Peter Peßl

Side-Channel Attacks on Lattice-Based Cryptography and Multi-Processor Systems

DOCTORAL THESIS
to achieve the university degree of
Doktor der technischen Wissenschaften
submitted to
Graz University of Technology

Supervisor
Prof. Stefan Mangard

Assessors
Prof. Stefan Mangard
Assoc.Prof. Peter Schwabe

Institute for Applied Information Processing and Communications (IAIK)
Graz, October 2018

AFFIDAVIT

I declare that I have authored this thesis independently, that I have not used other than the declared sources/resources, and that I have explicitly indicated all material which has been quoted either literally or by content from the sources used. The text document uploaded to TUGRAZonline is identical to the present doctoral thesis.

Date

Signature

Abstract

Cryptographic algorithms offer black-box security, i.e., just observing their inputs and outputs does not reveal the key. Electronic devices that execute these algorithms, however, do not fulfill the definition of a black box. Their physical properties, e.g., power consumption or timing behavior, can reveal sensitive information processed by the device. So-called side-channel attacks exploit this fact. To provide comprehensive protection against them, a deep understanding of attack vectors is necessary. Exactly this understanding is challenged by the constant evolution of cryptography and advancements in device design. In this thesis, we address this problem by analyzing implementations of lattice-based cryptography and by presenting the first side-channel attack capable of bridging the gap in multi-CPU systems.

First, we focus on lattice-based cryptography, which has gained a lot of traction in recent years. It boasts security against quantum computing and is thus a promising replacement of currently used public-key primitives. Despite its proven practicality, however, the implementation-security aspect is still largely unexplored. We tackle this and present some of the first side-channel and fault attacks targeting lattice-based cryptography. Our first attack targets the BLISS lattice-based signature scheme and offers several improvements over previous proposals, such as the ability to target an improved BLISS variant. Second, we analyze and circumvent a countermeasure aimed at preventing the previous attack. And third, by mounting fault attacks on deterministic lattice signatures we show that some countermeasures can have adverse side effects. All these attacks make use of algorithmic and algebraic features typical to this family of schemes, thereby highlighting the importance of dedicated analysis.

After lattices, we target multi-processor systems, which are becoming ubiquitous especially in cloud environments. Previous microarchitectural exploits, such as cache attacks, were cross-core but not able to bridge the gap between multiple physical CPUs. We show how DRAM can be exploited as a side-channel source not bound to this restriction. After reverse-engineering undisclosed mapping functions, we craft attacks breaking many isolation boundaries including that of different physical CPUs. This makes them a threat especially in shared systems, such as multi-tenant cloud machines.

Acknowledgments

This PhD thesis would not have been possible without the help and support of many great people, whom in the following I would like to express my deepest gratitude.

First and foremost, I would like to thank my advisor Stefan Mangard for his invaluable guidance and his faith in me. He gave me the freedom and trust to do research in areas of my interest and to dive into new and exciting topics. Despite his sometimes tight schedule, I could also always reach out to him and ask for advice. I would also like to express my thanks to Peter Schwabe for agreeing to be my assessor and for giving great feedback on this thesis.

Thanks also to all colleagues and collaborators I had (and still have) the pleasure of working with throughout my years at IAIK. I am very grateful for all the constructive discussions and successful joint work with my co-authors Robert Primas, Leon Groot Bruinderink, Yuval Yarom, Daniel Gruß, Michael Schwarz, and Clémentine Maurice. Special thanks also go to Michael Hutter, without whom I would have never started a PhD. Besides, I need to thank all my colleagues for, among many other things, interesting conversations over coffee, for joining my skiing trips, and for enduring my visits to their offices.

Last, but most certainly not least, I wish to express my sincere gratitude to my friends and my family. My parents Blasius and Cäcilia as well as my brother Martin always supported me and had an open ear. And finally, thanks to Nina and Happy, for showing me new perspectives in life and brightening my days.

Peter

Table of Contents

List of Tables

List of Figures

List of Algorithms

1

Introduction

Implementation attacks are a significant threat to electronic devices handling secrets such as cryptographic keys or sensitive user input. These attacks do not treat a device as a black box performing unobservable computations before returning a result, but instead, consider it in its entirety and use all information that the device offers (intentionally or unintentionally). That is, they exploit physical side channels, e.g., power consumption, electromagnetic emanations, and timing behavior, or manipulate the device to inject computational faults using, for instance, glitches on inputs or targeted bombardment with EM pulses. Initially, most such attacks required either the possession of or at least proximity to the attacked device. While this can often be achieved, in many cases even legitimately, it is still a hurdle. This hurdle, however, can be overcome by many timing-based remote attacks. Hence, devices ranging from servers of cloud providers all the way down to one-time-use electronic tickets are potential targets.

The danger of implementation attacks is well known and its discovery not recent. Seminal research uncovering possibilities to attack cryptographic implementations is more than two decades old [Koc96; BDL97; KJJ99]. Cryptography is the main tool for securing communication, thus recovering the key used to protect the sensitive information is a prime attack goal. Still, the actual secret, such as the plaintext, can also be targeted. Earliest reports of such (more general) side-channel attacks even date back to World War II [NSA72]. This might sound like enough time to fix this problem, but in reality, there are numerous challenges and the problem is far from being solved.

One reason is that the landscape of cryptography itself is in constant evolution. The first papers presenting side-channel attacks on symmetric cryptography targeted the now long obsolete Data Encryption Standard (DES) [KJJ99]. Since then, cryptographers have shown their love for competitions. The AES competition [NISa] looked for a successor to DES, eSTREAM [ECR] for new stream ciphers, the SHA-3 competition [NISd] for new hash functions, the still ongoing CAESAR contest [Ber] for ciphers offering authenticated encryption, and very recently NIST announced a call for lightweight ciphers [NISb]. Each such competition brings a swath of new algorithms and thus a constant stream of challenges for secure implementation.

These challenges get amplified when algorithms break new ground, which is the case for so-called post-quantum cryptography. This term refers to schemes that are secure even if large-scale quantum computers exist. Currently used public-key schemes, such as RSA and ECC, do not have this property and succumb to Shor's algorithm [Sho99]. They are thus up for replacement; the search for alternatives has gained high traction throughout the recent years. Fittingly, NIST currently runs a post-quantum cryptography standardization process [NISc], i.e., another competition (although NIST is reluctant in calling it that [Moo17]).

There exist several families of post-quantum-secure schemes, for a comprehensive overview we refer to Bernstein and Lange [BL17]. One family in particular, namely lattice-based cryptography, offers good security guarantees, very compelling performance, and has already seen first real-world tests [Bra16; Lan16]. The implementation security aspect, however, is pretty much unexplored. Unprotected implementations will undoubtedly be vulnerable against already known attack, but the mathematical structure underlying lattice-based primitives might open up many new exploitation paths and could be utilized for efficient side-channel attacks and subsequent analytic key recovery. If this is truly the case is an open question and thus, to say it in quantum speak and to invoke a famous cat, neither true nor false until analyzed.

However, not only (cryptographic) algorithms are changing, the devices we run them on are evolving as well. One recent trend is to outsource computations to someone else's machines, i.e., to the so-called cloud. There, multiple tenants are often co-located on a single machine; this makes enforcing isolation boundaries crucial. Microarchitectural attacks, such as cache attacks that exploit timing differences stemming from access patterns to the CPU cache [Tsu+03; Ber05; YF14], can cross some of the boundaries by exploiting hardware sharing. However, while they do allow cross-core attacks due to shared caches, they cannot bridge the gap between physical CPUs in multi-processor systems. Exactly such machines are becoming increasingly ubiquitous in the cloud. Thus, cache attacks could be trivially mitigated by simply assigning dedicated physical CPUs to each tenant. However, further analysis needs to determine if this offers protection against the full range of microarchitectural attacks.

1.1 Contribution and Outline

In the course of this thesis, we make progress in both these directions. That is, we show (1) efficient side-channel exploitation techniques for lattice-based cryptography and (2) a new microarchitectural side channel allowing cross-CPU attacks. This leads to the following outline of this thesis.

Chapter 2 provides a brief introduction to side-channel attacks, including power analysis and fault attacks as well as microarchitectural exploits such as cache attacks and the Rowhammer bug.

Side-Channel Attacks on Lattice-Based Cryptography

Chapter 3 describes the need for post-quantum cryptography in more depth and then goes on to explain one category of algorithms, namely lattice-based cryptography, in more detail. Apart from theoretical groundwork, we recall a concrete scheme with BLISS, discuss techniques for efficient implementation, and finally deal with previous and concurrent works on implementation security.

Chapter 4 describes a new side-channel attack on BLISS. Our method does not have restrictions of an earlier attack, as we can also target the improved BLISS-B scheme and perform the attack in a more realistic scenario. We achieve this by combining multiple techniques such as parity learning, statistical methods, integer programming, and lattice-basis reduction. This combination allows us to recover the key from the real-world BLISS-B implementation included in the strongSwan IPsec-based VPN suite.

Chapter 5 analyzes one potential mitigation technique against the attack discussed in Chapter 4. Concretely, shuffling was proposed as a protection technique for the Gaussian sampling component, which is the primary target of the described attacks. We analyze two concrete variants of this countermeasure and show that the simpler one cannot significantly improve security. The second one can also be circumvented, albeit at the cost of requiring significantly more side-channel measurements, which shows that this version is somewhat effective.

Chapter 6 steps away from BLISS and has a look at the more recent alternatives qTESLA and Dilithium, both of which have been submitted to the NIST call. The proposals already have implementation security in mind. They thus refrain from using Gaussian samplers and are deterministic, with the latter aiming to protect against nonce reuse. We show that this determinism opens the gates for differential fault attacks, where a single injected fault can already lead to full key recovery. We analyze the effects of the used abortion technique and demonstrate that the fault position has a high influence on the success probability.

Side-Channel Attacks on Multi-Processor Systems

Chapter 7 identifies DRAM, i.e., main memory of PCs, as a powerful side-channel source that is even shared across physical CPUs. DRAM contains so-called row buffers, which are required electrical components that exhibit some cache-like behavior. Namely, read and write timings depend on the previously accessed addresses. Before exploiting this, however, one needs to know how physical addresses are mapped to the used row buffer. This mapping is undisclosed, which is why we present two methods to reverse engineer it. We also demonstrate that the reverse-engineered functions improve previous microarchitectural attacks such as Rowhammer.

Chapter 8 shows how the timing differences can be used to build high-speed covert channels and mount low-noise side-channel attacks capable of crossing software (cross-VM) and physical (cross-CPU) boundaries. We dub our attack DRAMA (for DRAM Addressing), and demonstrate its capabilities by transmitting up to 2 Mbps across in our cross-core cross-CPU covert channel and by spying on keystroke timings using our side-channel attack.

Chapter 9 finally concludes this thesis.

2

Side-Channel Attacks

The idea of using side channels, i.e., information inadvertently leaked over a channel not intended for communication, to recover some secret is not particularly new. When talking to other people, we not only listen to their words but also instinctively interpret subtle and unintentional signals, such as the inability to look into one's eyes or sweating. From this, we infer whether someone is lying.

This type of "side-channel attack" obviously belongs to the realm of psychology, but attacks on electronic cryptographic devices also have a long and rich history. Already in 1943, a researcher at a Bell laboratory noticed that their then-used encryption machines send out parasitic electric signals which, when measured with an oscilloscope, can be used to recover the plaintext. This and other similar observations lead to the introduction of stringent shielding requirements for military encryption devices [NSA72].

The current understanding of the term side-channel attack is coined by the works of Paul Kocher. He showed that information such as timing [Koc96] or power consumption [KJJ99] of a cryptographic device can be used to recover the key. Fault injection has also proved to be an efficient attack technique. We discuss all these techniques in Section 2.1.

More recently, it was shown that microarchitectural optimizations of CPUs can introduce side-channel vulnerabilities. This opens up new attack vectors. Previous implementation attacks typically required physical access and thus targeted mostly embedded and smaller devices, such as smart cards. Microarchitectural attacks target larger systems, such as servers, and allow fully software-based exploitation. Thus, physical access is not needed anymore and remote attacks are a real possibility. Cache attacks, where timing differences between accesses to cache and RAM are exploited, are the most prominent example of a microarchitectural attack. Additionally, the Rowhammer bug demonstrates that aggressive optimization can even lead to exploitable faults. We give an overview of such microarchitectural attacks in Section 2.2.

2.1 Taxonomy and Overview

Implementation attacks can be grouped into two main categories. A passive adversary records and exploits available side-channel leakage, but does not disturb the computation. In contrast, an active attacker induces some error into the computation and then uses the faulty outcome to derive some secret.

2.1.1 Passive Attacks

Apart from the already mentioned timing and power leakage, other already exploited sources of side-channel information are, e.g., electromagnetic emanation [GMO01; QS01] (ranging from highly localized on-chip measurements to antennas placed several meters from the device [GS15]), acoustic signals [GST14], and heat [HS13]. The recording and required pre-processing of measurements (traces) can vastly differ for all these side-channel sources. The next steps in an attack, however, often use the same statistical techniques regardless of the underlying side-channel source.

Any passive attack exploits the fact that the side-channel information, such as the power consumption, depends on the data processed by the device. The side-channel information not only depends on the values of intermediate variables in an algorithm but also on the instruction being executed. We now describe two more concrete exploitation techniques, for a much more thorough explanation of passive side-channel attacks we refer to Mangard et al. [MOP07].

Differential Power Analysis. In a Differential Power Analysis (DPA) [KJJ99], one executes the same cryptographic operation, e.g., an encryption, many times. Each invocation uses the same secret key and some known but varying input or output, such as the ciphertext. One also records the instantaneous power consumption of the device for all executed operations. The key is then recovered by analyzing the difference in power consumption across the different executions. This coins the term Differential Power Analysis.

Trace analysis and key recovery involve several steps. First, one guesses a small part of the key (a subkey). Second, an intermediate is picked that depends both on the subkey and a known but varying input; the value of this intermediate is predicted for each possible value of the subkey and each measurement. Third, the power consumption caused by the intermediate is predicted. For this prediction, one typically uses power models such as the Hamming-weight model, which counts the bits set to one. Fourth and finally, the predicted power consumption is compared with the measured one using statistical methods; a popular tool is the Pearson correlation coefficient. The key candidate achieving the best match is then picked.

DPA requires little assumptions on the device and the exact implementation of the cryptographic primitive. The same power model can often be used for a large range of very different devices. An exact description of the power characteristics is thus not needed. What's more, DPA does not require knowing how and when exactly the targeted intermediate is manipulated, as the analysis is simply run for all points in the trace. However, DPA cannot be applied when the prerequisites are not met, e.g., if no fixed key is involved or if only a single measurement is possible. This prevents attacks on schemes which frequently change keys, but also on subroutines involving only ephemeral secrets. The latter is an integral part of many asymmetric primitives, thus leaving large parts unexploitable by DPA.

Simple Power Analysis and Template Attacks. The counterpart to DPA is commonly dubbed Simple Power Analysis (SPA). According to Kocher [KJJ99], it involves "a direct interpretation of power consumption measurements". This is not a tight definition. Techniques ranging from truly simple visual inspection of traces to much more involved template attacks are grouped into this category. The first allows recovery of chosen code paths, whereas the latter is seen as the (information theoretically) optimal attack.

In a template attack [CRR02], one first profiles the power consumption characteristic of a device. That is, one determines the typical consumption profiles for, e.g., different executed instructions or possible data values. This requires that the attacker can query the attacked device, or at least a very similar one, with known inputs, including the key. The computed profiles are called templates, often involve more than a single sample in the trace (Points of Interest POI), and are typically modeled as a multivariate Gaussian distribution. In the following template matching phase, the measured power consumption of the device-under-attack is then compared with the templates. One can then, e.g., assign a probability to each possibly processed value.

The term SPA is sometimes used synonymously with "single-trace attack", but please note this is not necessarily the case. One can use multiple traces in a template attack by using, e.g., Bayes theorem. Alternatively, one can also combine the outcome of many traces using algebraic methods, which is in fact done in Chapter 4 and Chapter 5.

2.1.2 Active Attacks

In contrast to a passive attack, an active adversary directly influences the device by injecting a computational fault. Such faults can be induced by, for instance, inserting glitches into the supply lines or external clock signals, electromagnetic pulses, or lasers [Bar+06].

Differential fault attacks are one powerful example of fault exploitation. There, one lets the device compute the same operation twice, but injects a fault during one of the invocations. The difference in the outcome can then be used to recover the key (or parts thereof). The seminal "Bellcore" attack [BDL97] demonstrated that this method allows recovering an RSA key using a single fault injection. Symmetric cryptography can also succumb to differential faults, as shown by Piret and Quisquater [PQ03].

Many cryptographic protocols prohibit running a primitive twice with the same input. This mitigates differential fault attacks, but other classes of attacks are still applicable. Statistical fault attacks [Fuh+13; Dob+16; Dob+18b] require more than a single fault but are unhindered by most algorithm-level fault countermeasures [Dob+18a].

2.2 Microarchitectural Side Channels

The above side-channel and fault attacks require possession of (or at least proximity to) the targeted device. Microarchitectural side-channels attacks are usually not bound to this restriction. They are made possible by the many performance optimizations such as caching, branch prediction, and out-of-order execution found in many modern CPUs. These optimizations can lead to data-dependent timing differences, which can be measured in software. What's more, the attacks allow to cross isolation boundaries and to attack, e.g., co-located tenants on a shared cloud server and PCs with sandboxed JavaScript executed by a browser. Recently, the microarchitectural attacks dubbed Spectre [Koc+18] and Meltdown [Lip+18] even brought widespread media attention to this topic.

We now describe two microarchitectural side-channels relevant for this work, namely cache attacks and the Rowhammer bug. For an in-depth survey of such attacks, see Ge et al. [Ge+18].

2.2.1 Cache Attacks

To bridge the speed gap between the faster processor and the slower memory, modern processor architectures employ multiple levels of *caches*, which store data that the processor predicts a program might use in the future. While the cache does not change the logical behavior of programs, it does affect their execution time. For the past 15 years, it has been known that timing variations due to the cache state can leak secret information about the execution of the program [Tsu+03; Ber05; OST06]. Over the years, many attacks that exploit the cache state have been designed, we now give one concrete technique.

The Flush+Reload Attack. The Flush+Reload attack [YF14] exploits read-only memory sharing, which is commonly used for sharing library code in modern operating systems. The attack consists of two phases. In the *flush* phase, the attacker evicts the contents of a monitored address from the cache. On Intel processors, this is typically achieved using the `clflush` instruction. The attacker then waits a bit before performing the *reload* phase of the attack. In the reload phase, the attacker reads the contents of the monitored memory address, while measuring the time it takes to perform the read. If the victim has accessed the monitored memory between the flush and the reload phases, the contents of the address will be cached and the attacker's read will be fast. Otherwise, the memory address will not be in the cache and the read will be slow.

By repeatedly interleaving the flush and the reload phases of multiple locations, the attacker can create a trace of the victim's uses of the monitored locations over time. When the victim access patterns depend on secret data, the attacker can use the trace to recover the data. Flush+Reload has been used to attack RSA [YF14], AES [Ira+14], ECDSA [YB14; Ben+14; PSY15], as well as non-cryptographic software [Zha+14; Ore+15].

Side-Channel Amplification. Because Flush+Reload only monitors victim accesses between the flush and the reload phases, accesses that occur during these phases may be missed, resulting in false negatives. The timing of victim accesses is mostly independent of the attacker's activity. Consequently, increasing the wait between these phases reduces the probability of false negatives, albeit at the cost of reduced temporal resolution. To mitigate the effects of the reduced temporal resolution, Allan et al. [All+16] suggest slowing down the victim. They demonstrate that by repeatedly evicting frequently-used code from the cache, they are able to slow programs down by a factor of up to 150. The combination of Flush+Reload and the Allen et al. attack has been used for attacks on ECDSA [All+16; PB17] and DSA [PBY16]

2.2.2 Rowhammer

The so-called Rowhammer bug [Kim+14; Hua+12; Par+14] crosses the gap between otherwise passive microarchitectural side-channels and active fault attacks. It allows corrupting data in main memory (RAM) using only software and thus can be seen as a remote fault attack.

It is caused by increasing the DRAM density, which has led to physically smaller DRAM cells that can thus store smaller charges. As a result, the cells have a lower noise margin, and the level of parasitic electrical interaction is potentially higher. This can be used to corrupt data, not in DRAM rows that are directly accessed, but rather in adjacent ones. When performing random memory accesses, the probability for such faults is virtually zero. However, it drastically increases when performing accesses in a certain pattern. Namely, flips can be caused by frequent activation (*hammering*) of adjacent rows. As data needs to be served from DRAM and not the cache, an attack needs to either flush data from the cache using the `clflush` instruction in native environments [Kim+14] or use cache eviction in other more restrictive environments, e.g., JavaScript [GMM16].

Seaborn [Sea15a] implemented two attacks that exploit the Rowhammer bug, showing the severity of faulting single bits for security. The first exploit is a kernel privilege escalation on a Linux system, caused by a bit flip in a page table entry. The second one is an escape of the Native Client sandbox caused by a bit flip in an instruction sequence for indirect jumps.

Part I

Side-Channel Attacks on Lattice-Based Cryptography

In this first part of the thesis, we present some of the first side-channel attacks and fault attacks targeting implementations of lattice-based cryptography. After covering the basics of lattices in Chapter 3, we present an attack on the BLISS-B signature scheme in Chapter 4, analyze a proposed countermeasure in Chapter 5, and perform a fault attack on the more recent Dilithium scheme in Chapter 6.

Publications and Contribution

This part is based on the following publications.

- Peter Pessl, Leon Groot Bruinderink, and Yuval Yarom. "To BLISS-B or not to be: Attacking strongSwan's Implementation of Post-Quantum Signatures." In: *CCS*. ACM, 2017, pp. 1843–1855. [PGY17]

  is the main basis for Chapter 4. The reported improvements to the attack on shuffling appear in Chapter 5.
  Contribution: I am the first author and contributed the main ideas and corresponding attack code. Leon Groot Bruinderink developed the integer-programming twos-recovery method; the practical attack as well as the simulated experiments were joint work. Yuval Yarom performed all cache measurements.

- Peter Pessl and Stefan Mangard. "Enhancing Side-Channel Analysis of Binary-Field Multiplication with Bit Reliability." In: *CT-RSA*. vol. 9610. LNCS. Springer, 2016, pp. 255–270. [PM16]

  originally targeted symmetric cryptography, but was later re-used in context of lattice-based cryptography. The relevant parts appear in Chapter 4.
  Contribution: I am the main author and provided all technical contributions.

