to BLISS-B, this new key recovery approach can also increase the efficiency (in the number of required side-channel measurements) of earlier attacks on the original BLISS (e.g. from the previous chapter or [Pes16]). And second, we use this new key-recovery approach to mount an asynchronous cache attack on the BLISS implementation provided by strongSwan. Hence, we attack a real-world implementation in a realistic setting.

Our key-recovery attack consists of four steps:

- In the first step, we use side channels to gather information on the noise vector. We use these leaked values, together with known challenge vector elements, to construct a linear system of equations. However, the signs in this system are unknown.
- In the second step, we solve the above system. We circumnavigate the problem of unknown signs by using the fact that $-1 \equiv 1 \bmod 2$. That is, we first solve the linear system in GF(2), instead of over the integers. Due to errors in the side channel the linear system may include some errors. Solving such a system is known as the Learning Parity with Noise (LPN) problem. We use an LPN solving algorithm to learn the parity of the secret key elements, i.e. to find $\mathbf{s} \bmod 2$.
- In some parameter sets of BLISS-B, the key $\mathbf{s} \in \{0, \pm 1\}^n$ and thus the above already uniquely determines the magnitude of the coefficients. In others, however, the secret key can also have some coefficients with ± 2, which have parity zero. In the third step, we employ one of two heuristics (depending on the parameter set) to identify those, both heuristics exploit the magnitude of the coefficients of $\mathbf{s} \cdot \mathbf{c}'$. The first heuristic uses an Integer Programming solver. The second uses a Maximum Likelihood estimate.
- At this stage we know the magnitude of each of the coefficients of the secret key $\mathbf{s}$. In the fourth step, we finalize the attack and extract $\mathbf{s}$. We construct a Shortest Vector Problem (SVP) based on the public key and the known information about the secret key. We solve this problem using the BKZ lattice-reduction algorithm.

For the idealized cache-attacks presented in the previous chapter (Section 8.5) and the BLISS-I parameter set, the new method can reduce the number of required signatures from 450 to 325.

We then perform a cache attack on the BLISS-B implementation which is deployed as part of the strongSwan VPN software. Unlike the attack in the previous chapter, we allow the adversary to be asynchronous and running in a different process than the victim. The adversary uses the FLUSH+RELOAD attack by [YF14], combined with the amplification attack of [ABF+16]. Furthermore, we target a real-world implementation and not a research-oriented reference implementation. Consequently, our attack scenario is much more realistic.

Differences with published version. There are several changes in this chapter compared to the published paper [PGY17]. The main changes are the removal of the details about the error correction mod 2 (Section 9.3.4), as well as the details of the Maximum Likelihood technique (Section 9.3.5), as the author of this thesis did not contribute to these parts. Section 9.6 is an additional section written solely by the author of this thesis.

Organization. In Section 9.2, we present BLISS-B and discrete Gaussians. Then, in Section 9.3 we discuss previous work on side channel analysis. We then show our improved key-recovery attack. We evaluate our new method in Section 9.4 by comparing it to earlier work. In Section 9.5, we perform a full attack on the BLISS implementation provided by strongSwan. We discuss a possible improvement in Section 9.6. We give specific countermeasures in Section 9.7.

9.2 — Preliminaries

In this section, we briefly describe background concepts required for the rest of the chapter. Most of the required background is already handled in the previous chapters. For the required background on lattices see Section 6.2. This chapter again relies on the knowledge of NTRU lattices and LLL/BKZ. For the required background on cache-attacks, see Section 6.3. The signature scheme BLISS and its algorithms, as well as several discrete Gaussian samplers are discussed in Section 8.2. Therefore, in this section we only state the differences between BLISS and BLISS-B.

The differences between the BLISS signature scheme and its optimized version BLISS-B are minor, although with a big impact on the applicability of side-channel attacks. Key-generation of both BLISS and BLISS-B are similar except that in BLISS-B keys no longer need to be rejected (Line 3 of Algorithm 8.1 is skipped). The verification algorithms of both schemes are identical (Algorithm 8.3), which means that signatures are both backward and forward compatible.

9.2.1 – BLISS-B. The BLISS-B signing procedure is given in Algorithm 9.1. The main difference to the original BLISS is Line 4 of the procedure. Recall that the challenge vector $\mathbf{c}$, used in the Fiat-Shamir transform [FS86], is computed by invoking a hash function H. In BLISS-B, this function returns a binary vector of length n and a Hamming weight of exactly κ. GreedySC (9.2) then computes the product $\mathbf{Sc}'$ for some vector $\mathbf{c}' \in \{-1, 0, +1\}^n$ that satisfies $\mathbf{c}' \equiv \mathbf{c} \mod 2$. Note that for the specific BLISS input $\mathbf{S} = (\mathbf{s}_1, \mathbf{s}_2) \in \mathcal{R}_{2q}^2$ in GreedySC, we have $m = 2n$ and $\mathbf{s}_i = \mathbf{S}_{1i}$ for $0 \leqslant i < n$ and $\mathbf{s}_i = \mathbf{S}_{2i}$ for $n \leqslant i < 2n$ where $\mathbf{S}_{1i}$ and $\mathbf{S}_{2i}$ are the rotations of $\mathbf{s}_1$ and $\mathbf{s}_2$, respectively, with possibly opposite sign matching the reduction in $\mathcal{R}_{2q}$. The generated $\mathbf{c}'$ contains information on the secret key, hence it is kept secret and not output as part of the signature. GreedySC is not part of the first version of BLISS (see Algorithm 8.2). Instead, the product $\mathbf{Sc}$ is used directly, i.e., $\mathbf{v}_1 = \mathbf{s}_1 \cdot \mathbf{c}$.

[DDLL13b] propose several parameter sets for different security levels. As these remain unchanged for BLISS-B, the parameters relevant for the attacks are (still) those in Table 8.1 of the previous chapter.

Discrete Gaussians. Like BLISS, also BLISS-B uses discrete Gaussian noise vectors to statistically hide the secret vector (Line 1 in Algorithm 9.1). For a background on this subject, see Section 8.2 of the previous chapter. The two described sampling methods, the CDT sampler and the Bernoulli sampler, are also analyzed in this work. In particular, the Bernoulli sampler is the method that is implemented in strongSwans BLISS-B implementation.

9.3 — An improved side-channel key-recovery technique

In this section, we present our new and improved side-channel attack on BLISS, that also works for BLISS-B. We first discuss why these steps are necessary by discussing the limitation of the attack described in the previous chapter.

Algorithm 9.1 BLISS-B Sign

```
Input: Message $\mu$, public key $\mathbf{A} = (\mathbf{a}_1, q-2)$, private key $\mathbf{S} = (\mathbf{s}_1, \mathbf{s}_2)$
Output: A signature $(\mathbf{z}_1, \mathbf{z}_2^\dagger, \mathbf{c})$
$\mathbf{y}_1 \leftarrow D_\sigma^n, \mathbf{y}_2 \leftarrow D_\sigma^n$
$\mathbf{u} = \zeta \cdot \mathbf{a}_1 \cdot \mathbf{y}_1 + \mathbf{y}_2 \bmod 2q$
$\mathbf{c} = H(\lfloor \mathbf{u} \rceil_d \bmod p \| \mu)$
$(\mathbf{v}_1, \mathbf{v}_2) = \mathrm{GreedySC}(\mathbf{S}, \mathbf{c})$
Sample a uniformly random bit $b$
$(\mathbf{z}_1, \mathbf{z}_2) = (\mathbf{y}_1, \mathbf{y}_2) + (-1)^b (\mathbf{v}_1, \mathbf{v}_2)$
Continue with some probability $f(\mathbf{v}, \mathbf{z})$, restart otherwise (details in [Duc14])
$\mathbf{z}_2^\dagger = (\lfloor \mathbf{u} \rceil_d - \lfloor \mathbf{u} - \mathbf{z}_2 \rceil_d) \bmod p$
return $(\mathbf{z}_1, \mathbf{z}_2^\dagger, \mathbf{c})$
```

Algorithm 9.2 GreedySC

```
Input: a matrix $\mathbf{S} \in \mathbb{Z}^{m \times n}$ and a binary vector $\mathbf{c} \in \mathbb{Z}^n$
Output: $\mathbf{v} = \mathbf{S}\mathbf{c}'$ for some $\mathbf{c}' \equiv \mathbf{c} \mod 2$
$\mathbf{v} = \mathbf{0} \in \mathbb{Z}^n$
Let $\mathcal{I}_c$ be the set of indices $0 \leqslant i \leqslant n-1$ where $\mathbf{c}_i = 1$
for $i \in \mathcal{I}_c$
    $\zeta_i = \mathsf{sign}(\langle \mathbf{v}, \mathbf{s}_i \rangle)$
    $\mathbf{v} = \mathbf{v} - \zeta_i \mathbf{s}_i$
return $\mathbf{v}$
```

9.3.1 – Limitations of Previous Attacks. The side-channel attacks on BLISS from the previous chapter have certain limitations and caveats. As already stated above, due to the unknown bit b, which is potentially different for each signature, the attacks in Chapter 8 only use samples where $z_i = y_i$ and thus $\langle \mathbf{s}, \mathbf{c}_i \rangle = 0$ (with high probability). This, however, only holds in roughly 15% of all samples (cf. Figure 10.2) and thus a lot of information is discarded. By finding a method to use all samples for the attack, the number of required signatures could drop drastically.

A second and more severe limitation is that the previous attacks do not apply to the improved BLISS-B signature scheme. The attacks recover the key by solving a (possibly erroneous) linear system $\mathbf{sL} \approx \mathbf{0}$, where $\mathbf{L}$ consists of the used challenge vectors $\mathbf{c}_i$. However, the GreedySC algorithm, which was added with BLISS-B, performs a multiplication of $\mathbf{s}$ with some unknown ternary $\mathbf{c}' \equiv \mathbf{c} \mod 2$, with $\mathbf{c}' \in \{-1, 0, 1\}^n$. In simple terms, the signs of the coefficients in $\mathbf{c}'$ (and thus also in the resulting lattice basis $\mathbf{L}'$) are unknown. Hence, a straight-forward solving of $\mathbf{sL}' \approx \mathbf{0}$ is not possible anymore.

On the practicality of previous attacks. A third limitation of the attack in Chapter 8 is the question of real-world applicability. The demonstrated attack (Section 8.6) targets an academic implementation that is not used in any “real-world” applications. Furthermore, the attack is synchronous. To achieve this, we modified the code of the BLISS implementation in order to interleave the phases of the FLUSH+RELOAD attack with the Gaussian sampler. In practice, it is not clear if an attacker can achieve such a level of synchronization without modifying the source, and an adversary that can modify the source can access the secret key directly without needing to resort to side channel attacks. Consequently, while we showed a proof-of-concept, the attack falls short in terms of real-world applicability.

9.3.2 – Overview of the improved attack. Our method consists of four major steps, each step reveals additional information on the secret signing key $\mathbf{s}$. The first step is equivalent to the attacks described in Section 8.3 and 8.4 of the previous chapter. That is, the attacker performs a side-channel attack, e.g., a cache attack, on the Gaussian-sampler component to recover some of the drawn samples y_i of $\mathbf{y}$. With this information we can construct a (possibly erroneous) system of linear equations over the integers, using knowledge on $z_i - y_i = (-1)^b(\mathbf{s} \cdot \mathbf{c}')$. (Section 9.3.3)

Due to the previously mentioned sign-uncertainty in BLISS-B (the recovered term $\mathbf{s} \cdot \mathbf{c}'$ instead of $\mathbf{s} \cdot \mathbf{c}$), the solution cannot be found with simple linear algebra in $\mathbb{Z}$. Instead, in Step 2 we solve this system in GF(2). For error correction, one can employ an LPN algorithm or simply brute-force if the error probabilities are small enough (Section 9.3.4)

This does not give us the full key, but instead $\mathbf{s}^* = \mathbf{s} \mod 2$. For some parameter sets however, there are some coefficients ± 2 (i.e., BLISS-0, BLISS-III and BLISS-IV have $\delta_2 > 0$). In Step 3, we retrieve their positions. We use the current knowledge on the secret key $\mathbf{s}^*$ to derive $\langle \mathbf{s}^*, \mathbf{c}_i \rangle$, and compare this with $z_i - y_i = \langle \mathbf{s}, \mathbf{c}'_i \rangle$ (obtained from the side channel). Based on that, we give two different methods in Section 9.3.5 to determine the positions of the ± 2 coefficients and derive $|\mathbf{s}| \in \{0, 1, 2\}^n$.

In the fourth step, we finally recover the full signing key. We use $|\mathbf{s}|$ to reduce the size of the lattice basis generated from the public key. We then perform a lattice reduction and search for $\mathbf{s}_2$ as a short vector in the lattice spanned by this smaller basis. Linear algebra then allows recovery of the full private key $(\mathbf{s}_1, \mathbf{s}_2)$. (Section 9.3.6)

We now give a more detailed description of these steps.

9.3.3 – Step 1: Gathering Samples. Akin to the previous chapter, we need to observe the generation of multiple signatures and use a side-channel to infer some of the elements of the corresponding noise vector $\mathbf{y} = \mathbf{y}_1$. In previous works, the exploited side channels were based on cache attacks (previous chapter) or on power analysis (in [Pes16]).

Side-channel analysis has to deal with measurement errors and other uncertainties. Due to these effects a recovered sample y_i might not be correct. In our scenario, the probability ϵ of such an error is known (or can be estimated to a certain extent) and can possibly be different for each sample.

For each recovered (and reassigned) sample y_i, we can write an equation $z_i = y_i + (-1)^b \langle \mathbf{s}, \mathbf{c}'_i \rangle$, which holds with probability $1 - \epsilon$. As the signs of coefficients of $\mathbf{c}'_i$ are unknown, we can simply ignore the multiplication with $(-1)^b$ and instead implicitly include this factor into $\mathbf{c}'_i$. Unlike in the previous chapter, we do not require that $\langle \mathbf{s}_1, \mathbf{c}_i \rangle = 0$ and thus can use all recovered samples. We compute the difference $t_i = z_i - y_i$ and rearrange all gathered $\mathbf{c}'_i$ into a matrix $\mathbf{L}'$ to get $\mathbf{s}\mathbf{L}' = \mathbf{t}$.

This system is defined over $\mathbb{Z}$. However, due to the unknown signs in the challenge $\mathbf{c}'$ it cannot be directly solved using straight-forward linear algebra, even in the case that all recovered samples are correct. Instead, a different technique is required.

9.3.4 – Step 2: Finding $\mathbf{s}_1$ mod 2. In the second attack step, we solve the above system by using the following observation. Line 6 of Algorithm 9.1, i.e., $\mathbf{z}_1 = \mathbf{y}_1 \pm \mathbf{s}_1 \cdot \mathbf{c}'$, is defined over $\mathbb{Z}$. That is, there is no reduction mod q involved[1]. Such an equivalence relation in $\mathbb{Z}$ also holds mod 2, i.e., in GF(2), whereas the reverse is not true.

In GF(2), we have that $-1 \equiv 1 \bmod 2$. This resolves the uncertainty in $\mathbf{L}'$ and we can, at least when assuming no errors in the recovered samples, solve the system $\mathbf{s}^*\mathbf{L}' = \mathbf{t}^*$ over GF(2). Here $\mathbf{s}^*$ and $\mathbf{t}^*$ denote $\mathbf{s}$ mod 2 and $\mathbf{t}$ mod 2, respectively. In the BLISS-I parameter set (Table 8.1), we have that $\delta_2 = 0$. Thus, $\mathbf{s}^*$ reveals the position of all $\lceil \delta_1 n \rceil = 154$ nonzero, i.e., (± 1), coefficients. However, a simple enumeration of all 2^{154} possibilities for $\mathbf{s}$ is still not feasible. Before we discuss a method to recover the signs of $\mathbf{s}$ and thus the full key, we briefly show how errors in $\mathbf{t}^*$ can be corrected.

Error Correction mod 2. As stated in Section 9.3.3, a recovered Gaussian sample y_i might not be correct. Hence, the right-hand-side of the system $\mathbf{s}^*\mathbf{L}' = \mathbf{t}^*$ is possibly erroneous. In the perfect side-channel attack on the Bernoulli sampler, there are no errors and the secret key can be found with simple linear algebra. However, in the cache attack on the CDT sampling algorithm (Section 8.3), errors cannot be avoided. For some cache weaknesses (see Table 8.2), it might be possible to find the secret key by gathering more signatures and trying a subset of challenges $\mathbf{c}$ that are hopefully error-free. However, the need for such an expensive technique increases the number of required signatures.

Another way is to rewrite the above equations in GF(2) as $\mathbf{s}^*\mathbf{L}' = \mathbf{t}^* + \mathbf{e}$. Here, $\mathbf{t}^*$ is errorless and the error is instead modeled as vector $\mathbf{e}$. Solving this system is exactly the LPN problem [Pie12], thus we employ an LPN solving algorithm [GJL14,LF06] to recover $\mathbf{s}^*$. The details for this part are outside the scope of this thesis, but can be found in the published version of this chapter [PGY17].

9.3.5 – Step 3: Recovering the Positions of Twos. After the above second attack step, we know $\mathbf{s}^* \equiv \mathbf{s} \bmod 2$. If we have $d_2 = \delta_2 n > 0$ (i.e., in BLISS-0, BLISS-III or BLISS-IV), we denote $\bar{\mathbf{s}} \in \{0,1\}^n$ the vector with $\bar{\mathbf{s}}_i = 1$ whenever $s_i = \pm 2$, i.e. this vector is non-zero at each coefficient where vector $\mathbf{s}$ has coefficient ± 2.

In the third attack step, we use one of two methods to recover $\bar{\mathbf{s}}$, one based on integer programming and the other based on a maximum likelihood test. Both make use of the fact that the weight κ of the challenge vector $\mathbf{c}$ (and hence also $\mathbf{c}'$) is relatively small. Thus, in any inner product $\langle \mathbf{s}, \mathbf{c}'_i \rangle$, only a small number of coefficients in $\mathbf{s}$ are relevant. From knowledge of $\mathbf{s}^*$, we can immediately derive how many of the selected coefficients are ± 1. We define this quantity as $\eta_1 = \langle \mathbf{s}^*, |\mathbf{c}_i| \rangle$. The other $\kappa - \eta_1$ are then either 0 or ± 2. We define the (unknown) number of twos as $\eta_2 = \langle \bar{\mathbf{s}}, |\mathbf{c}_i| \rangle$, this number is bound by $0 \leqslant \eta_2 \leqslant \min(d_2, \kappa - \eta_1)$

Both methods then compare the output of the side-channel analysis, i.e., $|z_i - y_i| = |\langle \mathbf{s}, \mathbf{c}'_i \rangle|$, to η_1 and use this to derive information on η_2. We will now discuss both methods.

Integer Programming Method. Our first method recovers $\bar{\mathbf{s}}$ by transforming the problem into an Integer Program. First, suppose we perfectly retrieved y_{ji} from a side-channel. If

$$|z_i - y_i| = |\langle \mathbf{s}, \mathbf{c}'_i \rangle| > \eta_1 + 1,$$

then we know that $\eta_2 > 0$, i.e. there has to be a at least one ± 2 involved making up for the difference in the above inequality. We save all $|\mathbf{c}_i|$ for which the above is true in a

[1] In fact, due to the parameter choices and the tailcut required by a real Gaussian sampler, $|\mathbf{y}_1 + \mathbf{s}_1 \cdot \mathbf{c}'|$ can never exceed q.

matrix $\mathbf{M}$. Then, we need to find a solution $\mathbf{r}$ for the following constraints:

$$\mathbf{Mr} \geqslant \mathbf{1}.$$

We also add another constraint stating that a solution must satisfy $\|\mathbf{r}\|_1 = \delta_2 n$, so that we end up with the correct number of coefficients in the solution. Also the indices where $\mathbf{s}^*$ is non-zero should be eliminated from the solution space.

Finding the solution $\bar{\mathbf{s}}$ can be seen as a minimal set cover problem with additional constraints. Here, the indices of $\mathbf{M}_i$ form sets and $\mathbf{r}$ a cover. We find the smallest solution for this problem using an Integer Program solver, namely GLPK [Prond]. Note that by adding more constraints, i.e., more rows in $\mathbf{M}$, the probability that the solver finds the correct solution increases.

The above method cannot be used if the errors in the recovered samples y exceed ± 2. Such errors could break the Integer program due to conflicting constraints. However, it is possible to deal with ± 1 errors, as the difference between $|z_i - y_i|$ and η_1 needs to be at least 2. Samples with an error of ± 1 can be detected and discarded, simply due to knowledge of the correct parity. Note that in the attacks of the previous Chapter (Section 8.5), a perfect side-channel adversary targeting the CDT sampling algorithm only makes errors of ± 1. Hence, this method can be used for this scenario.

Statistical Approach. A second approach that can recover the position of twos in secret key $\mathbf{s}_1$ is based on a statistical approach rather than integer programming. Thus, in some cases it withstands errors more easily. The details for this part are outside the scope of this thesis, but can be found in the published version of this chapter [PGY17].

9.3.6 – Step 4: Recovering $\mathbf{s}_1$ with the Public Key. After the above 3 steps we have recovered $|\mathbf{s}|$. In the fourth and final step, we recover the signs of all its nonzero coefficients and thereby the full signing key $\mathbf{s}$.

We do so by combining all knowledge on $|\mathbf{s}| = |\mathbf{s}_1|$ with the public key. Key generation (Algorithm 8.1) computes a public key $\mathbf{A} = \{2\mathbf{a}_q, q-2\}$, with $\mathbf{a}_q = \mathbf{s}_2/\mathbf{s}_1 = (2\mathbf{g}+1)/\mathbf{f}$ in the ring $\mathcal{R}_q$. In case of the BLISS-I and BLISS-II parameter sets (Table 8.1), both $\mathbf{f}, \mathbf{g}$ have $\lceil \delta_1 n \rceil = 154$ entries in $\{\pm 1\}$, while all other elements are zero. Thus, both these vectors are *small*.

When writing $\mathbf{s}_1 \cdot \mathbf{a}_q = \mathbf{s}_2$, it is easy to see that $\mathbf{s}_2 = 2\mathbf{g}+1$ is a short vector in the q-ary lattice generated by $\mathbf{a}_q$ (or more correctly, the rows of $\mathbf{A}_q$). Obviously, the parameters of BLISS were chosen in a way such that a straight-forward lattice-basis reduction approach is not feasible. However, knowledge of $|\mathbf{s}|$ allows a reduction of the problem size and thus the ability to recover the key.

With matrix-vector notation, i.e., $\mathbf{s}_1 \mathbf{A}_q = \mathbf{s}_2$, it becomes evident that all rows of $\mathbf{A}_q$ at indices where the coefficients of $|\mathbf{s}|$ (and thus $\mathbf{s}_1$) are zero can be simply ignored. Thus, we discard these rows and generate a matrix $\mathbf{A}_q^\star$ with size $(\lceil \delta_1 n \rceil \times n)$, i.e., (154×512) for parameter sets BLISS-I and BLISS-II). Hence, the rank of the lattice, i.e., the number of basis vectors, is decreased.

We further transform the key-recovery problem as follows. First, we do not search for $\mathbf{s}_2$ directly, but instead search for the even shorter $\mathbf{g}$ used in the key-generation process. We have that $\mathbf{f} \cdot \mathbf{a}_q = 2\mathbf{g} + 1$, thus $\mathbf{f} \cdot \mathbf{a}_q \cdot 2^{-1} = \mathbf{g} + 2^{-1}$ and we simply multiply all elements of $\mathbf{A}_q^\star$ with $2^{-1} \mod q$. We discard the computation of the first coefficient, which contains the added $2^{-1} \mod q$, and thus reduce the dimension of the lattice to $n-1$.

And second, we reduce the lattice dimension further to some d with $\delta_1 n < d < n-1$ by discarding the upper $n-1-d$ coefficients. Hence, we do not search for the full $\mathbf{g}$ but for the d-dimensional sub-vector $\mathbf{g}^\star$. If, on the one hand, this dimension d is too low, then $\mathbf{g}^\star$ is not the shortest vector in the q-ary lattice spanned by the now $(\lceil \delta_1 n \rceil \times n)$ matrix $\mathbf{A}_q^\star$. If, on the other hand, d is chosen too large, then a lattice-reduction algorithm might not be able to find the short $\mathbf{g}^\star$. For our experiments with parameter sets BLISS-I and BLISS-II, we set $d = 250$.

Finally, we feed the basis of the q-ary lattice generated by the columns of $\mathbf{A}_q^\star$ into a basis-reduction, i.e, the BKZ algorithm. If we picked d correct, the returned shortest-vector is the sought-after $\mathbf{g}^\star$. We then simply solve $\mathbf{f}^\star \mathbf{A}_q^\star = \mathbf{g}^\star$ for $\mathbf{f}^\star \in \mathbb{Z}^{\lceil \delta_1 n \rceil}$. This $\mathbf{f}^\star$ will only consist of elements in ± 1, which are the signs of the nonzero coefficients of the full $\mathbf{f}$. By simply putting the elements of $\mathbf{f}^\star$ into the nonzero coefficients of $\mathbf{s}_1'$, we can fully recover the first part of the signing key $\mathbf{f} = \mathbf{s}_1$. Finally, the second part of the key is $\mathbf{s}_2 = \mathbf{a}_q \cdot \mathbf{s}_1$. Thus, the full signing key is now recovered.

9.4 — Evaluation of key recovery

In this section, we give an evaluation of our new key-recovery technique. That is, we apply our algorithm to attacks presented in earlier work on original BLISS and compare its performance. Recall, however, that all previous work was unable to perform key-recovery for BLISS-B.

In order to allow a fair comparison, we reuse the modeled and idealized adversaries of earlier work. Concretely, we look at the idealized cache-adversary targeting the CDT sampling algorithm of [GHLY16] and the modeled adversaries for the attack on shuffling by [Pes16]. Thus, for the evaluation our Step 1 is identical to theirs.

We analyze the performance of the following steps in our key recovery. We analyze the key recovery mod 2, i.e., the LPN solving approach (Step 2). Then, we evaluate the success rate of both two-recovery approaches (Step 3). And finally, we state figures for the full-key recovery using a lattice reduction (Step 4).

9.4.1 – Step 2: Key-Recovery mod 2. For evaluation of the second attack step, i.e., mod-2 key recovery, we only consider the BLISS-I parameter set.

Our used LPN approach utilizes differing error probabilities of samples. Its first step is to filter samples, i.e., keep only those with lowest error probability. Evidently, this means that the success probability increases with the number of gathered LPN samples. Thus, we tested the performance for a broad set of observed signatures. For each test, we ran decoding on all 16 hyperthreads of a Xeon E5-2630v3 CPU running at 2.4GHz. If this does not find a solution after at most 10 minutes, then we abort and mark the experiment as failed.

Cache attack on CDT sampling. Figure 9.1 shows the results of the idealized cache-attack on the CDT sampler (as in Section 8.4). We do not perceive any significant differences between the original BLISS and BLISS-B here, so we perform experiments for both versions and give the average. We reach a success rate of about 0.9 when using 325 signatures. This is roughly 28 % less than the 450 signatures required in previous work. These savings can be explained as follows. We can now use all recovered samples, and not only those where $z = y$. However, this is somewhat offset by the fact that our LPN-based approach is not as error-tolerant as their lattice-based method which is not applicable in our setting.

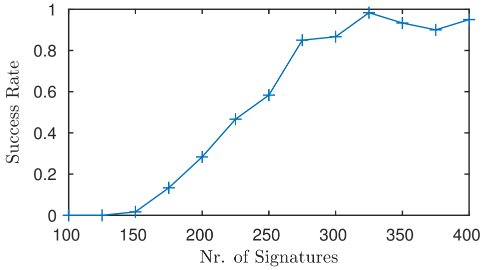
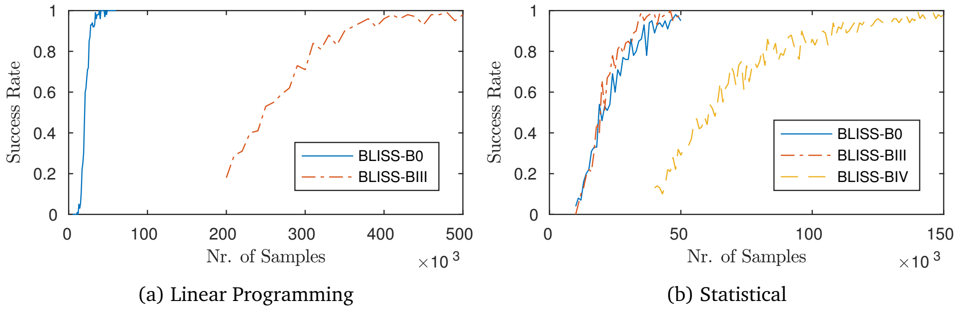


Figure 9.1: Success rate of LPN decoding for an idealized attack on CDT sampling

9.4.2 – Step 3: Recovery of Twos. For evaluation of the third attack step, we analyze the success rate of both twos-recovery procedures (subsection 9.3.5) with the idealized CDT adversary. We consider all parameter sets with $\delta_2 > 0$, i.e., BLISS-0, BLISS-III, and BLISS-IV.

We show the success rate as a function of the number of recovered samples in Figure 9.2. Please note that this is not equal to the number of required signatures (see Section 8.4). As seen in part a, the linear-programming approach requires 30 000 samples for BLISS-0 and 400 000 samples for BLISS-III, respectively. Here we do not evaluate the performance with BLISS-IV due to even higher requirements on the number of samples. The second approach, which is based on statistical methods, requires more samples for BLISS-0 (45 000) but performs better for BLISS-III (35 000) and BLISS-IV (130 000).



(a) Linear Programming

(b) Statistical

Figure 9.2: Success rate for Twos recovery

9.4.3 – Step 4: Key-Recovery using Lattice Reduction. In the last step, i.e., recovery of the full signing key $\mathbf{s}$ from $|\mathbf{s}|$ (subsection 9.3.6), we use the BKZ lattice-reduction algorithm. Concretely, we use the implementation provided by Shoup's Number Theory Library NTL [Sho15]. We set the BKZ block size to 25 and abort the reduction algorithm as soon as a fitting, i.e., short enough, candidate for the d-dimensional vector $\mathbf{g}^\star$ is found. Such a candidate vector must have a Hamming weight of at most $\lceil \delta_1 n \rceil$ and must consist solely of elements in $\{\pm 1\}$.

We evaluate the correctness and performance of this method by running over 250 key-recovery experiments for both the original BLISS-I and BLISS-BI. In each experiment, we generate a new key, perform a key recovery mod 2 (assuming a perfect and errorless side-channel), and finally perform a lattice reduction. All our experiments are successful, hence we can assume that once $\mathbf{s}^* = \mathbf{s}_1 \bmod 2$ is known, the full signing key can always be recovered. The average runtime of lattice reduction (with early abort) was roughly 4-5 minutes on an Intel Xeon E5-2660 v3 running at 2.6GHz.

Other parameter sets. For parameter sets BLISS-0, BLISS-III, and BLISS-IV, we were not able to perform full key-recovery using the above method. In case of BLISS-I and BLISS-II, the Hamming weight of $\mathbf{s}_1$ and hence the rank of the reduced q-ary lattice is $\delta_1 n = 154$. For BLISS-III and BLISS-IV, this quantity increases to 232 and 262, respectively. Due to the resulting increased rank of the lattice, we were not able to recover the key using BKZ.

9.5 — Attacking strongSwans BLISS-B

In this section, we perform a cache attack on the BLISS-B implementation of the strongSwan IPsec-based VPN suite [str15]. Concretely, we use the parameter set BLISS-I. We describe the setup and the execution of the cache attack in Section 9.5.1. Our adversary is not synchronized with the victim, thus we perform synchronization based on the signature output (Section 9.5.2). This corresponds to the first step of our key-recovery method. Afterwards one can apply the key-recovery attacks described in the previous section.

9.5.1 – Asynchronous Cache Attack. To gather samples with a side-channel attack, we use the `FR-trace` tool of the Mastik toolkit version 0.02 [Yar16]. `FR-trace` is a command line utility that allows mounting the FLUSH+RELOAD attack with amplification. We set `FR-trace` to perform the FLUSH+RELOAD attack every 30000 cycles. We describe the locations we monitor below. We set an amplification attack against the function `pos_binary`, which is used as part of Line 1 of the Bernoulli sampler (Algorithm 8.6). This slows the average running time of the function from 500 to 233000 cycles, creating a temporal separation between calls to the Bernoulli sampler. However, this slowdown is not uniform and 26% of the calls take less than 30000 cycles, i.e. below the temporal resolution of our attack.

strongSwan's implementation of BLISS uses the Bernoulli-sampling approach as described in Section 8.2. Thus, we reuse the exploit described in the previous Chapter (Section 8.4) and detect if the input to Algorithm 8.7 was 0. Our cache adversary is asynchronous. Thus, to detect the zero input we have to keep track of several events. First, we detect calls to the Gaussian sampler (Algorithm 8.6). Second, strongSwan interleaves the sampling of the two noise vectors $\mathbf{y}_1$ and $\mathbf{y}_2$, i.e., it calls the sampler twice in each of the 512 iterations of a loop. As we only target the generation of $\mathbf{y}_1$, we detect the end of each iteration and only use the first call to the Gaussian sampler in each iteration. Third, we track the entry to Algorithm 8.7 and only use the last entry per sampled value. Other calls to this function correspond to rejections and thus cannot be used. Finally, if we detect that Line 4 of Algorithm 8.7 was not executed, we know that $x = 0$. In this case, the sampled value y is a multiple of $K = 254$.

For BLISS-I, the above events, which we will dub *zero events* from now on, happen on average twice per signature. In order to minimize the error rate, we apply aggressive filtering. Also, we found that possibly due to prefetching, access to Line 4 of Algorithm 8.7 is often detected although $x = 0$. As a result, we detect zero events on average 0.74 times per signature. 92% of these detections were correct, the other 8% were false positives in which the access to Line 4 was missed by the cache attack.

We carry out the experiment on a server featuring an 8 core Intel Xeon E5-2618L v3 2.3GHz processor and 8GB of memory, running a CentOS 6.8 Linux, with gcc 4.4.7. We use strongSwan version 5.5.2, which is the current version at the time of writing this work. We build strongSwan from the sources with BLISS enabled and with C compile options `-g -falign-functions=64`. To validate the side-channel results against the ground-truth, we collect a trace of key operations executed as part of the signature generation. The trace only has a negligible effect on the timing behaviour of the code and is not used for key extraction.

9.5.2 – Resynchronization. Even though zero events can be detected by an adversary, due to the asynchronous nature of the attack it is not obvious which of the 512 samples corresponds to this detection. In other words, we can detect (with high probability) that there exists a sample $y \in \{0, \pm K, \pm 2K, \ldots\}$, but we do not know *which* sample.

We recover the index i of a detected zero event as follows. First, we locate the first and the last call to the Gaussian sampler in the cache trace. We then estimate the positions of the other 510 calls by placing them evenly in between. Note that Algorithm 8.6 does not run in constant time, hence this can only give a rough approximation. However, we found that run-time differences average out and that the estimated positions are relatively close to the real calls. In fact, this method gives better results than counting the calls to Algorithm 8.6 in the trace, as some calls are missed and counting errors accumulate. We also found that the error, i.e., the difference from the estimated index of an event to its real index in the signature, roughly follows a Gaussian distribution with standard deviation 3.5. We then compute the time span between the detected event and the estimated calls to the sampler, match it against the above Gaussian distribution, and then apply Bayes theorem to derive the probability that the detected call to the Gaussian sampler corresponds to each index $0 \ldots 511$ in the signature.

This alone, however, does not allow a sufficient resynchronization. We use the signature output $\mathbf{z}$ in order to further narrow down the index i. For each coefficient in $\mathbf{z}$, we compute the distance d to the closest multiple of parameter K used in Algorithm 8.6. Then we look up the prior-probability that the sample y corresponding to any signature coefficient z was a multiple of K, this is simply the probability that a coefficient of $\mathbf{s}_1 \cdot \mathbf{c}'$ is equal to d. We estimated this distribution using a histogram approach, it is shown in Figure 10.2 (for BLISS-I). As $K = 254$ and the coefficient-wise probability distribution of $\mathbf{s}_1 \cdot \mathbf{c}'$ is narrow, many elements of the unknown $\mathbf{y}$ have a zero or very small probability of being a multiple of K.

Finally, we combine the prior-probabilities derived from the signature output $\mathbf{z}$ with the matching of the trace, which we do by applying Bayes theorem once more. We then use only these zero events that can be reassigned to a single signature index with high probability, i.e., > 0.975, and where the prior-probability $\mathbb{P}(\langle \mathbf{s}_1, \mathbf{c}_i' \rangle = d)$ is also high, i.e., $d < 3$.

Roughly 1/3 of detected zero events fulfill both criteria. Out of these, 95% are correct, i.e., correspond to a real zero event and were reassigned to the correct index. Recall that our key-recovery approach only requires the value of $z_i - y_i \bmod 2$. Thus, 97.5% of all

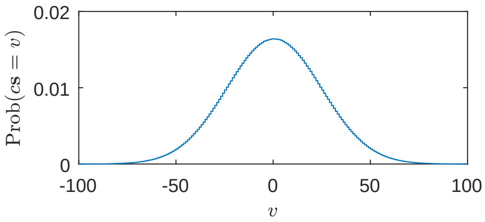


Figure 9.3: Coefficient-wise probability distribution of $\mathbf{s}_1 \cdot \mathbf{c}'$

recovered samples are correct in GF(2).

We can now apply either a brute-force search for error-less samples or we apply the LPN approach to recover the secret over GF(2). As we used the parameter set BLISS-I and thus have $\mathbf{s} \in \{0, \pm 1\}^n$, the third attack step described in Section 9.3.5 is not required. The fourth attack step, the lattice reduction, then finally returns the secret signing key. The runtime of this step was already stated in Section 9.4.3.

9.6 — A little bit more side-channel

The results in Section 9.4.3 are a bit unsatisfactory: even after full recovery of the coefficients that are $\pm 1, \pm 2$, the remaining rank of the reduced q-ary lattice is too big to solve with BKZ. This is independent of the number of recovered samples that one is able to retrieve via side-channel attacks. However, if we add a little more power to the side-channel attack, the attack can progress and reduce the rank of the lattice bit by bit.

In the implementation of BLISS in strongSwan [str15] (but also in the research-oriented implementation [DDLL13a]), the sign bit b in Line 6 of Algorithm 9.1 is not protected against side-channel attacks. A naive way of implementing Line 6 of Algorithm 9.1 is given in Algorithm 9.3. As the loop proceeds similar for every index, a cache attack should be able to catch the triggered events, i.e. the branch depending on bit b. As this is at a much later stage than the discrete Gaussian sampling, both attacks could be performed one after another. Although this might be considered a naive implementation, the original goal of adding this bit b to BLISS is for increased performance, not for increased protection against (side-channel) attacks. Ironically, when only this additional sign bit is retrieved via side-channel information, the sign bits of the secret key can be recovered bit by bit, when $|\mathbf{s}|$ from the previous chapter is known. This means the attack can progress.

From the previous two sections, we get all non-zero coefficients of $|\mathbf{s}|$, which is already quite a lot of information. However, for the last step of the key-recovery, it might be necessary to reduce the size of the unknown coefficients even more. Especially in the case of parameter sets I, III and IV, which we were unable to break due to the large lattice dimensions that is left after the side-channel attack. Let $|\mathbf{s}| \in \{0, 1, 2\}^n$ be the secret vector with unknown signs. Note that in the greedy way of computing $\mathbf{v} = \mathrm{GreedySC}(\mathbf{S}, \mathbf{c})$ (Algorithm 9.2), the first coefficient of $\mathbf{c}'$ will be -1: due to the $\mathbf{v} = \mathbf{0}$ in the first iteration. We can use this to our advantage. Let $|s|_{c_1}$ be the coefficient of $|\mathbf{s}|$ such that coefficient c_1 of $\mathbf{c}$ is the first non-zero element. If $|s|_{c_1} \neq 0$, it is possible that in some cases, there is only one possibility for the sign of $|s|_{c_1}$. For example, imagine that $|s|_{c_1} \neq 0$ and consider

Algorithm 9.3 Vector addition with random bit

```
Input: Vectors $\mathbf{u}, \mathbf{v} \in \mathbb{Z}^n$ and bit $b \in \{0, 1\}$
Output: Vector $\mathbf{z} = \mathbf{u} + (-1)^b \mathbf{v}$
initialize $\mathbf{z} := \mathbf{0}$
for $i \in [0, n)$:
    if $b = 0$:
        $\mathbf{z}_i = \mathbf{u}_i + \mathbf{v}_i$
    else if $b = 1$:
        $\mathbf{z}_i = \mathbf{u}_i - \mathbf{v}_i$
return $\mathbf{z}$
```

the extreme case that

$$z_{ji} - y_{ji} = |\langle \mathbf{s}, \mathbf{c}'_{ji} \rangle| = \langle \mathbf{c}_{ji}, |\mathbf{s}| \rangle = -1|s|_{c_1}$$

i.e., the only coefficient that is non-zero in both $\mathbf{s}$ and $\mathbf{c}$ is c_1. Thus this automatically means that

$$-1 \cdot sign(|s|_{c_1}) = sign(\langle \mathbf{s}, \mathbf{c}'_{ji} \rangle) = sign(z_{ji} - y_{ji})$$

Since we now have all the unknown signs, we can determine $\mathsf{sign}(|s|_{c_1})$. Although the probability for this extreme case is quite small, there are other, less extreme cases. If the retrieval of bit b is done from the start (together with the attack on the discrete Gaussian sampler), the signatures retrieved for steps 1-3 could be reused. We did not run experiments for this last step.

9.7 — Countermeasures

The discussion in Section 8.7 also applies to this chapter: the attacks are mitigated when the whole implementation is written in constant-time. More specifically for this chapter and for strongSwans implementation: it is crucial that Algorithm 8.7 is implemented in constant-time and without secret-dependent branching, i.e. the handling of rejections and table look-ups should not depend on the input. As shown in Algorithm 9.4, this can be done by performing all ℓ steps in the loop and always sample an A_i. The return value ν is then updated according to the values of A_i and x_i in constant time. We use C-style bitwise-logic operands to describe this update.