- Peter Pessl. "Analyzing the Shuffling Side-Channel Countermeasure for Lattice-Based Signatures." In: *INDOCRYPT*. vol. 10095. LNCS. 2016, pp. 153–170. [Pes16]

  is the basis for Chapter 5.
  Contribution: I am the sole author.

- Leon Groot Bruinderink and Peter Pessl. "Differential Fault Attacks on Deterministic Lattice Signatures." In: *TCHES* 2018.3 (2018), pp. 21–43. [GP18]

  is used in Chapter 6.
  Contribution: This is joint work with Leon Groot Bruinderink, who contributed the initial idea and developed the partial reuse scenario, which is why this latter part is not included in this thesis. I implemented the other attack scenarios, developed the modified signing algorithm, performed the practical evaluation, and wrote the discussion on countermeasures.

3

Lattice-Based Cryptography
An Overview

Quantum computers are a serious threat to a majority of currently in-use public-key cryptosystems which rely on the difficulty of factoring large integers (RSA) and finding discrete logarithms (DH, ECC). Shor's algorithm [Sho99], however, can solve both these problems in polynomial time on such a quantum device.

It is uncertain when large-enough quantum computers will see the light of day, or even if they will do it at all. An estimate made in 2014 [Mar14] states that quantum computers able to factor currently-used RSA moduli could be available as early as 2030. Other predictions are less optimistic [Sha16]. Still, there is steady and undeniable progress as, e.g., shown by the recent unveiling of a 72-qubit machine [Kel18] and their beginning public accessibility [Kni17].

This outlook causes serious concerns and has, for instance, already led to official recommendations from government bodies. The NSA [NSA16; Sch15] recommends to skip ECC if it is not already adopted, and instead wait for suitable quantum resistant algorithms. The search for such quantum-secure alternatives is already in full swing and draws a lot of attention. This is demonstrated by the high interest in the currently ongoing NIST Post-Quantum Cryptography standardization process[1] [NISc]. At the late-2017 deadline, NIST received 69 proper submissions. This trumps the submission count for previous cryptographic competitions, such as for AES and SHA-3, by far. Apart from these mostly academic efforts, modern post-quantum cryptography has also already seen (limited) real-world evaluation, e.g., Google experimented with the NewHope [Alk+16] key-exchange in their Chrome browser [Bra16; Lan16].

---

[1] NIST repeatedly stated that this process is not supposed to be a competition [Moo17].

Apart from the black-box security of all these new proposals[2], efficiency both in terms of computational complexity and communication overhead (key and ciphertext sizes) are another important aspect. In this regard, schemes based on the hardness of certain lattice problems reign supreme and offer a favorable trade-off. Maybe for this very reason, they form the largest group in terms of submissions to the NIST call, outnumbering other contenders such as code-based, hash-based, supersingular isogeny, or multivariate cryptography.

Keys and ciphertexts for lattice-based primitives are somewhat larger than current RSA moduli, but still acceptably so. Regarding runtime, lattice-based schemes can even outperform RSA and ECC. This is clearly shown by an ever-growing body of work targeting their efficient implementation. These implementations target a wide set of different platforms, ranging from desktop PCs with vector instructions (e.g., the optimized reference implementations of Kyber and Dilithium [Ava+17; Lyu+17]) over microcontrollers [OPG14; Liu+15b; POG15] and FPGAs [Roy+14b; PDG14; How+18] to re-purposed RSA/ECC processors available on contemporary smart cards [Alb+18]. Please note that this list is by no means exhaustive.

In contrast to all these advancements, the implementation security aspect is just now starting to gain some traction and was almost entirely unexplored when the work on this thesis started. The structure and used implementation techniques for lattice-based cryptosystems differ drastically from that of RSA/ECC-based ones. Thus, their susceptibility to this form of attack has to be analyzed. New algebraic constructs, efficient algorithms, and implementation techniques can open up a swath of new vulnerabilities. Also, standard countermeasures used for ECC do not apply.

Outline. Before we can dive into the intricacies of implementing and attacking lattice-based cryptography, some groundwork has to be done first. For this reason, in Section 3.1 we give a brief introduction to lattices as such and also discuss computational problems forming the basis of many reductionist security arguments. In Section 3.2, we describe one concrete lattice-based primitive, namely the BLISS signature scheme. A crucial component of BLISS and many other proposals is Gaussian sampling, which is why we also discuss some sampler architectures. Finally, in Section 3.3 we recall previous and concurrent work on implementation security of lattice-based schemes.

3.1 Lattices

A lattice Λ is a discrete subgroup of $\mathbb{R}^n$. When given m linearly independent vectors $\mathbf{b}_1, \ldots, \mathbf{b}_m \in \mathbb{R}^n$, the lattice $\Lambda(\mathbf{b}_1, \ldots, \mathbf{b}_m)$ contains all of the points that

---

[2] Official forum for the NIST standardization process at `https://groups.google.com/a/list.nist.gov/forum/#!forum/pqc-forum`

are integer linear combinations of the basis vectors:

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

We call $\mathbf{B} = (\mathbf{b}_1, \ldots, \mathbf{b}_m)$ the basis matrix of the lattice, with n the dimension and m the rank of the lattice. Lattice bases are not unique: for each full-rank basis $\mathbf{B} \in \mathbb{R}^{n \times n}$ of Λ, one can apply a unimodular matrix $\mathbf{U} \in \mathbb{Z}^{n \times n}$, such that $\mathbf{UB}$ is also a basis of Λ. There exist lattice-basis reduction algorithms that are aimed at finding a *good* basis, which consists of short and nearly orthogonal vectors. The most important of these algorithms are the LLL [LLL82] as well as BKZ and its improved versions [CN11]. These algorithms output a new basis $\mathbf{B}'$ which satisfies certain conditions. Besides outputting $\mathbf{B}'$, LLL and BKZ implementations (such as those provided by the libraries NTL [Sho] and fplll [tea16]) can also output $\mathbf{U}$ such that $\mathbf{B}' = \mathbf{UB}$.

For cryptographic purposes one typically uses q-ary lattices. In this type it holds that for any vector $\mathbf{v} \in \Lambda$, all vectors $\mathbf{u}$ with $\mathbf{u} \equiv \mathbf{v} \bmod q$ are also in the lattice. Two hard hard problems using such q-ary lattices and forming the base of lattice-based cryptography are the Learning with Errors (LWE) problem [Reg05] and the Short Integer Solution (SIS) problem [Ajt96]. For LWE, one samples a uniformly random matrix $\mathbf{A} \in \mathbb{Z}_q^{m \times n}$ and two vectors $\mathbf{s}_1 \in \mathbb{Z}_q^n, \mathbf{s}_2 \in \mathbb{Z}_q^m$ from some narrow distribution, such as a discrete Gaussian (cf. Section 3.2.2) with low standard deviation. The adversary is then given the target $\mathbf{t} = \mathbf{A}\mathbf{s}_1 + \mathbf{s}_2 \bmod q$ and is tasked to recover $(\mathbf{s}_1, \mathbf{s}_2)$. For SIS, an attacker has to find a short but nonzero vector $\mathbf{s} \in \mathbb{Z}_q^n$ such that $\mathbf{As} \equiv \mathbf{0} \bmod q$.

Cryptographic constructions based on LWE or SIS require storing large matrices and computing matrix-vector products. In order to save memory and decrease execution time, the most efficient lattice-based cryptographic constructions introduce additional structure into their underlying lattice problems. That is, they work with the polynomial ring $\mathcal{R}_q = \mathbb{Z}_q[x]/(x^n + 1)$, with q being a prime and n a power of 2. An element $\mathbf{a} \in R_q$ can be described by its coefficient vector $\mathbf{a} = (a_0, \ldots, a_{n-1})$. Note that we will use bold-face to interchangeably denote polynomials and their coefficient vectors. Addition of two polynomials $\mathbf{a}, \mathbf{b}$ is simply the component-wise addition mod q. Multiplication of two polynomials $\mathbf{a}, \mathbf{b} \in \mathcal{R}_q$ will be denoted by $\mathbf{a} \cdot \mathbf{b}$, and can be represented as a matrix-vector product, i.e., $\mathbf{a} \cdot \mathbf{b} = \mathbf{aB} = \mathbf{bA}$, where the columns of $\mathbf{A}, \mathbf{B} \in \mathbb{Z}_q^{n \times n}$ are negacyclic rotations of $\mathbf{a}$ and $\mathbf{b}$, respectively. The computation of the i-th coefficient of the product $\mathbf{a} \cdot \mathbf{b}$ can be written as $\langle \mathbf{a}, \mathbf{b}_i \rangle$, with $\mathbf{b}_i$ the i-th column of matrix $\mathbf{B}$.

The versions of LWE and SIS using such an additional structure are aptly named Ring-LWE and Ring-SIS [LPR13]. From the above descriptions, it is easy to see that in the ring-setting the $(n \times n)$ matrix $\mathbf{A}$ can be replaced with the polynomial $\mathbf{a}$; the attacker is now given $\mathbf{t} = \mathbf{a} \cdot \mathbf{s}_1 + \mathbf{s}_2$. What's more, the parameters n and q are usually chosen such that polynomial multiplication can be efficiently computed using the Number Theoretic Transform (NTT). In essence, the NTT is a Fast-Fourier Transform over $\mathbb{Z}_q$ instead of over the complex numbers and allows to compute a polynomial multiplication in $\mathcal{O}(n \log n)$ time, instead of the $\mathcal{O}(n^2)$ required for the matrix-vector multiplication of standard LWE.

Module-LWE/Module-SIS are further generalizations of Ring-LWE/Ring-SIS, respectively. There, one uses matrices and vectors of polynomials. More concretely, 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 $\mathbf{t} \equiv \mathbf{A} \cdot \mathbf{s}_1 + \mathbf{s}_2 \bmod q$.

3.2 BLISS and Gaussian Samplers

We now give a brief description of one concrete lattice-based primitive, namely the BLISS [Duc+13] signature scheme and its improved variant BLISS-B [Duc14]. A crucial component of BLISS, and many other lattice-based schemes, is sampling from a discrete Gaussian distribution. For this reason we will describe this distribution and also methods to sample from it.

Notation. With $=$ we denote deterministic assignments (and also comparisons in cases where the meaning is clear). With $\leftarrow$ we refer to probabilistic sampling from either some distribution or uniformly from a set. We denote the probability of an event e as $\Pr(e)$. The ℓ_2 norm of a vector/polynomial $\mathbf{w} \in \mathcal{R}_q$ is defined as $\|\mathbf{w}\|_2 = \|\mathbf{w}\| = \sqrt{\sum_{i=0}^{n-1} w_i^2}$, with $w_i \in [-(q-1)/2, \ldots, -1, 0, 1, \ldots, (q-1)/2]$.

3.2.1 Bimodal Lattice Signature Scheme (BLISS)

The most efficient instantiation of BLISS works with polynomials over the ring $\mathcal{R}_q = \mathbb{Z}_q[x]/(x^n + 1)$ and thus allows working with efficient polynomial arithmetic.

Key generation for the improved version BLISS-B is shown in Algorithm 3.1. Apart from an additional rejection step, this algorithm is identical for the original BLISS scheme. During key generation, two polynomials $\mathbf{f}, \mathbf{g}$ with exactly $d_1 = \delta_1 n$ coefficients in $\{\pm 1\}$, $d_2 = \delta_2 n$ coefficients in $\{\pm 2\}$, and all remaining elements being 0, are sampled. n, δ_1, δ_2, and q are part of the parameter set.

Algorithm 3.1 BLISS-B Key Generation Algorithm

```
Output: Public key $\mathbf{A} \in \mathcal{R}_{2q}^2$, private key $\mathbf{S} \in \mathcal{R}_{2q}^2$
Choose random polynomials $\mathbf{f}, \mathbf{g}$ with $d_1$ entries in $\{\pm 1\}$ and $d_2$ entries in $\{\pm 2\}$ until $\mathbf{f}$ is invertible
$\mathbf{S} = (\mathbf{s}_1, \mathbf{s}_2) = (\mathbf{f}, 2\mathbf{g} + 1)$
$\mathbf{a}_q = \mathbf{s}_2/\mathbf{s}_1 \bmod q$
return $(\mathbf{A}, \mathbf{S})$, with $\mathbf{A} = (2\mathbf{a}_q, q - 2) \bmod 2q$
```

Signature generation is described in Algorithm 3.2. It takes as input a message μ, a public key $\mathbf{A}$, and a private key $\mathbf{S} = (\mathbf{s}_1, \mathbf{s}_2)$. First, two noise polynomials $\mathbf{y}_1, \mathbf{y}_2$ are sampled from a discrete Gaussian distribution D_σ. The intermediate

$\mathbf{u}$ is hashed together with the message, where the hash function H outputs a bit vector $\mathbf{c}$ of length n and (small) hamming weight κ. Depending on a random and secret bit b, the products $\mathbf{s}_1\mathbf{c}$ and $\mathbf{s}_2\mathbf{c}$ are then either added to or subtracted from the noise polynomials $\mathbf{y}_1$ and $\mathbf{y}_2$, respectively. BLISS is based on the Fiat-Shamir with Aborts Framework [Lyu09]. Simply speaking, in this framework a signature σ is rejected and signing restarted if $\mathbf{z}$ does (statistically) not follow the distribution of $\mathbf{y}$. This rejection sampling done in Line 7 hides any secret information in the signature and thus provides the zero-knowledge property. Parameters ζ, d, and p are used for signature compression.

Algorithm 3.2 BLISS Signature Algorithm

```
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 $(\mathbf{z}_1, \mathbf{z}_2^\dagger, \mathbf{c})$
1: $\mathbf{y}_1 \leftarrow D_\sigma^n$, $\mathbf{y}_2 \leftarrow D_\sigma^n$
2: $\mathbf{u} = \zeta \cdot \mathbf{a}_1\mathbf{y}_1 + \mathbf{y}_2 \bmod 2q$
3: $\mathbf{c} = \mathsf{H}(\lfloor \mathbf{u} \rceil_d \bmod p \| \mu)$
4: $b \leftarrow \{0,1\}$
5: $\mathbf{z}_1 = \mathbf{y}_1 + (-1)^b \mathbf{s}_1\mathbf{c}$
6: $\mathbf{z}_2 = \mathbf{y}_2 + (-1)^b \mathbf{s}_2\mathbf{c}$
7: Continue with probability $\left(M \exp\left(-\frac{\|\mathbf{Sc}\|^2}{2\sigma^2}\right) \cosh\left(\frac{\langle \mathbf{z}, \mathbf{Sc}\rangle}{\sigma^2}\right)\right)^{-1}$, else restart
8: $\mathbf{z}_2^\dagger = (\lfloor \mathbf{u} \rceil_d - \lfloor \mathbf{u} - \mathbf{z}_2 \rceil_d)$
9: return $(\mathbf{z}_1, \mathbf{z}_2^\dagger, \mathbf{c})$
```

The signing procedure of the improved BLISS-B variant is given in Algorithm 3.3. It differs from the original version by replacing the direct multiplication of $\mathbf{Sc}$ with $\mathsf{GreedySC}$ (Algorithm 3.4). It computes the product $\mathbf{Sc}'$ for some ternary vector $\mathbf{c}'$ (this means $\mathbf{c}' \in \{-1, 0, +1\}^n$) that satisfies $\mathbf{c}' \equiv \mathbf{c} \mod 2$. This $\mathbf{c}'$ is chosen such that $\|\mathbf{Sc}'\|$ is (heuristically) minimized, which in turn lowers the repetition rate and thus gives on average a performance improvement. The concrete speed-up depends on the used parameter set, it ranges from a factor of 1.2 to at most 2.8 [Duc14]. Note that for the specific BLISS input $\mathbf{S} = (\mathbf{s}_1, \mathbf{s}_2) \in \mathcal{R}_{2q}^2$ in $\mathsf{GreedySC}$, we have $m = 2n$ and $\mathbf{s}_i = \mathbf{S}_{1i}$ for $0 \leq i < n$ and $\mathbf{s}_i = \mathbf{S}_{2i}$ for $n \leq i < 2n$ where $\mathbf{S}_{1i}$ and $\mathbf{S}_{2i}$ are the negacyclic rotations of $\mathbf{s}_1$ and $\mathbf{s}_2$, respectively. The generated $\mathbf{c}'$ contains information on the secret key; hence it is kept secret and not output as part of the signature.

For completeness, we also present the verification algorithm in Algorithm 3.5. The verification algorithm is the same for both BLISS and BLISS-B. For a more detailed explanation, we refer to the original publications [Duc+13; Duc14].

Ducas et al. [Duc+13] propose several parameter sets for different security levels. These remain unchanged for BLISS-B. We give the proposed parameters in Table 3.1.

Algorithm 3.3 BLISS-B Signature Algorithm

```
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 $(\mathbf{z}_1, \mathbf{z}_2^\dagger, \mathbf{c})$
$\mathbf{y}_1 \leftarrow D_\sigma^n$, $\mathbf{y}_2 \leftarrow D_\sigma^n$
$\mathbf{u} = \zeta \cdot \mathbf{a}_1 \cdot \mathbf{y}_1 + \mathbf{y}_2 \bmod 2q$
$\mathbf{c} = \mathsf{H}(\lfloor \mathbf{u} \rceil_d \bmod p || \mu)$
$(\mathbf{v}_1, \mathbf{v}_2) = \mathsf{GreedySC}(\mathbf{S}, \mathbf{c})$
$b \leftarrow \{0, 1\}$
$(\mathbf{z}_1, \mathbf{z}_2) = (\mathbf{y}_1, \mathbf{y}_2) + (-1)^b (\mathbf{v}_1, \mathbf{v}_2)$
Continue with probability $\left(M \exp\left(-\frac{\|\mathbf{Sc}\|^2}{2\sigma^2}\right) \cosh\left(\frac{\langle \mathbf{z}, \mathbf{Sc} \rangle}{\sigma^2}\right)\right)^{-1}$, else restart
$\mathbf{z}_2^\dagger = (\lfloor \mathbf{u} \rceil_d - \lfloor \mathbf{u} - \mathbf{z}_2 \rceil_d) \bmod p$
return $(\mathbf{z}_1, \mathbf{z}_2^\dagger, \mathbf{c})$
```

Algorithm 3.4 GreedySC

```
Input: a matrix $\mathbf{S} \in \mathbb{Z}^{m \times n}$ and a binary vector $\mathbf{c} \in \mathbb{Z}^n$
Output: $\mathbf{v} = \mathbf{Sc}'$ for some $\mathbf{c}' \equiv \mathbf{c} \mod 2$
$\mathbf{v} = \mathbf{0} \in \mathbb{Z}^n$
for $i \in \mathcal{I}_c$ do
    $\zeta_i = \mathrm{sgn}(\langle \mathbf{v}, \mathbf{s}_i \rangle)$
    $\mathbf{v} = \mathbf{v} - \zeta_i \mathbf{s}_i$
return $\mathbf{v}$
```

Algorithm 3.5 BLISS Verification Algorithm

```
Input: Message $\mu$, public key $\mathbf{A} = (\mathbf{a}_1, q-2) \in \mathcal{R}_{2q}^2$, signature $(\mathbf{z}_1, \mathbf{z}_2^\dagger, \mathbf{c})$
Output: Accept or reject the signature
if $\mathbf{z}_1, \mathbf{z}_2^\dagger$ violate certain bounds (details in [Duc+13]) then reject
accept iff $\mathbf{c} = H(\lfloor \zeta \cdot \mathbf{a}_1 \cdot \mathbf{z}_1 + \zeta \cdot q \cdot \mathbf{c} \rceil_d + \mathbf{z}_2^\dagger \bmod p, \mu)$
```

Table 3.1: BLISS Parameter Sets

Parameter Set	n	q	σ	δ_1	δ_2	κ
BLISS-0 (Toy)	256	7681	100	0.55	0.15	12
BLISS-I	512	12289	215	0.3	0	23
BLISS-II	512	12289	107	0.3	0	23
BLISS-III	512	12289	250	0.42	0.03	30
BLISS-IV	512	12289	271	0.45	0.03	39

3.2.2 Discrete Gaussians

BLISS, just like many other lattice-based schemes, requires (high-precision) sampling from the so-called discrete Gaussian distribution. We denote such a distribution having standard deviation σ and mean zero by D_σ. When writing $y \leftarrow D_\sigma$, we have that a variable y is sampled from this distribution D_σ. The probability of sampling a value x is given by $D_\sigma(x) = \rho_\sigma(x)/\rho_\sigma(\mathbb{Z})$, with $\rho_\sigma(x) = \exp(-x^2/(2\sigma^2))$ and the normalization constant $\rho_\sigma(\mathbb{Z}) = \sum_{k=-\infty}^{\infty} \rho_\sigma(k)$. D_σ^n denotes an n-dimensional vector with elements independently sampled from D_σ.

The emergence of lattice-based cryptography and its reliance on such discrete Gaussian noise led to an increased interest in efficient samplers and a large number of proposed sampler architectures. Apart from generic methods like rejection sampling and inversion sampling, these also include, e.g., the Knuth-Yao random walk [DG14], the Ziggurat method [Buc+13], and arithmetic coding [Saa18].

Compared to lattice-based public-key encryption [LP11], the standard deviation required for BLISS and similar signature schemes is relatively high. This makes samplers requiring large precomputed tables less attractive, especially for constrained devices and their usually low storage capacities. A high-precision evaluation of transcendental functions, as required for straight-forward rejection samplers, is also not a viable option. For these reasons, specialized samplers were constructed, two of which we will now present.

Efficient CDT sampling. Pöppelmann et al. [PDG14] proposed an optimized sampler which is based on the inversion method. For generic inversion sampling, one first precomputes a cumulative distribution table (CDT), i.e., a table $T[y] = \Pr(x < y | x \leftarrow D_\sigma^+)$ for $y \in [0, \tau\sigma]$. Here, τ denotes the tail-cut factor which is required due to the infinite support of D_σ. Thanks to the symmetry of D_σ, sampling can be easily reduced to sampling from the one-sided distribution D_σ^+ with support $[0, \tau\sigma]$ followed by multiplication with a random sign bit. For actual sampling, one generates a uniformly random $r \in [0, 1)$ and returns the y satisfying $T[y] \leq r < T[y+1]$ (using a binary search in T).