Algorithm 9.4 Sampling a bit from $\mathcal{B}(\exp(-x/(2\sigma^2)))$ for $x \in [0, 2^\ell)$, constant-time version

```
Input: $x \in [0, 2^\ell)$ an integer in binary form $x = x_{\ell-1} \ldots x_0$. Precomputed table $E$ with
    $E[i] = \exp(-2^i/(2\sigma^2))$ for $0 \leqslant i < \ell$
Output: A bit $b$ from $\mathcal{B}(\exp(-x/(2\sigma^2)))$
$\nu = 1$
for $i = \ell - 1$ downto 0 do
    sample $A_i$ from $\mathcal{B}(E[i])$
    $\nu = \nu \,\&\, (A_i \mid {\sim}x_i)$
return $\nu$
```

CHAPTER 10

Learning with differential faults

10.1 — Overview

Context. We have seen in the previous two chapters that securely sampling from a discrete Gaussian distribution is hard to achieve in practice. The attacks, especially the practical asynchronous attacks of Chapter 9, show the importance of protection against side-channel attacks. Secure implementation is therefore another aspect of the NIST Post-Quantum Cryptography standardization process [NIS]. The proposals should be easy to implement both correctly (ideally also model-mismatch resistant) and securely. Proposals that offer such characteristics on a wide variety of platforms, including PCs as well as constrained devices like smart cards, are more desirable. Naturally this requires analysis of many implementation attacks, not only the passive side-channel attacks (i.e. cache-attacks from the previous two chapters), but also active fault attacks ([BCN+06]). The latter are a well-known threat to embedded devices. Rowhammer.js, a remote software-only fault attack [KDK+14, GMM16], demonstrated that also high-performance PCs are vulnerable. As implementations are evaluated in terms of both security and performance, they should be made resistant to such attacks ideally without too much costs.

In this regard, an interesting property of many lattice-based signature schemes (including BLISS and BLISS-B) is that they make use of the Fiat-Shamir transform [FS86]. Concretely, two NIST submissions, qTESLA [BAA+17] and Dilithium [LDK+17], use a variant of the transform called Fiat-Shamir with Aborts [Lyu09]. However, signature schemes built using the Fiat-Shamir transform, such as ECDSA, have a well-known caveat: signing requires a nonce and a nonce reuse for different messages leads to trivial key recovery. This requirement was sometimes violated in the past, as, e.g., shown by the infamous attack on the PlayStation3 console [BMS10]. In order to sidestep this problem, the signature scheme can be made entirely deterministic. That is, the nonce is derived by hashing the message and a special part of the key, which leads to each input having a unique signature. Both Dilithium and qTESLA[1] use this approach and thus follow in the footsteps of proposals such as EdDSA [BDL+11] and deterministic ECDSA [Por13].

This solution, however, creates problems when it comes to fault attacks. An attacker can let a victim sign the same message twice, but introduce a computational fault in one of the signature computations. This results in different signatures using the same nonce and

[1] Following the initial publication of this work, a very recent update of the qTESLA specification added a mandatory countermeasure which makes the algorithm non-deterministic and prohibits our attacks. We refer to the originally submitted version of qTESLA for the remainder of the chapter, and discuss the countermeasure and update in Section 10.6

thus in a key recovery. In fact, recent work [BP16, ABF+18, PSS+18, SB18] explored the vulnerability of elliptic-curve signatures against such differential fault attacks, including Rowhammer-based ones [PSS+18].

The vulnerability of lattice-based deterministic signatures, however, is less clear. The possibility of such differential attacks was already hinted at [LDK+17, BAA+17], yet many questions remain open. Concretely, the abortion technique introduced by Lyubashevsky [Lyu09] and used by both qTESLA and Dilithium may hamper the attack. Furthermore, the different algebraic structure might open up new attack venues. Understanding the possibilities of such fault attacks is relevant in the standardization process and possible deployment of these schemes. In this chapter, we investigate the applicability of differential fault attacks on deterministic lattice-based signature schemes

Summary. We show the applicability of differential fault attacks on deterministic lattice-based signature schemes. We focus on Dilithium, but all our attacks apply to qTESLA as well. We explore how and where these schemes are vulnerable to single random faults and show how fault-induced nonce reuse allows extracting the secret key. Furthermore, we show attacks that can easily create and then efficiently exploit a *partial* nonce-reuse. This scenario yields valid signatures and thus allows to bypass some generic countermeasures.

In Dilithium and qTESLA, a unique signature vector $\mathbf{z} = \mathbf{y} + c\mathbf{s}$ is constructed out of a challenge c, a secret element $\mathbf{s}$, and a deterministically computed nonce $\mathbf{y}$. The attack is focused on faulting the computation of challenge c, leaving the nonce $\mathbf{y}$ untouched and thus creating a nonce reuse scenario. By carefully examining two signatures of the same message yet with a (due to a fault) different challenge c, $\mathbf{s}$ can be extracted using linear algebra. We identify multiple operations inside the Dilithium signing algorithm that are vulnerable, i.e., where a random fault can lead to nonce reuse. We say "can", as the use of the Fiat-Shamir with Aborts framework leads to not all faults being exploitable. We determine the success probabilities for all fault scenarios, they range from 14 % to 91 %. In addition to these scenarios, we also explore fault-induced *partial* nonce reuse. There, the fault attack is specifically focused on the computation of nonce $\mathbf{y}$, but in such a way that *only a portion* of the computation is different. We exploit this by transforming key recovery into a unique shortest-vector problem, and show how to solve it using the BKZ lattice-reduction algorithm. While previous work already exploited such partial reuse scenarios for ECC [ABF+18], our attacks are much less restrictive regarding injected faults.

Successful extraction of $\mathbf{s}$ alone, however, does not directly allow to run the signing algorithm. This is due to Dilithium's public-key compression, which causes that some additional elements of the secret key cannot be computed from just $\mathbf{s}$. Thus, we show a tweaked signature algorithm that can still produce valid signatures on any new message despite lacking some parts of the key.

We verified the vulnerabilities by performing clock glitching on an ARM Cortex-M4 microcontroller. In particular, we induced random faults during polynomial multiplication and in the SHAKE extendable output function. We show that an attacker with detailed knowledge of the executed code can easily inject faults at correct locations despite some non-constant time behavior. An unprofiled attacker who injects a fault anywhere during the signing process still has a high chance of succeeding. Up to 65.2 % of the execution time of Dilithium is vulnerable to our attacks.

We finally give a discussion on generic countermeasures against the attacks and reason about their applicability and implementation costs. We conclude that probably the simplest yet most effective countermeasure is a rerandomization of deterministic sampling, which, however, is not covered by the security proof of Dilithium.

Organization. In Section 10.2, we give the necessary background to understand the remainder of the chapter. In Section 10.3, we explore the possibilities of differential fault attacks on Dilithium. In Section 10.4, we show that the signature algorithm should be modified since the secret key element extracted by our attacks does not suffices yet to compute valid signatures for any message. In Section 10.5 we verify the vulnerabilities with real experiments on an ARM Cortex-M4 microcontroller. In Section 10.6 we end the chapter with a discussion on countermeasures. In Section 10.7, we show how to apply the results to qTESLA.

10.2 — Preliminaries

In this section, we introduce additional background on lattices and the Dilithium signature scheme. We also provide a summary of previous attacks on implementations of lattice-based cryptography.

10.2.1 – Lattice-Based Cryptography. For a general introduction on lattices, see Section 6.2. In this chapter, we furthermore extend the preliminaries on lattices a little bit. As described in Section 6.2, two hard problems underlying many lattice-based cryptography schemes are Ring-LWE/Ring-SIS, which are defined over the ring $\mathcal{R}_q$. Given a public key $(a, t) \in \mathcal{R}_q^2$, for Ring-LWE an attacker is asked to find short polynomials s_1, s_2 such that $t \equiv a \cdot s_1 + s_2 \bmod q$. With short, we mean polynomials whose coefficients are small, i.e. in absolute value less or equal to some small $\eta > 0$. Module-LWE/Module-SIS are generalizations of Ring-LWE/Ring-SIS, respectively. There the problems are defined over $\mathcal{R}_q^{k \times \ell}$ for some positive integers $k, \ell > 1$: given a matrix $\mathbf{A} \in \mathcal{R}_q^{k \times \ell}$ and a vector $\mathbf{t} \in \mathcal{R}_q^k$, find two short elements $\mathbf{s}_1 \in \mathcal{R}_q^\ell, \mathbf{s}_2 \in \mathcal{R}_q^k$ such that $\mathbf{t} \equiv \mathbf{A} \cdot \mathbf{s}_1 + \mathbf{s}_2 \bmod q$. For the attacks described in this chapter, this means that an attacker needs to find multiple secret key elements.

Additional Notation. In this chapter we need to introduce some more notation to improve readability. As we have both vectors and vectors of vectors, we define for each polynomial $f \in \mathcal{R}_q$, the corresponding vector of coefficients in $\mathbb{Z}_q$ as $\underline{f} = (f_0, f_1, \ldots, f_{n-1})$. With $:=$ we denote deterministic assignments, with $\leftarrow$ we refer to uniform probabilistic sampling from some set. We define the ℓ_2 and ℓ_∞ norm for $w \in \mathcal{R}_q$ by $\|w\|_2 = \sqrt{\sum_{i=0}^{n-1} w_i^2}$ and $\|w\|_\infty = \max\{|w_0|, |w_1|, \ldots, |w_{n-1}|\}$, where all w_i are represented by an element in the interval $[-\frac{q-1}{2}, \frac{q-1}{2}]$. This definition can be naturally expanded to vectors of polynomials. The set S_η denotes the subset of $\mathcal{R}_q$ that includes all elements w that satisfy $\|w\|_\infty \leqslant \eta$, i.e. the short polynomials described in the previous paragraph.

10.2.2 – Deterministic Lattice Signatures. We now describe the two deterministic lattice-based signature schemes Dilithium [LDK+17] and qTESLA [BAA+17], both of which were submitted to the NIST call. For design rationale, associated security proofs, and more details (e.g., on various subroutines) we refer to the respective submission documents.

Algorithm 10.1 Dilithium Key Generation

```
Output: Keypair $(pk, sk)$
$\rho \leftarrow \{0,1\}^{256}, \mathsf{K} \leftarrow \{0,1\}^{256}$
$(\mathbf{s}_1, \mathbf{s}_2) \leftarrow S_\eta^l \times S_\eta^k$
$\mathbf{A} \in \mathcal{R}_q^{k \times \ell} := \mathsf{ExpandA}(\rho)$
$\mathbf{t} := \mathbf{A}\mathbf{s}_1 + \mathbf{s}_2$
$(\mathbf{t}_1, \mathbf{t}_0) := \mathsf{Power2Round}_d(\mathbf{t})$
$tr \in \{0,1\}^{384} := \mathsf{CRH}(\rho \| \mathbf{t}_1)$
return $(pk = (\rho, \mathbf{t}_1), sk = (\rho, \mathsf{K}, tr, \mathbf{s}_1, \mathbf{s}_2, \mathbf{t}_0))$
```

Dilithium. In this chapter we focus mainly on Dilithium, which is why we give a more in-depth description of this scheme. Dilithium is based on the Module-LWE/SIS assumption. It operates over the fixed base ring $\mathcal{R}_q = \mathbb{Z}_q[x]/(x^{256} + 1)$, $q = 8380417$ and allows for flexibility by allowing different module parameters (k, ℓ). This means that code used for arithmetic in $\mathcal{R}_q$ can be reused for any module $\mathcal{R}_q^{k \times \ell}$, which makes an adaptation to other security levels easier.

Key generation is given in Algorithm 10.1. First, two random seeds ρ, K, and two key elements $\mathbf{s}_1$, $\mathbf{s}_2$ are sampled. The function $\mathsf{ExpandA}$ deterministically expands the seed ρ into a matrix $\mathbf{A} \in \mathcal{R}_q^{k \times \ell}$ using the extendable-output function (XOF) $\mathsf{SHAKE128}$. This is done to minimize public and private key sizes as only ρ needs to be stored instead of the full $\mathbf{A}$. The public key $\mathbf{t} = \mathbf{A}\mathbf{s}_1 + \mathbf{s}_2$ is compressed by feeding it into the $\mathsf{Power2Round}_q$ function, which computes a pair $(\mathbf{t}_1, \mathbf{t}_0)$ such that $\mathbf{t} = \mathbf{t}_1 \cdot 2^d + \mathbf{t}_0$. Only the upper part $\mathbf{t}_1$ is published. The lower bits $\mathbf{t}_0$ and a hash of the public key $tr = \mathsf{CRH}(\rho \| \mathbf{t}_1)$ are included in the private key sk. CRH is shorthand for Collision Resistant Hash, Dilithium uses $\mathsf{SHAKE256}$ with an output length of 384 bits.

Dilithium is based on the Fiat-Shamir with Aborts Framework [Lyu09]. Simply speaking, in this framework a signature σ is rejected and signing restarted to make σ follow some fixed distribution. This rejection sampling statistically hides any secret information in the signature and thus provides the zero-knowledge property. The structure of rejection sampling can be easily seen in Algorithm 10.2, which shows a slightly simplified[2] version of the Dilithium signature algorithm. The comments in Algorithm 10.2 refer to our attack scenarios and can be ignored for now.

Signature generation starts off by recomputing $\mathbf{A}$ and hashing the message M together with the hashed public key tr. The abort loop starts off by using the function $\mathsf{DeterministicSample}$ to generate the noise $\mathbf{y} \in S_{\gamma_1 - 1}^{\ell}$. The product $\mathbf{w} = \mathbf{A}\mathbf{y}$ is compressed to $\mathbf{w}_1$ using $\mathsf{HighBits}$. The hint $\mathbf{h}$ later allows the verifier to recompute this $\mathbf{w}_1$. The hash function H instantiates the random oracle needed in the proof. It returns a sparse ternary polynomial $c \in B_{60}$, i.e., a polynomial with Hamming weight 60 and all non-zero coefficients in ± 1. The function $\mathsf{Decompose}$ returns both $\mathsf{HighBits}$ and $\mathsf{LowBits}$ of its input. Finally, several checks are performed that determine if the current signature is accepted or rejected. In case of a rejection, a counter value κ is updated and the signature generation is repeated with this new value.

Note that all operations in Algorithm 10.2 are completely deterministic and thus gen-

[2] Some additional checks and constant subroutine arguments are omitted.

Algorithm 10.2 Dilithium Sign (simplified[2])

```
Input: Message $M$, private key $sk = (\rho, K, tr, \mathbf{s}_1, \mathbf{s}_2, \mathbf{t}_0)$
Output: Signature $\sigma = (\mathbf{z}, \mathbf{h}, c)$
$\mathbf{A} \in \mathcal{R}_q^{k \times \ell} := \mathsf{ExpandA}(\rho)$                          ▷ $fA_\rho$, $fA_E$
$\mu \in \{0,1\}^{384} := \mathsf{CRH}(tr \| M)$
$\kappa := 0, (\mathbf{z}, \mathbf{h}) := \perp$
while $(\mathbf{z}, \mathbf{h}) = \perp$ do
    $\mathbf{y} \in S_{\gamma_1 - 1}^{l} := \mathsf{DeterministicSample}(K \| \mu \| \kappa)$    ▷ $fY$
    $\mathbf{w} := \mathbf{Ay}$                                                   ▷ $fW$
    $\mathbf{w}_1 := \mathsf{HighBits}(\mathbf{w})$
    $c \in B_{60} := \mathsf{H}(\mu \| \mathbf{w}_1)$                                ▷ $fH$
    $\mathbf{z} := \mathbf{y} + c\mathbf{s}_1$
    $\mathbf{h} := \mathsf{MakeHint}(-c\mathbf{t}_0, \mathbf{w} - c\mathbf{s}_2 + c\mathbf{t}_0)$
    $(\mathbf{r}_1, \mathbf{r}_0) := \mathsf{Decompose}(\mathbf{w} - c\mathbf{s}_2)$
    if $\|\mathbf{z}\|_\infty \geqslant \gamma_1 - \beta$ or $\|\mathbf{r}_0\|_\infty \geqslant \gamma_2 - \beta$ or $\mathbf{r}_1 \neq \mathbf{w}_1$ then $(\mathbf{z}, \mathbf{h}) := \perp$
    $\kappa := \kappa + 1$
return $\sigma = (\mathbf{z}, \mathbf{h}, c)$
```

Algorithm 10.3 Dilithium Verify (simplified[2])

```
Input: Public key $pk = (\rho, \mathbf{t}_1)$, message $M$, signature $\sigma = (\mathbf{z}, \mathbf{h}, c)$
$\mathbf{A} \in R_q^{k \times \ell} := \mathsf{ExpandA}(\rho)$
$\mu \in \{0,1\}^{384} := \mathsf{CRH}(\mathsf{CRH}(\rho \| \mathbf{t}_1) \| M)$
$\mathbf{w}_1 := \mathsf{UseHint}(\mathbf{h}, \mathbf{Az} - c\mathbf{t}_1)$
accept iff $c = \mathsf{H}(\mu \| \mathbf{w}_1)$
```

erate a unique signature for message M[3]. This property is also used in the proof of Dilithium in the Quantum Random Oracle Model (QROM) [KLS18]. The proof does allow a non-deterministic version, albeit at the cost of tightness and a loss in security proportional to the number of distinct signatures an adversary can observe per message.

For completeness, we also provide a simplified version of the verification procedure (Algorithm 10.3). Throughout this chapter we use the recommended Dilithium parameter set III shown in Table 10.1. The designers claim 128 bits of security against a Quantum adversary. Other parameter sets mainly differ in the used (k, ℓ), so our later attacks are possible for all proposed sets.

qTESLA. Structurally, the signature scheme qTESLA [BAA+17] is very similar to that of Dilithium. It also uses a variant of the Fiat-Shamir with Aborts framework and is deterministic. Unlike Dilithium, its proof in the QROM model [ABB+17] allows for a non-deterministic version as well (without losing tightness). The main difference however is that qTESLA is based on the Ring-LWE/SIS assumptions instead of the module counterparts. Thus, it operates on $\mathcal{R}_q = \mathbb{Z}_q[x]/(x^n + 1)$ (so $k = \ell = 1$) with $n \geqslant 1024$. We will later demonstrate our attacks on the example of Dilithium. Still, with some mi-

[3] A previous Dilithium description [DLL+17] is probabilistic, but did not include a proof in the QROM.

Table 10.1: Dilithium parameter sets

	I	II	III	IV
	weak	medium	recommended	high
n	256	256	256	256
q	8380417	8380417	8380417	8380417
d	14	14	14	14
weight(c)	60	60	60	60
γ_1	523776	523776	523776	523776
γ_2	261888	261888	261888	261888
(k, ℓ)	(3, 2)	(4, 3)	(5, 4)	(6, 5)
η	7	6	5	3
β	375	325	275	175
ω	64	80	96	120

nor modifications they all also apply to qTESLA. We defer the description of qTESLA to Section 10.7, where we also highlight the similarities to Dilithium.

10.2.3 – The SHAKE Extendable Output Function. Dilithium makes heavy use of the SHAKE Extendable Output Function (XOF). It is also an important target of our fault attack, which is why we now very briefly describe it. SHAKE uses the sponge construction [BDPV07], which we already handled in quite some detail in Chapter 2. For the sake of completeness, we restate the important parts for this chapter. The sponge construction has two internal parameters r and c called the rate and the capacity, where the capacity is chosen such that the sponge construction meets a desired level of security. We call the internal state of the sponge x, consisting of $r + c$ bits, with all bits initialized to zero. The sponge starts with the *absorb* phase. Any input to the sponge function is first padded, using some injective padding function, resulting in $k \geqslant 1$ input blocks $m_1 \| m_2 \| \ldots \| m_k$ of length r bits. These message blocks are then XORed with the first r bits of the state x, interleaved with applications of a permutation $f : \{0, 1\}^{r+c} \to \{0, 1\}^{r+c}$. In SHAKE, the Keccak-$f$ permutation is used. After all message blocks are processed, the *squeeze* phase starts. Depending on the desired output length, the function iteratively returns the first r bit blocks of the internal state x, interleaved with applications of the permutation f. In Figure 10.1, we show an example for three input blocks and three output blocks. Note that this construction allows for any number of input and output blocks.

In the context of fault attacks, the important thing to note is that any manipulations in f corrupt the state x and thus affect all subsequent operations. For instance, faulting the first application of f in the squeeze phase leads to a correct H_0 but faulty $H_1, H_2, \ldots$.

10.2.4 – Implementation Security of Lattice-Based Cryptography. In the previous two chapters we discussed various attacks on lattice-based schemes that require high-precision sampling from a discrete Gaussian distribution. We also discussed several countermeasures. We call these passive implementation attacks: attacks that do not interfere with the computation itself. Active implementation attacks on lattice-based cryp-

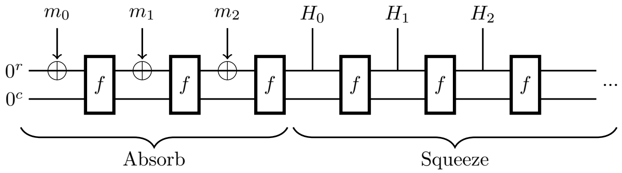


Figure 10.1: Sponge construction with three block input $m_0||m_1||m_2$ and three block output $H_0||H_1||H_2$. With $0^r, 0^c$ we denote the all-zero bit string of length r, c. The application of the padding function is not shown.

tography also received some prior attention. Two concurrent works [BBK16, EFGT16] investigated fault attacks on non-deterministic lattice-based signature schemes, such as BLISS [DDLL13b], GLP [GLP12], PASSSign [HPS+14], and ring-TESLA [ABB+16]. Espitau et al. [EFGT16] investigated loop-abort faults in the generation of noise-polynomial $y \in \mathcal{R}_q$. This means that the sampling algorithm for this polynomial is cut off after the m'th noise coefficient, i.e. $\underline{y} = (y_0, \ldots, y_{m-1}, 0, \ldots, 0)$. A key-recovery is possible if $m \ll n$. The target distribution of y is not relevant; the general attack framework applies to both BLISS (which uses the discrete Gaussian distribution) and the other previously mentioned schemes (which all use the uniform distribution).

Like Dilithium and qTESLA, the above mentioned lattice-based signatures compute $z = y + cs$, where c, z are part of the signature σ and $s \in \mathcal{R}_q$ is a small secret key element. We can rewrite this equation as:

$$c^{-1}z \equiv c^{-1}y + s \bmod q \tag{10.1}$$

where we assume that $c \in \mathcal{R}_q$ is invertible (which is true with very high probability). As s is a small element, the target $t = c^{-1}z$ is close to a point in the lattice generated by the vectors $\{\underline{w}_i = c^{-1}x^i \bmod q \mid i \in \{0, \ldots, m-1\}\}$ and $q\mathbb{Z}^n$, and the difference is exactly s. This means that the closest-vector problem in (10.1) can be solved by, e.g., a lattice reduction followed by application of Babai's nearest plane algorithm. As this sub-lattice is of full dimension and too hard to solve at once, one can reduce the size of the problem to solve (10.1) for a subset $I \subseteq \{0, \ldots, n-1\}$ of indices, using the projection $\psi_I : \mathbb{Z}^n \to \mathbb{Z}^I$ given by $\psi_I((u_i)_{0 \leqslant i < n}) = (u_i)_{i \in I}$. It can be shown that if the cardinality of any subset I is slightly larger than m (see the analysis in [EFGT16]), then (10.1) is solvable for subset I. By repeating this for multiple subsets, the complete secret key element s can be recovered. With knowledge of s the full secret key could be recovered using linear algebra.

10.2.5 – Differential Fault Attacks on ECC. In this chapter we concentrate on differential fault attacks, in which the difference between a faulty and a correct output is used to determine information about the secret key. Previous work [BP16, ABF+18, PSS+18, SB18] explored such attacks on two deterministic elliptic curve signature schemes: EdDSA and deterministic ECDSA. Both of these signature schemes use the Fiat-Shamir transform, thus requiring the usage of a unique nonce per message. The fault attacks mainly focus on achieving nonce reuse, as this leads to a very efficient key-recovery.

Concretely, Poddebniak et al. [PSS+18] exploit the fact that the message is hashed twice in EdDSA. By manipulating the message in between these hashing operations with

Rowhammer, one can induce a nonce reuse and perform a key recovery. Ambrose et al. [ABF+18] inspect a wider range of scenarios. They show that even random faults in certain operations can allow attacks. Additionally, they show that faults affecting the nonce itself are also usable. However, for this they require a very restrictive fault model. They need that the resulting error is limited to a few bits, as an exhaustive search is required to find the exact difference between the faulty and correct nonce.

10.3 — Differential faults on deterministic lattice signatures

In this section, we present our differential fault attacks on Dilithium. As previously mentioned, these attacks apply to qTESLA as well, as we provide the attacks for arbitrary values of ℓ and k. First, we briefly describe our fault model. Then we explain the main intuition of our attacks. We identified multiple vulnerable operations, for each of them we finally describe how faulting can lead to key recovery. We also discuss additional properties, such as ease of fault injection, for the scenarios.

Fault model. In this chapter we assume the possibility of injecting a single random fault. These can encompass instruction skips, arithmetic faults, glitches in storage, and more. The faults are not restricted to specific operations but can be applied during a large section of execution time. This model is also used for some of the previously mentioned attacks on EdDSA [ABF+18] (some scenarios require a more restrictive fault model). In contrast, previous active attacks on lattice-based signatures required more control, such as the ability to abort a loop [EFGT16].

10.3.1 – Intuition. The intuition behind our fault attacks is as follows. We let the signer sign the same message M twice. In the first invocation we do not inject any fault and receive a valid and proper signature $\sigma = (\mathbf{z}, \mathbf{h}, c)$. We inject a fault in the second run; we use $'$, e.g., $\mathbf{z}'$, to denote variables in this faulted invocation. More concretely, we inject a fault such that $\mathbf{y}'$ is undisturbed and due to the determinism equal to $\mathbf{y}$, yet $c' \neq c$ and thus $\mathbf{z}' = \mathbf{y} + c'\mathbf{s}_1$.

Thus, the fault induces a nonce-reuse scenario. When defining $\Delta\mathbf{z} = \mathbf{z} - \mathbf{z}'$ (and $\Delta c, \Delta\mathbf{y}$ analogously), we have $\Delta\mathbf{z} = \Delta\mathbf{y} + \Delta c \cdot \mathbf{s}_1 = \Delta c \cdot \mathbf{s}_1$ as $\Delta\mathbf{y} = 0$. Thus, under the requirement that Δc is invertible, which is true with very high probability, then $\mathbf{s}_1 = \Delta c^{-1} \cdot \Delta\mathbf{z}$.

The Fiat-Shamir with Abort structure, however, introduces an additional hurdle. We require that both the valid as well as the faulty signature computation terminate in the same iteration of the abortion loop. In other words, when using κ_f to denote the final value of the loop counter κ, we need that $\Delta\kappa_f = \kappa_f - \kappa_f' = 0$. Observe that in Algorithm 10.2, loop counter κ is input to DeterministicSample. Hence, to achieve $\mathbf{y} = \mathbf{y}'$ we have the requirement that $\Delta\kappa_f = 0$. Due to faulty intermediates and the influence of the rejection tests, this is obviously not guaranteed.

In the remainder of this section we discuss concrete fault scenarios. That is, we explain which operations in Algorithm 10.2 can be faulted such that key-recovery is possible. For each scenario we will give the exploitation technique as well as state its success probability, i.e., the chance that it terminates in the same loop iteration and thus $\Delta\kappa_f = 0$. This probability was estimated using at least 10 000 fault simulations per scenario. An

Table 10.2: Fault scenarios discussed in this chapter

Name	Section	Description
fH	10.3.2	Random fault in call to H
fW	10.3.3	Random fault in polynomial multiplication $\mathbf{w} := \mathbf{A}\mathbf{y}$
fA_ρ	10.3.4	Corrupt ρ during import of *sk*
fA_E	10.3.4	Random fault in expansion $\mathbf{A} := \mathsf{ExpandA}(\rho)$
fY	10.3.5	Random fault in sampling $\mathbf{y} := \mathsf{DeterministicSample}(\cdot)$

overview of the scenarios is given in Table 10.2, they are listed in order of appearance in Algorithm 10.2. The order of description will be different.

10.3.2 – Scenario: fH. Probably the most intuitive way to achieve a nonce-reuse is the fH scenario, where a random fault is injected into the computation $c \in \mathcal{B}_{60} := \mathsf{H}(\mu \| \mathbf{w}_1)$. This can be achieved by either manipulating one of the inputs $\mu, \mathbf{w}_1$ immediately before they are being used in H, or by directly injecting a fault into the hash function H itself.

We will show in Section 10.5.1 that it is a very reasonable assumption that an attacker can inject a fault in the correct iteration κ_f, i.e., the last one in the non-faulty computation. If the rejection step is then passed with the different c', secret element $\mathbf{s}_1$ can be recovered as described in Section 10.3.1.

Since c is a sparse ternary polynomial and $\mathbf{s}_1 \in S_\eta^l$ has small coefficients, their product is also small. We depict its coefficient-wise probability distribution in Figure 10.2, it can be approximated with a (discretized) Gaussian distribution having zero mean and $\sigma \approx 24.3$. As $\|c\mathbf{s}\|_2 \ll \|\mathbf{y}\|_2, \|\mathbf{w}\|_2$, the acceptance conditions for $\mathbf{z}$ and $\mathbf{r}_0$ are likely to hold for a different c as well. This results in a high success probability of over 90 %.

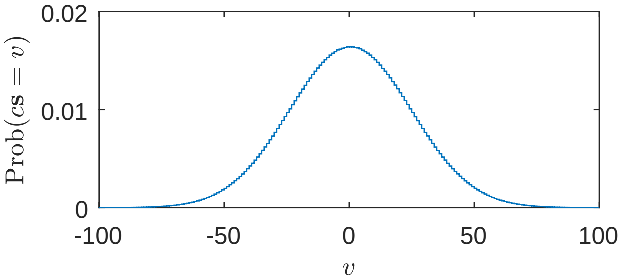


Figure 10.2: Coefficient-wise probability distribution of $c\mathbf{s}$

Determining success. There are two ways to test if $\Delta\kappa_f = 0$ and thus key recovery is successful. The first method is to simply recover $\mathbf{s}_1$ and then test if it is small, i.e., $\mathbf{s}_1 \in S_\eta^l$. If $\Delta\mathbf{y} \neq 0$ then the recovered key will be a random vector in $\mathcal{R}_q^\ell$ which will not fulfill the bound on the ℓ_∞ norm. Alternatively, one can also exploit the small norm of $c\mathbf{s}$

by computing $\|\Delta\mathbf{z}\|_2$ (or also $\|\Delta\mathbf{z}\|_\infty$) and test if it is below a certain threshold. Again, $\Delta\mathbf{y} \neq 0$ will lead to a very large value of $\|\Delta\mathbf{z}\|_2$.

Apart from $\Delta\kappa_f = 0$, we also require that Δc is invertible. This is true with very high probability. The fraction of invertible polynomials in $\mathcal{R}_q$ is $(1-1/q)^n$ [LPR13], which is about $1-2^{-15}$ for the Dilithium parameters. In the remainder of this chapter, we assume this fraction also holds for the polynomials described by Δc (i.e. the difference of two random sparse ternary polynomials $c, c' \in B_{60}$). We test for invertibility and consider the attack to have failed in the rare case that Δc is not invertible.

10.3.3 – Scenario: fW. Instead of directly faulting the hash function H, it is also possible to alter $c := \mathsf{H}(\mu\|\mathbf{w}_1)$ by manipulating the computation of its inputs $\mu, \mathbf{w}_1$. The message/public key hash μ is also used as seed for DeterministicSample, hence faults in the computation $\mu := \mathsf{CRH}(tr\|M)$ are not exploitable.

Faults in the computation of $\mathbf{w} := \mathbf{A}\mathbf{y}$ which lead to an incorrect $\mathbf{w}_1$, however, can be exploited. The required polynomial multiplications in $\mathcal{R}_q$ can be efficiently implemented using the Number Theoretic Transform (NTT). Because the runtime of multiplication is higher than that of hashing, it can be a more viable target for fault attacks. An NTT is essentially an FFT-like transform over a prime field and uses similar implementation techniques, i.e., butterfly networks. Due to these techniques, the number of coefficients in $\mathbf{w}$ affected by a single random fault can range from 1 to all $n \cdot k$.

As unaffected coefficients of $\mathbf{w}$ clearly pass rejection and a single altered one is sufficient to achieve $\Delta c \neq 0$, minimizing the number of faulty coefficients increases the success probability. Thus, unlike in our other scenarios the concrete fault position has a much stronger impact. To give a sense of possible success probabilities, we evaluated the two most extreme cases. First, we inject a fault in the forward-NTT of $\mathbf{y}$. Such a fault spreads to all $n \cdot k$ coefficients of $\mathbf{w}$ and thus leads to a low success probability (25.3 %). Second, we fault the inverse-NTT applied to $\mathbf{w}$ such that only two coefficients are affected. With a success probability of over 90 %, this sub-scenario is similar to directly faulting H. Note that while single-coefficient faults are also possible, they are slightly less likely to lead to a successful key-recovery. This is due to the chance that a faulty coefficient w' still rounds to the correct $w_1' = w_1$, which results in $\Delta c = 0$ and the fault not being exploitable.

10.3.4 – Scenarios: $\mathbf{fA}_\rho$, $\mathbf{fA}_\mathbf{E}$. Another possibility to achieve a faulty $\mathbf{w} = \mathbf{A}\mathbf{y}$ is to manipulate the expansion of seed ρ into the matrix $\mathbf{A}$. As seen in Algorithm 10.2, this is done before entering the abort loop and is thus always executed at the same time. Furthermore, ExpandA is a major contributor to overall runtime (cf. Section 10.5.2). Both these properties drastically simplify fault injection for this scenario. Also, $\mathbf{A}$ has a larger footprint (20 kB in Dilithium-III) than other variables and is potentially kept in memory for a prolonged time, i.e., by caching it one does not need to re-run ExpandA for every signing operation. These properties make $\mathbf{A}$ a particularly interesting target for memory-based faults, such as Rowhammer.

When focusing on more traditional faulting techniques, then differences in $\mathbf{A}$ can be achieved by either manipulating the seed ρ, e.g., during loading of the private key (scenario fA_ρ), or by inserting a glitch into the expansion $\mathbf{A} \in \mathcal{R}_q^{k\times\ell} := \mathsf{ExpandA}(\rho)$ (scenario fA_E). On first glance these scenarios might seem identical. There are, however, some major differences. Observe that in Algorithm 10.4, which sketches the method for expand-

Algorithm 10.4 ExpandA(ρ)

```
Input: Seed $\rho$
Output: uniform $\mathbf{A} \in \mathcal{R}_q^{k \times \ell}$
for $i := 0 \ldots k-1$ do
    for $j := 0 \ldots \ell-1$ do
        $\mathbf{A}_{i,j} :=$ SamplePoly($\rho \| i \| j$)
return A

function SamplePoly($s$)
    $t \in \{0,1\}^{5 \cdot r} :=$ SHAKE128($s$)
    $u := 0$
    while $u < n$ do
        $v :=$ next $\lceil \log_2 q \rceil$ bits of $t$
        if $v < q$ then
            $a_i := v$
            $u := u + 1$
    return a
```

ing ρ into $\mathbf{A}$, the $k \cdot \ell$ polynomials comprising $\mathbf{A}$ are generated using independent calls to SHAKE. Thus, any single fault in the SHAKE permutation leads to just one corrupted polynomial. Consequently, after the matrix-vector multiplication $\mathbf{Ay}$ we have n differing coefficients of $\mathbf{w}$. This leads to a success probability of approximately 54 %.

Directly faulting ρ, either during import or in storage, obviously results in an all different $\mathbf{A}$ and thus $\mathbf{w}$. This decreases the success probability to only 14 %. However, this type of fault has a major advantage when it comes to defeating countermeasures. It is potentially (semi-)permanent and can thus, at least under certain circumstances, not be detected by the generic double-computation countermeasure. In Section 10.6 we discuss this in more detail.

10.3.5 – Scenario: fY. So far, we have only discussed fault-induced nonce-reuse scenarios, i.e. the case where $\mathbf{y}' = \mathbf{y}$. For our final scenario, we will switch to *partial* nonce-reuse. Unlike all previous scenarios, inducing a partial reuse still leads to valid signatures and is thus not detectable with a signature verification.

We introduce some additional notation for element $\mathbf{t} \in \mathcal{R}_q^{\ell}$: we define $t_u \in \mathcal{R}_q$ to be the u'th element of $\mathbf{t}$ and $(\underline{t}_u)_v \in [-\frac{q-1}{2}, \frac{q-1}{2}]$ to be its v'th coefficient, for $0 \leqslant u < \ell$ and $0 \leqslant v < n$. We define $\underline{e}_j$ for $0 \leqslant j < n$ to be the j-th unit vector, i.e. the vector with a 1 at position j and zero otherwise.

Simple example. First, let us assume the following. We inject a fault in $\mathbf{y}' \in S_{\gamma_1 - 1}^{\ell}$ such that only a single coefficient $(\underline{y}'_u)_v \in \mathbf{y}$ (with index $u \in \{0, \ldots, \ell-1\}$ and $v \in \{0, \ldots, n-1\}$) is changed to a random value (while preserving $|(\underline{y}'_u)_v| \leqslant \gamma_1 - 1$). Still, this leads to a completely different $\mathbf{w}_1$, and therefore to a different c' and $\mathbf{z}' = \mathbf{y}' + c'\mathbf{s}_1$.

We then compute $\mathbf{s}_1 = \Delta c^{-1} \cdot \Delta \mathbf{z}$ and determine u by simply using the one index for which $s_{1,u} \notin S_\eta$. For all $i \neq u$ we have that $\Delta y_i = 0$, thus recovery of these key polynomials succeeds. If we now compute the difference Δz_u, we will notice the following: for indices $i \neq v$ we see $|\Delta z_i| \leqslant 2\delta$ for some threshold δ chosen such that

Algorithm 10.5 DeterministicSample$_{\gamma_1-1}(s)$ (simplified[4])

```
Input: Seed $s$
Output: $\mathbf{y} \in S^{\ell}_{\gamma_1-1}$
1: for $u := 0 \ldots \ell - 1$ do                          ▷ Sample $\ell$ nonce polynomials
2:     $t \in \{0,1\}^{5 \cdot r} :=$ SHAKE256$(s \| u)$
3:     $v := 0$
4:     while $v < n$ do                                      ▷ Rejection sampling
5:         $x :=$ next $2\lceil \log_2 \gamma_1 \rceil$ bits of $t$
6:         if $x \leqslant 2(\gamma_1 - 1)$ then
7:             $(\underline{y}_u)_v := q + \gamma_1 - 1 - x$
8:             $v := v + 1$
9: return $\mathbf{y}$
```

$\|c\mathbf{s}_1\|_\infty \leqslant \delta$ holds for any c and $\mathbf{s}_1$. Concretely, we can set $\delta = 60\eta$. The injected fault in $(\underline{y}'_u)_v$ can cause any difference to the value, but on average it will be large. On expectation $|(\underline{y}'_u)_v - (\underline{y}_u)_v|$ will be $\frac{2\gamma_1-1}{3}$ as both of these coefficients are random values in $[-(\gamma_1-1), (\gamma_1-1)]$ (by our assumption). Since $\|c\mathbf{s}_1\|_\infty \leqslant \delta \ll \frac{2\gamma_1-1}{3}$, we can detect index v and thus the position of the fault by using the index of maximum $|\Delta z_u|$.

We finally recover $s_{1,u}$ as follows. Simply speaking, we eliminate row v of the linear system $s_{1,u} = \Delta c^{-1} \cdot \Delta z_u$, guess the value of $(\underline{s}_{1,u})_v$ (exhaustive search), solve for the full $s_{1,u}$ and test if it is in S_η. Note that similarly we could also directly guess the value of $(\Delta \underline{y}_u)_v$ (instead of $(\underline{s}_u)_v$), albeit there the search-space is much larger. This latter scenario is the direct counterpart to the partial nonce reuse fault attack on elliptic curve signatures (as described in [ABF+18] and mentioned in Section 10.2.5): an exhaustive search is used to determine the exact error in the faulted nonce.

We will show next that for lattice-based signatures these partial-reuse attacks are way more powerful. The exhaustive search can be replaced with solving a lattice problem, which is much more efficient. The far larger number of tolerable errors allows replacing the very restrictive fault model (influencing a small number of bits of the nonce) with random faults in SHAKE.

Efficient partial nonce reuse attack. The nonce $\mathbf{y} \in S^{\ell}_{\gamma_1-1}$ is generated by function DeterministicSample, a simplified[4] version is given in Algorithm 10.5. Note that input seed s changes whenever a signature is rejected (Algorithm 10.2), and the counter u will change the individual elements of $\mathbf{y}$. The idea is that we now fault SHAKE (Line 2), but in such a way that it only changes a few coefficients of y_u for some $u \in \{0, \ldots, \ell - 1\}$. Since all coefficients of $(\underline{y}_u)$ are sampled sequentially, a fault that only affects the last few bytes of t' will only change the last few coefficients of $(\underline{y}_u)$.