As the statistical distance to a true discrete Gaussian must be kept low, a straight-forward implementation of the above approach requires that the entries of T are stored with very high precision, e.g., 128 bits. To reduce the table size and speed up sampling, Pöppelmann et al. propose the following optimizations. First, they save memory by using Gaussian convolution. They set $k = 11$, $\sigma' = \sigma/\sqrt{1+k^2} \approx 19.53$ and sample two values $y', y'' \leftarrow D_{\sigma'}$. They then combine them to $y \leftarrow D_\sigma$ by setting $y = ky' + y''$. And second, they speed up sampling by using a byte-oriented guide table I. Each entry $I[r_0]$ stores the smallest interval $(\min_{r_0}, \max_{r_0})$ with $T[\min_{r_0}] \leq r_0/256$ and $T[\max_{r_0}] \geq (r_0+1)/256$. By using this table, the range for the following binary search can be immediately reduced to the interval $[\min_{r_0}, \max_{r_0})$.

The detailed sampling procedure is given in Algorithm 3.6. It uses a byte-wise approach, where $T_j[i]$ denotes the j-th byte of $T[i]$. To save memory, the table T is stored in floating-point representation, using a mantissa table M and an exponent table E. For efficiency reasons Pöppelmann et al. actually store

$T[y] = \Pr(x \geq y | x \leftarrow D_\sigma^+)$, i.e., $T[0] = 1$ and $T[y] > T[y+1]$. This is accounted for in the binary-search part. For further explanations, we refer to [PDG14].

Algorithm 3.6 CDT Sampler using Guide Tables [PDG14]

Input: Guide table I, mantissa table M, exponent table E
Output: A value y' sampled according to $D_{\sigma'}$

```
$r_0 \leftarrow \{0,1\}^8$
$[\min, \max] = I[r_0]$
$i = (\min + \max)/2$, $j = 0$, $k = 0$
while max-min $> 1$ do
    $t = T_j[i]$, with $T_j[i] = M_{j-E[i]}[i]$ or 0
    if $t > r_j$ then
        $\min = i$, $i = (i + \max)/2$, $j = 0$
    else if $t < r_j$ then
        $\max = i$, $i = (\min + i)/2$, $j = 0$
    else
        $j = j + 1$
        if $k < j$ then
            $r_j \leftarrow \{0,1\}^8$, $k = j$
$s \leftarrow \{0,1\}$
if $s$ then return $-i$
else return $i$
```

Bernoulli rejection sampler. Rejection sampling is a generic method to sample from one distribution $f(x)$ when only having access to samples from another distribution $g(x)$. For sampling, one draws a value $y \leftarrow g(x)$, and then accepts this value with a probability of $f(y)/(M \cdot g(y))$, where M is a constant satisfying $f(x) \leq M \cdot g(x)$ for all x. If the sample is rejected, then sampling simply restarts.

In our case $f(x) = D_\sigma(x)$; $g(x)$ is typically some easy-to-sample-from distribution, e.g., uniform. Hence, $y \in [-\tau\sigma, \tau\sigma]$, which is then accepted with probability $\rho_\sigma(y)/\rho_\sigma(0)$. For this, a uniformly random value $r \in [0, 1)$ is sampled and y is accepted if $r \leq \rho_\sigma(y)/\rho_\sigma(0)$. A naive implementation of this method requires a high-precision evaluation of transcendental functions and is slow due to high rejection rates.

For this reason, Ducas et al. [Duc+13] introduce a more efficient method called *Bernoulli sampler*, which is tailored for BLISS and its high standard deviation. It uses the subroutine described in Algorithm 3.7 to sample a bit b from $\mathcal{B}(\exp(-x/f))$, i.e., the Bernoulli distribution $\mathcal{B}$ parametrized such that $\Pr(b = 1) = \exp(-x/f)$. The constant f depends on the standard deviation σ, while x varies. Pseudocode for the Bernoulli sampler appears in Algorithm 3.8. For a more detailed description, we refer to [Duc+13].

Algorithm 3.7 Sampling a bit from $\mathcal{B}(\exp(-x/(2\sigma^2)))$ for $x \in [0, 2^\ell)$

```
Input: $x \in [0, 2^\ell)$ an integer in binary form $x = x_{\ell-1} \dots x_0$. Precomputed table
    $E$ with $E[i] = \exp(-2^i/(2\sigma^2))$ for $0 \leq i < \ell$
Output: A bit $b$ from $\mathcal{B}(\exp(-x/(2\sigma^2)))$
for $i = \ell - 1$ downto 0 do
    if $x_i = 1$ then
        $A_i \in \{0,1\} \leftarrow \mathcal{B}(E[i])$
        if $A_i = 0$ then return 0
return 1
```

Algorithm 3.8 Bernoulli Sampler

```
Input: Standard deviation $\sigma$, integer $K = \lfloor \frac{\sigma}{\sigma_2} + 1 \rfloor$ with $\sigma_2^2 = \frac{1}{2 \ln 2}$
Output: A value $y'$ sampled according to $D_\sigma$
$x \in \mathbb{Z} \leftarrow D^+_{\sigma_2}$ (details in [Duc+13])
$z \in \mathbb{Z} \leftarrow \{0, \dots, K-1\}$
Set $y = Kx + z$
$b \in \{0,1\} \leftarrow \mathcal{B}(\exp(-z(z + 2Kx)/(2\sigma^2)))$ using Algorithm 3.7
if $b = 0$ then restart
if $y = 0$ then restart with probability 1/2
$s \leftarrow \{0,1\}$ and return $(-1)^s y$
```

3.3 Implementation Security of Lattice-Based Cryptography

Since lattice-based cryptography has gained traction in recent years, interest in its implementation-security aspect is also increasing. For the case of passive side-channel attacks, previous and concurrent works showed, e.g., the applicability of cache attacks to lattice-based signatures [Gro+16; Bin+17b; SZM17], specialized power analysis of lattice-based cryptography [PPM17], and cold boot attacks adapted specifically for lattices [ADP18].

Active implementation attacks on lattice-based cryptography also received some attention. Two concurrent works [BBK16; Esp+16] investigated fault attacks on the lattice-based signature schemes BLISS [Duc+13], GLP [GLP12], PASSSign [Hof+14], and ring-TESLA [Akl+16]. Espitau et al. [Esp+16] investigated loop-abort faults in the generation of the noise-polynomial $\mathbf{y}$.

These attacks make the need for appropriate countermeasures evident. In fact, many of the reference implementations submitted to the NIST call come with a basic countermeasure, namely constant (read: key-independent) runtime and control flow. This measure prohibits timing and cache attacks, but cannot protect against adversaries having physical access to the device and performing, e.g., power measurements. For such cases, more in-depth protection mechanisms such as masked implementations [Bar+18; Ode+18; Rep+15] as well as shuffling and other randomization techniques [Ode+18; Saa18] were proposed.

3.3.1 Security of Gaussian Samplers

Gaussian samplers are an integral part of many lattice-based primitives, but also an interesting side-channel target. As will later be shown, learning the sampled value via side-channel information enables key recovery. However, implementing thoroughly secured samplers seems to be a difficult task. Due to their complex structure, implementing them both correctly and efficiently is challenging and error-prone already, even without considering implementation security.

In fact, there already are side-channel attacks [Gro+16; Esp+17] which exploit the non-constant time nature of some samplers. Additionally, the use of Gaussians can lead to complex rejection conditions. For BLISS, it requires the evaluation of an exponential and a hyperbolic cosine (Line 7 of Algorithm 3.2). The BLISS authors do present methods to perform rejection efficiently and without the need of running high-precision floating-point arithmetic, but do not address the implementation security issue. However, non-constant time rejection was also already exploited in previous work [Esp+17].

Countermeasures. There do already exist first approaches to implementing discrete Gaussian samplers securely [Bos+15; MW17; HLS18; Kar+18]. Some of these approaches, however, can incur a large performance penalty [Bos+15] or might not be suitable for the large standard deviations required by lattice-based signatures [Kar+18].

An intermediate approach is to use an unprotected, or only somewhat protected, sampler and then shuffle, i.e., randomly permute, the entries in the generated noise vector [Saa18]. In Chapter 5, this countermeasure will be analyzed in detail.

Finally, the problem of securely implementing Gaussian samplers can be entirely sidestepped by not using discrete Gaussians at all. Instead, a different and easy to sample from distribution can be used. The lattice-based encryption/key exchange schemes NewHope [Alk+16] and Kyber [Ava+17], for instance, make use of the binomial distribution with single-trial success probability $p = 0.5$ and a small number of trials n. Samples from this distribution can be trivially generated by simply counting the number of set bits, i.e., computing the Hamming weight, of a uniformly random variable with fixed bit-length n.

For signatures, this distribution is not well suited, which is why schemes like GLP [GLP12], Dilithium [Lyu+17], and qTESLA [Bin+17a] use the uniform distribution over the integers in some fixed range instead. For the used parameters, constant-time sampling is trivial. Furthermore, this choice also drastically simplifies the rejection sampling present in all these schemes. Instead of evaluating transcendental functions, it is sufficient to simply test if the signature output is in the same range. A downside of using the uniform distribution is that signatures become slightly larger. Compared to a Gaussian-based instantiation of Dilithium described in [Duc+17], signatures of the uniform version are larger by approximately 10 %.

3.3.2 A Cache Attack on BLISS

At CHES 2016, Groot Bruinderink et al. [Gro+16] presented the first side-channel attack on BLISS and with that the first implementation attack targeting a lattice-based signature scheme. Since this attack will be often referenced (Chapter 4 and Chapter 5), we now give a more detailed explanation.

Their attack targets the Gaussian sampler and can thus stand as an example for the attacks mentioned in the previous section. As their side channel they use cache timing, i.e., they observe and exploit timing differences caused by the CPU cache (cf. Section 2.2.1). They use this information to infer some elements of the noise vector $\mathbf{y}$.

They then focus on Line 5 of Algorithm 3.2, i.e., $\mathbf{z}_1 = \mathbf{y}_1 + (-1)^b \mathbf{s}_1 \mathbf{c}$. As their attack targets the original BLISS variant, the $\mathbf{c} \in \{0,1\}^n$ used in this equation is output as part of the signature. Due to the rounding used for compression and the resulting loss of linearity they cannot use the second part of the signature $\mathbf{z}_2^\dagger$ and solely use $\mathbf{z}_1$. Thus, for the sake of simplicity we will from now on omit the index 1 of $\mathbf{z}_1, \mathbf{y}_1$, and $\mathbf{s}_1$, and always imply it, if not mentioned otherwise.

For each recovered Gaussian sample, an attacker can create an equation of form

$$z_{ji} = y_{ji} + (-1)^{b_j} \langle \mathbf{s}_1, \mathbf{c}_{ji} \rangle. \tag{3.1}$$

Here, i denotes the index of the recovered Gaussian sample in the signature. The values z_{ji} and y_{ji} are the i-th coefficients of $\mathbf{z}_1$ and $\mathbf{y}_1$ in the j-th signature; $\mathbf{c}_{ji}$ denotes the i-th column of $\mathbf{C}_j$, which is the matrix used in the matrix-vector representation of polynomial multiplication. After gathering enough of these equations over the course of multiple signing operations, they then recover the secret key $\mathbf{s}_1$ using linear algebra or a lattice reduction. $\mathbf{s}_2$ can then be trivially reconstructed by using the connection between public and private key.

Groot Bruinderink et al. apply this technique to two different Gaussian samplers, namely a CDT-based sampler similar to the one described in Algorithm 3.6 and the Bernoulli rejection sampler described in Algorithm 3.8.[3] For both, they performed an evaluation using ideal adversaries and practical experiments using the Flush+Reload attack technique. We now describe their attacks on the two samplers in detail.

Attacking the CDT sampler. The CDT sampler (Algorithm 3.6) uses two tables (CDT table T and interval table I). Accessing these tables can leak the accessed cache-line. This information, in turn, leaks a range of possible values for y_i. Groot Bruinderink et al. describe two approaches to estimate y_i more precisely than naively using these leaked ranges. The first approach is to intersect the ranges of possible values learned from each table. The second approach is to track down the binary search steps done in the sampling procedure by looking at multiple accesses in table T.

As the search step in table T is a binary search, one of two adjacent values is returned after the last table-lookup. This means that a sample y_{ji} can only

---

[3] Further sampler architectures are discussed in the full version of [Gro+16].

be determined up to an uncertainty of ±1. However, there is generally a bias in the value that is returned, as the targeted distribution is a discrete Gaussian and not uniform. If this bias is large enough, Groot Bruinderink et al. guess the returned value to be the more likely one.

Another obstacle is that they do not get the sign of y_{ji}, but only know $|y_{ji}|$ from the accessed cache-lines. However, they use the knowledge of the corresponding coefficient z_{ji} of the signature vector $\mathbf{z}$. It is possible to derive the sign from z_{ji}, as $\langle \mathbf{s}, \mathbf{c}_{ji} \rangle$ is small and thus the sign of y_{ji} will most likely be the sign of z_{ji}.

After the above procedure, they have approximate knowledge of y_{ji}. However, bit b of the signature is still unknown. Instead of guessing or recovering the value of this bit for each signature, they only use samples where, with a high probability, $z_{ji} = y_{ji}$. In these samples one has that $\langle \mathbf{s}, \mathbf{c}_{ji} \rangle = 0$ which makes the value of b irrelevant. After collecting enough samples, they use the challenge vectors $\mathbf{c}_{ji}$ that satisfy the above restrictions to construct a matrix $\mathbf{L}$ such that $\mathbf{sL} \approx \mathbf{0}$ is a small vector in the lattice spanned by $\mathbf{L}$. They then use the LLL lattice-reduction algorithm [LLL82] on $\mathbf{L}$ to find a small lattice basis. With a high probability, the secret key $\mathbf{s}$ is part of the unimodular transformation matrix retrieved from LLL. The correctness of the key can be verified by matching against the known public key.

Attacking the Bernoulli sampler. The Bernoulli sampler (Algorithm 3.8) uses the table E which stores (high precision) exponential values required to do rejection steps. As this table is only accessed for every set bit of input x (Line 2 in Algorithm 3.7), no table access is done in the case that input $x = 0$. This only happens when input z to the Bernoulli sampler (Line 4 in Algorithm 3.8) is zero, leading to a small subset of possible values $y_{ji} \in \{0, \pm K, \pm 2K, \ldots\}$. As K is in general large, this can lead to a complete retrieval of y_{ji} by also using knowledge of the corresponding signature coefficient z_{ji}. By again restricting to the cases when $y_{ji} = z_{ji}$, Groot Bruinderink et al. used the challenge vectors $\mathbf{c}_{ji}$ to construct a matrix $\mathbf{L}$ such that $\mathbf{sL} = \mathbf{0}$. The secret vector $\mathbf{s}$ can then be found by calculating the (integer left) kernel of $\mathbf{L}$.

4

Attacking StrongSwan's Implementation of BLISS-B

The side-channel attack on BLISS by Groot Bruinderink et al. [Gro+16] described on the previous pages breaks new ground and offers the first insights into implementation security of lattice-based cryptography. Nevertheless, their attack has some limitations.

First, in their proof-of-concept cache attack, they target the "research-oriented" reference implementation[1] of BLISS. They also modified its code 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 and practical setting. Second, and maybe more importantly, their attack does not apply to the improved BLISS-B. Due to its better performance, this newer variant is used per default in the first real-world adoption of BLISS, namely in the production-grade implementation deployed by the strongSwan IPsec-based VPN suite [str].

The attack target. An important operation in BLISS is the computation $\mathbf{z} = \mathbf{y} + (-1)^b(\mathbf{s} \cdot \mathbf{c})$. Using the recovered values of $\mathbf{y}$ over many signatures, Groot Bruinderink et al. 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 (cf. Section 3.3.2).

In BLISS-B, however, the secret $\mathbf{s}$ is multiplied with a ternary polynomial $\mathbf{c}' \in \{-1, 0, 1\}^n$ 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

---

[1] The reference implementation is available at `http://bliss.di.ens.fr/`

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 on the original BLISS, both for side-channel attacks [Esp+17] as well as fault attacks [BBK16; Esp+16]. Hence, one might be tempted to think of BLISS-B as a "free" side-channel countermeasure.

Contribution. In this chapter, we show that this is not the case. 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 applying 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 [Gro+16]. 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 under realistic settings.

Our key-recovery method 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. (Section 4.2.1)

- 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 over the bits, i.e., 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 present a tweaked LPN solving algorithm in Section 4.6 and use it to learn the parity of the secret key elements, i.e., to find $\mathbf{s} \bmod 2$ (Section 4.2.2).

- In some parameter sets (cf. Section 3.2.1), 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 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. (Section 4.2.3)

- 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. (Section 4.2.4)

When using the idealized cache-attack presented by Groot Bruinderink et al. [Gro+16] and the BLISS-I parameter set, our 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. We recover the secret signing key after observing roughly 6 000 signature generations. Unlike Groot Bruinderink et al., our adversary is asynchronous and runs in a different process than the victim. The adversary uses the Flush+Reload attack by Yarom and Falkner [YF14], combined with the amplification attack of Allan et al. [All+16]. Furthermore, we target a real-world implementation and not a research-oriented reference implementation. Consequently, our attack scenario is much more realistic. While strongSwan does not claim any side-channel security, our results still show that practical attacks on the BLISS family are feasible.

Outline. In Section 4.1 we give some additional background, namely a quick introduction to LPN and a closer look at limitations of earlier attacks on BLISS. We then show our improved key-recovery attack in Section 4.2. We evaluate our new method in Section 4.3 by comparing it to earlier work. In Section 4.4, we perform a full attack on the BLISS implementation provided by strongSwan. We briefly discuss countermeasures in Section 4.5. As the used error correction technique is only one step in key recovery, but requires a more in-depth explanation and additional background, we defer its detailed description to Section 4.6.

4.1 Preliminaries

In this section, we give some additional preliminaries required for this chapter. First we recap the LPN problem. Then we further describe some limitations of a previous side-channel attack on BLISS.

4.1.1 Learning Parity with Noise (LPN)

We now recall the Learning Parity with Noise (LPN) problem, whose search version appears in Definition 1.

Definition 1 (Learning Parity with Noise)**.** *Let* $\mathbf{k} \in GF(2^n)$ *and* $\epsilon \in (0, 0.5)$ *be a constant noise rate. Then, given* ν *vectors* $\mathbf{a}_i \in GF(2^n)$ *and noisy observations* $b_i = \langle \mathbf{a}_i, \mathbf{k} \rangle + e_i$*, the* $\mathbf{a}_i$ *sampled uniformly, and the* e_i *sampled from the Bernoulli distribution with parameter* ϵ*, find* $\mathbf{k}$*.*

The most efficient algorithms aimed at solving this problem are based on the work of Blum et al. [BKW03]. Later work then modified and improved the BKW algorithm [LF06; GJL14]. While these algorithms run in sub-exponential time, they tend to require a large number of LPN samples as well as a lot of memory. A different approach is to view LPN as decoding a random linear code over the binary field GF(2). While this second approach runs in exponential time, it typically offers a negligible memory consumption and lower sample requirements.

LPN is a well-researched problem and is used as a basis for cryptographic constructions [Pie12]. Furthermore, LPN solving algorithms have also been used in side-channel attacks on binary-field multiplication [Bel+15; BFG14; PM16]. The extension of this problem from the binary field GF(2) to a prime field GF(q) is known as Learning with Errors (LWE) [Reg05] and is a major cornerstone in lattice-based cryptography.

4.1.2 Limitations of Previous Attacks

The side-channel attack on BLISS described in Section 3.3.2 has certain limitations and caveats. Due to the unknown bit b, which is potentially different for each signature, Groot Bruinderink et al. [Gro+16] only use samples where $z_{ji} = y_{ji}$ and thus $\langle \mathbf{s}, \mathbf{c}_{ji} \rangle = 0$ (with high probability). This, however, only holds in roughly 15 % of all samples (cf. Figure 4.3) 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 attack does not apply to the improved BLISS-B signature scheme. Groot Bruinderink et al. recover the key by solving a (possibly erroneous) linear system $\mathbf{sL} \approx \mathbf{0}$, where $\mathbf{L}$ consists of the used challenge vectors $\mathbf{c}_{ji}$. However, the GreedySC algorithm (Algorithm 3.4), 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.

A third limitation of the attack of Groot Bruinderink et al. is the question of practicality. The attack targets an academic implementation that is not used in any "real-world" applications. Furthermore, the attack is synchronous. To achieve this, Groot Bruinderink et al. modify the code of the BLISS implementation 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 Groot Bruinderink et al. show the exploitation potential and a proof-of-concept, their attack falls short of being practical.

We now present a new key-recovery technique that resolves the issues discussed in this section. That is, it works even for BLISS-B and can reduce the number of required signatures by using all recovered samples. Furthermore, in Section 4.4 we give results on our improvements on the practicality of the previous attack. That is, we present the asynchronous attack on strongSwan's implementation of BLISS-B.

4.2 An Improved Side-Channel Key-Recovery Technique

In this section, we present our new and improved side-channel attack on BLISS, which also works for BLISS-B. Our method consists of four main steps, each step reveals additional information on the secret signing key $\mathbf{s}$.

The first step is equivalent to previous works. That is, the attacker performs a side-channel attack, e.g., a cache attack or power analysis, 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 4.2.1)

Due to the previously mentioned sign-uncertainty in BLISS-B (the recovered terms $\mathbf{s} \cdot \mathbf{c}'$ 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 over the bits, i.e., in GF(2). For error correction, we employ an LPN algorithm that is based on a decoding approach and can incorporate differing error probabilities. (Section 4.2.2)

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 $\langle \mathbf{s}^*, \mathbf{c}_{ji} \rangle$, and compare this with $z_{ji} - y_{ji} = \langle \mathbf{s}, \mathbf{c}'_{ji} \rangle$ (obtained from the side channel). Based on that, we give two different methods in Section 4.2.3 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 public key. We then perform a lattice reduction and search for $\mathbf{s}_2$ as a short vector in the lattice spanned by this reduced key. Linear algebra then allows recovery of the full private key $(\mathbf{s}_1, \mathbf{s}_2)$ (Section 4.2.4).

We now give a more detailed description of these steps.

4.2.1 Step 1: Gathering Samples

Akin to previous attacks (cf. Section 3.3.2), 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 other works, the exploited side channels were cache timings (in [Gro+16]) or power consumption (in [Pes16]).

Side-channel analysis has to deal with noise and other uncertainties. Due to these effects a recovered sample y_{ji} 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 be different for each sample. We will later use these probabilities to optimize our attack.

For each recovered sample y_{ji}, we can write an equation $z_{ji} = y_{ji} + (-1)^b \langle \mathbf{s}, \mathbf{c}'_{ji} \rangle$, which holds with probability $1 - \epsilon$. As the signs of coefficients of $\mathbf{c}'_{ji}$ are unknown, we can simply ignore the multiplication with $(-1)^b$ and instead implicitly include this factor into $\mathbf{c}'_{ji}$. Unlike Groot Bruinderink et al., we do not require that

$\langle \mathbf{s}_1, \mathbf{c}_{ji} \rangle = 0$ and thus can use all recovered samples. We compute the difference $t_{ji} = z_{ji} - y_{ji}$ and rearrange all gathered $\mathbf{c}'_{ji}$ 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 $\mathbf{c}'$ 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.

4.2.2 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 3.3, i.e., $\mathbf{z}_1 = \mathbf{y}_1 + \mathbf{s}_1 \cdot \mathbf{c}'$, is defined over $\mathbb{Z}$. That is, there is no reduction mod q involved[2]. Such an equivalence relation in $\mathbb{Z}$ obviously 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 $\mathbf{s}^*\mathbf{L}' = \mathbf{t}^*$ in 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 3.1), 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 show how errors in $\mathbf{t}^*$ can be corrected.

Error Correction mod 2. As stated in Section 4.2.1, a recovered Gaussian sample y_{ji} might not be correct. Hence, the right-hand-side of the system $\mathbf{s}^*\mathbf{L}' = \mathbf{t}^*$ is possibly erroneous. For instance, in the cache attack on CDT sampling algorithm of Groot Bruinderink et al., errors cannot be avoided. Hence, the capability of error correction is crucial.

We can 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 described in Section 4.1.1, thus we employ an LPN solving algorithm to recover $\mathbf{s}^*$. The most time-efficient algorithms to solve LPN are based on the work of Blum et al. [BKW03]. A caveat of this and improved versions [LF06; GJL14] are large memory and LPN-sample requirements. For instance, with $n = 512$ and an error probability ϵ of just 0.01, the often quoted LF1 algorithm by Levieil et al. [LF06] requires 2^{52} bytes of memory. Thus, for BLISS and the already somewhat high dimension of $n = 512$ this class of algorithms is not ideal for the problem at hand.

Also, note that in the definition of the LPN problem (Definition 1) the error probability ϵ is constant for all samples. This, however, does not reflect the reality of our side-channel attack. There, each recovered Gaussian sample can be assigned a potentially different error probability ϵ_i. By making use of this additional knowledge, the solving process can potentially be sped up.

We use a new LPN-solving algorithm that can make use of such differing probabilities and that does not require an extensive amount of memory. First, we

---

[2] 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.

perform filtering, i.e., only keep the samples with the lowest error probabilities. All other samples are discarded. Then we use a decoding approach, i.e., solving LPN by decoding a random linear code, on the remaining samples. We tweaked Stern's decoding algorithm [Ste88] such that it can incorporate varying error probabilities. As a detailed description of the algorithm requires additional background, we defer it to Section 4.6. This method was originally proposed in the context of a side-channel attack on polynomial multiplication in GF(2) [PM16], but the algorithmic problem is identical.

It is easy to see that due to the initial filtering of highly reliable equations, there exists a possible trade-off between gathered samples and computational runtime. That is, with more equations one can expect a lower error probability of the few best samples, which decreases the runtime of decoding. We will explore this trade-off in Section 4.3.

Determining error probabilities. Thus far, we did not discuss how the error probabilities of the samples are computed. They mainly depend on the used side-channel attack. Groot Bruinderink et al. [Gro+16] attack two different samplers using a cache attack. In their (idealized) attack on a Bernoulli sampler, they can recover samples perfectly. Hence, no error correction is required. The attack on a CDT sampler, however, cannot exclude errors. There, the error probability depends on the used cache weakness[3].

4.2.3 Step 3: Recovering the Position 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 the 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 $0 \le \eta_2 \le \min(d_2, \kappa - \eta_1)$.

Both methods then compare the output of the side-channel analysis, i.e., $|z_{ji} - y_{ji}| = |\langle \mathbf{s}, \mathbf{c}'_{ji} \rangle|$, 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 formulating it into an Integer Program. First, suppose we perfectly retrieved y_{ji} from a

---

[3] The error probabilities are specified in Appendix B of the full version of [Gro+16].

side-channel. If

$$|z_{ji} - y_{ji}| = |\langle \mathbf{s}, \mathbf{c}'_{ji} \rangle| > \eta_1 + 1,$$

we know that $\eta_2 > 0$, i.e. there has to be at least one ±2 involved making up for the difference in the above inequality. We save all $|\mathbf{c}_{ji}|$ for which the above is true in a list $\mathbf{M}$. Then, we need to find a solution $\mathbf{r}$ for the following constraints:

$$\mathbf{Mr} \geq \mathbf{1}.$$

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.

Finding the solution $\bar{\mathbf{s}}$ can be seen as a minimal-set-cover problem. 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 [Pro]. 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 $|z_{ji} - y_{ji}|$ and η_1 needs to be at least 2. Samples with an error of ±1 can be detected an discarded, simply due knowing the correct parity. Note that in the work of Groot Bruinderink et al. [Gro+16], an (idealized) adversary targeting the CDT sampling algorithm only makes errors of ±1. Hence, this method can be used for this scenario.

Statistical Approach. We now give a second approach that can recover the position of twos in $\mathbf{s}_1$. It differs from the first as we use a statistical approach rather than integer programming. Thus, it can withstand errors more easily.

We use the following observation. The probability that a certain $z_i - y_i$ is observed clearly depends on η_1 and η_2. If $\eta_2 = 0$, then the probability density function is essentially a binomial distribution picking from $\{\pm 1\}$ instead of the usual $\{0, 1\}$. If $\eta_2 \neq 0$ but $\eta_1 = 0$, the same goes with $\{\pm 2\}$. We compute the joint distribution for all possible combinations of η_1 and η_2

We then perform a standard hypothesis testing. That is, for every recovered sample we compute $\Pr(Z - Y = z_i - y_i | \mathrm{H}_1 = \eta_1, \mathrm{H}_2 = \eta_2)$ for the correct η_1 and for all $0 \leq \eta_2 \leq \min(d_2, \kappa - \eta_1)$. Note that all distributions, and thus also the joint one, are symmetric. Thus, the actual sign of $z_i - y_i$ is not relevant. Then, we apply Bayes' Theorem to get every $\Pr(\mathrm{H}_2 = \eta_2 | Z - Y = z_i - y_i)$, compute the expected value of H_2, and divide this number by $\min(d_2, \kappa - \eta_1)$. This gives us the probability that any one of the $\min(d_2, \kappa - \eta_1)$ unknown but involved key coefficients is 2.

Finally, we perform a log-likelihood test. For each unknown coefficient s_k in $\mathbf{s}_1$, we compute the mean of the logarithm of the above probability, over the recovered samples where $\mathbf{c}_{ji}$ is 1 at index k. We then set the d_2 coefficients with the highest score to 2.

4.2.4 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 3.1) 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 the BLISS-I and BLISS-II parameter sets (Table 3.1), 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 such a way such 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 $n-1$.

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. The returned shortest-vector is the sought-after $\mathbf{g}^\star$. We then 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 non-zero coefficients of the full $\mathbf{f}$. By 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.

4.3 Evaluation of Key Recovery

In this section, we give an evaluation of our new key-recovery technique. That is, we analyze its performance and compare it to the attack of Groot Bruinderink et al. [Gro+16] on original BLISS. 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 Groot Bruinderink et al.. 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). Finally, we state figures for the full-key recovery using a lattice reduction (Step 4).

4.3.1 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 2.4 GHz. If this does not find a solution after at most 10 minutes, then we abort and say that the experiment has failed.

Cache attack on CDT sampling. Figure 4.1 shows the results of the idealized cache-attack on a CDT sampler by Groot Bruinderink et al. [Gro+16]. We did not perceive any significant differences between BLISS and BLISS-B here, so we performed 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.

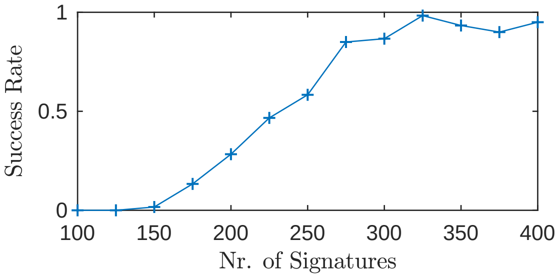
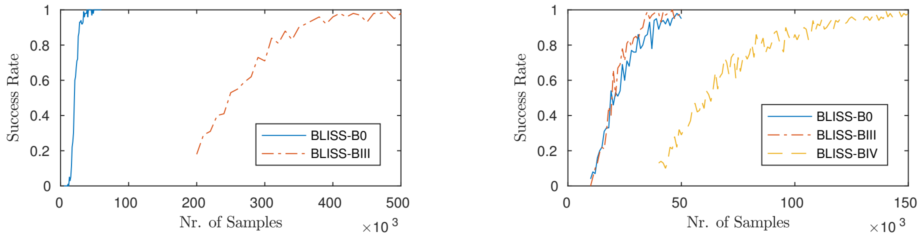


Figure 4.1: Success rate of LPN decoding for an idealized attack on CDT sampling

4.3.2 Step 3: Recovery of Twos

For evaluation of the third attack step, we analyzed the success rate of both twos-recovery procedures (Section 4.2.3) 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 4.2. Please note that this is not equal to the number of required signatures (see [Gro+16]). As seen in Figure 4.2a, the linear-programming approach requires 30 000 samples for BLISS-0 and 400 000 samples for BLISS-III, respectively. Here we did not evaluate the performance with BLISS-IV due to even higher sample requirements. 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 4.2: Success rate for Twos recovery

4.3.3 Step 4: Key-Recovery using Lattice Reduction

In the last step, i.e., recovery of the full signing key $\mathbf{s}$ from $|\mathbf{s}|$ (Section 4.2.4), we use the BKZ lattice-reduction algorithm. Concretely, we use the implementation provided by Shoup's Number Theory Library (NTL) [Sho]. 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 evaluated the correctness and performance of this method by running over 250 key-recovery experiments for both BLISS-I and BLISS-BI. In each experiment, we generated a new key, performed a key recovery mod 2 (assuming a perfect and errorless side-channel), and finally performed a lattice reduction. All our experiments were 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 2.6 GHz.

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.

4.4 Attacking strongSwan's BLISS-B

In this section, we perform a cache attack on the BLISS-B implementation of the strongSwan IPsec-based VPN suite [str]. Concretely, we use the parameter set BLISS-I. We describe the setup and the execution of the cache attack in Section 4.4.1. Our adversary is not synchronized with the victim, thus we perform synchronization based on the signature output (Section 4.4.2). This corresponds to the first step of our key-recovery method. Finally, we apply the other three steps and describe the outcome.

4.4.1 Asynchronous Cache Attack

We carry out the experiment on a server featuring an 8-core Intel Xeon E5-2618L v3 2.3 GHz processor and 8 GB of memory, running a CentOS 6.8 Linux, with gcc 4.4.7. We use strongSwan version 5.5.2, which was the current version at the time of running the experiments. 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 behavior of the code and is not used for key extraction.

For the 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 Algorithm 3.8. This slows the average running time of the function from 500 to 233000 cycles, creating a temporal separation between calls to Algorithm 3.8. 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.

The BLISS implementation included in strongSwan uses the Bernoulli-sampling approach described in Section 3.2.2. Thus, we reuse the exploit of Groot Bruinderink et al. [Gro+16] and detect if the input to Algorithm 3.7 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 3.8). 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 3.7 and only use the last entry per sampled value. Other calls to this function correspond to rejections (Lines 5 and 6 of Algorithm 3.8) and thus cannot be used. Finally, if we detect that Line 3 of Algorithm 3.7 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 3 of Algorithm 3.7 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 3 was missed by the cache attack.

4.4.2 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 3.8 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 3.8 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 a standard deviation of 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. . . 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 3.8. 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 4.3 (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. Note that this approach is somewhat similar to the attack on the shuffling countermeasure described in Section 5.2.

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

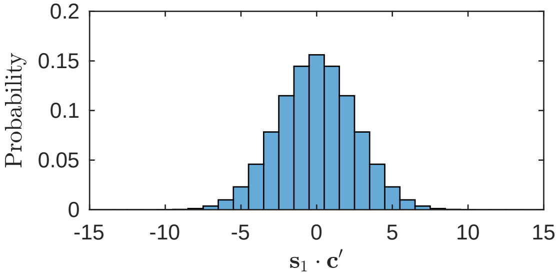


Figure 4.3: Coefficient-wise probability distribution of $\mathbf{s}_1 \cdot \mathbf{c}'$

single signature index with high probability, i.e., > 0.975, and where the prior-probability $\Pr(\langle \mathbf{s}_1, \mathbf{c}'_i \rangle = d)$ 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., they 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 recovered samples are correct in GF(2).

4.4.3 LPN and Results

For LPN-decoding (Section 4.2.2) we set the code length to 1024. With the above detection rates, we require on average 6 000 signatures in order to collect this number of samples. Note that for the selection of used samples we again made use of probabilities. For instance, we use only zero events that can be reassigned to a single sample with high probability. Unlike in the case of the attack on the CDT sampling algorithm, however, we were not able to accurately determine any differing error probabilities within the selected 1024 samples. Thus, we used a non-modified algorithm for decoding the random linear code. Our used decoding algorithm is based on the descriptions in [BLP08].

We performed 100 decoding experiments using the error distribution obtained from the previous step. In the 1024 used samples we encountered between 26 and 36 errors. We ran decoding using 64 threads on two Xeon E5-2699 v4 running at 2.2 GHz. Similar to Section 4.3.1, we abort decoding after 10 minutes and then consider it to have failed. 98 experiments were successful, 82 of them finished within the first minute.

As we used the parameter set BLISS-I and thus have $\mathbf{s} \in \{0, \pm 1\}$, the third attack step is not required. The fourth attack step, lattice reduction, then finally returns the secret signing key. The runtime of this step was already stated in Section 4.3.3.

4.5 Countermeasures

To protect against the side-channel attack described in this paper, it is vital that Algorithm 3.7 is implemented in constant time and without secret-dependent branching. More specifically, the handling of rejections and table look-ups should not depend on the input. As shown in Algorithm 4.1, this can be done by performing all ℓ steps in the loop and always sample an A_i. The return value v is then updated according to the values of A_i and x_i in constant time. We use C-style bitwise-logic operations to describe this update.