As mentioned in Section 10.2.3, SHAKE operates on a state x of $r + c$ bits and consists of an absorb phase and a squeeze phase. If a fault is injected during the absorb phase, the output of SHAKE will be completely different. However, if the fault is injected near the end of the squeeze phase, only the last few applications of f will operate

[4] For example, with very small probability the $5 \cdot r$ bits are not enough to generate enough values for any y_i. In that case, another call to SHAKE and more rejection sampling is done.

on a faulty state x and thus return different outputs (cf. Section 10.2.3). In particular, as DeterministicSample requests 5 output blocks of SHAKE (Line 2), an injected fault in, e.g., the last or second-to-last application of Keccak-f during the squeeze phase will cause changes in the last few bytes of t'. Thus, only the last few coefficients of y'_u will differ i.e. $\Delta y_u = (0, \ldots, 0, \zeta_\nu, \zeta_{\nu+1}, \ldots, \zeta_{n-1})$ for some index $\nu \in \{0, \ldots, n-1\}$. Note that the index ν for which the values start to differ will vary depending on how many elements were accepted from the first few output blocks of SHAKE. We can again detect index ν similarly as mentioned previous: by taking the first index where $|\Delta z_u| \geqslant 2\delta$. However, we cannot apply the brute-force search for the corresponding elements in $(\underline{s}_{1,u})_{\nu \leqslant i < n}$ anymore: the search-space will be too large.

Instead, we will transform the search for these $n - \nu$ secret coefficients to the lattice problem as described in Section 10.2.4. Thus, when writing:

$$t = \Delta c^{-1} \Delta z_u = \Delta c^{-1} \Delta y_u + s_{1,u}$$

we have a target t and want to determine the closest point on the lattice generated by Δc^{-1}. The difference between t and its closest lattice point is exactly $s_{1,u}$. Solving this closest-vector problem is made possible by using that the first ν coefficients of Δy_u are 0. We use the lattice generated by basis vectors $\{\underline{w}_i = \Delta c^{-1} x^i \bmod q \mid i \in \{\nu, \nu+1, \ldots, n-1\}\}$ and $q\mathbb{Z}^n$. Take $I = \{m, m+1, \ldots, n-1\}$ to be the target subset of indices, where $m < \nu$ and apply ψ_I to these basis vectors to project to the target search space, where ψ_I as defined in Section 10.2.4. We then cast the problem at hand to a unique shortest-vector problem as, e.g., described by Albrecht, Fitzpatrick and Göpfert [AFG13], and then apply a lattice-reduction algorithm (like LLL or BKZ). If successful, we retrieve a small $n - m$ dimensional vector $\underline{s}^{\text{guess}}$, whose coefficients correspond to the last $n - m$ coefficients of $\underline{s}_{1,u}$. To get the full $s_{1,u}$, we replace the last $n - m$ coefficients of $\Delta \underline{z}_u$ by the coefficients of $\underline{s}^{\text{guess}}$, transform rotation-matrix $\Delta \mathbf{C}$ of Δc into $\overline{\mathbf{C}}$ by replacing the last $n - m$ columns by the identity columns $\underline{e}_m, \underline{e}_{m+1}, \ldots, \underline{e}_{n-1}$ and compute the full $\underline{s}^{\text{guess}} = \Delta \underline{z} \overline{\mathbf{C}}^{-1}$. We can verify correctness by checking that $\underline{s}^{\text{guess}} \in S_\eta$.

In our experiments, we injected a random fault in the last (denoted by 1P) or second-to-last (denoted by 2P) application of Keccak-f inside SHAKE (called in Algorithm 10.5, line 2). Since the input to SHAKE is shorter than the rate r, out of the total five applications of Keccak-f these are the fourth (2P) and fifth (1P), respectively. Faults in the 3 earlier applications of Keccak-f did not yield a solvable lattice problem. We performed 1000 experiments for both 1P and 2P and determined the average number of errors (so $n - \nu$) and the average running time for BKZ (on an Intel Xeon E5-4669 v4 @ 2.20GHz). In our experiments, we took m such that the cardinality of I is about $1.4(n - \nu)$. For the lattice reduction, we used BKZ with block-size 25 but included an early abort, i.e., we abort reduction as soon as a potential key-candidate (a vector in S_η) is found. The results are shown in Table 10.3. The success probability of the lattice reduction was 100%. Thus, if a fault is correctly injected and $\Delta \kappa_f = 0$, then the key $\mathbf{s}_1$ can always be recovered. The probability that $\Delta \kappa_f = 0$ is between 24 and 25 % (Table 10.4).

10.3.6 – Summary of scenarios. We now give a summary of the different fault scenarios. In Table 10.4 we restate the success probability of all fault scenarios. Recall that in scenario fW a large number of outcomes is possible, but we analyzed the best and worst possible outcomes. For scenarios fY, fH, and fW we assume that the fault is injected in the last iteration κ_f.

Table 10.3: Results of injecting faults in DeterministicSample

Squeeze phase iteration	Average number of errors in $\mathcal{Y}_u$	Average running time lattice reduction
1P (last)	39	2.3s
2P (second to last)	93	35.8s

Table 10.4: Fault-attack success probability in percent

fA_ρ	fA_E	fY-1P	fY-2P	fW	fH
14.3	54.4	24.8	23.9	25.4 - 90.3	91.0

fH is the most intuitive scenario and also achieves the highest success probability. However, it is also the smallest of all targets (cf. Section 10.5.2). The lowest success probability is achieved for fA_ρ, yet with the huge advantage of being potentially permanent. Faulting the expansion of $\mathbf{A}$ offers both a large and fixed-time target. Finally, scenario fY lead to valid yet still exploitable signatures.

10.4 — Signing with the recovered key

In the previous sections we showed how to recover $\mathbf{s}_1$ after a successful fault injection. However, $\mathbf{s}_1$ is only one component of the private key $sk = (\rho, K, tr, \mathbf{s}_1, \mathbf{s}_2, \mathbf{t}_0)$. The seed ρ, which is used for generating the matrix $\mathbf{A}$, is also part of the public key. tr can be trivially recomputed as $\mathsf{CRH}(\rho \| \mathbf{t}_1)$ (cf. Algorithm 10.1). K is used as a secret input to the deterministic sampler and cannot be recovered with our attack. However, an attacker can just choose any random K and still produce valid signatures. The only downside here is that the owner of the full private key can test whether or not a signature is forged. He simply runs the signature algorithm and tests for equality, a new K will obviously result in a different yet still valid signature.

The situation for the two remaining components, namely $\mathbf{s}_2$ and $\mathbf{t}_0$, is less clear. Recall that $\mathbf{t} := \mathbf{A}\mathbf{s}_1 + \mathbf{s}_2$ (cf. Algorithm 10.1). If $\mathbf{t}$ is known, then recovering $\mathbf{s}_2$ boils down to simple linear algebra. However, for compression the key generation computes a pair $(\mathbf{t}_1, \mathbf{t}_0)$ satisfying $\mathbf{t}_1 \cdot 2^d + \mathbf{t}_0 = \mathbf{t}$ and includes only the upper part $\mathbf{t}_1$ in the public key. Thus, the equation $\mathbf{t}_1 \cdot 2^d + \mathbf{t}_0 = \mathbf{A}\mathbf{s}_1 + \mathbf{s}_2$ cannot be directly solved.

Note also that during signature computation $\mathbf{s}_2$ and $\mathbf{t}_0$ are only used for hint generation and rejection purposes. Thus, there are no simple equations that can be exploited for recovering this part of the private key. This obviously does not imply that there is no information on $\mathbf{s}_2$ present. For instance, in a valid signature we have that $\|\mathbf{r}_0\|_\infty < \gamma_2 - \beta$, with $(\mathbf{r}_1, \mathbf{r}_0) := \mathsf{Decompose}(\mathbf{w} - c\mathbf{s}_2)$. As $\mathbf{w}$ is recoverable since $\mathbf{s}_1$ is already known, an attacker will get constraints for the possible values for $\mathbf{s}_2$. A large number of such constraints could result in a fully determined $\mathbf{s}_2$. However, we expect that a very large number of valid signatures and high computational effort is needed to perform such a recovery.

Instead, we now present a modified signing procedure (Algorithm 10.6) that does not

Algorithm 10.6 Dilithium Sign with recovered key $\mathbf{s}_1$

```
Input: Message $M$, private key part $\mathbf{s}_1$, public key $pk = (\rho, \mathbf{t}_1)$
Output: Signature $\sigma = (\mathbf{z}, \mathbf{h}, c)$
$tr \in \{0,1\}^{384} := \mathsf{CRH}(\rho \| \mathbf{t}_1)$        ▷ Recompute tr from public information
$K \leftarrow \{0,1\}^{256}$        ▷ Sample a random seed
$\mathbf{u} := \mathbf{A}\mathbf{s}_1 - \mathbf{t}_1 \cdot 2^d$        ▷ $\mathbf{A}\mathbf{s}_1 - \mathbf{t}_1 \cdot 2^d = \mathbf{t}_0 - \mathbf{s}_2$
$\mathbf{A} \in \mathcal{R}_q^{k \times \ell} := \mathsf{ExpandA}(\rho)$
$\mu \in \{0,1\}^{384} := \mathsf{CRH}(tr \| M)$
$\kappa := 0, (\mathbf{z}, \mathbf{h}) := \perp$
while $(\mathbf{z}, \mathbf{h}) = \perp$ do
    $\mathbf{y} \in S_{\gamma_1 - 1}^{l} := \mathsf{DeterministicSample}(K \| \mu \| \kappa)$
    $\mathbf{w} := \mathbf{A}\mathbf{y}$
    $\mathbf{w}_1 := \mathsf{HighBits}(\mathbf{w})$
    $c \in B_{60} := \mathsf{H}(\mu \| \mathbf{w}_1)$
    $\mathbf{z} := \mathbf{y} + c\mathbf{s}_1$
    $\mathbf{h} := \mathsf{MakeHint}(-c\mathbf{u}, \mathbf{w} + c\mathbf{u})$        ▷ $\mathsf{MakeHint}(-\mathbf{c}(\mathbf{s}_2 - \mathbf{t}_0, ), \mathbf{w} - c\mathbf{s}_2 + c\mathbf{t}_0)$
    if $\|\mathbf{z}\|_\infty \geqslant \gamma_1 - \beta$ then        ▷ Remove rejection conditions
        $(\mathbf{z}, \mathbf{h}) := \perp$
    else
        if not $\mathsf{Verify}(pk, M, (\mathbf{z}, \mathbf{h}, c))$ then $(\mathbf{z}, \mathbf{h}) := \perp$        ▷ Test for correctness
    $\kappa := \kappa + 1$
return $\sigma = (\mathbf{z}, \mathbf{h}, c)$
```

require knowledge of $\mathbf{s}_2$. Thus, the property that only a single valid/faulty signature pair is needed for the attack is preserved. Algorithm 10.6 starts off by recomputing *tr* and sampling a random K, as described earlier. Then we compute $\mathbf{u} := \mathbf{A}\mathbf{s}_1 - \mathbf{t}_1 \cdot 2^d$, which is exactly the difference of the unknown quantities, i.e., $\mathbf{u} = \mathbf{t}_0 - \mathbf{s}_2$. Signature generation then continues as usual up until the computation of the hint $\mathbf{h}$.

In the original signing algorithm we have $\mathbf{h} := \mathsf{MakeHint}(-c\mathbf{t}_0, \mathbf{w} - c\mathbf{s}_2 + c\mathbf{t}_0)$. The second argument to MakeHint can be trivially rewritten as $\mathbf{w} - c\mathbf{s}_2 + c\mathbf{t}_0 = \mathbf{w} + c\mathbf{u}$. The first argument $-c\mathbf{t}_0$ cannot be computed without knowledge of $\mathbf{t}_0$. We get around this by exploiting the fact that $\mathbf{t}_0$ is vastly larger than $\mathbf{s}_2$, with coefficients in the intervals $[\pm 2^{d-1}]$ and $[\pm\eta]$, respectively. Thus, we have that $\mathbf{u} = \mathbf{t}_0 - \mathbf{s}_0 \approx \mathbf{t}_0$ and simply substitute $-c\mathbf{t}_0$ with $-c\mathbf{u}$.

We then skip all rejection conditions that cannot be tested without knowing $\mathbf{s}_2$ or $\mathbf{t}_0$. Essentially, we just test if $\|\mathbf{z}\|_\infty \geqslant \gamma_1 - \beta$ and reject the signature if this is the case. Finally, we perform a verification of the signature to catch the very improbable case that $\mathsf{MakeHint}(-c\mathbf{u}, \mathbf{w} + c\mathbf{u}) \neq \mathsf{MakeHint}(-c\mathbf{t}_0, \mathbf{w} + c\mathbf{u})$, as that is the only thing that a real verifier would test.

Due to the removal of rejection conditions, this modified signing algorithm potentially leaks secret information. Thus, anyone being aware of the fact that signatures are computed by our modified algorithm could maybe also recover the secret key. Since all produced signatures are valid, there is no trivial way to detect this (without already knowing the key, as explained earlier).

10.5 — Experimental verification

In this section, we back up our previous theoretical expositions and simulations by running our attack on an actual device. After discussing our platform, we show how an attacker can inject a fault in the iteration κ_f without determining the concrete value. This requires at least some knowledge of the implementation. For this reason, we also demonstrate that a random fault anywhere during the signing procedure has a high chance of being exploitable.

Platform. For our experiments, we use an STM32F405 microcontroller (ARM Cortex-M4F) running on a ChipWhisperer CW308 side-channel evaluation board. We run the Dilithium C reference implementation[5] (compiled with `-O3`) and clock our device at 30 MHz. For attack evaluation, we signal the start and end of signing with a trigger pin. As faulting method we make use of clock glitches.

We mounted attacks for all scenarios except fA_ρ, all with success. For the scenarios targeting the SHAKE XOF, i.e., fA_E, fY, and fH, the ability to precisely time clock glitches and thus to attack very specific instructions is not needed. A single such permutation takes approximately 40 000 clock cycles and we only require that its output is different, thus any random fault suffices. In fact, we did not determine the exact location or effect of the fault. Attacks on the polynomial multiplication (scenario fW) can benefit from more precise fault injection (see Section 10.3.3). However, even random faults yield a high success rate (Section 10.5.2).

10.5.1 – Injecting a Fault in the Correct Iteration. Recall that a fault is only exploitable if both the faulted and the non-faulted execution of the signing algorithm terminate in the same iteration of the abort loop, i.e., $\Delta\kappa_f = 0$. Clearly, in the scenarios fY, fW, and fH, an attacker can maximize the success probability by injecting the fault in this last iteration κ_f.

The Dilithium reference implementation is constant (read: key-independent) time. The individual rejection conditions (line 12 of Algorithm 10.2) are still tested as soon as possible. This minimizes the runtime of failed iterations but does not leak sensitive information on the key. Quite on the contrary, this non-constant-time behavior somewhat complicates the fault attack. Even an attacker knowing κ_f cannot exactly pinpoint the time of execution of vulnerable operations and thus the best time to inject a fault.

We get around this by using the observation that the last loop iteration κ_f is, unlike the previous ones, constant time. Only there all operations are guaranteed to be performed and apart from the rejections the code is constant time. Thus, we determine the time of execution of vulnerable operations as follows. First, we perform the undisturbed signing and measure its runtime. And second, we simply subtract a fixed offset (depending on the to-be-faulted operation) from this overall runtime. We used this method for our attacks in the scenarios fY, fH, and fW, and were successful for any κ_f.

10.5.2 – Unprofiled Attacks. The above method is highly accurate, yet requires some device/code profiling. Concretely, an attacker needs to determine the time offsets (either from the start or finish of the signing operation) of the vulnerable code. This might not always be a realistic assumption. For this reason, we now show that an attacker injecting

[5] Reference implementation available at `https://pq-crystals.org/dilithium/software.shtml`

Table 10.5: Runtime-percentage of vulnerable code

	fA_E	fY	fW	fH	**Sum**
$\kappa_f = 1$	47.4	3.8	11.2	2.9	**65.2**
Overall	24.3	2.0	5.7	1.5	**33.5**

a random fault anywhere in the signing process still has a high chance of succeeding. We do so by measuring the runtime (in cycles) of the vulnerable code and relating it to the overall execution time (Table 10.5).

In the best-case scenario for such an attacker, the signing algorithm terminates in the first iteration ($\kappa_f = 1$). In this case, 65.2 % of execution time are vulnerable. In the general case (no restriction to $\kappa_f = 1$), the success probability goes down to one-third of the total execution time.

In both cases, sampling of the matrix **A** takes by far the most time. Additionally, it is performed at a fixed time in the execution, shortly after the invocation of the signing algorithm. Thus, in reality a unprofiled attacker faulting somewhere in this region has a much higher chance of hitting ExpandA than stated in Table 10.5.

In Figure 10.3, we further visualize the general case and compare runtime to success probability for different scenarios. Recall that depending on the concrete fault position, the success probability of scenario fW varies drastically (see Table 10.4). For the case of the unprofiled attacker, we narrowed down this probability by performing 1000 fault attacks on our target device, with faults at random positions inside fW. Approximately 62 % of these faults were exploitable. Faulting the call to H yields the highest success probability (Table 10.4), but also has the smallest footprint. As discussed in Section 10.3.5, 2/5 of the time spent on the SHAKE call by DeterministicSample is vulnerable to the attack. This makes it a slightly larger target compared to fH, but also with a much lower success probability. In total, a fault inside the vulnerable portions can be exploited with a probability of 56 %. These cover 33.5 % of execution time, thus approximately 19 % of random faults anywhere during signing lead to key recovery.

10.6 — Countermeasures

When presenting new attacks, a discussion on potential countermeasures should never be missing. For this reason, we present the applicability and effectiveness of three generic countermeasures against the fault attacks described in this chapter. For each of these methods, we give the runtime costs and state which fault scenarios will be mitigated by it. A summary of the latter is shown in Table 10.6.

Double computation. While determinism leads to the applicability of differential fault attacks in the first place, it can also be used as a countermeasure against such attacks. Concretely, many faults can be detected by running the signature algorithm twice and testing the output for equality. With double computation, we mean that the secret keys are loaded from memory and the signature algorithm as described in Algorithm 10.2 is run twice. This obviously doubles execution time. The countermeasure can be defeated by

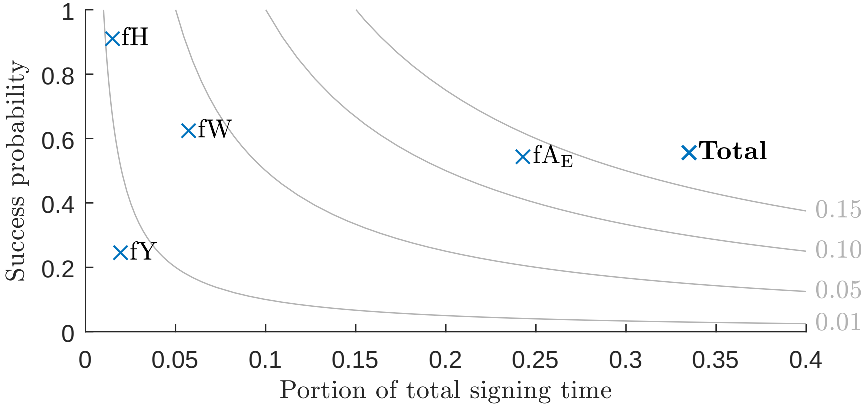


Figure 10.3: Comparison of scenarios regarding runtime as portion of total signing time (from Table 10.5) vs. success probability (from Table 10.4). Lines of constant product are drawn in solid gray.

either injecting an identical fault twice, which can be challenging, or by using a permanent fault, e.g., in scenario fA_ρ with the seed ρ.

Verification-after-sign.[6] Many of the presented attack scenarios lead to signatures being invalid. Thus, performing signature verification after signing is an effective countermeasure. As runtime costs of verification are less than one-third of signing (see the runtimes [LDK+17]), this option is also much more efficient than double computation. As a downside, however, it cannot detect faults injected into the sampling of $\mathbf{y}$ as this yields valid signatures.

Additional randomness. A final and very simple countermeasure is to re-randomize the deterministic sampling of the noise $\mathbf{y}$. One can simply sample a random $x \leftarrow \{0,1\}^{256}$ and then invoke $\mathbf{y} := \mathsf{DeterministicSample}(K\|\mu\|\kappa\|x)$. This effectively mitigates the differential fault attack as the faulted call to the signing algorithm uses different $\mathbf{y}$ and thus $\Delta\mathbf{y} \neq 0$.

Furthermore, this method might also hamper further side-channel attacks coming as side-effects of determinism. As observed by Seuschek, Heyszl and De Santis [SHS16], as well as by Samwell, Batina, Bertoni, Daemen and Susella [SBB+18], mixing the known message μ with the secret seed K in a hash function (in Dilithium this is SHAKE in $\mathsf{DeterministicSample}$) opens the gates for DPA-like attacks. Hash functions are hard to protect against such attacks; using an additional random input can be a cheap alternative. How x needs to be introduced to maximize the protection while keeping the necessary size of x small likely depends on the used hash function, further investigations are needed to answer this question for the case of SHAKE. However, as SHAKE is (quantum) indifferentiable from a random oracle [CMSZ19], this method should be fine.

[6] An earlier version of this work (published at TCHES and ePrint) stated that permanent faults in $\mathbf{A}$ can, at least under certain circumstances, not be detected with an additional verification of signatures. This is incorrect, verify-after-sign does protect against fA_ρ, as the verifier will use an unmodified $\mathbf{t}_1$.

Table 10.6: Applicable countermeasures

	fA_ρ	fA_E	fY	fW	fH
Double computation	✗	✓[7]	✓	✓	✓
Verification-after-sign	✓[6]	✓	✗	✓	✓
Additional randomness†	✓	✓	✓	✓	✓

†Not supported by proof of Dilithium [KLS18].