Algorithm 4.1 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 $x = x_{\ell-1} \dots x_0$. Precomputed table
    $E$ with $E[i] = \exp(-2^i/(2\sigma^2))$ for $0 \leq i < \ell$
Output: A bit $b$ from $\mathcal{B}(\exp(-x/(2\sigma^2)))$
$v = 1$
for $i = \ell - 1$ downto 0 do
    $A_i \in \{0, 1\} \leftarrow \mathcal{B}(E[i])$
    $v = v \,\&\, (A_i \mid {\sim}x_i)$
return $v$
```

Note that while this ad-hoc countermeasure can fix the exploited leak in this specific implementation, different attack techniques and side-channels might still allow key recovery. Thus, high-precision Gaussian samplers will likely stay a prime target for attacking lattice-based schemes. Some cryptographers and implementers seem to have noted this problem, as there already exist approaches at thoroughly secure sampling. These include different sampler architectures as well as other and more easy to sample from noise distributions (cf. Section 3.3.1).

4.6 A Closer Look at the Error Correction

In Section 4.2.2, we used a tweaked LPN algorithm to recover $\mathbf{s}_1 \bmod 2$ from possibly erroneous noise samples y. Previously we only gave the basic intuition of this error-correction approach, but did not discuss its details. We now make up for this and give an in-depth description.

Notation. For a random variable X, we use $\mathbb{E}(X)$ to denote its mean. We use side-channel leakage to derive the probability that a bit b is set to 1. We write $p_b = \Pr(b = 1)$. We use τ_b as the respective bias, i.e., $\tau_b = |p_b - 1/2|$. When performing a classification, we set $b = \lfloor p_b \rceil$, with $\lfloor \cdot \rceil$ the rounding operator. This classification has an error probability $\epsilon_i = 1/2 - \tau_i$.

The above is exactly the Bernoulli distribution with parameter p_b, we denote it as $\mathcal{B}(p_b)$. We also make use of the so-called Poisson binomial distribution. This distribution describes the sum of N independent Bernoulli trials, where each trial has a possibly different Bernoulli parameter p_k. Given the vector $(p_1, \dots, p_N)$, the respective density function can be computed by using the closed-form expression by Fernandez and Williams [FW10].

4.6.1 LPN and Decoding

In this section, we explore the connection between LPN (Definition 1) and coding theory. Then we give approaches for decoding random linear codes.

Connection to random linear codes. The LPN problem (Definition 1) can be restated as decoding a random linear code over GF(2) [Pie12]. Let $\mathbf{A} = [\mathbf{a}_i]_{0 \leq i < \nu}$ be the matrix whose rows are the $\mathbf{a}_i$. Further, let $\mathbf{b}$ and $\mathbf{e}$ be row vectors of the b_i and e_i, respectively, Then, one can think of $\mathbf{A}$ as generator matrix of a random linear code. Decoding requires to find the message $\mathbf{k}$ given a noisy word $\mathbf{b} = \mathbf{kA} + \mathbf{e}$, which is exactly LPN.

Linear codes are characterized by the three main parameters $[n, k, d]$, with n the code length, k the code dimension, and d the minimum Hamming distance between any two valid codewords. In the case of LPN, the dimension k is equal to the size of the secret. For random linear codes, the obtained code rate $R = k/n$ is, with very high probability, close to the Gilbert-Varshamov bound [CG90]. That is, $R \approx 1 - \mathsf{H}(d/n)$, with H the binary entropy function. The code length n is chosen according to this bound, with $d/n \approx \epsilon$.

The fastest algorithms for decoding random linear codes rely on *Information-Set Decoding* (ISD). First proposed by Prange in 1962 [Pra62], these algorithms have quite a long history, with probably the most notable version being Stern's algorithm [Ste88].

Syndrome decoding. Before discussing decoding algorithms in detail, we briefly describe syndrome decoding. For a $(k \times n)$ generator matrix $\mathbf{G}$ in standard form, i.e., $\mathbf{G} = (I_k|Q)$, the so-called parity-check matrix $\mathbf{H}$ is given as $\mathbf{H} = (-Q^T|I_{n-k})$, with Q a $(k \times (n-k))$ matrix. The set of valid codewords C forms the kernel of the check matrix, i.e., $\mathbf{Hc} = 0, \forall \mathbf{c} \in C$. For a noisy word $\mathbf{y} = \mathbf{c} + \mathbf{e}$, we have $\mathbf{Hy} = \mathbf{He} = \mathbf{s}$. $\mathbf{s}$ is called the syndrome, it only depends on the error $\mathbf{e}$.

For decoding, we now want to find an error term $\mathbf{e}$ with some maximum weight w such that $\mathbf{He} = \mathbf{s}$. In other words, we are searching for at most w columns of $\mathbf{H}$ summing up to $\mathbf{s}$.

Stern's attack (and improvements). We now review Stern's algorithm. This algorithm takes as input a check matrix $\mathbf{H}$, a syndrome $\mathbf{s}$, and a maximum error weight w.

In a first step, Stern partitions the n columns of the parity-check matrix $\mathbf{H}$ into two distinct sets $\mathcal{I}, \mathcal{Q}$. $\mathcal{I}$ is made up of $(n-k)$ randomly selected columns which must form an invertible subset. $\mathcal{Q}$ is comprised of the remaining k columns. For simplicity, we assume that the columns of the check matrix are permuted such that $\mathbf{H}' = (\mathcal{Q}|\mathcal{I})$. Note that such a permutation also affects the syndrome and the position of error bits.

Next, he selects a size ℓ subset $\mathcal{Z}$ of $\mathcal{I}$, where ℓ is an algorithm parameter. $\mathcal{Q}$ is randomly split into two size $k/2$ subsets $\mathcal{X}, \mathcal{Y}$. The second part of $\mathbf{H}'$ is then transformed into identity form by applying elementary row operations. Stern then searches for (permuted) error terms $\mathbf{e}$ with a maximum weight w having exactly p nonzero bits in $\mathcal{X}$, p nonzero bits in $\mathcal{Y}$, no nonzero bits in $\mathcal{Z}$, and at most $w - 2p$ in the remaining columns. This is visualized in (4.1). This search uses a collision technique. If it fails, then the algorithm is restarted by selecting new $\mathcal{Q}$ and $\mathcal{I}$. For more details on the algorithm we refer to [BLP08].

$$\mathbf{H}' = (\mathcal{Q}|\mathcal{I}) = \left(\begin{array}{cc|cccc}
\overbrace{1\ 0\ 0\cdots}^{k/2:\ p\text{ err.}} & \overbrace{\cdots 0\ 1\ 0}^{k/2:\ p\text{ err.}} & \overbrace{1\quad\ }^{\ell:\ 0\text{ err.}} & & & \\
1\ 1\ 0\cdots & \cdots 0\ 0\ 0 & & 1 & & \\
0\ 1\ 1\cdots & \cdots 1\ 1\ 1 & & & 1 & \\
\vdots & \vdots & & & \ddots & \\
0\ 1\ 1\cdots & \cdots 1\ 0\ 1 & & & & 1
\end{array}\right) \tag{4.1}$$

Canteaut and Chabaud [CC98] proposed an improvement of this algorithm, which was later refined by Bernstein, Lange, and Peters [BLP08]. Instead of choosing $\mathcal{Q}, \mathcal{I}$ randomly at each iteration and spending considerable time to transform $\mathbf{H}'$ to the desired form, one can use a simple column swapping. In each iteration, c elements of $\mathcal{Q}$ are exchanged with c from $\mathcal{I}$, where c is an algorithm parameter.

Stern and reliability. The possibility of enhancing the performance of Stern's algorithm by using reliability information was briefly mentioned by Valembois [Val00]. However, thus far it was not used in a cryptographic or side-channel context. Also, it lacks proper description and an in-depth runtime analysis.

We will now explain our new algorithm and thus show how reliability information can be leveraged to reduce the computation time. First however, we introduce a new version of LPN which better describes the problem at hand.

4.6.2 LPVN: A new LPN Variant

In standard LPN (Definition 1), the error probability ϵ is constant for all samples. This, however, does not reflect the reality of the side-channel information, where every LPN sample can be assigned a possibly different error probability. We formalize this by introducing a new problem dubbed *Learning Parity with Variable Noise* (LPVN).

Definition 2 (Learning Parity with Variable Noise)**.** *Let* $\mathbf{k} \in GF(2^n)$ *and* ψ *be a probability distribution over* $[0, 0.5]$*. Then, given* ν *vectors* $\mathbf{a}_i \in GF(2^n)$*,* v *error probabilities* ϵ_i*, and noisy observations* $b_i = \langle \mathbf{a}_i, \mathbf{k} \rangle + e_i$*, the* $\mathbf{a}_i$ *sampled uniformly, the* ϵ_i *sampled from* ψ^4*, and the* e_i *sampled from* $\mathcal{B}(\epsilon_i)$*, find* $\mathbf{k}$*.*

Casting LPVN to LPN is possible by simply setting $\epsilon = \mathbb{E}(\epsilon_i)$. However, the additional information in the form of the ϵ_i allows designing more efficient algorithms. Also, it is easy to see that with a non-zero meta-probability distribution ψ in close vicinity of 0, the problem becomes trivial given enough samples.

4.6.3 Filtering

In the context of side-channel analysis, the overall average error rate $\mathbb{E}(\epsilon_i)$ can be expected to be high, i.e., beyond 0.25. The resulting large code length n might lead to excruciating decoding runtimes.

In order to cut this time down drastically, we perform filtering of the samples. When given a certain number ν of LPVN samples, only those n with the lowest error probability are kept. All other samples are simply discarded.

The number of available samples ν plays a crucial role in the expected attack runtime. By increasing ν, the quality of the best samples is also expected to rise. This in turn decreases the required code length n and the runtime of the decoding algorithm. Hence, a trade-off between the number of samples ν and computational complexity is possible. This is in stark contrast to standard LPN, where the decoding runtime is mostly independent of the number of samples.

4.6.4 Using Reliability in Stern's Attack

After filtering, the n remaining samples are used as input for Stern's algorithm. More concretely, we use the improved version given in [BLP08]. This algorithm does not directly cope with reliability information. Hence, by setting $b_i = \lfloor p_i \rceil$ a classification is performed.

Instead of discarding the reliability information at this point, we use it to further speed up the decoding process. Recall that Stern's algorithm involves a column-swapping step. We now tweak the algorithm by replacing the uniform selection of the swapped columns with a reliability-guided one. The goal is to minimize the expected error in $\mathcal{Q}$, while still assuring high randomness in the chosen columns.

Column-swapping procedure. The probability that a column $t \in \mathcal{Q}$ is deselected in the next step is set to be directly proportional to its error probability ϵ_t, i.e., $\Pr(t) = \epsilon_t / \sum_{t^* \in \mathcal{Q}} \epsilon_{t^*}$. Analogously, we use the *squared* bias to select the new column, i.e., for every $u \in \mathcal{I}$, $\Pr(u) = \tau_u^2 / \sum_{u^* \in \mathcal{I}} \tau_{u^*}^2$. Experimentally we found that this combination gives the best performance.

We use rejection sampling in order to sample from these continuously changing probability density functions. Concretely, we sample a $t \in \mathcal{Q}$ and a $u \in [0, \max_{0 \le i < n}(\epsilon_i)]$ uniformly and accept t if $u < \epsilon_t$. Note that computation of the normalized probabilities $\Pr(t)$ is not required for this method.

The impact of inaccurate probabilities. It might not always be possible to derive accurate probabilities for all samples. For instance, when using a template attack, the leakage characteristic of the profiling device can slightly differ from that of the attacked device. As long as the error in probability is not too large, the attack will still work. However, the algorithm runtime and its analysis might suffer from inaccuracies.

4.6.5 Runtime Analysis of the Tweaked Algorithm

In this section, we present the method used for runtime estimation of our tweaked information-set decoding algorithm. This analysis follows along the lines of [BLP08].

The runtime of ISD algorithms is typically measured in the number of required bit operations. As the amount of bit operations per iteration of Stern's algorithm is essentially unchanged when using our tweak, we refer to [BLP08] for the calculation of this quantity. The number of required iterations however is expected to decrease. We now detail the estimation of this quantity.

Probability that a column is part of $\mathcal{Q}$. In a first step, for each column t of the initial parity-check matrix we retrieve $\Pr(t \in \mathcal{Q})$, i.e., the average probability of t being part of $\mathcal{Q}$ when using the replacement rules given in Section 4.6.4. This is done by using a method similar to a Markov-chain analysis.

We define $\mathbf{s} = (s_0, s_1, \ldots, s_{n-1})$ as the vector containing these probabilities (the state). This vector is initialized uniformly, i.e., $s_i = 512/n, 0 \leq i \leq n$ (for BLISS-I). Then we construct a transition matrix $\mathbf{A}$ which depends on the current state. A column in this matrix corresponds to the event that the respective column of the check matrix is drawn in the uniform-sampling step of the rejection-sampling procedure. The main diagonal of $\mathbf{A}$ contains the probability that this selection is rejected, which is 2τ in our case. The remaining entries contain the swapping probabilities, i.e., the probability that column $j \in \mathcal{Q}$ is exchanged with column $i \notin \mathcal{Q}$. Note that these depend on the current state, as columns already part of $\mathcal{Q}$ can not be swapped into it. Thus, we have

$$\mathbf{A}_{i,j,i\neq j} = (1 - 2\tau_j)\frac{(1 - s_i)\tau_i^2}{\sum_{j^* \neq j}(1 - s_j^*)\tau_{j^*}^2} \text{ and } \mathbf{A}_{i,i} = 2\tau_i.$$

Finally, we update $\mathbf{s} = \mathbf{A}\mathbf{s}$. The recalculation of $\mathbf{A}$ and updating is repeated until the probability vector $\mathbf{s}$ reaches a steady state.

Error-count density function. In the subsequent step, we compute the probability density function for the error count for both $\mathcal{Q}$ and $\mathcal{I}$. For that we acquire the Poisson binomial PDF with Bernoulli probabilities $\epsilon_t \cdot \Pr(t \in \mathcal{Q})$ and $\epsilon_t \cdot (1 - \Pr(t \in \mathcal{Q}))$, respectively. We use the closed-form expression of the Poisson binomial PDF as provided by Fernandez and Williams [FW10].

The resulting quantities are then used to compute the conditional probabilities of selecting or deselecting an erroneous column into $\mathcal{Q}$, depending on the current number of errors in this set. This then directly gives us the probability to increase or decrease the number of errors in $\mathcal{Q}$ for each swapping step.

Final Markov-chain analysis. Finally, a Markov-chain analysis akin to [BLP08] is used to estimate the number of expected iterations. Here we simply use our increase/decrease probabilities from above instead of the one given in [BLP08].

5

Analyzing the Shuffling Side-Channel Countermeasure for Lattice Signatures

The previous chapter, alongside other work such as [Gro+16; Esp+17], clearly demonstrates the need to protect Gaussian samplers against side channels. Still, countermeasures should be affordable regarding runtime. This is especially true for embedded devices and the typically already high cost of running asymmetric primitives on such platforms.

In this regard, an interesting proposal by Saarinen [Saa18] is to simply use an unprotected (or somewhat protected) sampler to generate a vector of n samples and then randomly permute this vector. Such a shuffling is easy to implement and has a low runtime overhead. However, the concrete security gains achieved by shuffling have thus far never been analyzed. As a consequence, convincing security arguments are still sorely lacking.

Contribution. In this chapter, we tackle the above problem and present an in-depth analysis of shuffling in the context of lattice-based signatures. Our analysis consists of two main parts, a side-channel analysis and a new attack on shuffling.

In the first part, we perform a side-channel attack on a Gaussian sampler implementation running on an ARM microcontroller. Our attack combines two methods. First, we recover the control flow of the sampling procedure. As many samplers, including the one used by us, require data-dependent branches and are not inherently constant runtime, this already allows to narrow down the possible samples. And second, we use templates to uniquely identify the sampled value.

While this attack is not able to identify all samples, it can recover certain values with very high confidence.

In the second part of our shuffling analysis, we present a new attack on the countermeasure. We perform an *un-shuffling*, i.e., reassign some recovered samples to the corresponding part of the signature output. After having collected enough matching pairs over multiple signatures, we can recover the private signing key. We stress that we do not attack the shuffling algorithm as such, we do not even consider its leakage in our analysis. Instead, we exploit the difference in distributions of Gaussian samples (high standard deviation) and a specific key-dependent intermediate (low standard deviation).

As we aim for a broad analysis, we evaluate this attack given several modeled side-channel adversaries. They are largely based on the previous side-channel analysis, but to test the theoretical boundaries of the countermeasure we also include an ideal attacker who can recover all samples. We also consider two different versions of the shuffling countermeasure. Our analysis shows that the simpler variant does not provide a noteworthy increase in side-channel security. Our ideal attacker succeeds using only 40 signatures. With 7 000 signatures, the modeled adversary who is closest to our side-channel analysis can also easily recover the key. However, the second shuffling version, which uses Gaussian convolution and shuffles twice, can increase the number of observed signatures required for an attack significantly. Nevertheless, with around 70 000 to 90 000 signatures (and some additional processing time for error correction) an attack is still practical and possible.

Finally, note that while we focus on BLISS, Gaussian sampling is required for many lattice-based schemes. Thus, the shuffling countermeasure and also our attack could be used for a wide range of implementations.

Outline. First, in Section 5.1, we describe the countermeasure in more detail. Then, we evaluate the side-channel leakage of a concrete Gaussian sampler implementation in Section 5.2. Using the results of this side-channel analysis, we present an attack on the shuffling countermeasure and also discuss its outcome in Section 5.3. Finally, we conclude this chapter in Section 5.4.

5.1 The Shuffling Countermeasure

Instead of protecting the sampler itself, one could also simply use an unprotected (or somewhat protected) sampler implementation to generate n samples and then randomly permute them. This breaks the connection between time of sampling and index in the signature and thus makes attacks more difficult. This shuffling countermeasure was first proposed by Roy et al. [Roy+14a], albeit in the context of lattice-based public-key encryption. More recently, Saarinen [Saa18] proposed a variant that uses shuffling multiple times (in conjunction with Gaussian convolution) for use in BLISS. For m stages, he sets $\sigma' = \sigma/\sqrt{m}$, then samples $\mathbf{y}^i \leftarrow D^n_{\sigma'}$ for $i = 1 \ldots m$ and computes $\mathbf{y} = \sum_i \mathsf{Shuffle}(\mathbf{y}^i)$. However, neither

Roy nor Saarinen provided an analysis of this countermeasure. Thus, its true effectiveness has still been unknown.

In this chapter, we will investigate the security of two versions of shuffling. First, we have a look at simple shuffling in combination with the sampler of Pöppelmann et al. [PDG14]. Second, we will analyze an instantiation of multi-stage shuffling that was concretely proposed by Saarinen [Saa18]. He suggests combining two stages of shuffling with the Gaussian-convolution parameters of Pöppelmann et al.. Below we describe both versions.

Single-Stage Shuffling: $\mathbf{y}', \mathbf{y}'' \leftarrow D^n_{\sigma'}$, $\mathbf{y} = \mathsf{Shuffle}(k\mathbf{y}' + \mathbf{y}'')$

Two-Stage Shuffling: $\mathbf{y}', \mathbf{y}'' \leftarrow D^n_{\sigma'}$, $\mathbf{y} = k \cdot \mathsf{Shuffle}(\mathbf{y}') + \mathsf{Shuffle}(\mathbf{y}'')$

5.2 A Side-Channel Attack on a Gaussian Sampler

Before evaluating the shuffling countermeasure, it is crucial to understand how much information on Gaussian samples a side-channel attacker can realistically expect. For this reason, we now present a side-channel analysis of a sampler implementation. Recall that Gaussian sampling is a random process that does not involve any keying material. Also, its output is typically used only once. Hence, we are limited to single-trace SPA-style attacks.

5.2.1 Implementation and Measurement Setup

For our experiments, we implemented the Gaussian sampling procedure proposed by Pöppelmann et al. [PDG14] (cf. Section 3.2.2) in software. The contents of all required lookup-tables are directly taken from their open-sourced BLISS FPGA implementation. Note that our analysis focuses solely on sampling from $D_{\sigma'}$ (Algorithm 3.6), i.e., we do not use any leakage stemming from the Gaussian convolution step. Whats more, for this chapter we exclusively use BLISS-I (cf. Table 3.1), thus we have a fixed $\sigma \approx 215$ and corresponding lookup-table contents.

As a target platform, we chose a Texas Instruments MSP432 (ARM Cortex-M4F) microcontroller on an MSP432P401R LaunchPad development board[1]. For pseudo-random number generation, we used the on-chip hardware AES accelerator in counter mode. While this setup is likely susceptible to DPA attacks [Jaf07; MH15], we do not use any leakage of the AES execution.

In our attack, we exploit the EM side channel. As shown in Figure 5.1, we placed a Langer RF-B 3-2 near-field probe in proximity to the external core-voltage regulation circuitry. Note that for this setup, no spatial profiling of on-chip EM leakage is required. Also, we expect the results of power measurements to be somewhat similar. For our evaluation, we use a dedicated trigger that signals the start of a sampling procedure. Real-world attackers do not have this option and need to detect the 1024 calls to Algorithm 3.6 required for sampling $\mathbf{y}_1$.

---

[1] The design files of this development board are available online [Tex].

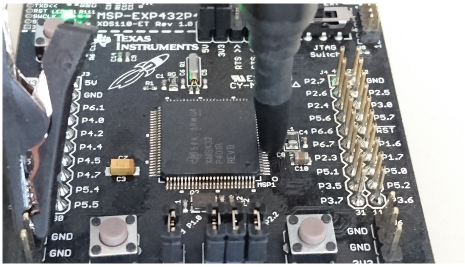


Figure 5.1: Measurement setup. The EM probe is placed directly to the left of the external core-voltage regulation circuitry.

Such adversaries can use, e.g., trace alignment in combination with the methods described in the next section.

5.2.2 Reconstructing the Control Flow

When analyzing Algorithm 3.6, it becomes obvious that the data-dependent branches offer much information on the sampled value. In fact, the return value can be uniquely determined by the first random byte r_0 and the control flow.

We recover the control flow using a trace-matching approach. For each possible conditional jump, we record a reference trace by computing the mean of multiple profiling traces at select points in time (in some cases just a single point) near the first occurrence of this branch. During the attack, we then compare these references to the attack trace by computing the mean of squared differences. Figure 5.2 illustrates that for some branches, the most information lies within a time shift of subsequent operations. In these cases, we use a single reference and match them at both locations. We then use the case with the lowest score. We repeat this matching process until the algorithm exits. The position of the respective next matching process is determined on the basis of the previously taken branches. The final branch detection then also reveals the sign of the sampled value.

With the described method, we can reconstruct the control flow with perfect accuracy. This should not be surprising, when, e.g., observing the huge trace differences illustrated in Figure 5.2.

Note that, while we use device profiling for deriving the reference traces, there exist non-profiled alternatives. An attacker could, e.g., build the references on the fly after a visual inspection of a limited number of traces. Alternatively, he could use a clustering approach for determining the branches.

5.2.3 Determining the Sampled Values via Templates

To uniquely determine the sampled value, we recover the value of r_0 using a template attack [CRR02]. For each possible control flow (up to a certain depth), we built templates for each value of r_0 that can potentially result in this flow.

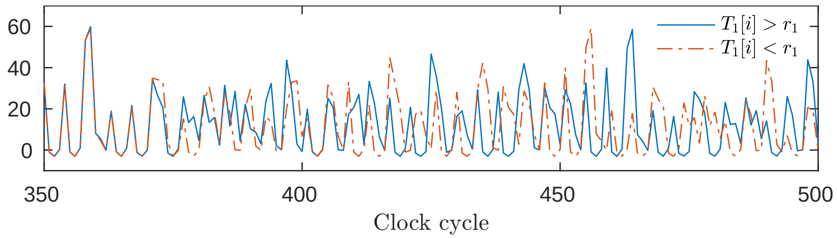
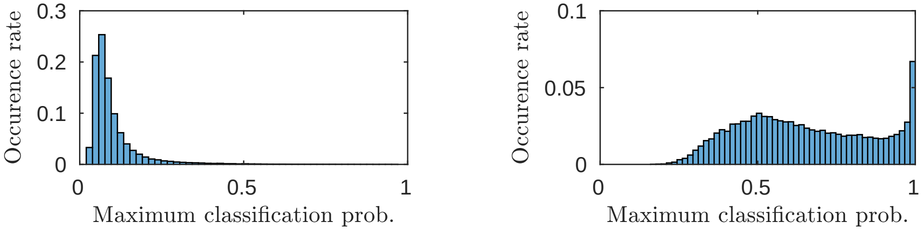


Figure 5.2: Demonstration of a timing difference stemming from a branch inside the first loop iteration. After around cycle 420, the trace for $T_1[i] > r_1$ (blue, solid) trails by 8 clock cycles.

The points-of-interest for the attack were determined using a t-test, as proposed by Gierlichs et al. [GLP06]. We limited the maximum number of used points to 8.

The outcome of the template attack is depicted in Figure 5.3. There we show a histogram of the maximum classification probabilities. In our implementation, the guide-table lookup already yields the final sample for 206 values of r_0. As seen in Figure 5.3a, we cannot determine the correct samples with high confidence in these cases. As our later analysis on the shuffling countermeasure requires such a high confidence, we have to discard these samples. This situation changes in cases that require a single comparison step in the binary-search algorithm. Figure 5.3b shows that 6.5 % of these samples can be determined with probability close to 1.

If more than a single comparison is required, then the template attack can recover the sampled value almost perfectly. The overall success rate here is 99.5 %. If we discard the 1 % of samples whose probability is lower than 0.90, then the success rate reaches 99.9 %.



(a) No comparisons

(b) 1 comparison

Figure 5.3: Results of the template attacks for no or 1 comparison

5.3 An Analysis of the Shuffling Countermeasure

In this section, we give an in-depth analysis of the shuffling countermeasure. First, we give a brief discussion on its cost. Afterwards, we present an attack that can circumvent this countermeasure, albeit at the cost of requiring a higher number of recorded signatures. We state the performance of this attack with regards to several modeled side-channel adversaries and variations of the countermeasure.

Please note that we focus on the original BLISS variant in this chapter. Thus any mention of BLISS refers to this version unless stated otherwise.

5.3.1 Cost

We evaluated the cost of shuffling by implementing the Fisher-Yates shuffling algorithm [Knu98]. When running at 48 MHz, which is the maximum for our MSP432 evaluation platform, shuffling a vector of $n = 512$ entries took 1.5 ms. For comparison, sampling an element from $D^n_{\sigma'}$, which requires 512 calls to Algorithm 3.6, needs about 2.5 ms. For creating a single signature, 4 elements of $D^n_{\sigma'}$ need to be sampled. The shuffling operation is called either 2 or 4 times, depending on whether single-stage or two-stage shuffling is used. In the latter case, the total runtime of sampling is increased by 57 %, which is still relatively little when it comes to SCA countermeasures.

5.3.2 Considered Attackers

In order to allow a broad analysis of the shuffling countermeasure and to achieve easier reproducibility, we do not directly use the outcome of the attack described in Section 5.2. Instead, we use the results as a basis to model three side-channel adversaries. Each one is based on a different assumption on his capabilities. Note that all following descriptions are in the context of sampling from the "small" $D_{\sigma'}$ and thus Algorithm 3.6, which is called 2048 times during signature generation. We do not use any leakage from the multiplication with k, the addition needed for Gaussian convolution, and even the shuffling algorithm itself. We do so to keep the analysis as generic and implementation-independent as possible.

A1 - perfect SCA adversary. This attacker can recover all generated samples. We use this adversary to evaluate the theoretical limits of the shuffling countermeasure.

A2 - profiled SCA adversary. This attacker can profile the device and perform a template attack. We assume that the attacker can correctly determine the entire control flow and is able to correctly classify all samples which required at least 2 comparisons in the binary-search step. For the analysis, we make a further simplification and only use samples with absolute above a certain threshold. This threshold is set so that all samples larger than it require at least 2 comparisons. All samples at and below the threshold are considered to be unknown.

A3 - non-profiled SCA adversary. This attacker is not able to profile and thus cannot perform a template attack. However, he is still able to reconstruct the control flow. All samples which are not uniquely determined by the control flow are considered to be unknown.

Adversary A2 is closest to the side-channel analysis given in the previous section. However, he does not use any potentially classified samples which used only a single comparison (cf. Figure 5.3b) or the small portion of samples requiring 2 comparisons but being below the threshold. In return, we also ignore the tiny error probability and assume that all reconstructed samples are correct.

For our particular BLISS parameter set and sampler implementation, we have the following concrete implications. For A2, the mentioned threshold is 47, i.e., we say that the adversary can correctly classify all samples with an absolute value larger than 47. Approximately 1.5 % of the samples from $D_{\sigma'}$ meet this restriction. The adversary A3 can correctly classify all samples with an absolute value larger than 54, which amounts to only 0.53 % of all samples.

5.3.3 Attack without Shuffling

For key recovery, we use the relation also exploited by Groot Bruinderink et al. (cf. Section 3.3.2). We gather equations of form $z_{ji} = y_{ji} + (-1)^{b_j} \langle \mathbf{s}_1, \mathbf{c}_{ji} \rangle$ and then solve the resulting linear system. We do not consider error correction and require that these equations are correct.

If the entire $\mathbf{y}_1$ is known, which is the case for adversary A1, and no shuffling is used, then key recovery is trivial and requires only a single signature. $\mathbf{s}_1$ (or $-\mathbf{s}_1$) can be computed by solving the linear system given as $\mathbf{z}_1 - \mathbf{y}_1 = (-1)^b \mathbf{s}_1 \mathbf{c}$ for any value of b. Attackers A2 and A3 require multiple signatures in order to recover the key, even in the non-shuffled scenario. As our sampling procedure combines two samples $y', y'' \leftarrow D_{\sigma'}$ to $y = ky' + y''$, we can only recover samples y where the side-channel information reveals both y' and y''. Hence, A2 can recover a portion of $0.015^2 \approx 2.2 \cdot 10^{-4}$ of all samples, whereas for A3 this quantity decreases to $2.2 \cdot 10^{-5}$.

We need to combine $n = 512$ equations of form $z_{ji} = y_{ji} + (-1)^{b_j} \langle \mathbf{s}_1, \mathbf{c}_{ji} \rangle$ into a system. If the b_j are recoverable by using side-channel information and the original BLISS variant is used, then this system can trivially be solved for $\mathbf{s}_1$. If the b_j are not known, or BLISS-B is used (where $\mathbf{s}_1$ is multiplied with the unknown $\mathbf{c}'$), then the approach presented in Chapter 4 can be used. That is, the system can be solved over GF(2) to learn the parity of $\mathbf{s}_1$, which is then further used to mount a lattice attack resulting in the full $\mathbf{s}_1$.

The expected number of signatures required to gather $n = 512$ classified samples and corresponding signature values is roughly 4 400 for A2 and 36 000 for A3. Note that in this non-shuffled scenario, the differences between A2 and the SCA from Section 5.2, i.e., not using all classifiable samples, have a significant impact. Including them reduces the attack cost to only around 1 000 signatures.

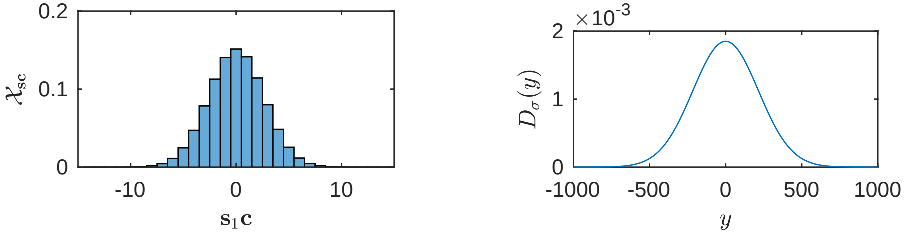


Figure 5.4: Comparison of the coefficient-wise distribution of $\mathbf{s}_1\mathbf{c}$ ($\mathcal{X}_{\mathbf{sc}}$) and $\mathbf{y}$ (D_σ^n)

5.3.4 An Attack on Shuffling - Basic Concept

If the elements of $\mathbf{y}_1$ are shuffled after sampling, then the above attack is not directly applicable. To still use it, we first need to do an *un-shuffling*, i.e., we need to re-assign recovered Gaussian samples to their respective index in the signature and thus to the correct $z_i \in \mathbf{z}_1$.

We do that by exploiting the differing (coefficient-wise) distributions of $\mathbf{s}_1\mathbf{c}$ and $\mathbf{y}_1$; they are shown in Figure 5.4. The distribution of $\mathbf{s}_1\mathbf{c}$, which we denote with $\mathcal{X}_{\mathbf{sc}}$, was estimated using a histogram approach, whereas $\mathbf{y}_1$ follows D_σ^n. Observe that the standard deviation of $\mathbf{y}_1$ is much larger than that of $\mathbf{s}_1\mathbf{c}$. Thus, we can say that $\mathbf{z}_1 \approx \mathbf{y}_1$.

We use this observation as the basis of our attack. If we know one particular coefficient y of $\mathbf{y}_1$ but not its position due to shuffling, then we can test all coefficients of the public $\mathbf{z}_1$ for proximity to y. If only a single $z_i \in \mathbf{z}_1$ is "close" to y, then we can assign y to the position of z_i and compute $z_i - y$ to retrieve the value of $(-1)^b\langle \mathbf{s}_1, \mathbf{c}_i\rangle$. As actual metric for closeness, we use $\mathcal{X}_{\mathbf{sc}}(z_i - y)$. Observe that this approach is expected to succeed mostly for large absolute values of y and thus z_i, i.e., in the tail of D_σ. Due to the high dimension $n = 512$, there will be many similar values of y and z_i near the center; thus a unique assignment will not be possible in those cases.

5.3.5 Attack Details

The previous description of our attack is relatively informal; we now give a more in-depth explanation. For now, consider the case of single-stage shuffling, we adapt the approach to the two-stage variant later on.

Given two values z_i and y, we define $z_i \sim y$ as the event that z_i and y belong to the same index i in the signature. Without considering knowledge of other processed values, we have a likelihood $\Pr(z_i \sim y) = \mathcal{X}_{\mathbf{sc}}(z_i - y)$.

When now given the public $\mathbf{z}_1$ and a single sample y of $\mathbf{y}_1$, we can compute, for each $z_i \in \mathbf{z}_1$, $\Pr(z_i \sim y|\mathbf{z}_1)$. We do that by using Bayes' theorem with uniform prior, i.e., $\Pr(z_i \sim y|\mathbf{z}_1) = \Pr(z_i - y)/\sum_{z_j \in \mathbf{z}_1} \Pr(z_j - y)$. Analogously, for a single z_i and a fully reconstructed but shuffled $\mathbf{y}_1$, we can compute $\Pr(z_i \sim y_j|\mathbf{y}_1)$.

We perform this analysis on every possible combination of y and z_i. Thus, we compute a likelihood matrix $\mathbf{L} \in (n \times n)$, with $\mathbf{L}_{i,j} = \mathcal{X}_{\mathbf{sc}}(z_i - y_j)$. Afterwards, we apply the Bayesian step to both the columns and the rows of this matrix in order to derive the aforementioned conditional probabilities. Then, we combine both normalized matrices by taking their maximum, i.e., we set $\Pr(z_i \sim y_j) = \max(\Pr(z_i \sim y_j|\mathbf{z}_1), \Pr(z_i \sim y_j|\mathbf{y}_1))$. In other words, we both search for z_i that fit to only one y, and y that fit to only one z_i. Finally, for each $z_i \in \mathbf{z}_1$, we pick the most likely y as $\operatorname{argmax}_{y_j} \Pr(z_i \sim y_j)$. This shuffling analysis is repeated for each recorded signature.

The straight-forward key-recovery algorithm described in Section 5.3.3 just involves linear algebra and hence requires errorless information. Thus, we keep only pairs (z_i, y) that match with very high probability; we set the threshold to 0.99. Error correction can be achieved by employing the methods presented in Chapter 4; we will discuss this later for the two-stage approach.

Merging equal $\mathbf{y}$. For key recovery, we compute $z_i - y$ for each recovered pair (z_i, y). Here, the actual *index* of y is irrelevant, only the *value* of y needs to be correct. Consequently, if $\mathbf{y}_1$ contains multiple copies of the same value, then they can be treated as a single entity.

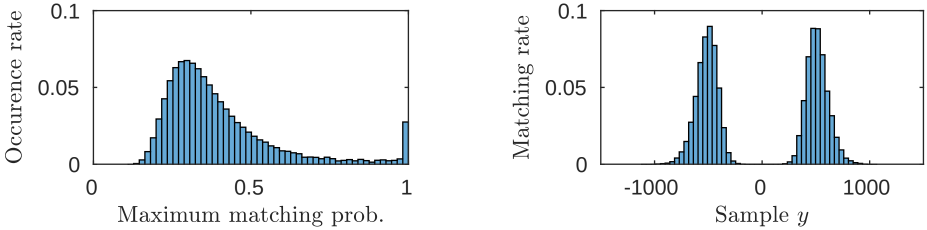
We use this observation as follows. We create a vector $\mathbf{u}$ which contains the unique elements of $\mathbf{y}_1$. We then compute $\Pr(z_i \sim u_j|\mathbf{u})$. For that, we use the number of times each u_j appears in $\mathbf{u}$ as prior probabilities (instead of the uniform distribution). For $\mathbf{y} \leftarrow D_\sigma$, the average number of unique elements in $\mathbf{y}$ is 377. For $\mathbf{y}' \leftarrow D_{\sigma'}$ only 92 elements are unique on average. Especially in the latter case, the merging of equal y' increases the rate of matches and also decreases the computation time of the subsequent analysis. From now on, we will always use this optimization implicitly.

Note that merging equal values of $\mathbf{z}_1$ is not useful. As already hinted by always using subscripts, each z_i is coupled to one specific $\mathbf{c}_i$, i.e., a negacyclic rotation of the signature part $\mathbf{c}$.

Results on single-stage shuffling. We evaluated our described attack against (single-stage) shuffling by running it with 2^{20} signatures. The results for A1 are shown in Figure 5.5. 2.5 % of all samples match with a probability of at least 0.99. With this number, only 40 signatures are required to gather $n = 512$ equations. As expected, the successfully matched y are in the tail of D'_σ (Figure 5.5b).

For A2 and A3, we do not know the entire $\mathbf{y}_1$ and so did not compute $\Pr(z_i \sim y_j|\mathbf{y}_1)$. When only using $\Pr(z_i \sim y|\mathbf{z}_1)$, we can match a proportion of $1.4 \cdot 10^{-4}$ (A2) and $2.2 \cdot 10^{-5}$ (A3) of all samples. This translates to requiring 7 000 and 46 000 signatures, respectively. The number of expected errors in $n = 512$ equations is well below 1 for all considered adversaries.

When compared to the signature requirements without shuffling, one can observe only a marginal increase. All numbers are well low enough to be practical, thus shuffling once is not an effective countermeasure.



(a) Histogram of maximum matching probabilities

(b) Histogram of the value of successfully matched y

Figure 5.5: Result for the attack on single-stage shuffling, attacker A1

5.3.6 Adaptation to Two-Stage Shuffling

For two-stage shuffling, the $\mathbf{y}', \mathbf{y}''$ are independently permuted. Thus, we cannot compute any elements of $\mathbf{y}_1$ in a straight-forward manner which makes the above attack not directly applicable. A similar one, however, is still possible; we now state the required modifications. As the sampling (and shuffling) process proceeds in two steps, we also adapt a two-stage approach in the attack.

Assume we are given $\mathbf{z}_1$ and the *shuffled* $\mathbf{y}', \mathbf{y}''$, with $\mathbf{y}_1 = k\mathbf{y}' + \mathbf{y}''$ and hence $\mathbf{z}_1 = k\mathbf{y}' + \mathbf{y}'' + (-1)^b\mathbf{s}_1\mathbf{c}$. We first aim at finding matching pairs for elements of $\mathbf{z}_1$ and $\mathbf{y}'$. Afterwards, for each pair (z_i, y'), we compute $z_i - ky'$ and then match this difference with the elements of the second vector $\mathbf{y}''$. We now explain the details of this process.

First stage. The first part differs from the previous attack; we now test the proximity of elements of $\mathbf{z}_1$ to those of $k\mathbf{y}'$. As $\mathbf{z}_1 - k\mathbf{y}' = \mathbf{y}'' + (-1)^b\mathbf{s}_1\mathbf{c}$, we cannot test proximity with regards to $\mathcal{X}_{\mathbf{sc}}$. Instead, we could use the distribution of an $x = x_1 + x_2$, with $x_1 \leftarrow D_{\sigma'}$ and $x_2 \leftarrow \mathcal{X}_{\mathbf{sc}}$. We denote it as $\mathcal{X}_{\mathbf{sc}+D_{\sigma'}}$. However, as the attacker has (at least partial) knowledge on $\mathbf{y}''$, this would be suboptimal. Hence, we set (with some abuse of notation) $x_2 \leftarrow \mathbf{y}''$, i.e., randomly chosen elements from $\mathbf{y}''$. We call the resulting distribution $\mathcal{X}_{\mathbf{sc}+\mathbf{y}''}$ and use it to fill our likelihood matrix $\mathbf{L}$ with $\mathbf{L}_{i,j} = \mathcal{X}_{\mathbf{sc}+\mathbf{y}''}(z_i - ky'_j)$. The remainder of the analysis, i.e., the Bayesian steps and picking the maximum, are then the same. Finally, all samples matched with probability greater than 0.99 are fed to the second stage of the recovery.

For A2 and A3, we require additional modifications. First, we cannot compute $\mathcal{X}_{\mathbf{sc}+\mathbf{y}''}$, as $\mathbf{y}''$ is only partially known. Instead, we construct a hybrid distribution that merges $D_{\sigma'}$ (for all unknown samples up to the threshold of 47 and 54, respectively) and the known samples of $\mathbf{y}''$. Then, unlike in the single-stage attack, we would also like to compute (or rather estimate) $\Pr(z_i \sim y'_j | \mathbf{y}')$ despite not having the full $\mathbf{y}'$. We do so by introducing a dummy sample y'_d, which represents all (unknown) samples below the model threshold. Thus, we test if z_i matches with any of the known y'_j or with any element below the threshold. We set the likelihood of y'_d as in (5.1), the remaining steps are then equivalent to those of A1.

$$\Pr(z_i \sim y'_d) = \sum_{y=-\text{threshold}}^{\text{threshold}} \mathcal{X}_{\mathbf{sc}+D_{\sigma'}}(z_i - y) \tag{5.1}$$

Second stage. In the second stage, we test each pair (z_i, y'_i) found in the previous stage with the elements of $\mathbf{y}''$. We do so by computing $z_i - ky'_i$ and then testing for proximity to the elements of $\mathbf{y}''$ with regards to $\mathcal{X}_{\mathbf{sc}}$.

Even for A1, the expected number of matched pairs per signature in the first stage is relatively small. Thus, we cannot compute $\Pr((z_i - ky'_i) \sim y''_j | (\mathbf{z}_1 - k\mathbf{y}'))$ and are left with $\Pr((z_i - ky'_i) \sim y''_j | \mathbf{y}'')$. Like in the first stage, A2 and A3 only have partial knowledge of $\mathbf{y}''$. We use the same trick as above and introduce a dummy sample y''_d representing all elements below the modeled threshold of 47 and 55, respectively. Here we use (5.2) and then again perform the Bayesian step and filtering of the most probable matches.

$$\Pr((z_i - ky'_i) \sim y''_d) = \sum_{y=-\text{threshold}}^{\text{threshold}} \mathcal{X}_{\mathbf{sc}}(z_i - ky'_i - y) \tag{5.2}$$

Results on two-stage shuffling. Like earlier, we evaluated our described attack against two-stage shuffling by running it with 2^{20} signatures. For our ideal adversary A1, we can match 0.26 % of samples in the first stage (with probability greater than 0.99). Out of the found pairs, 0.15 % can also be matched in the second stage. This results in requiring 260 000 signatures in order to find $n = 512$ equations.

Interestingly, the losses incurred by the restrictions of A2 are relatively small. We can match 0.25 % in the first and 0.15 % of samples in the second stage. With 285 000, the number of required signatures is virtually identical to the previous case. As to be expected, A3 performs slightly worse. The matching rates decrease to 0.18 % and 0.10 %, respectively. This results in requiring 575 000 signatures

Error Correction and BLISS-B. Up until now, we required that all reassignments are correct. This, of course, requires aggressive filtering and in turn a high number of observed signature generations. When using the decoding-based error correction discussed in Section 4.2.2 and Section 4.6, samples with somewhat lower confidence can also be admitted, thus allowing a trade-off between computational effort (for decoding) and number of side-channel measurements.

Even without error correction, shuffling once cannot significantly increase security. For this reason, we evaluate error correction only for the two-stage approach. For our performance evaluation, we used the same setup as in Section 4.3.1.

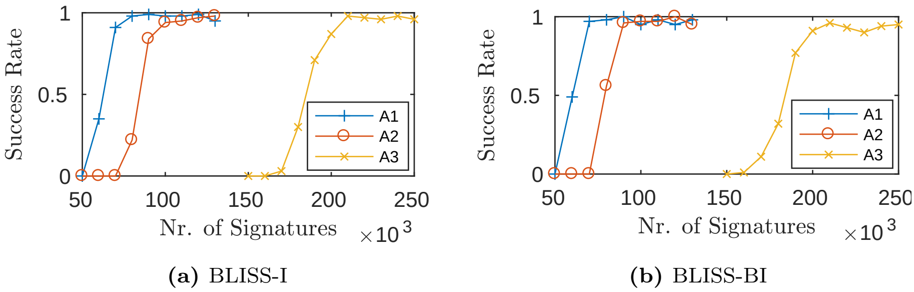
Recall that the attack of Chapter 4 solves a system over GF(2) and uses probabilities in the decoding process. This can be used for unshuffling, as now we only require knowledge of $\Pr(z_{ji} - y \equiv 0 \mod 2)$ instead of $\Pr(z_{ji} - y)$ Assume for now that the adversary is in possession of the full, but shuffled, $\mathbf{y}$. This $\mathbf{y}$ contains two large elements of values, for example values 1498 and 1502, and there is one signature coefficient that is large, for example one element $z = 1500$. Only one of the two elements of y belong to this z, and both elements have a probability of 50 %. However, the probability that the difference of $z - y$ is even is $\Pr(z_i - y \equiv 0 \bmod 2) = 1$. In general, if given full (or parts of) shuffled $\mathbf{y}$, all these probabilities can be easily computed as:

$$\Pr(z_i - y \equiv 0 \bmod 2) = \sum_{y_j \in \mathbf{y}: z_i - y_j \equiv 0 \bmod 2} \Pr(z_i \sim y_j)$$

Thus, the error probability for computation over GF(2) at least as small as over $\mathbb{Z}$, and in most cases significantly smaller. Thus, fewer signatures are required to gather enough samples with low-enough error probability.

Figure 5.6 finally gives the outcome of this process. We give results for BLISS-I (Figure 5.6a) and BLISS-BI (Figure 5.6b) separately, as the introduction of the GreedySC algorithm leads to different results and slightly better performance for BLISS-B[2]. For A1, one needs approximately 70 000 signatures to achieve a success rate larger than 0.9. A2 needs 90 000 signatures, A3 200 000. Thus, error correction can cut the number of required signatures by a factor of around 3.

Finally, note that this error-correction approach could also be used to relax the restriction on errorless samples. That is, samples that were not correctly recovered by the side-channel analysis (Section 5.2) would not break the attack. The final error probabilities could be computed as the product of reassignment probability and the probabilities as output by the template attack. We do not follow this approach here and stay with modeled adversaries.



(a) BLISS-I

(b) BLISS-BI

Figure 5.6: Success rate of LPN decoding for the attack on shuffling

Discussion. Unlike single-stage shuffling, two-stage shuffling can increase the number of required signatures for an attack significantly and thus can be considered an effective countermeasure, at least against the presented attack. However,

---

[2] GreedySC aims at minimizing the norm of $\mathbf{s} \cdot \mathbf{c}'$; thus the difference $z - y$ is, on average, smaller. The attack on shuffling benefits from this, as it tests this difference.

while the given numbers are high, they are still within reach for a dedicated attacker (especially when using error correction).

The large difference between the shuffling variants could be explained as follows. For single-stage shuffling, we tested elements from D_σ, with $\sigma \approx 215$, against a distance of $\mathcal{X}_{\mathbf{sc}}$. For the two-stage attack, the ratio of the matched standard deviations is much smaller. In the second stage, for instance, we match elements from $D_{\sigma'}$, with $\sigma' \approx 19.5$, against the same $\mathcal{X}_{\mathbf{sc}}$. As a result, the matchable samples are even further out in the tail of $D_{\sigma'}$ and so less frequent than was the case for single-stage shuffling. This also explains the compared to A1 maybe surprisingly small losses of A2 and A3. These adversaries can only recover a small number of samples, but the ones they can find are already in tail $D_{\sigma'}$ and thus more likely to be usable. Obviously, the attack degradation caused by the smaller difference of deviations is amplified by requiring two matching steps. So, we can only rewind shuffling for indices i where *both* y'_i and y''_i are outliers.

5.4 Conclusion

Our work shows that shuffling is, at least if done correctly, an effective and cheap countermeasure in the context of lattice-based signatures. However, while it can drastically increase the attack complexity, relying on two-stage shuffling alone might not be enough to protect against attacks on Gaussian samplers. The reported signature requirements for attacks are still practical, especially when using the error correction techniques from Chapter 4. In this regard, recall that we did not use leakage from either multiplication with k and addition of two samples in the Gaussian convolution, the shuffling itself, or from the PRNG. This information can be used to further decrease the number of required signatures. Thus, a mix of countermeasures and reducing the leakage of the sampling algorithm itself is necessary for sufficient protection. Increasing the number of sampling (and shuffling) stages as well as the use of different convolution parameters might also offer better protection.

6

Differential Fault Attacks on Deterministic Lattice Signatures

The previous chapters show that the advantages of using the discrete Gaussian distribution, namely small signature and key sizes, come at the cost of challenging implementation and protection thereof. For this very reason, more recent proposals such as Dilithium [Lyu+17] and qTESLA [Bin+17a] (both submitted to the NIST call) use samples from the uniform distribution instead.

In addition, both these schemes also add measures aimed at countering implementation errors. They make use of the Fiat-Shamir with Aborts framework of Lyubashevsky [Lyu09]. A well known caveat of signature schemes using the (plain) Fiat-Shamir transform, such as ECDSA, is that signing requires a nonce. Reusing that nonce 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 [Bye+10]. 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 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 [Ber+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

---

[1] Following the initial publication of [GP18], which is the basis for this chapter, 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 this chapter, and discuss the countermeasure and update in Section 6.5

computational fault in one of the signature computations. This results in different signatures using the same nonce and thus in a key recovery. In fact, recent work [BP16; Amb+18; Pod+17; SB18] explored the vulnerability of elliptic-curve signatures against such differential fault attacks, including Rowhammer-induced faults [Pod+17].

The vulnerability of lattice-based deterministic signatures, however, is less clear. The possibility of such differential attacks was already hinted at [Lyu+17; Bin+17a], yet many questions remain open. Concretely, the abortion technique introduced by Lyubashevsky 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.

Contribution. In this chapter, we show the applicability of differential fault attacks on deterministic lattice-based signature schemes. We mainly 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.

Dilithium and qTESLA are somewhat similar to BLISS, all these schemes are based on Fiat-Shamir with Aborts and compute a signature vector of form $\mathbf{z} = \mathbf{y} + c\mathbf{s}$ out of a challenge c, a secret element $\mathbf{s}$, and a nonce/noise $\mathbf{y}$. Unlike BLISS, however, nonce generation is deterministic in Dilithium and qTESLA. Our 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 [Amb+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 sign 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. Still, 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 tight variant of the security proof of Dilithium.

Outline. In Section 6.1, we give some additional background required for this chapter. In Section 6.2, we explore the possibilities of differential fault attacks on Dilithium. In Section 6.3, we show how to modify the signature algorithm such that the secret key element extracted by our attacks suffices to compute valid signatures for any message. In Section 6.4 we verify the vulnerabilities with real experiments on an ARM Cortex-M4 microcontroller. In Section 6.5 we conclude this chapter with a discussion on countermeasures.

6.1 Background

In this section, we introduce both the Dilithium and the qTESLA signature schemes. We also discuss previous attacks on deterministic elliptic-curve signatures, such as EdDSA and deterministic ECDSA.

Changed and additional notation. Please note that in this chapter we use slightly different notation for polynomials and their coefficients. That is, we now use plain letters, e,g., f, for polynomials in $\mathcal{R}_q$. Bold-face is now reserved for vectors $\mathbf{a}$ and matrices $\mathbf{A}$ comprised of multiple polynomials. This change in notation is done to keep consistency with the original descriptions of qTESLA and Dilithium. Also, the introduction of yet another dedicated notation for vectors and matrices is likely more distracting than this change.

Apart from the standard ℓ_2 norm, Dilithium also makes use of the ℓ_∞ norm defined as $\|w\|_\infty = \max\{|w_0|, |w_1|, \ldots, |w_{n-1}|\}$, 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. S_η denotes the subset of $\mathcal{R}_q$ that includes all elements w that satisfy $\|w\|_\infty \leq \eta$.

6.1.1 Deterministic Lattice Signatures

We now describe the two deterministic lattice-based signature schemes Dilithium [Lyu+17] and qTESLA [Bin+17a], 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 6.1 Dilithium Key Generation

```
Output: Keypair $(pk, sk)$
$\rho \leftarrow \{0,1\}^{256}, K \leftarrow \{0,1\}^{256}$
$(\mathbf{s}_1, \mathbf{s}_2) \leftarrow S_\eta^l \times S_\eta^k$
$\mathbf{A} \in \mathcal{R}_q^{k \times \ell} = \mathsf{ExpandA}(\rho)$
$\mathbf{t} = \mathbf{A}\mathbf{s}_1 + \mathbf{s}_2$
$(\mathbf{t}_1, \mathbf{t}_0) = \mathsf{Power2Round}_d(\mathbf{t})$
$tr \in \{0,1\}^{384} = \mathsf{CRH}(\rho || \mathbf{t}_1)$
return $(pk = (\rho, \mathbf{t}_1), sk = (\rho, K, tr, \mathbf{s}_1, \mathbf{s}_2, \mathbf{t}_0))$
```

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 6.1. 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.

Just like BLISS, Dilithium is based on the Fiat-Shamir with Aborts Framework [Lyu09]. The structure of rejection sampling can be easily seen in Algorithm 6.2, which shows a slightly simplified[2] version of the Dilithium signature algorithm. The comments in Algorithm 6.2 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.

---

[2] Some additional checks and constant subroutine arguments are omitted.

Algorithm 6.2 Dilithium Sign (simplified[2])