The added protection against implementation attacks does not negate the protection against incorrect implementation and resulting nonce reuse (using the same $\mathbf{y}$ for different messages). For instance, using a constant r effectively reverts signing to its deterministic version. Additional upsides of this countermeasure are its simplicity and negligible runtime overhead. Furthermore, unlike straight-forward implementations of the two previous countermeasures, it is single-pass and so does not require to keep a copy of the message in memory. Note that this countermeasure was already proposed in the context of EdDSA [ABF+18, SBB+18] and SPHINCS [BHH+15], but it can also be applied to lattice-based signatures. In fact, a very recent update of the qTESLA specification made use of this countermeasure mandatory and cites the presented attacks as reason.

There are, however, also considerable downsides of this countermeasure. First, unlike the two previous countermeasures, this countermeasure is probabilistic and requires some source of entropy, i.e., a true random number generator. Such a generator might not be available on all devices, especially low-resource ones. And second, this countermeasure violates the security proof of Dilithium. Kiltz, Lyubashevksy, and Schaffner [KLS18] present a tight proof in the quantum random oracle model (QROM) based on the hardness of MLWE, MSIS, and a new problem called SelfTargetMSIS. They require the signature scheme to be deterministic. They do give an alternative proof for a probabilistic version of Dilithium, yet it is not tight and loses security linearly in the number of observed different signatures per message.

Thus, introducing this countermeasure voids provable security guarantees, albeit no concrete attack is known. The Dilithium authors "*still recommend using deterministic signatures except in environments that may be vulnerable to the aforementioned side-channel attacks*" [LDK+17]. However, determining whether or not an environment is vulnerable is not easy, as clearly shown by the Rowhammer attack.

10.7 — Description of qTESLA

In this section we briefly describe the qTESLA signature scheme [BAA+17]. Please note that we give the originally submitted version of qTESLA. Following the initial publication of this work, in a very recent update of the qTESLA specification the *additional randomness* countermeasure was incorporated as part of the algorithm description. As the security proof of qTESLA allows for a probabilistic version, this countermeasure can

[7] For this countermeasure to work, we assume that $\mathbf{A}$ is not cached in memory after it is used for the first time, i.e. $\mathsf{expandA}$ is called again in the second signature computation. If $\mathbf{A}$ would be cached in memory, the two signature computations would be identical and the countermeasure would fail.

be used without violating any security guarantees. Hence the attacks are no longer applicable.

In Algorithms 10.7, 10.8, and 10.9, we give slightly simplified versions of key generation, signing, and verification, respectively. Note qTESLA's similarity to Dilithium, we highlight this similarity by stating the corresponding variable and function names in Table 10.7. The main difference between Dilithium and qTESLA is that the latter is based on Ring-LWE and thus operates on polynomials in $\mathbb{Z}_q[x]/(x^n+1)$ with $n \geqslant 1024$. Dilithium is based on the Module-LWE assumption and uses vectors/matrices of polynomials in a fixed base ring $\mathbb{Z}_q[x]/(x^{256}+1)$.

Algorithm 10.7 qTESLA Key Generation

```
Output: Keypair $(pk, sk)$
seed_a ← $\{0,1\}^{256}$, seed_y ← $\{0,1\}^{256}$
$a \in \mathcal{R}_q$ := GenA(seed_a)
repeat
    $s \in \mathcal{R}_q \leftarrow D_\sigma, e \in \mathcal{R}_q \leftarrow D_\sigma$        ▷ Discrete Gaussian distribution $D_\sigma$
until $s$ and $e$ fulfill certain criteria
$t := as + e \bmod q$
return $(pk = (\text{seed}_a, t), sk = (s, e, \text{seed}_y, \text{seed}_a))$
```

Algorithm 10.8 qTESLA Sign (simplified)

```
Input: Message $M$, private key $sk = (s, e, \text{seed}_y, \text{seed}_a)$
Output: Signature $\sigma = (c, z)$
$a \in \mathcal{R}_q := \text{GenA}(\text{seed}_a)$
counter := 0
rand := $\text{PRF}_1(\text{seed}_y, M)$
repeat
    $y := \text{PRF}_2(\text{rand}, \text{counter})$
    $v := ay \bmod q$
    $c := \text{H}(\text{Round}(v), M)$
    $z := y + sc$
    counter := counter + 1
until Reject$(z, v, c, sk)$
return $\sigma = (c, z)$
```

Algorithm 10.9 qTESLA Verify (simplified)

```
Input: Public key $pk = (\text{seed}_a, t)$, message $M$, signature $\sigma = (c, z))$
$a \in \mathcal{R}_q := \text{GenA}(\text{seed}_a)$
$w := az - tc \bmod q$
return $c = \text{H}(\text{Round}(w), M)$
```

Table 10.7: Comparison of variable/parameter names and function names for Dilithium and qTESLA. Only differing names are listed.

Variables:

Dilithium	ρ	K	$\mathbf{s}_1$	$\mathbf{s}_2$	κ	μ	$\mathbf{w}$
qTESLA	seed$_a$	seed$_y$	s	e	counter	rand	ν

Functions:

Dilithium	$\mathsf{ExpandA}$	CRH	$\mathsf{DeterministicSample}$	$\mathsf{HighBits}$
qTESLA	GenA	PRF_1	PRF_2	Round

Applicability of our attacks. All attacks for Dilithium described in Section 10.3 can easily be adapted to the original *deterministic* version of qTESLA, with obviously differing success probabilities due to different parameter sets, rejection conditions, and algebraic structure. In particular, the major fault scenario (a random fault in SHAKE) would be the same: SHAKE is used in qTESLA in a similar way to build the functions described in Table 10.7. A subtle difference however is that Dilithium samples multiple smaller polynomials, e.g., $\mathbf{y} \in \mathcal{R}_q^\ell$, using independent calls to SHAKE, whereas qTESLA uses just one call to SHAKE to sample a single but larger polynomial. This most likely affects success rates and also the available time for fault injection. For instance, in scenario fY in Dilithium one can inject a fault in the last 2 permutations in any one of the ℓ independent SHAKE calls, whereas in qTESLA only the last permutations of the single SHAKE call can be faulted.

After faulting, key recovery is exactly the same, i.e., computing $s = \Delta c^{-1} \cdot \Delta z$. Note that in qTESLA the public key t is not compressed, thus recovering e (which corresponds to $\mathbf{s}_2$ in Dilithium) is trivial as soon as s (corresponds to $\mathbf{s}_1$) is known. No adapted signature algorithm (as described in Section 10.4) is needed.

Chapter 11

Conclusions and future work

We now revisit the research questions posed in Chapters 7 to 10 of this thesis.

Q1: How much does left-to-right sliding window exponentiation hurt security?

In Chapter 7 we analyzed the common belief about sliding window methods: only 40% (or 33% depending on the window size) of the bits are leaking trough a side-channel attack: not enough for a full key-recovery. We show that that belief is incorrect for the left-to-right recoding: this recoding actually leaks many more bits than previously estimated. We showed two possibilities in analyzing the additional information that can be recovered from the square and multiply sequence. The first way is to determine specific rules that transform this sequence into known bits. For RSA-1024 and window-size of 4, this already means that the number of recovered bits is enough to finish the key-recovery. However, for RSA-2048 with window-size of 5, this was not enough. Instead of translating the square and multiply sequence to known bits, the sequence was used directly in the pruning algorithms by Heninger-Shacham, which greatly improved the attack. This also incorporated information that is not directly translatable to known bits. This way, 13% of the RSA-2048 keys were vulnerable to key-recovery.

We have disclosed this issue to the Libgcrypt maintainers and have worked with them to produce and validate a patch to mitigate our attack. The vulnerability was assigned CVE-2017-7526. This motivates the necessity to also carefully handle side-channel attacks on post-quantum cryptography.

Q2: Are side-channel attacks possible on the discrete Gaussian sampler?

We showed several side-channel attacks on five different samplers for the discrete Gaussian distribution in Chapter 8. In particular, we carried out the analysis for two prominent samplers: the CDT sampler (with guide table) and the Bernoulli sampler. We showed how the memory access patterns of these samplers leaked enough information to determine the sampled value, possibly up to minor errors. The attacks were performed on the lattice-based signature scheme BLISS, that uses noise sampled from the discrete Gaussian distribution to hide the secret key within the signature. To remove (possible) measurement errors, we built a lattice whose basis is generated from multiple challenge vectors. This basis was then reduced using LLL, returning possible secret keys in the unitary transformation matrix retrieved from LLL. A proof-of-concept attack verified the vulnerability. All

BLISS parameter suggestions turned out to be vulnerable to the cache attacks. Roughly up to 4200 signature executions were needed to perform the attacks.

This work is the basis of many follow-ups (e.g. [Pes16, PGY17]), and is possibly the reason why BLISS was not submitted to the NIST post-quantum standardization competition. Instead, several of the authors of BLISS were involved in the submitted scheme Dilithium (Chapter 10), which does not use noise sampled from the discrete Gaussian distribution and cites this work (and in general side-channel attacks) as motivation for this choice.

Q3: Is BLISS-B a free countermeasure against side-channel attacks?

In Chapter 9 we showed that BLISS-B is also vulnerable to side-channel attacks. However, this required significantly more steps than the attacks against BLISS. We presented a new side-channel key-recovery algorithm against both the original BLISS and the BLISS-B variant. Our key-recovery algorithm draws from a wide array of techniques, including learning-parity with noise, integer programming, maximum likelihood tests, and a lattice-basis reduction. With each application of a technique, we reveal additional information on the secret key culminating in a complete key recovery. Finally, we showed that cache attacks on post-quantum cryptography are not only possible, but also practical. We mount an asynchronous cache attack on the production-grade BLISS-B implementation of strongSwan. The attack recovers the secret signing key after observing roughly 6000 signature generations.

We have disclosed the vulnerability to strongSwan, but since the attacks are very technical and non-trivial to mount (and for some parameter sets do not even work), the maintainers decided not to fix this vulnerability. This means the side-channel vulnerability is still there.

Q4: Do deterministic versions of lattice-based signature schemes hurt security?

In Chapter 10 we extended the applicability of differential fault attacks to lattice-based cryptography. In particular, we showed how two deterministic lattice-based signature schemes, Dilithium and qTESLA, are vulnerable to such attacks. In particular, we demonstrated that single random faults can result in a nonce-reuse scenario which allows key recovery. We also expand this to fault-induced partial nonce-reuse attacks, which do not corrupt the validity of the computed signatures and thus are harder to detect. Using linear algebra and lattice-basis reduction techniques, an attacker can extract one of the secret key elements after a successful fault injection. Some other parts of the key cannot be recovered, but a tweaked signature algorithm can be used to sign any message using the partial info of the secret key. We provide experimental verification of our attacks by performing clock glitching on an ARM Cortex-M4 micro-controller. In particular, we show that up to 65.2% of the execution time of Dilithium is vulnerable to an unprofiled attack, where a random fault is injected anywhere during the signing procedure and still leads to a successful key-recovery.

Shortly after the first publication of our result, the qTESLA team modified its specification to make the additional randomness countermeasure (i.e. remove the deterministic feature) mandatory and cites the presented attacks as reason.

Open problems

We end Part II on this thesis by posing some open problems that remain after the questions answered in Chapters 7 to 10.

I: Can the attacks be applied to other window methods?

The sliding-window method for RSA is not the only "window method" in cryptography. There are many cryptographic implementations for e.g. elliptic curves that use similar techniques. For example, in previous works [BvdPSY14] attacks were shown against elliptic-curve signature schemes where a so-called window Non-Adjacent Form (wNAF) was used for the double-and-add sequence. The attack was finished using lattice-basis reduction. But there are many other window-forms for efficient elliptic-curve arithmetic [MSJ13]: some are sliding windows, some are fixed windows; some can only be used in a right-to-left manner, while others can only be used left-to-right. It would be interesting to see which of these methods are vulnerable to attacks such as in Chapter 7. And more importantly, which attacks can benefit from the improved analysis, e.g. additional rules that capture more bits before applying a full key-recovery algorithm.

II: Can we give formal arguments why the side-channel attack on BLISS works?

In Chapter 8, we used a lattice-basis reduction algorithm (LLL) to find the secret key, given a lattice spanned by challenge vectors. Actually, the secret key is found in the unimodular transformation matrix given as part of the LLL output. This means that one of the short vectors that LLL recovers, is the vector containing the errors from measurements (or the errors in the attack on CDT). This is unexpected, as usually the short vector that LLL/BKZ finds (or more exact algorithms that find the shortest vector), is the secret key itself. There are several questions about why our technique actually works: i.e. why does LLL even find such a short vector? Although the challenge vectors are sparse (only κ non-zero coefficients), the determinant of the lattice is in that case still large. This means that the approximated length of the vectors found by LLL is also large. But instead, LLL finds vectors that are very close to zero, i.e. vectors with fewer than κ non-zeros. It would be interesting to know why LLL works so well in this case. It could be that the sparsity of the challenge vectors improves the working of the algorithm a lot, as in that case the vectors are also relatively more orthogonal than for, e.g., random lattices. To state this question in more generality, it would be interesting to give formal arguments why the attacks on BLISS work.

III: Can we perform a branch-and-prune algorithm on the signs of the secret key?

In Chapter 9, we apply several steps to recover additional information on the secret key. For the first couple of steps, we use the information recovered from samples (signatures coupled with the side channel), up to the point where we know the magnitudes of the secret key (everything except for the signs of the non-zero coefficients). From there, we apply a lattice-basis reduction algorithm on the lattice that remains after removing the zero coefficients. Unfortunately, it turned out that the rank of the remaining lattice was still too large for certain BLISS-B parameter sets (BLISS-III and BLISS-IV), meaning we were unable to finish the key-recovery. The question remains if we can do better than a lattice-reduction algorithm. A possible technique might be some sort of branch-and-prune algorithm, similar to the Heninger-Shacham algorithm for RSA (see Chapter 7). Each branch would fix the sign for a certain coefficient, and all the information in the recovered samples decide whether to prune. This means that if, at a certain point, a branch has conflicting samples (i.e. the chosen signs of the secret key do not match the possibilities given by certain samples), the branch is pruned. It might be that such a technique is able to recover the signs of the secret key more efficiently, even tough the search space is still very large. Such a technique would be valuable, as this means that other side-channel attacks on lattice-based schemes could mainly focus on recovering the magnitudes of the secret key and from there apply the key-recovery.

IV: Can we adapt the fault attacks to deterministic lattice-based encryption schemes?

In Chapter 10, we showed the applicability of differential fault attacks to deterministic lattice-based signature schemes. We showed several possible attacks on two schemes, Dilithium and qTESLA, which are both deterministic, by generating randomness from the secret key and the message. However, similar techniques also exist for encryption schemes. Many lattice-based encryption schemes (including HILA5, see Chapter 4) require randomness as well, but can be made deterministic in an analogous way to the signature schemes: by generating randomness deterministically from the message (and possibly the public key). In fact, turning a lattice-based public-key encryption scheme into a post-quantum IND-CCA KEM requires the scheme to be deterministic [SXY18]. It would be interesting to see if such schemes are also vulnerable to (differential) fault attacks. If such an attack can recover the encrypted message, it is probably not enough to recover the secret decryption key. However, additional steps might lead to a possible reaction attack as in Chapter 4, i.e. a reaction attack coupled with fault injection on a deterministic IND-CCA lattice-based encryption scheme.

Bibliography

[ABB$^+$16] Sedat Akleylek, Nina Bindel, Johannes A. Buchmann, Juliane Krämer, and Giorgia Azzurra Marson. An efficient lattice-based signature scheme with provably secure instantiation. In David Pointcheval, Abderrahmane Nitaj, and Tajjeeddine Rachidi, editors, *AFRICACRYPT*, volume 9646 of *Lecture Notes in Computer Science*, pages 44–60. Springer, 2016.

[ABB$^+$17] Erdem Alkim, Nina Bindel, Johannes A. Buchmann, Özgür Dagdelen, Edward Eaton, Gus Gutoski, Juliane Krämer, and Filip Pawlega. Revisiting TESLA in the quantum random oracle model. In Tanja Lange and Tsuyoshi Takagi, editors, *PQCrypto*, volume 10346 of *Lecture Notes in Computer Science*, pages 143–162. Springer, 2017.

[ABD$^+$17] Jean-Philippe Aumasson, Daniel J. Bernstein, Christoph Dobraunig, Maria Eichlseder, Scott Fluhrer, Stefan-Lukas Gazdag, Andreas Hülsing, Panos Kampanakis, Stefan Kölbl, Tanja Lange, Martin M. Lauridsen, Florian Mendel, Ruben Niederhagen, Christian Rechberger, Joost Rijneveld, and Peter Schwabe. SPHINCS$^+$ stateless hash-based signatures. Submission to the NIST Post-Quantum Cryptography Standardization [NIS], 2017. `https://sphincs.org/`.

[ABF$^+$16] Thomas Allan, Billy Bob Brumley, Katrina E. Falkner, Joop van de Pol, and Yuval Yarom. Amplifying side channels through performance degradation. In Stephen Schwab, William K. Robertson, and Davide Balzarotti, editors, *ACSAC*, pages 422–435. ACM, 2016.

[ABF$^+$18] Christopher Ambrose, Joppe W. Bos, Björn Fay, Marc Joye, Manfred Lochter, and Bruce Murray. Differential attacks on deterministic signatures. In Nigel P. Smart, editor, *CT-RSA*, volume 10808 of *Lecture Notes in Computer Science*, pages 339–353. Springer, 2018.

[AD97] Miklós Ajtai and Cynthia Dwork. A public-key cryptosystem with worst-case/average-case equivalence. In Frank Thomson Leighton and Peter W. Shor, editors, *STOC*, pages 284–293. ACM, 1997.

[ADPS16] Erdem Alkim, Léo Ducas, Thomas Pöppelmann, and Peter Schwabe. Post-quantum key exchange - A new hope. In Thorsten Holz and Stefan Savage, editors, *USENIX*, pages 327–343. USENIX Association, 2016.

[AFG13] Martin R. Albrecht, Robert Fitzpatrick, and Florian Göpfert. On the efficacy of solving LWE by reduction to unique-svp. In Hyang-Sook Lee and Dong-Guk Han, editors, *ICISC*, volume 8565 of *Lecture Notes in Computer Science*, pages 293–310. Springer, 2013.

[AHMN10] Jean-Philippe Aumasson, Luca Henzen, Willi Meier, and María Naya-Plasencia. Quark: A lightweight hash. In Stefan Mangard and François-Xavier Standaert, editors, *CHES*, volume 6225 of *Lecture Notes in Computer Science*, pages 1–15. Springer, 2010.

[AIES14] Gorka Irazoqui Apecechea, Mehmet Sinan Inci, Thomas Eisenbarth, and Berk Sunar. Wait a minute! A fast, cross-VM attack on AES. In Angelos Stavrou, Herbert Bos, and Georgios Portokalidis, editors, *RAID*, volume 8688 of *Lecture Notes in Computer Science*, pages 299–319. Springer, 2014.

[AP13] Nadhem J. AlFardan and Kenneth G. Paterson. Lucky thirteen: Breaking the TLS and DTLS record protocols. In *IEEE Symposium on Security and Privacy*, pages 526–540. IEEE Computer Society, 2013.

[Ayi19] Raji Ayinla. OpenSSH 8.0 Releasing With Quantum Computing Resistant Keys, March 2019. `https://codesmithdev.com/openssh-8-0-releasing-with-quantum-computing-resistant-keys/`.

[BAA$^+$17] Nina Bindel, Sedat Akleylek, Erdem Alkim, Paulo S. L. M. Barreto, Johannes Buchmann, Edward Eaton, Gus Gutoski, Juliane Krämer, Patrick Longa, Harun Polat, Jefferson E. Ricardini, and Gustavo Zanon. qTESLA. Submission to the NIST Post-Quantum Cryptography Standardization [NIS], 2017. `https://qtesla.org`.

[BBF$^+$17] Hayo Baan, Souvik Bhattacharya, Scott Fluhrer, Oscar Garcia-Morchon, Thijs Laarhoven, Rachel Player, Ronald Rietman, Markku-Juhani O. Saarinen, Ludo Tolhuizen, Jose Luis T. Arce, and Zhenfei Zhang. Round5. Submission to the NIST Post-Quantum Cryptography Standardization [NIS], 2017. `https://round5.org/`.

[BBG$^+$17a] Daniel J. Bernstein, Joachim Breitner, Daniel Genkin, Leon Groot Bruinderink, Nadia Heninger, Tanja Lange, Christine van Vredendaal, and Yuval Yarom. Sliding right into disaster: Left-to-right sliding windows leak. Cryptology ePrint Archive, Report 2017/627, 2017. `https://eprint.iacr.org/2017/627`.

[BBG$^+$17b] Daniel J. Bernstein, Joachim Breitner, Daniel Genkin, Leon Groot Bruinderink, Nadia Heninger, Tanja Lange, Christine van Vredendaal, and Yuval Yarom. Sliding right into disaster: Left-to-right sliding windows leak. In Wieland Fischer and Naofumi Homma, editors, *CHES*, volume 10529 of *Lecture Notes in Computer Science*, pages 555–576. Springer, 2017.