```
Input: Message $M$, private key $sk = (\rho, K, tr, \mathbf{s}_1, \mathbf{s}_2, \mathbf{t}_0)$
Output: Signature $\sigma = (\mathbf{z}, \mathbf{h}, c)$
$\mathbf{A} \in \mathcal{R}_q^{k \times \ell} = \mathsf{ExpandA}(\rho)$                                  ▷ $\mathrm{fA}_\rho$, $\mathrm{fA}_\mathrm{E}$
$\mu \in \{0,1\}^{384} = \mathsf{CRH}(tr||M)$
$\kappa = 0, (\mathbf{z}, \mathbf{h}) = \perp$
while $(\mathbf{z}, \mathbf{h}) = \perp$ do
    $\mathbf{y} \in S_{\gamma_1 - 1}^{l} = \mathsf{DeterministicSample}(K||\mu||\kappa)$          ▷ fY
    $\mathbf{w} = \mathbf{A}\mathbf{y}$                                                    ▷ 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 $\|\mathbf{z}\|_\infty \geq \gamma_1 - \beta$ or $\|\mathbf{r}_0\|_\infty \geq \gamma_2 - \beta$ 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 6.3 Dilithium Verify (simplified[2])

```
Input: Public key $pk = (\rho, \mathbf{t}_1)$, message $M$, signature $\sigma = (\mathbf{z}, \mathbf{h}, c)$
$\mathbf{A} \in R_q^{k \times \ell} = \mathsf{ExpandA}(\rho)$
$\mu \in \{0,1\}^{384} = \mathsf{CRH}(\mathsf{CRH}(\rho||\mathbf{t}_1)||M)$
$\mathbf{w}_1 = \mathsf{UseHint}(\mathbf{h}, \mathbf{A}\mathbf{z} - c\mathbf{t}_1)$
accept iff $c = \mathsf{H}(\mu||\mathbf{w}_1)$
```

Note that all operations in Algorithm 6.2 are completely deterministic and thus generate 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 6.3). All specified parameter sets for Dilithium are given in Table 6.1. Throughout this chapter we use the recommended Dilithium parameter set III. It claims 128 bits of security against a quantum adversary. The other sets mainly differ in the used (k, ℓ), so our later attacks are possible for all proposed sets.

---

[3] A previous Dilithium description [Duc+17] is probabilistic, but did not include a proof in the QROM.

Table 6.1: Dilithium Parameter Sets

	I weak	II medium	III recommended	IV 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

qTESLA. Structurally, the signature scheme qTESLA [Bin+17a] is very similar to Dilithium. It also uses a variant of the Fiat-Shamir with Aborts framework and is deterministic. Unlike Dilithium, its proof in the QROM model [Alk+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 $n \geq 1024$.

We now restate the qTESLA algorithms, but please note that we give the version originally submitted to the NIST call. Following the initial publication of [GP18], in an update of the qTESLA specification a countermeasure was added which makes qTESLA probabilistic again. Hence the attacks presented in this chapter are no longer applicable. We will discuss the countermeasure in Section 6.5. As [GP18] is a potential reason for these changes and is cited in the update, we still describe the attacks.

In Algorithms 6.4, 6.5, and 6.6, we give slightly simplified versions of key generation, signing, and verification, respectively. The proposed parameter sets are given in Table 6.2. Note the similarity of qTESLA and Dilithium, we highlight this by stating the corresponding variable and function names in Table 6.3. 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 $n \geq 1024$. 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 6.4 qTESLA Key Generation

```
**Output:** Keypair $(pk, sk)$
$\mathrm{seed}_a \leftarrow \{0,1\}^{256}, \mathrm{seed}_y \leftarrow \{0,1\}^{256}$
$a \in \mathcal{R}_q = \mathsf{GenA}(\mathrm{seed}_a)$
**repeat**
    $s \in \mathcal{R}_q \leftarrow D_\sigma, e \in \mathcal{R}_q \leftarrow D_\sigma$
**while** $s$ and $e$ do not fulfill certain criteria
$t = as + e \bmod q$
**return** $(pk = (\mathrm{seed}_a, t), sk = (s, e, \mathrm{seed}_y, \mathrm{seed}_a)$
```

Algorithm 6.5 qTESLA Sign (simplified)

```
**Input:** Message $M$, private key $sk = (s, e, \mathrm{seed}_y, \mathrm{seed}_a)$
**Output:** Signature $\sigma = (c, z)$
$a \in \mathcal{R}_q = \mathsf{GenA}(\mathrm{seed}_a)$
counter $= 0$
rand $= \mathsf{PRF}_1(\mathrm{seed}_y, M)$
**repeat**
    $y = \mathsf{PRF}_2(\mathrm{rand}, \mathrm{counter})$
    $v = ay \bmod q$
    $c = \mathsf{H}(\mathsf{Round}(v), M)$
    $z = y + sc$
    counter $=$ counter $+ 1$
**while** $\mathsf{Reject}(z, v, c, sk)$
**return** $\sigma = (c, z)$
```

Algorithm 6.6 qTESLA Verify (simplified)

```
**Input:** Public key $pk = (\mathrm{seed}_a, t)$, message $M$, signature $\sigma = (c, z))$
$a \in \mathcal{R}_q = \mathsf{GenA}(\mathrm{seed}_a)$
$w = az - tc \bmod q$
**return** $c = \mathsf{H}(\mathsf{Round}(w), M)$
```

Table 6.2: qTESLA Parameter Sets

	qTESLA-128	qTESLA-192	qTESLA-256
n	1024	2048	2048
q	8058881	12681217	27627521
σ	8.5	8.5	8.5
weight(c)	36	50	72
B	$2^{20}-1$	$2^{21}-1$	$2^{2}-1$
d	21	22	23
L_E	798	1117	1534
L_S	758	1138	1516

Table 6.3: 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	v

Functions:

Dilithium	ExpandA	CRH	DeterministicSample	HighBits
qTESLA	GenA	PRF_1	PRF_2	Round

6.1.2 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; Amb+18; Pod+17; 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. [Pod+17] exploit the fact that the message is hashed twice in EdDSA. By manipulating the message in between these hashing operations with Rowhammer, they can induce a nonce reuse and thus recover the key. Ambrose et al. [Amb+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.

6.2 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 general ℓ, 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. We assume the possibility of injection 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 [Amb+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 [Esp+16].

6.2.1 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 $\mathbf{z}' = \mathbf{y} + c'\mathbf{s}_1$.

Table 6.4: Fault scenarios discussed in this chapter

Name	Section	Description
fA_ρ	6.2.4	Corrupt ρ during import of *sk*
fA_E	6.2.4	Random fault in expansion $\mathbf{A} = \mathsf{ExpandA}(\rho)$
fY	6.2.5	Random fault in sampling $\mathbf{y} = \mathsf{DeterministicSample}(\cdot)$
fW	6.2.3	Random fault in polynomial multiplication $\mathbf{w} = \mathbf{A}\mathbf{y}$
fH	6.2.2	Random fault in call to H

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 $\Delta\mathbf{z} = \Delta\mathbf{y} + \Delta c \cdot \mathbf{s}_1 = \Delta c \cdot \mathbf{s}_1$. 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 6.2, the loop counter κ is input to $\mathsf{DeterministicSample}$. 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 6.2 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 overview of the scenarios is given in Table 6.4, they are listed in order of appearance in Algorithm 6.2. The order of description will be different.

6.2.2 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 $c \in B_{60} = \mathsf{H}(\mu || \mathbf{w}_1)$. 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 6.4.1 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 6.2.1.

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 6.1, 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 rejection 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 %.

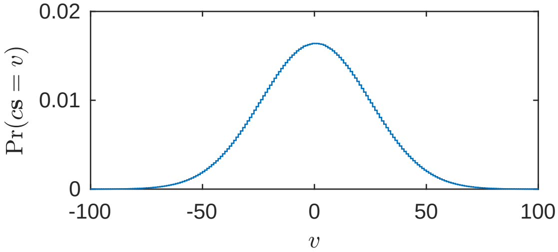


Figure 6.1: Coefficient-wise probability distribution of $c\mathbf{s}$

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. We experimentally verified that 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.

6.2.3 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 $\mathsf{DeterministicSample}$, 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). Still, the runtime of multiplication is higher than that of hashing, thus 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.

6.2.4 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 6.2, this is done before entering the abort loop and is thus always executed at the same time. Furthermore, $\mathsf{ExpandA}$ is a major contributor to overall runtime (cf. Section 6.4.2). 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 $\mathsf{ExpandA}$ for every singing 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 6.7, which sketches the method for expanding 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 $\mathbf{A}\mathbf{y}$ 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 just 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 6.5 we discuss this in more detail.

Algorithm 6.7 ExpandA(ρ)

```
Input: Seed $\rho$
Output: uniform $\mathbf{A} \in \mathcal{R}_q^{k \times \ell}$
for $i = 0 \dots k-1$ do
    for $j = 0 \dots \ell - 1$ do
        $\mathbf{A}_{i,j} = \mathsf{SamplePoly}(\rho || i || j)$
return A

function SamplePoly($s$)
    $t \in \{0,1\}^{5 \cdot \mathrm{SHAKErate}} = \mathsf{SHAKE128}(s)$
    $u = 0$
    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 $a$
```

6.2.5 Scenario: fY

So far, we have only discussed fault-induced nonce-reuse scenarios, i.e. the case where $\mathbf{y}' = \mathbf{y}$. In [GP18], a method to also exploit *partial* nonce-reuse is presented. It uses the observation that in DeterministicSample (Algorithm 6.8) the elements in $\mathbf{y}$ are sampled sequentially, and that injecting a fault in just one of the calls to SHAKE does not affect the $\ell - 1$ other nonce polynomials. Whats more, when faulting one of the later invocations of the Keccak-f permutations inside SHAKE, only last few coefficients of the affected y' are different, i.e., most but not the entire nonce is reused. As y' is *close* to y, one can use techniques originally proposed for loop-abort faults [Esp+16] and cast key recovery to solving a closest-vector problem in a lattice.

An important advantage of this scenario is that even faulted signatures will be valid, thus the attack is not detectable with a subsequent signature verification. This attack scenario was developed by Groot Bruinderink, which is why we do not give a detailed description of this approach here and instead refer to [GP18].

6.2.6 Summary of Scenarios

We now give a summary of the different fault scenarios. In Table 6.5 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.

fH is the most intuitive scenario and also achieves the highest success probability. However, it is also the smallest of all targets (cf. Section 6.4.2). The lowest success probability is achieved for fA$_\rho$, yet with the huge advantage of

Algorithm 6.8 $\mathsf{DeterministicSample}_{\gamma_1-1}(s)$ (simplified[4])

```
Input: Seed $s$
Output: $\mathbf{y} \in S^{\ell}_{\gamma_1-1}$
for $u = 0 \ldots \ell - 1$ do                          ▷ Sample $\ell$ nonce polynomials
    $t \in \{0,1\}^{5 \cdot \mathrm{SHAKErate}} = \mathsf{SHAKE256}(s||u)$
    $v = 0$
    while $v < n$ do                                     ▷ Rejection sampling
        $r =$ next $2\lceil \log_2 \gamma_1 \rceil$ bits of $t$
        if $r \leq 2(\gamma_1 - 1)$ then
            $(\underline{y}_u)_v = q + \gamma_1 - 1 - r$
            $v = v + 1$
return $\mathbf{y}$
```

Table 6.5: Fault-attack success probability in percent

fA_ρ	fA_E	fY	fW	fH
14.3	54.4	24.4	25.4 - 90.3	91.0

being potentially permanent. Faulting the expansion of **A** offers both a large and fixed-time target. Finally, scenario fY leads to valid yet still exploitable signatures.

6.2.7 Attacking qTESLA

All attacks for Dilithium described in this section 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 6.3. After faulting, key recovery is exactly the same, i.e., computing $s = \Delta c^{-1} \cdot \Delta z$.

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 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.

---

[4] For example, with very small probability the $5 \cdot \mathrm{SHAKErate}$ 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.

6.3 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. The value tr can be trivially recomputed as $\mathsf{CRH}(\rho||\mathbf{t}_1)$ (cf. Algorithm 6.1). 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 by simply running the signature algorithm and testing for equivalence. 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 6.1). If $\mathbf{t}$ is known, then recovering $\mathbf{s}_2$ boils down to simple linear algebra. However, for compression one 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 6.9) that does not 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 6.9 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 $\mathsf{MakeHint}$ 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 that if $\|\mathbf{z}\|_\infty \geq \gamma_1 - \beta$ and reject the signature if this is the case. Finally, we perform a verification of the signature to catch the very improbable case that $\mathsf{MakeHint}(-c\mathbf{u}, \mathbf{w} + c\mathbf{u}) \neq \mathsf{MakeHint}(-c\mathbf{t}_0, \mathbf{w} + c\mathbf{u})$.