[BBK16] Nina Bindel, Johannes A. Buchmann, and Juliane Krämer. Lattice-based signature schemes and their sensitivity to fault attacks. In *FDTC*, pages 63–77. IEEE Computer Society, 2016.

[BCG$^+$13] Johannes A. Buchmann, Daniel Cabarcas, Florian Göpfert, Andreas Hülsing, and Patrick Weiden. Discrete Ziggurat: A time-memory trade-off for sampling from a Gaussian distribution over the integers. In Tanja Lange, Kristin E. Lauter, and Petr Lisonek, editors, *SAC*, volume 8282 of *Lecture Notes in Computer Science*, pages 402–417. Springer, 2013.

[BCLvV17] Daniel J. Bernstein, Chitchanok Chuengsatiansup, Tanja Lange, and Christine van Vredendaal. NTRU prime: Reducing attack surface at low cost. In Carlisle Adams and Jan Camenisch, editors, *SAC*, volume 10719 of *Lecture Notes in Computer Science*, pages 235–260. Springer, 2017.

[BCN$^+$06] Hagai Bar-El, Hamid Choukri, David Naccache, Michael Tunstall, and Claire Whelan. The Sorcerer's Apprentice Guide to Fault Attacks. *Proceedings of the IEEE*, 94(2):370–382, 2006.

[BCNS15] Joppe W. Bos, Craig Costello, Michael Naehrig, and Douglas Stebila. Post-quantum key exchange for the TLS protocol from the ring learning with errors problem. In *IEEE Symposium on Security and Privacy*, pages 553–570. IEEE Computer Society, 2015.

[BDE$^+$11] Johannes A. Buchmann, Erik Dahmen, Sarah Ereth, Andreas Hülsing, and Markus Rückert. On the security of the winternitz one-time signature scheme. In Abderrahmane Nitaj and David Pointcheval, editors, *AFRICACRYPT*, volume 6737 of *Lecture Notes in Computer Science*, pages 363–378. Springer, 2011.

[BDF$^+$11] Dan Boneh, Özgür Dagdelen, Marc Fischlin, Anja Lehmann, Christian Schaffner, and Mark Zhandry. Random oracles in a quantum world. In Dong Hoon Lee and Xiaoyun Wang, editors, *ASIACRYPT*, volume 7073 of *Lecture Notes in Computer Science*, pages 41–69. Springer, 2011.

[BDK$^+$07] Johannes A. Buchmann, Erik Dahmen, Elena Klintsevich, Katsuyuki Okeya, and Camille Vuillaume. Merkle signatures with virtually unlimited signature capacity. In Jonathan Katz and Moti Yung, editors, *ACNS*, volume 4521 of *Lecture Notes in Computer Science*, pages 31–45. Springer, 2007.

[BDL$^+$11] Daniel J. Bernstein, Niels Duif, Tanja Lange, Peter Schwabe, and Bo-Yin Yang. High-Speed High-Security Signatures. In Bart Preneel and Tsuyoshi Takagi, editors, *CHES*, volume 6917 of *Lecture Notes in Computer Science*, pages 124–142. Springer, 2011.

[BDM$^+$12] Thierry P. Berger, Joffrey D'Hayer, Kevin Marquet, Marine Minier, and Gaël Thomas. The GLUON family: A lightweight hash function family based on fcsrs. In Aikaterini Mitrokotsa and Serge Vaudenay, editors, *AFRICACRYPT*, volume 7374 of *Lecture Notes in Computer Science*, pages 306–323. Springer, 2012.

[BDPA08] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. On the indifferentiability of the sponge construction. In Nigel P. Smart, editor, *EUROCRYPT*, volume 4965 of *Lecture Notes in Computer Science*, pages 181–197. Springer, 2008.

[BDPV07] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. Sponge functions. ECRYPT Hash Workshop, 2007.

[Ber05] Daniel J. Bernstein. Cache-timing attacks on AES, 2005. Preprint available at `http://cr.yp.to/antiforgery/cachetiming-20050414.pdf`.

[BGD+06] Johannes A. Buchmann, Luis Carlos Coronado García, Erik Dahmen, Martin Döring, and Elena Klintsevich. CMSS - an improved merkle signature scheme. In Rana Barua and Tanja Lange, editors, *INDOCRYPT*, volume 4329 of *Lecture Notes in Computer Science*, pages 349–363. Springer, 2006.

[BGLP18] Daniel J. Bernstein, Leon Groot Bruinderink, Tanja Lange, and Lorenz Panny. HILA5 Pindakaas: On the CCA security of lattice-based encryption with error correction. In Antoine Joux, Abderrahmane Nitaj, and Tajjeeddine Rachidi, editors, *AFRICACRYPT*, volume 10831 of *Lecture Notes in Computer Science*, pages 203–216. Springer, 2018.

[BHH+15] Daniel J. Bernstein, Daira Hopwood, Andreas Hülsing, Tanja Lange, Ruben Niederhagen, Louiza Papachristodoulou, Michael Schneider, Peter Schwabe, and Zooko Wilcox-O'Hearn. SPHINCS: practical stateless hash-based signatures. In Elisabeth Oswald and Marc Fischlin, editors, *EUROCRYPT*, volume 9056 of *Lecture Notes in Computer Science*, pages 368–397. Springer, 2015.

[BHHP19] Nina Bindel, Mike Hamburg, Andreas Hülsing, and Edoardo Persichetti. Tighter proofs of CCA security in the quantum random oracle model. Cryptology ePrint Archive, Report 2019/590, 2019. `https://eprint.iacr.org/2019/590`.

[BKL+13] Andrey Bogdanov, Miroslav Knezevic, Gregor Leander, Deniz Toz, Kerem Varici, and Ingrid Verbauwhede. SPONGENT: the design space of lightweight cryptographic hashing. *IEEE Trans. Computers*, 62(10):2041–2053, 2013.

[BMS10] "Bushing", "Marcan", and "Sven". PS3 epic fail. 27th Chaos Communication Congress, 2010. `https://events.ccc.de/congress/2010/Fahrplan/events/4087.en.html`.

[BP16] Alessandro Barenghi and Gerardo Pelosi. A note on fault attacks against deterministic signature schemes. In Kazuto Ogawa and Katsunari Yoshioka, editors, *IWSEC*, volume 9836 of *Lecture Notes in Computer Science*, pages 182–192. Springer, 2016.

[BR93] Mihir Bellare and Phillip Rogaway. Random oracles are practical: A paradigm for designing efficient protocols. In Dorothy E. Denning, Raymond Pyle, Ravi Ganesan, Ravi S. Sandhu, and Victoria Ashby, editors, *ACM CCS*, pages 62–73. ACM, 1993.

[Bra16] Matt Braithwaite. Experimenting with Post-Quantum Cryptography, July 2016. `https://security.googleblog.com/2016/07/experimenting-with-post-quantum.html`.

[BvdPSY14] Naomi Benger, Joop van de Pol, Nigel P. Smart, and Yuval Yarom. "ooh aah... just a little bit" : A small amount of side channel can go a long way. In Lejla Batina and Matthew Robshaw, editors, *CHES*, volume 8731 of *Lecture Notes in Computer Science*, pages 75–92. Springer, 2014.

[CA74] Hui-Chuan Chen and Yoshinori Asau. On generating random variates from an empirical distribution. *AIIE Transactions*, 6(2):163–166, 1974.

[CDF$^+$07] J. Callas, L. Donnerhacke, H. Finney, D. Shaw, and R. Thayer. OpenPGP message format. RFC 4880, November 2007.

[CGH$^+$18] Jan Czajkowski, Leon Groot Bruinderink, Andreas Hülsing, Christian Schaffner, and Dominique Unruh. Post-quantum security of the sponge construction. In Tanja Lange and Rainer Steinwandt, editors, *PQCrypto*, volume 10786 of *Lecture Notes in Computer Science*, pages 185–204. Springer, 2018.

[CGHS17] Jan Czajkowski, Leon Groot Bruinderink, Andreas Hülsing, and Christian Schaffner. Quantum preimage, 2nd-preimage, and collision resistance of SHA3. Cryptology ePrint Archive, Report 2017/302, 2017. `https://eprint.iacr.org/2017/302`.

[CHS19] Jan Czajkowski, Andreas Hülsing, and Christian Schaffner. Quantum indistinguishability of random sponges. Cryptology ePrint Archive, Report 2019/069, 2019. `https://eprint.iacr.org/2019/069`.

[CMP18] Laurent Castelnovi, Ange Martinelli, and Thomas Prest. Grafting trees: A fault attack against the SPHINCS framework. In Tanja Lange and Rainer Steinwandt, editors, *PQCrypto*, volume 10786 of *Lecture Notes in Computer Science*, pages 165–184. Springer, 2018.

[CMSZ19] Jan Czajkowski, Christian Majenz, Christian Schaffner, and Sebastian Zur. Quantum lazy sampling and game-playing proofs for quantum indifferentiability. *CoRR*, abs/1904.11477, 2019.

[CN11] Yuanmi Chen and Phong Q. Nguyễn. BKZ 2.0: Better lattice security estimates. In Dong Hoon Lee and Xiaoyun Wang, editors, *ASIACRYPT*, volume 7073 of *Lecture Notes in Computer Science*, pages 1–20. Springer, 2011.

[DAS$^+$17] Jintai Ding, Saed Alsayigh, R. V. Saraswathy, Scott R. Fluhrer, and Xiaodong Lin. Leakage of signal function with reused keys in RLWE key exchange. In *ICC*, pages 1–6. IEEE, 2017.

[dCRVV15] Ruan de Clercq, Sujoy Sinha Roy, Frederik Vercauteren, and Ingrid Verbauwhede. Efficient software implementation of ring-LWE encryption. In Wolfgang Nebel and David Atienza, editors, *DATE*, pages 339–344. ACM, 2015.

[DDLL13a] Léo Ducas, Alain Durmus, Tancrède Lepoint, and Vadim Lyubashevsky. BLISS: Bimodal Lattice Signature Schemes, 2013. `http://bliss.di.ens.fr/`.

[DDLL13b] Léo Ducas, Alain Durmus, Tancrède Lepoint, and Vadim Lyubashevsky. Lattice signatures and bimodal gaussians. In Ran Canetti and Juan A. Garay, editors, *CRYPTO*, volume 8042 of *Lecture Notes in Computer Science*, pages 40–56. Springer, 2013.

[DG14] Nagarjun C. Dwarakanath and Steven D. Galbraith. Sampling from discrete gaussians for lattice-based cryptography on a constrained device. *Appl. Algebra Eng. Commun. Comput.*, 25(3):159–180, 2014.

[DLL$^+$17] Léo Ducas, Tancrède Lepoint, Vadim Lyubashevsky, Peter Schwabe, Gregor Seiler, and Damien Stehlé. CRYSTALS – Dilithium: Digital signatures from module lattices. Cryptology ePrint Archive, Report 2017/633, 2017. Publication to [LDK$^+$17].

[Doc05] Christophe Doche. Exponentiation. In *Handbook of Elliptic and Hyperelliptic Curve Cryptography.*, pages 144–168. Chapman and Hall/CRC, 2005.

[Duc14] Léo Ducas. Accelerating bliss: the geometry of ternary polynomials. Cryptology ePrint Archive, Report 2014/874, 2014. `https://eprint.iacr.org/2014/874`.

[EFGT16] Thomas Espitau, Pierre-Alain Fouque, Benoît Gérard, and Mehdi Tibouchi. Loop-abort faults on lattice-based Fiat-Shamir and hash-and-sign signatures. In Roberto Avanzi and Howard M. Heys, editors, *SAC*, volume 10532 of *Lecture Notes in Computer Science*, pages 140–158. Springer, 2016.

[EGM96] Shimon Even, Oded Goldreich, and Silvio Micali. On-line/off-line digital signatures. *Journal of Cryptology*, 9(1):35–67, 1996.

[Flu16] Scott R. Fluhrer. Cryptanalysis of ring-LWE based key exchange with key share reuse, 2016. IACR Cryptology ePrint Archive 2016/085 `https://ia.cr/2016/085`.

[FO99] Eiichiro Fujisaki and Tatsuaki Okamoto. Secure integration of asymmetric and symmetric encryption schemes. In Michael J. Wiener, editor, *CRYPTO*, volume 1666 of *Lecture Notes in Computer Science*, pages 537–554. Springer, 1999.

[FS86] Amos Fiat and Adi Shamir. How to prove yourself: Practical solutions to identification and signature problems. In Andrew M. Odlyzko, editor, *CRYPTO*, volume 263 of *Lecture Notes in Computer Science*, pages 186–194. Springer, 1986.

[GBK11] David Gullasch, Endre Bangerter, and Stephan Krenn. Cache games - bringing access-based cache attacks on AES to practice. In *IEEE Symposium on Security and Privacy*, pages 490–505. IEEE Computer Society, 2011.

[GH17] Leon Groot Bruinderink and Andreas Hülsing. "Oops, I did it again" – Security of one-time signatures under two-message attacks. In Carlisle Adams and Jan Camenisch, editors, *SAC*, volume 10719 of *Lecture Notes in Computer Science*, pages 299–322. Springer, 2017.

[GHLY16] Leon Groot Bruinderink, Andreas Hülsing, Tanja Lange, and Yuval Yarom. Flush, Gauss, and Reload – A cache attack on the BLISS lattice-based signature scheme. In Benedikt Gierlichs and Axel Y. Poschmann, editors, *CHES*, volume 9813 of *Lecture Notes in Computer Science*, pages 323–345. Springer, 2016.

[GJL14] Qian Guo, Thomas Johansson, and Carl Löndahl. Solving LPN using covering codes. In Palash Sarkar and Tetsu Iwata, editors, *ASIACRYPT (1)*, volume 8873 of *Lecture Notes in Computer Science*, pages 1–20. Springer, 2014.

[GLP12] Tim Güneysu, Vadim Lyubashevsky, and Thomas Pöppelmann. Practical Lattice-Based Cryptography: A Signature Scheme for Embedded Systems. In Emmanuel Prouff and Patrick Schaumont, editors, *CHES*, volume 7428 of *Lecture Notes in Computer Science*, pages 530–547. Springer, 2012.

[GMM16] Daniel Gruss, Clémentine Maurice, and Stefan Mangard. Rowhammer.js: A remote software-induced fault attack in JavaScript. In Juan Caballero, Urko Zurutuza, and Ricardo J. Rodríguez, editors, *DIMVA*, volume 9721 of *Lecture Notes in Computer Science*, pages 300–321. Springer, 2016.

[GNUa] GNU Privacy Guard. URL: `https://www.gnupg.org`.

[Gnub] GnuPG Frontends. URL: `https://www.gnupg.org/related_software/frontends.html`.

[GP18] Leon Groot Bruinderink and Peter Pessl. Differential fault attacks on deterministic lattice signatures. *TCHES*, 2018(3):21–43, 2018.

[GPP11] Jian Guo, Thomas Peyrin, and Axel Poschmann. The PHOTON family of lightweight hash functions. In Phillip Rogaway, editor, *CRYPTO*, volume 6841 of *Lecture Notes in Computer Science*, pages 222–239. Springer, 2011.

[GPPT15] Daniel Genkin, Lev Pachmanov, Itamar Pipman, and Eran Tromer. Stealing keys from PCs using a radio: Cheap electromagnetic attacks on windowed exponentiation. In Tim Güneysu and Helena Handschuh, editors, *CHES*, volume 9293 of *Lecture Notes in Computer Science*, pages 207–228. Springer, 2015.

[GPV08] Craig Gentry, Chris Peikert, and Vinod Vaikuntanathan. Trapdoors for hard lattices and new cryptographic constructions. In Cynthia Dwork, editor, *STOC*, pages 197–206. ACM, 2008.

[Gro96] Lov K. Grover. A fast quantum mechanical algorithm for database search. In Gary L. Miller, editor, *STOC*, pages 212–219. ACM, 1996.

[GSM15] Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. Cache template attacks: Automating attacks on inclusive last-level caches. In Jaeyeon Jung and Thorsten Holz, editors, *USENIX*, pages 897–912. USENIX Association, 2015.

[GST14] Daniel Genkin, Adi Shamir, and Eran Tromer. RSA key extraction via low-bandwidth acoustic cryptanalysis. In Juan A. Garay and Rosario Gennaro, editors, *CRYPTO (1)*, volume 8616 of *Lecture Notes in Computer Science*, pages 444–461. Springer, 2014.

[HBG$^+$18] Andreas Hülsing, Denis Butin, Stefan-Lukas Gazdag, Joost Rijneveld, and Aziz Mohaisen. XMSS: eXtended Merkle Signature Scheme. RFC 8391, 2018. URL: `https://tools.ietf.org/html/rfc8391`.

[HGS99] Chris Hall, Ian Goldberg, and Bruce Schneier. Reaction attacks against several public-key cryptosystems. In Vijay Varadharajan and Yi Mu, editors, *ICICS*, volume 1726 of *Lecture Notes in Computer Science*, pages 2–12. Springer, 1999.

[HLS18] Andreas Hülsing, Tanja Lange, and Kit Smeets. Rounded Gaussians – fast and secure constant-time sampling for lattice-based crypto. In Michel Abdalla and Ricardo Dahab, editors, *PKC (2)*, volume 10770 of *Lecture Notes in Computer Science*, pages 728–757. Springer, 2018.

[HM96] Shai Halevi and Silvio Micali. Practical and provably-secure commitment schemes from collision-free hashing. In Neal Koblitz, editor, *CRYPTO*, volume 1109 of *Lecture Notes in Computer Science*, pages 201–215. Springer, 1996.

[HNP$^+$03] Nick Howgrave-Graham, Phong Q. Nguyễn, David Pointcheval, John Proos, Joseph H. Silverman, Ari Singer, and William Whyte. The impact of decryption failures on the security of NTRU encryption. In Dan Boneh, editor, *CRYPTO*, volume 2729 of *Lecture Notes in Computer Science*, pages 226–246. Springer, 2003.

[HPS98] Jeffrey Hoffstein, Jill Pipher, and Joseph H. Silverman. NTRU: A ring-based public key cryptosystem. In Joe Buhler, editor, *ANTS*, volume 1423 of *Lecture Notes in Computer Science*, pages 267–288. Springer, 1998.

[HPS$^+$14] Jeffrey Hoffstein, Jill Pipher, John M. Schanck, Joseph H. Silverman, and William Whyte. Practical Signatures from the Partial Fourier Recovery Problem. In Ioana Boureanu, Philippe Owesarski, and Serge Vaudenay, editors, *ACNS*, volume 8479 of *Lecture Notes in Computer Science*, pages 476–493. Springer, 2014.

[HRB13] Andreas Hülsing, Lea Rausch, and Johannes A. Buchmann. Optimal parameters for XMSS^{MT}. In Alfredo Cuzzocrea, Christian Kittl, Dimitris E. Simos, Edgar R. Weippl, and Lida Xu, editors, *Security Engineering and Intelligence Informatics*, volume 8128 of *Lecture Notes in Computer Science*, pages 194–208. Springer, 2013.

[HRS16] Andreas Hülsing, Joost Rijneveld, and Fang Song. Mitigating multi-target attacks in hash-based signatures. In Chen-Mou Cheng, Kai-Min Chung, Giuseppe Persiano, and Bo-Yin Yang, editors, *PKC(1)*, volume 9614 of *Lecture Notes in Computer Science*, pages 387–416. Springer, 2016.

[HRSS17] Andreas Hülsing, Joost Rijneveld, John M. Schanck, and Peter Schwabe. High-speed key encapsulation from NTRU. In Wieland Fischer and Naofumi Homma, editors, *CHES*, volume 10529 of *Lecture Notes in Computer Science*, pages 232–252. Springer, 2017.

[HS00] Jeffrey Hoffstein and Joseph H. Silverman. Reaction attacks against the NTRU public key cryptosystem, 2000. NTRU Cryptosystems Technical Report 015, version 2. https://web.archive.org/web/20000914041434/http://www.ntru.com:80/NTRUFTPDocsFolder/NTRUTech015.pdf.

[HS09] Nadia Heninger and Hovav Shacham. Reconstructing RSA private keys from random key bits. In Shai Halevi, editor, *CRYPTO*, volume 5677 of *Lecture Notes in Computer Science*, pages 1–17. Springer, 2009.

[Hül13] Andreas Hülsing. W-OTS+ – shorter signatures for hash-based signature schemes. In Amr Youssef, Abderrahmane Nitaj, and Aboul Ella Hassanien, editors, *AFRICACRYPT*, volume 7918 of *Lecture Notes in Computer Science*, pages 173–188. Springer, 2013.

[IGI$^+$16] Mehmet Sinan Inci, Berk Gülmezoglu, Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. Cache attacks enable bulk key recovery on the cloud. In Benedikt Gierlichs and Axel Y. Poschmann, editors, *CHES*, volume 9813 of *Lecture Notes in Computer Science*, pages 368–388. Springer, 2016.

[Int12] Intel Corporation. *Intel 64 and IA-32 Architectures Optimization Reference Manual*, April 2012.

[JY00] Marc Joye and Sung-Ming Yen. Optimal left-to-right binary signed-digit recoding. *IEEE Trans. Computers*, 49(7):740–748, 2000.

[JZC$^+$18] Haodong Jiang, Zhenfeng Zhang, Long Chen, Hong Wang, and Zhi Ma. IND-CCA-secure key encapsulation mechanism in the quantum random oracle model, revisited. In Hovav Shacham and Alexandra Boldyreva, editors, *CRYPTO (3)*, volume 10993 of *Lecture Notes in Computer Science*, pages 96–125. Springer, 2018.

[KDK$^+$14] Yoongu Kim, Ross Daly, Jeremie Kim, Chris Fallin, Ji-Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu. Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors. In *ISCA*, pages 361–372. IEEE Computer Society, 2014.

[KLS18] Eike Kiltz, Vadim Lyubashevsky, and Christian Schaffner. A Concrete Treatment of Fiat-Shamir Signatures in the Quantum Random-Oracle Model. In Jesper Buus Nielsen and Vincent Rijmen, editors, *EUROCRYPT (3)*, volume 10822 of *Lecture Notes in Computer Science*, pages 552–586. Springer, 2018.

[KY76] Donald E Knuth and Andrew C Yao. The complexity of nonuniform random number generation. *Algorithms and complexity: new directions and recent results*, pages 357–428, 1976.

[Lam79] Leslie Lamport. Constructing digital signatures from a one way function. Technical Report SRI-CSL-98, SRI International Computer Science Laboratory, 1979.

[Lan16] Adam Langley. CECPQ1 results, November 2016. `https://www.imperialviolet.org/2016/11/28/cecpq1.html`.

[Lan18] Adam Langley. SECPQ2, December 2018. `https://www.imperialviolet.org/2018/12/12/cecpq2.html`.

[LDK$^+$17] Vadim Lyubashevsky, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Peter Schwabe, Gregor Seiler, and Damien Stehlé. CRYSTALS-Dilithium. Submission to the NIST Post-Quantum Cryptography Standardization [NIS], 2017. `https://pq-crystals.org/dilithium`.

[L'E11] Pierre L'Ecuyer. Non-uniform random variate generations. In *International Encyclopedia of Statistical Science*, pages 991–995. Springer, 2011.

[LF06] Éric Levieil and Pierre-Alain Fouque. An improved LPN algorithm. In Roberto De Prisco and Moti Yung, editors, *SCN*, volume 4116 of *Lecture Notes in Computer Science*, pages 348–359. Springer, 2006.

[LLL82] Arjen K. Lenstra, Hendrik Jr. W. Lenstra, and László Lovász. Factoring polynomials with rational coefficients. *Mathematische Annalen*, 261(4):515–534, 1982.

[LM95] Frank T Leighton and Silvio Micali. Large provably fast and secure digital signature schemes based on secure hash functions, July 11 1995. US Patent 5,432,852.

[LPR10] Vadim Lyubashevsky, Chris Peikert, and Oded Regev. On ideal lattices and learning with errors over rings. In Henri Gilbert, editor, *EUROCRYPT*, volume 6110 of *Lecture Notes in Computer Science*, pages 1–23. Springer, 2010.

[LPR13] Vadim Lyubashevsky, Chris Peikert, and Oded Regev. A Toolkit for Ring-LWE Cryptography. In Thomas Johansson and Phong Q. Nguyễn, editors, *EUROCRYPT*, volume 7881 of *Lecture Notes in Computer Science*, pages 35–54. Springer, 2013.

[LYG$^+$15] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee. Last-level cache side-channel attacks are practical. In *IEEE Symposium on Security and Privacy*, pages 605–622. IEEE Computer Society, 2015.

[Lyu09] Vadim Lyubashevsky. Fiat-Shamir with Aborts: Applications to Lattice and Factoring-Based Signatures. In Mitsuru Matsui, editor, *ASIACRYPT*, volume 5912 of *Lecture Notes in Computer Science*, pages 598–616. Springer, 2009.

[MCF19] David McGrew, Michael Curcio, and Scott Fluhrer. Leighton-Micali hash-based signatures. RFC 8554, 2019. URL: `https://tools.ietf.org/html/rfc8554`.

[ME78] J.J. Moder and S.E. Elmaghraby. *Handbook of operations research: models and applications*. Van Nostrand Reinhold Co., 1978.

[Mer89] Ralph C. Merkle. A certified digital signature. In Gilles Brassard, editor, *CRYPTO*, volume 435 of *Lecture Notes in Computer Science*, pages 218–238. Springer, 1989.

[MKF$^+$16] David A. McGrew, Panos Kampanakis, Scott R. Fluhrer, Stefan-Lukas Gazdag, Denis Butin, and Johannes A. Buchmann. State management for hash-based signatures. In Lidong Chen, David A. McGrew, and Chris J. Mitchell, editors, *SSR*, volume 10074 of *Lecture Notes in Computer Science*, pages 244–260. Springer, 2016.

[MSJ13] Hani Mimi, Azman Samsudin, and Shahram Jahani. Elliptic curve point multiplication algorithm using precomputation. *WSEAS Transactions on Computers*, 12, 2013.

[MvOV96] Alfred J. Menezes, Paul C. van Oorschot, and Scott A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, October 1996.

[MW17] Daniele Micciancio and Michael Walter. Gaussian sampling over the integers: Efficient, generic, constant-time. In Jonathan Katz and Hovav Shacham, editors, *CRYPTO (2)*, volume 10402 of *Lecture Notes in Computer Science*, pages 455–485. Springer, 2017.

[Nat15] National Institute of Standards and Technology (NIST). Secure hash standard (SHS). FIPS PUBS 180-4, 2015. `doi:10.6028/NIST.FIPS.180-4`.

[NC10] Michael A. Nielsen and Isaac L. Chuang. *Quantum Computation and Quantum Information*. Cambridge University Press, Cambridge, 10th anniversary edition, 2010.

[NIS] NIST. Post-Quantum Cryptography Standardization. `https://csrc.nist.gov/projects/post-quantum-cryptography/post-quantum-cryptography-standardization`.

[NIS14] NIST. SHA-3 standard: Permutation-based hash and extendable-output functions. Draft FIPS 202, 2014. Available at `http://csrc.nist.gov/publications/drafts/fips-202/fips_202_draft.pdf`.

[NIS16] NIST. Announcing request for nominations for public-key post-quantum cryptographic algorithms, 2016. `https://csrc.nist.gov/news/2016/public-key-post-quantum-cryptographic-algorithms`.

[NIS19] NIST. Request for Public Comments on Stateful Hash-Based Signatures (HBS). `https://www.nist.gov/news-events/news/2019/02/request-public-comments-stateful-hash-based-signatures-hbs`, 2019.

[O'C99] Luke O'Connor. *An Analysis of Exponentiation Based on Formal Languages*, pages 375–388. Springer Berlin Heidelberg, Berlin, Heidelberg, 1999.

[OST06] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache attacks and countermeasures: The case of AES. In David Pointcheval, editor, *CT-RSA*, volume 3860 of *Lecture Notes in Computer Science*, pages 1–20. Springer, 2006.

[PDG14] Thomas Pöppelmann, Léo Ducas, and Tim Güneysu. Enhanced lattice-based signatures on reconfigurable hardware. In Lejla Batina and Matthew Robshaw, editors, *CHES*, volume 8731 of *Lecture Notes in Computer Science*, pages 353–370. Springer, 2014.

[Pei10] Chris Peikert. An efficient and parallel Gaussian sampler for lattices. In Tal Rabin, editor, *CRYPTO*, volume 6223 of *Lecture Notes in Computer Science*, pages 80–97. Springer, 2010.

[Pei14] Chris Peikert. Lattice cryptography for the internet. In Michele Mosca, editor, *PQCrypto*, volume 8772 of *Lecture Notes in Computer Science*, pages 197–219. Springer, 2014.

[Per05] Colin Percival. Cache missing for fun and profit. In *BSDCan 2005*, Ottawa, CA, 2005.

[Pes16] Peter Pessl. Analyzing the shuffling side-channel countermeasure for lattice-based signatures. In Orr Dunkelman and Somitra Kumar Sanadhya, editors, *INDOCRYPT*, volume 10095 of *Lecture Notes in Computer Science*, pages 153–170, 2016.

[PG13] Thomas Pöppelmann and Tim Güneysu. Towards practical lattice-based public-key encryption on reconfigurable hardware. In Tanja Lange, Kristin E. Lauter, and Petr Lisonek, editors, *SAC*, volume 8282 of *Lecture Notes in Computer Science*, pages 68–85. Springer, 2013.

[PGY17] Peter Pessl, Leon Groot Bruinderink, and Yuval Yarom. To BLISS-B or not to be: Attacking strongSwan's implementation of post-quantum signatures. In Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *ACM CCS*, pages 1843–1855. ACM, 2017.

[Pie12] Krzysztof Pietrzak. Cryptography from learning parity with noise. In Mária Bieliková, Gerhard Friedrich, Georg Gottlob, Stefan Katzenbeisser, and György Turán, editors, *SOFSEM*, volume 7147 of *Lecture Notes in Computer Science*, pages 99–114. Springer, 2012.

[Por13] T. Pornin. Deterministic Usage of the Digital Signature Algorithm (DSA) and Elliptic Curve Digital Signature Algorithm (ECDSA). RFC 6979, 2013. `https://tools.ietf.org/html/rfc6979`.

[PPS12] Kenneth G. Paterson, Antigoni Polychroniadou, and Dale L. Sibborn. A coding-theoretic approach to recovering noisy RSA keys. In Xiaoyun Wang and Kazue Sako, editors, *ASIACRYPT*, volume 7658 of *Lecture Notes in Computer Science*, pages 386–403. Springer, 2012.

[Prond] GNU Project. GLPK (GNU Linear Programming Kit), n.d. `https://www.gnu.org/software/glpk/`.

[PSS$^+$18] Damian Poddebniak, Juraj Somorovsky, Sebastian Schinzel, Manfred Lochter, and Paul Rösler. Attacking deterministic signature schemes using fault attacks. In *EuroS&P*, pages 338–352. IEEE, 2018.

[Reg09] Oded Regev. On lattices, learning with errors, random linear codes, and cryptography. *J. ACM*, 56(6):34:1–34:40, 2009.

[Ren61] Alfred Renyi. On measures of entropy and information. In *Proceedings of the Fourth Berkeley Symposium on Mathematical Statistics and Probability*, volume 1, pages 547–561, Berkeley, Calif., 1961.

[Rog02] Phillip Rogaway. Authenticated-encryption with associated-data. In Vijayalakshmi Atluri, editor, *ACM CCS*, pages 98–107. ACM, 2002.

[Ros83] Sheldon Mark Ross. *Stochastic processes*. Probability and mathematical statistics. Wiley, 1983.

[RR02] Leonid Reyzin and Natan Reyzin. Better than biba: Short one-time signatures with fast signing and verifying. In Lynn Margaret Batten and Jennifer Seberry, editors, *ACISP*, volume 2384 of *Lecture Notes in Computer Science*, pages 144–153. Springer, 2002.

[RRVV14] Sujoy Sinha Roy, Oscar Reparaz, Frederik Vercauteren, and Ingrid Verbauwhede. Compact and side channel secure discrete Gaussian sampling. Cryptology ePrint Archive, `http://eprint.iacr.org/2014/591`, 2014.

[RVV13] Sujoy Sinha Roy, Frederik Vercauteren, and Ingrid Verbauwhede. High precision discrete gaussian sampling on FPGAs. In Tanja Lange, Kristin E. Lauter, and Petr Lisonek, editors, *SAC*, volume 8282 of *Lecture Notes in Computer Science*, pages 383–401. Springer, 2013.

[Saa17a] Markku-Juhani O. Saarinen. HILA5: Key encapsulation mechanism (KEM) and public key encryption algorithm. Submission to the NIST Post-Quantum Cryptography Standardization [NIS], 2017. `https://github.com/mjosaarinen/hila5`.

[Saa17b] Markku-Juhani O. Saarinen. HILA5: on reliability, reconciliation, and error correction for Ring-LWE encryption. In Carlisle Adams and Jan Camenisch, editors, *SAC*, volume 10719 of *Lecture Notes in Computer Science*, pages 192–212. Springer, 2017.

[Saa18] Markku-Juhani O. Saarinen. Arithmetic coding and blinding countermeasures for lattice signatures – engineering a side-channel resistant post-quantum signature scheme with compact signatures. *J. Cryptographic Engineering*, 8(1):71–84, 2018.

[SB18] Niels Samwel and Lejla Batina. Practical Fault Injection on Deterministic Signatures: The Case of EdDSA. In Antoine Joux, Abderrahmane Nitaj, and Tajjeeddine Rachidi, editors, *AFRICACRYPT*, volume 10831 of *Lecture Notes in Computer Science*, pages 306–321. Springer, 2018.

[SBB+18] Niels Samwel, Lejla Batina, Guido Bertoni, Joan Daemen, and Ruggero Susella. Breaking Ed25519 in WolfSSL. In Nigel P. Smart, editor, *CT-RSA*, volume 10808 of *Lecture Notes in Computer Science*, pages 1–20. Springer, 2018.

[Sho15] Victor Shoup. NTL: A library for doing number theory, 2015. `http://www.shoup.net/ntl/`.

[SHS16] Hermann Seuschek, Johann Heyszl, and Fabrizio De Santis. A Cautionary Note: Side-Channel Leakage Implications of Deterministic Signature Schemes. In Martin Palkovic, Giovanni Agosta, Alessandro Barenghi, Israel Koren, and Gerardo Pelosi, editors, *CS2@HiPEAC*, pages 7–12. ACM, 2016.

[str15] strongSwan. strongSwan 5.2.2 released, January 2015. `https://www.strongswan.org/blog/2015/01/05/strongswan-5.2.2-released.html`.

[SXY18] Tsunekazu Saito, Keita Xagawa, and Takashi Yamakawa. Tightly-secure key-encapsulation mechanism in the quantum random oracle model. In Jesper Buus Nielsen and Vincent Rijmen, editors, *EUROCRYPT (3)*, volume 10822 of *Lecture Notes in Computer Science*, pages 520–551. Springer, 2018.

[TU16] Ehsan Ebrahimi Targhi and Dominique Unruh. Post-quantum security of the Fujisaki-Okamoto and OAEP transforms. In Martin Hirt and Adam D. Smith, editors, *TCC (B2)*, volume 9986 of *Lecture Notes in Computer Science*, pages 192–216, 2016.

[Unr16a] Dominique Unruh. Collapse-binding quantum commitments without random oracles. In Jung Hee Cheon and Tsuyoshi Takagi, editors, *ASIACRYPT (2)*, volume 10032 of *Lecture Notes in Computer Science*, pages 166–195, 2016.

[Unr16b] Dominique Unruh. Computationally binding quantum commitments. In Marc Fischlin and Jean-Sébastien Coron, editors, *EUROCRYPT (2)*, volume 9666 of *Lecture Notes in Computer Science*, pages 497–527. Springer, 2016.

[Unr17] Dominique Unruh. Collapsing sponges: Post-quantum security of the sponge construction. Cryptology ePrint Archive, Report 2017/282, 2017. `https://eprint.iacr.org/2017/282`.

[vdPSY15] Joop van de Pol, Nigel P. Smart, and Yuval Yarom. Just a little bit more. In Kaisa Nyberg, editor, *CT-RSA*, volume 9048 of *Lecture Notes in Computer Science*, pages 3–21. Springer, 2015.

[VDvT02] Eric R. Verheul, Jeroen M. Doumen, and Henk C.A. van Tilborg. Sloppy Alice attacks! Adaptive chosen ciphertext attacks on the McEliece public-key cryptosystem. In *Information, Coding and Mathematics*, pages 99–119. Springer, 2002.

[Wal03] Colin D. Walter. Longer keys may facilitate side channel attacks. In Mitsuru Matsui and Robert J. Zuccherato, editors, *SAC*, volume 3006 of *Lecture Notes in Computer Science*, pages 42–57. Springer, 2003.

[Yar16] Yuval Yarom. Mastik: A micro-architectural side-channel toolkit. `http://cs.adelaide.edu.au/~yval/Mastik/Mastik.pdf`, September 2016.

[YB14] Yuval Yarom and Naomi Benger. Recovering OpenSSL ECDSA nonces using the FLUSH+RELOAD cache side-channel attack. Cryptology ePrint Archive, Report 2014/140, 2014. `https://eprint.iacr.org/2014/140`.

[YF14] Yuval Yarom and Katrina Falkner. FLUSH+RELOAD: A high resolution, low noise, L3 cache side-channel attack. In Kevin Fu and Jaeyeon Jung, editors, *USENIX*, pages 719–732. USENIX Association, 2014.

[YGH16] Yuval Yarom, Daniel Genkin, and Nadia Heninger. CacheBleed: A timing attack on OpenSSL constant time RSA. In Benedikt Gierlichs and Axel Y. Poschmann, editors, *CHES*, pages 346–367. Springer, 2016.

[Zha12] Mark Zhandry. How to construct quantum random functions. In *FOCS*, pages 679–687. IEEE Computer Society, 2012.

[Zha15] Mark Zhandry. A note on the quantum collision and set equality problems. *Quantum Information & Computation*, 15(7&8):557–567, 2015.

[ZJRR12] Yinqian Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. Cross-VM side channels and their use to extract private keys. In Ting Yu, George Danezis, and Virgil D. Gligor, editors, *ACM CCS*, pages 305–316. ACM, 2012.

[ZJRR14] Yinqian Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. Cross-tenant side-channel attacks in PaaS clouds. In Gail-Joon Ahn, Moti Yung, and Ninghui Li, editors, *ACM CCS*, pages 990–1003. ACM, 2014.

Summary

Attacking Post-Quantum Cryptography

Cryptography is a vital part of today's internet, protecting any data packet containing sensitive information (such as emails, passwords, banking details) from attackers. These attackers will try literally everything to break the used cryptography, so a natural question is how to evaluate the strength of a cryptographic scheme. This evaluation has several aspects in order to be meaningful: e.g. what is the exact security goal that the cryptographic scheme provides and what are the attacker's capabilities? This second question becomes very important when we look at the future: many big companies such as IBM and Google, as well as many universities and governments, are actively trying to build large quantum computers. These computers are able to solve the computationally hard mathematical problems that underlie the cryptography that we are using everywhere today. The area of post-quantum cryptography is dealing with this additional attacker capability by developing cryptography that is able to resist attacks from large quantum computers. Naturally, post-quantum cryptography also needs evaluation of possible attacks before it can be deployed on the internet. This thesis covers several attacks on post-quantum cryptography.

Part I: Model-mismatch attacks. The first part of this thesis focuses on model-mismatch attacks: what happens when (some of) the assumptions made in the security model for a cryptographic scheme become invalid? There are several reasons that such a model mismatch might occur. For example: the concept of quantum computing introduces a whole new computing paradigm, possibly invalidating the (previous) assumptions on the capabilities of the attacker, that now has a (large enough) quantum computer. On the other hand, model mismatch might also occur when cryptography is incorrectly used, possibly due to accidental events in applications or simply because the security model did not match the cryptographic scheme itself. The first part of this thesis covers several model-mismatch attacks and ways to prevent these.

Part II: Side-channel attacks. The second part of this thesis covers side-channel attacks. These attacks use physical information leakage of devices (such as (cache) memory access patterns or EM radiation) that perform cryptographic operations as a short-cut to break the cryptographic implementation. As post-quantum cryptography introduces new cryptographic primitives, it also introduces new algorithms and implementations. It is not always straightforward to achieve a secure implementation, but at the same time it is also not straightforward to perform such a side-channel attack. The second part of this thesis covers several side-channel attacks on post-quantum cryptography.

Curriculum Vitae

Leon Groot Bruinderink was born on November 17, 1990 in Wageningen, the Netherlands. In 2009 he graduated from Dorenweerd College in Doorwerth with a VWO degree. In the same year, he started studying Industrial and Applied Mathematics at the Technische Universiteit Eindhoven (TU/e). He completed the Bachelor program in 2013 and the Master program in 2016 with distinction (cum laude). During his study, he spent a trimester in the US as a visiting scholar at the University of Pennsylvania, under supervision of Nadia Heninger. His Master project was supervised by Tanja Lange (TU/e), Andreas Hülsing (TU/e) and Lodewijk Bonebakker (ING). The thesis was entitled "Towards Post-Quantum Bitcoin – Side-channel Analysis of Bimodal Lattice Signatures". The scientific paper based on this work was nominated for the Dutch Cyber Security best Research Paper (DCSRP) Award in 2017. During his studies, Leon was actively involved in organizing extracurricular activities at the Eindhoven Student Association.

In 2016, Leon started his PhD project in the Coding Theory and Cryptology group at Technische Universiteit Eindhoven, under supervision of Tanja Lange, Daniel J. Bernstein and Andreas Hülsing. This project was supported by the Commission of the European Communities through the Horizon 2020 program under project number ICT–645622 (PQCRYPTO). The present thesis contains the results of his work from 2016 to 2019. His scientific contributions range from theoretical works, e.g. modeling quantum attacks, to applied works, including side-channel attacks.