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

Algorithm 6.9 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
    $\mathbf{y} \in S_{\gamma_1 - 1}^{l} = \mathsf{DeterministicSample}(K || \mu || \kappa)$
    $\mathbf{w} = \mathbf{A}\mathbf{y}$
    $\mathbf{w}_1 = \mathsf{HighBits}(\mathbf{w})$
    $c \in B_{60} = \mathsf{H}(\mu || \mathbf{w}_1)$
    $\mathbf{z} = \mathbf{y} + c\mathbf{s}_1$
    $\mathbf{h} = \mathsf{MakeHint}(-c\mathbf{u}, \mathbf{w} + c\mathbf{u})$        ▷ $\mathsf{MakeHint}(-c(\mathbf{s}_2 - \mathbf{t}_0,), \mathbf{w} - c\mathbf{s}_2 + c\mathbf{t}_0)$
    if $\|\mathbf{z}\|_\infty \geq \gamma_1 - \beta$ 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 $\sigma = (\mathbf{z}, \mathbf{h}, c)$
```

secret key. Since all produced signatures are valid, there is no trivial way to test for this condition (without already knowing the key, as explained earlier).

The case of qTESLA. As a side note, observe 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 (corresponding to $\mathbf{s}_1$) is known. No adapted signature algorithm is needed.

6.4 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 6.2.3). However, even random faults yield a high success rate (Section 6.4.2).

6.4.1 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 6.2) 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.

6.4.2 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 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 6.6).

---

[5] Reference implementation available at https://pq-crystals.org/dilithium/software.shtml

Table 6.6: 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**

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 $\mathbf{A}$ 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 an unprofiled attacker faulting somewhere in this region has a much higher chance of hitting ExpandA than stated in Table 6.6.

In Figure 6.2, 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 6.5). 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 6.5), but also has the smallest footprint. As discussed in Section 6.2.5, 40 % 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.

6.5 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 work. 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 6.7.

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. This obviously doubles execution time. The countermeasure can be defeated by either injecting

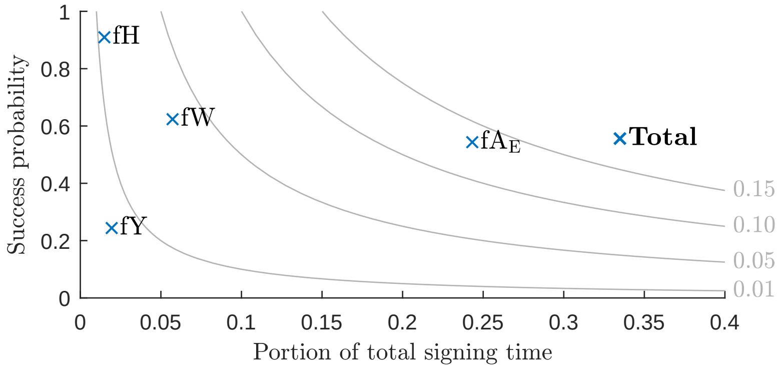


Figure 6.2: Comparison of scenarios regarding runtime as portion of total signing time (from Table 6.6) vs. success probability (from Table 6.5). Lines of constant product are drawn in solid gray.

an identical fault twice, which can be challenging, or by using a permanent fault, e.g., in scenario fA$_\rho$ with the seed ρ.

Verification-after-sign. 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 [Lyu+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 rerandomize the deterministic sampling of the noise $\mathbf{y}$. One can simply sample a random $r \leftarrow \{0,1\}^{256}$ and then invoke $\mathbf{y} = \mathsf{DeterministicSample}(K||\mu||\kappa||r)$. This effectively mitigates the differential fault attack as the faulted call to the signing algorithm uses different $\mathbf{y}$ and thus $\Delta\mathbf{y} \neq 0$.

Whats more, this method might also hamper further side-channel attacks coming as side-effects of determinism. As observed by Seuschek et al. [SHS16] and Samwell et al. [Sam+18], mixing the known message μ with the secret seed K in a hash function (in Dilithium this is SHAKE in 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 r needs to be introduced to maximize the protection while keeping the necessary size of r small likely depends on the used hash function, further investigations are needed to answer this question for the case of SHAKE.

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

Table 6.7: Applicable countermeasures

	fA_ρ	fA_E	fY	fW	fH
Double computation	✗	✓	✓	✓	✓
Verification-after-sign	✓	✓	✗	✓	✓
Additional randomness†	✓	✓	✓	✓	✓

† Not covered by proof of Dilithium [KLS18].

are its simplicity and negligible runtime overhead. Furthermore, unlike straightforward 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 [Amb+18; Sam+18], 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 unique 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"* [Lyu+17]. However, determining whether or not an environment is vulnerable is not easy, as clearly shown by the Rowhammer bug.

Part II

Side-Channel Attacks on Multi-Processor Systems

In this second part of the thesis, we present microarchitectural side-channel attacks capable of attacking multi-processor systems. In Chapter 7, we identify DRAM as a suitable shared resource and reverse engineer the mapping from physical addresses to DRAM banks. Then, we propose our concrete attacks in Chapter 8.

Publications and Contribution

This part is based on the following publication.

- Peter Pessl, Daniel Gruss, Clémentine Maurice, Michael Schwarz, and Stefan Mangard. "DRAMA: Exploiting DRAM Addressing for Cross-CPU Attacks." In: *USENIX Security Symposium.* USENIX Association, 2016, pp. 565–581. [Pes+16]

  forms the base for Chapter 7 and Chapter 8.
  Contribution: I am the first author and contributed the physical probing and covert channel concept/implementation. The software-based reverse engineering is due to Michael Schwarz, Daniel Gruss developed the template side-channel attack.

7

Reverse-Engineering DRAM Addressing

Due to the popularity of cloud services, multiple tenants sharing the same physical server through different virtual machines (VMs) is now a common situation. In such settings, a major requirement is that no sensitive information is leaked between tenants, therefore proper isolation mechanisms are crucial to the security of these environments. While software isolation is enforced by hypervisors, shared hardware presents risks of information leakage between tenants. Previous research shows that microarchitectural attacks can leak secrets of victim processes, e.g., by clever analysis of data-dependent timing differences. Such side-channel measurements allow the extraction of information like cryptographic keys or enable communication over isolation boundaries via covert channels.

Cloud providers can deploy different hardware configurations; however, multi-processor systems are becoming ubiquitous due to their numerous advantages. They offer high peak performance for parallelized tasks while enabling sharing of other hardware resources such as the DRAM. They also simplify load balancing while still keeping the area and cost footprint low. Additionally, cloud providers now commonly disable memory deduplication, i.e., merging of memory pages having identical content, between VMs for security reasons.

To attack such configurations, successful and practical attacks must comply with the following requirements:

1. *Work across processors:* As these configurations are now ubiquitous, an attack that does not work across processors is severely limited and can be trivially mitigated by, e.g., exclusively assigning processors to tenants.
2. *Work without any shared memory:* With memory deduplication disabled, shared memory is not available between VMs. All attacks that require shared memory are thus completely mitigated in cross-VM settings with such configurations.

In the last years, the most prominent and well-studied example of shared-hardware exploits is that of cache attacks (cf. Section 2.2.1). They use the processor-integrated cache and were shown to be effective in a multitude of settings, such as cross-VM key-recovery attacks [Ris+09; Ira+14; Zha+12; Inc+15], including attacks across cores [YF14; Liu+15a; Mau+15a; Gru+16]. However, due to the cache being local to the processor, these attacks do not work across processors and thus violate requirement 1. Note that Irazoqui et al. [IES16] presented a cross-CPU cache attack which exploits cache coherency mechanisms in multi-processor systems. However, their approach requires shared memory and thus violates requirement 2. The whole class of cache attacks is therefore not applicable in multi-processor systems without any shared memory.

Other attacks leverage the main memory that is a shared resource even in multi-processor systems. Xiao et al. [Xia+13] presented a covert channel that exploits memory deduplication. This covert channel has a low capacity and requires the availability of shared memory, thus violating requirement 2. Wu et al. [WXW15] presented a covert channel exploiting the locking mechanism of the memory bus. While this attack works across processors, the capacity of the covert channel is orders of magnitude lower than that of current cache covert channels.

Therefore, none of the above allows efficient attacks with the two previously mentioned requirements. However, we now go on to show that attacks are still possible in this restricted setting.

Contribution. We identify the DRAM row buffer as a resource shared even across CPUs. Before exploiting this resource, however, one needs to know how physical memory addresses map to DRAM channels, ranks, and banks, i.e., to the used row buffer. This mapping is undocumented; therefore we present two methods to reverse engineer it. The first method retrieves the correct addressing functions by performing physical probing of the memory bus. The second method is entirely software-based, fully automatic, and relies only on timing differences.[1] Thus, it can be executed remotely and enables finding DRAM address mappings even in VMs in the cloud. We reverse engineered the addressing functions on a variety of processors and memory configurations. Besides consumer-grade PCs, we also analyzed a dual-CPU server system – similar to those found in cloud setups – and multiple recent smartphones.

We then show how the reverse-engineered mapping can be used to improve existing attacks. Existing Flush+Reload cache attacks use an incorrect cache-miss threshold, introducing noise and reducing the spatial accuracy. Knowledge of the DRAM address mapping also enables practical Rowhammer attacks on DDR4.

---

[1] The source code of this reverse-engineering tool and exemplary DRAMA attacks can be found at `https://github.com/IAIK/drama`.

Outline. The remainder of the chapter is organized as follows. In Section 7.1, we describe some of the inner workings of DRAM. In Section 7.2, we provide definitions that we use throughout the chapter. In Section 7.3, we describe our two approaches to reverse engineer the DRAM addressing and we provide the reverse-engineered functions. In Section 7.4, we show how knowledge of the DRAM addressing improves cache attacks like Flush+Reload and we show how it makes Rowhammer attacks practical on DDR4 and more efficient on DDR3.

7.1 DRAM Organization

Modern DRAM is organized in a hierarchy of channels, DIMMs, ranks, and banks. A system can have one or more *channels*, which are physical links between the DRAM modules and the memory controller. Channels are independent and can be accessed in parallel. This allows distribution of the memory traffic, increasing the bandwidth, and reducing the latency in many cases. Multiple *Dual Inline Memory Modules* (*DIMMs*), which are the physical memory modules attached to the mainboard, can be connected to each channel. A DIMM typically has one or two *ranks*, which often correspond to the front and back of the physical module. Each rank is composed of *banks*, typically 8 on DDR3 DRAM and 16 on DDR4 DRAM. In the case of DDR4, banks are additionally grouped into *bank groups*, e.g., 4 bank groups with 4 banks each. Banks finally contain the actual memory arrays which are organized in *rows* (typically 2^{14} to 2^{17}) and *columns* (often 2^{10}). On PCs, the DRAM word size (bus width) is 64 bits, resulting in a typical row size of 8 KB. As channel, rank, and bank form a hierarchy, two addresses can only be physically adjacent in the DRAM chip if they are in the same channel, DIMM, rank, and bank. In this case we just use the term same bank.

The memory controller, which is integrated into modern processors, translates physical addresses to channels, DIMMs, ranks, and banks. AMD publicly documents the addressing function used by its products (see, e.g., [Adv13, p. 345]), however to the best of our knowledge Intel does not. The mapping for one Intel Sandy Bridge machine in one memory configuration has been reverse engineered by Seaborn [Sea15b]. However, Intel has changed the mapping used in its more recent microarchitectures. Also, the mapping necessarily differs when using other memory configurations, e.g., a different number of DIMMs.

The row buffer. Apart from the memory array, each bank also features a row buffer between the DRAM cells and the memory bus. From a high-level perspective, it behaves like a directly-mapped cache and stores an entire DRAM row. Requests to addresses in the currently active row are served directly from this buffer. If a different row needs to be accessed, then the currently active row is first closed (with a pre-charge command) and then the new row is fetched (with a row-activate command). We call such an event a row conflict. Naturally, such a conflict leads to significantly higher access times compared to requests to the active row. This timing difference will later serve as the basis for our attacks and the software-based reverse-engineering method. Note that after each refresh operation, a bank is already in the pre-charged state. In this case, no row is currently activated.

Independently of our work, Hassan et al. [HKP15] also proposed algorithms to reverse engineer DRAM functions based on timing differences. However, their approach requires customized hardware performance-monitoring units. Thus, they tested their approach only in a simulated environment and not on real systems. Concurrently to our work, Xiao et al. [Xia+16] proposed a method to reverse engineer DRAM functions based on the timing differences caused by row conflicts. Although their method is similar to ours, their focus is different, as they used the functions to perform Rowhammer attacks across VMs.

DRAM organization for multi-CPU systems. In modern multi-CPU server systems, each CPU features a dedicated memory controller and attached memory. The DRAM is still organized in one single address space and is accessible by all processors. Requests for memory attached to other CPUs are sent over the CPU interconnect, e.g., Intel's QuickPath Interconnect (QPI). This memory design is called Non-Uniform Memory Access (NUMA), as the access time depends on the memory location.

On our dual Haswell-EP setup, the organization of this single address space can be configured for the expected workload. In *interleaved mode*, the memory is split into small slices which are spliced together in an alternating fashion. In *non-interleaved* mode, each CPUs memory is kept in one contiguous physical-address block. For instance, the lower half of the address space is mapped to the first CPUs memory, whereas the upper half is mapped to the second CPUs memory.

7.2 Definitions

In this section we provide definitions for the terms *row hit* and *row conflict*. These definitions provide the basis for our reverse engineering as well as the covert and side-channel attacks.

Every physical memory location maps to one out of many rows in one out of several banks in the DRAM. Considering a single access to a row i in a bank there are two major possible cases:

1. The row i is already opened in the row buffer. We call this case a *row hit*.
2. A different row $j \neq i$ in the same bank is opened. We call this case a *row conflict*.

Considering frequent alternating accesses to two (or more) addresses we distinguish three cases:

1. The addresses map to different banks. In this case the accesses are independent and whether the addresses have the same row indices has no influence on the timing. Row hits are likely to occur for the accesses, i.e., access times are low.
2. The addresses map to the same row i in the same bank. The probability that the row stays open in between accesses is high, i.e., access times are low.

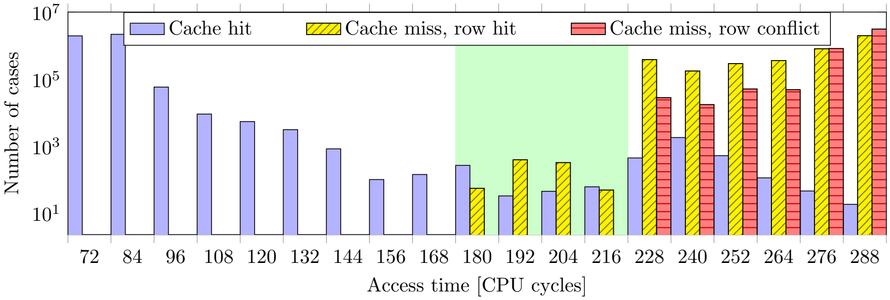


Figure 7.1: Histogram for cache hits and cache misses divided into row hits and row conflicts on the Ivy Bridge i5 test system. Measurements were performed after a short idle period to simulate non-overlapping accesses by victim and spy. From 180 to 216 cycles row hits occur, but no row conflicts.

3. The addresses map to the different rows $i \neq j$ in the same bank. Each access to an address in row i will close row j and vice versa. Thus, row conflicts occur for the accesses, i.e., access times are high.

For measuring the timing differences of row hits and row conflicts, data has to be flushed from the cache. Figure 7.1 shows a comparison of standard histograms of access times for cache hits and cache misses. Cache misses are further divided into row hits and row conflicts. For this purpose an unrelated address in the same row was accessed to cause a row hit and an unrelated address in the same bank but in a different row was accessed to cause a row conflict. We see that from 180 to 216 cycles row hits occur, but no row conflicts (cf. highlighted area in Figure 7.1). In the remainder, we build different attacks that are based on this timing difference between row hits and row conflicts.

7.3 Reverse Engineering DRAM Addressing

In this section, we present our reverse engineering of the DRAM address mapping. We discuss two approaches, the first one is based on physical probing, whereas the second one is entirely software-based and fully automated. Finally, we present the outcome of our analysis, i.e., the reverse-engineered mapping functions. In the remainder of this chapter, we denote by a a physical memory address. a_i denotes the i-th bit of an address.

7.3.1 Linearity of Functions

The DRAM addressing functions are reverse engineered in two phases. First, a measuring phase and second, a subsequent solving phase. Our solving approaches require that the addressing functions are linear, i.e., they are XORs of physical-address bits.

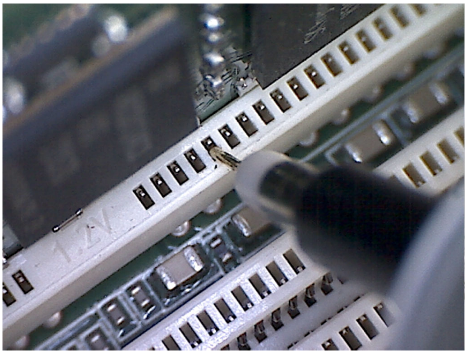


Figure 7.2: Physical probing of the DIMM slot.

In fact, Intel used such functions in earlier microarchitectures. For instance, Seaborn [Sea15b] reports that on his Sandy Bridge setup the bank address is computed by XORing the bits $a_{14}..a_{16}$ with the lower bits of the row number ($a_{18}..a_{20}$) (cf. Figure 7.4a). This is done in order to minimize the number of row conflicts during runtime. Intel also uses linear functions for CPU-cache addressing. Maurice et al. [Mau+15b] showed that the *complex addressing* function, which is used to select cache slices, is an XOR of many physical-address bits.

As it turns out, linearity holds on all our tested configurations. However, there are setups in which it might be violated, such as triple-channel configurations. We did not test such systems and leave a reverse engineering to future work.

7.3.2 Reverse Engineering Using Physical Probing

Our first approach to reverse engineer the DRAM mapping is to physically probe the memory bus and to directly read the control signals. As shown in Figure 7.2, we use a standard passive probe to establish contact with the pin at the DIMM slot. We then repeatedly accessed a selected physical address[2] and used a high-bandwidth oscilloscope to measure the voltage and subsequently deduce the logic value of the contacted pin. Note that due to the repeated access to a single address, neither a timely location of specific memory requests nor distinguishing accesses to the chosen address from other random ones are required.

We repeated this experiment for many selected addresses and for all pins of interest, namely the bank-address bits (BA0, BA1, BA2 for DDR3 and BG0, BG1, BA0, BA1 for DDR4) for one DIMM and the chip select CS for half the DIMMs.

For the solving phase we use the following approach. Starting from the top-layer (channel or CPU addressing) and drilling down, for each DRAM addressing function we create an over-defined system of linear equations in the physical address bits. The left-hand side of this system is made up of the relevant tested

---

[2] Resolving virtual to physical addresses requires root privileges in Linux. Given that we need physical access to the internals of the system, this is a very mild prerequisite.

physical addresses. For instance, for determining the bank functions we only use addresses that map to the contacted DIMMs channel. The right-hand side of the system of equations are the previously measured logic values for the respective address and the searched-for function. The logic values for CPU and channel addressing are computed by simply ORing all respective values for the chip-select pins. We then solve this system using linear algebra. The solution is the corresponding DRAM addressing function.

Obviously, this reverse-engineering approach has some drawbacks. First, expensive measurement equipment is needed. Second, it requires physical access to the internals of the tested machine. However, it has the big advantage that the address mapping can be reconstructed for each control signal individually and exactly. Thus, we can determine the exact individual functions for the bus pins. Furthermore, every platform only needs to be measured only once in order to learn the addressing functions. Thus, an attacker does not need physical access to the concrete attacked system if the measurements are performed on a similar machine.

7.3.3 Fully Automated Reverse Engineering

For our second approach to reverse engineer the DRAM mapping we exploit the fact that row conflicts lead to higher memory access times. We use the resulting timing differences to find sets of addresses that map to the same bank but to a different row. Subsequently, we determine the addressing functions based on these sets. The entire process is fully automated and runs in unprivileged and possibly restricted environments.

Timing analysis. In the first step, we aim to find same-bank addresses in a large array mapped into the attackers' address space. For this purpose, we perform repeated alternating access to two addresses and measure the average access time. We use `clflush` to ensure that each access is served from DRAM and not from the CPU cache. As shown in Figure 7.3, for some address pairs the access time is significantly higher than for most others. These pairs belong to the same bank but to different rows. The alternating access causes frequent row conflicts and consequently the high latency.

The tested pairs are drawn from an address pool, which is built by selecting random addresses from a large array. A small subset of addresses in this pool is tested against all others in the pool. The addresses are subsequently grouped into sets having the same channel, DIMM, rank, and bank. We try to identify as many such sets as possible in order to reconstruct the addressing functions.

Function reconstruction. In the second phase, we use the identified address sets to reconstruct the addressing functions. This reconstruction requires (at least partial) resolution of the tested virtual addresses to physical ones. Similar as later in Section 8.2.1, one can use either the availability of 2 MB pages, 1 GB pages, or

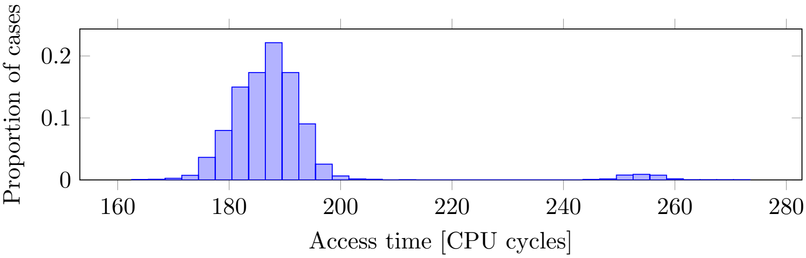


Figure 7.3: Histogram of average memory access times for random address pairs on our Haswell test system. A clear gap separates the majority of address pairs causing no row conflict (lower access times), because they map to different banks, from the few address pairs causing a row conflict (higher access times), because they map to different rows in the same bank.

privileged information such as the virtual-to-physical address translation that can be obtained through `/proc/pid/pagemap` in Linux systems.

In the case of 2 MB pages we can recover all partial functions up to bit a_{20}, as the lowest 21 bit of virtual and physical address are identical. On many systems the DRAM addressing functions do not use bits above a_{20} or only a few of them, providing sufficient information to mount covert and side-channel attacks later on. In the case of 1 GB pages we can recover all partial functions up to bit a_{30}. This is sufficient to recover the full DRAM addressing functions on all our test systems. If we have full access to physical address information we will still ignore bits a_{30} and upwards. These bits are typically only used for DRAM row addressing and they are very unlikely to play any role in bank addressing. Additionally, we ignore bits $(a_0..a_5)$ as they are used for addressing within a cache line, which makes it unlikely that they are used for bank addressing.

The search space is then small enough to perform a brute-force search of linear functions within seconds. For this, we generate all linear functions that use exactly n bits as coefficients and then apply them to all addresses in one randomly selected set. We start with $n = 1$ and increment n subsequently to find all functions. Only if the function has the same result for all addresses in a set, we test this potential function on all other sets. However, in this case we only pick one address per set and test whether the function is constant over all sets. If so, the function is discarded. We obtain a list of possible addressing functions that also contains linear combinations of the actual DRAM addressing functions. We prioritize functions with a lower number of coefficients, i.e., we remove higher-order functions which are linear combinations of lower-order ones. Depending on the random address selection, we now have a complete set of correct addressing functions. We verify the correctness either by comparing it to the results from the physical probing, or by performing a software-based test. One option to perform the latter is to verify the timing differences on a larger set of addresses. Another option is to perform Rowhammer tests, where usage of

Table 7.1: Experimental setups.

CPU / SoC	Microarch.	Mem.
i5-2540M	Sandy Bridge	DDR3
i5-3230M	Ivy Bridge	DDR3
i7-3630QM	Ivy Bridge	DDR3
i7-4790	Haswell	DDR3
i7-6700K	Skylake	DDR4
2x Xeon E5-2630 v3	Haswell-EP	DDR4
Qualcomm Snapdragon S4 Pro	ARMv7	LPDDR2
Samsung Exynos 5 Dual	ARMv7	LDDDR3
Qualcomm Snapdragon 800	ARMv7	LPDDR3
Qualcomm Snapdragon 820	ARMv8-A	LPDDR3
Samsung Exynos 7420	ARMv8-A	LPDDR4

the addressing functions should increase the number of bit flips per second by a factor that is the number of sets we found.

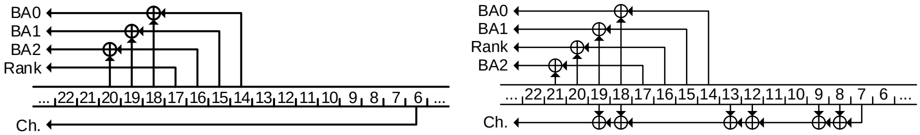
Compared to the probing approach, this purely software-based method has significant advantages. It does not require any additional measurement equipment and can be executed on a remote system. We can identify the functions even from within VMs or sandboxed processes if 2 MB or 1 GB pages are available. Furthermore, even with only 4 KB pages we can group addresses into sets that can be directly used for covert or side-channel attacks. This software-based approach also allows reverse engineering in settings where probing is not easily possible anymore, such as on mobile devices with hard-wired ball-grid packages. Thus, it allowed us to reverse engineer the mapping on current ARM processors.

One downside of the software-based approach is that it cannot recover the exact labels (BG0, BA0, ...) of the functions. Thus, we can only guess whether the reconstructed function computes a bank address bit, rank bit, or channel bit. Note that assigning the correct labels to functions is not required for any of our attacks.

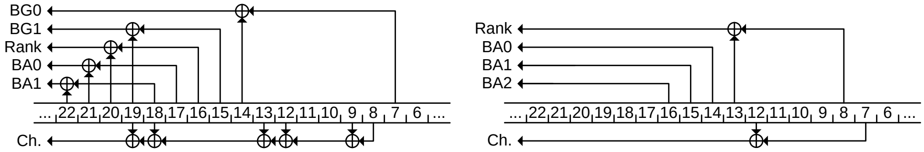
7.3.4 Results

We now present the reverse-engineered mappings for all our experimental setups. We analyzed a variety of systems (Table 7.1), including a dual-CPU Xeon system, that can often be found in cloud systems, and multiple current smartphones. Where possible, we used both presented reverse-engineering methods and cross-validated the results.

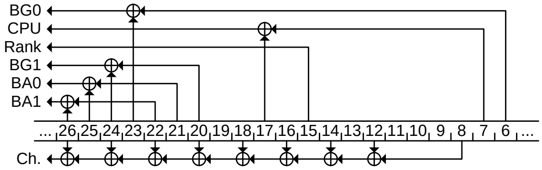
We found that the basic scheme is always as follows. On PCs, the memory bus is 64 bits wide, yet the smallest addressable unit is a byte. Thus, the three lower bits ($a_0..a_2$) of the physical address are used as byte index into a 64-bit (8-byte) memory word and they are never transmitted on the memory bus. Then,



(a) Sandy Bridge – DDR3 [Sea15b]. **(b)** Ivy Bridge / Haswell – DDR3.



(c) Skylake – DDR4. **(d)** Samsung Exynos 7420 – LPDDR4.



(e) Dual Haswell-EP (Interleaved Mode) – DDR4.

Figure 7.4: Reverse engineered dual channel mapping (1 DIMM per channel) for different architectures.

the next bits are used for column selection. One bit in between is used for channel addressing. The following bits are responsible for bank, rank, and DIMM addressing. The remaining upper bits are used for row selection.

The detailed mapping, however, differs for each setup. To give a quick overview of the main differences, we show the mapping of one selected memory configuration for multiple Intel microarchitectures and ARM-based SoCs in Figure 7.4. Here we chose a configuration with two equally sized DIMMs in dual-channel configuration, as it is found in many off-the-shelf consumer PCs. All our setups use dual-rank DIMMs and use 10 bits for column addressing. Figure 7.4a shows the mapping on the Sandy Bridge platform, as reported by Seaborn [Sea15b]. Here, only a_6 is used to select the memory channel, a_{17} is used for rank selection. The bank-address bits are computed by XORing bits $a_{14}..a_{16}$ with the lower bits of the row index ($a_{18}..a_{20}$).

The channel selection function changed with later microarchitectures, such as Ivy Bridge and Haswell. As shown in Figure 7.4b, the channel-selection bit is now computed by XORing seven bits of the physical address. Further analysis showed that bit a_7 is used exclusively, i.e., it is not used as part of the row- or column address. Additionally, rank selection is now similar to bank addressing and also uses XORs.

Our Skylake test system uses DDR4 instead of DDR3. Due to DDR4's introduction of bank grouping and the doubling of the available banks (now 16), the addressing function necessarily changed again. As shown in Figure 7.4c, a_7 is not used for channel selection anymore, but for bank addressing instead.

Figure 7.4e depicts the memory mapping of a dual-CPU Haswell-EP system equipped with DDR4 memory. It uses 2 modules in dual-channel configuration *per CPU* (4 DIMMs in total). In interleaved mode (cf. Section 7.1), the chosen CPU is determined as $a_7 \oplus a_{17}$. Apart from the different channel function, there is also a difference in the bank addressing, i.e., bank addressing bits are shifted. The range of bits used for row indexing is now split into address bits ($a_{17}..a_{19}$) and a_{23} upwards.

The mapping used on one of our mobile platforms, a Samsung Galaxy S6 with an Exynos 7420 ARMv8-A SoC and LPDDR4 memory, is much simpler (cf. Figure 7.4d). Here physical address bits are mapped directly to bank address bits. Rank and channel are computed with XORs of only two bits each. The bus width of LPDDR4 is 32 bits, so only the two lowest bits are used for byte indexing in a memory word.

Table 7.2 shows a comprehensive overview of all platforms and memory configurations we analyzed. As all found functions are linear, we simply list the index of the physical address bits that are XORed together. With the example of the Haswell microarchitecture, one can clearly see that the indices are shifted to accommodate for the different memory setups. For instance, in single-channel configurations a_7 is used for column instead of channel selection, which is why bank addressing starts with a_{13} instead of a_{14}.

7.4 Improving Attacks

In this section, we describe how the DRAM addressing functions can be used to improve the accuracy, efficiency, and success rate of existing attacks.

Flush+Reload. The first step when performing Flush+Reload attacks is to compute a cache-hit threshold, based on a histogram of cache hits and cache misses (memory accesses). However, as we have shown (cf. Figure 7.1) row hits have a slightly lower access time than row conflicts. To get the best performance in a Flush+Reload attack it is necessary to take row hits and conflicts into account. Otherwise, if a process accesses any memory location in the same row, a row hit will be misclassified as a cache hit. This introduces a significant amount of noise as the spatial accuracy of a cache hit is 64 bytes and the one of a row hit can be as low as 8 KB, depending on how actively the corresponding pages of the row are used. We found that even after a call to `sched_yield` and thus at least two context switches, a row hit is still observed in 2 % of the cases on a Linux system that is mostly idle. In a Flush+Reload attack the victim computes in parallel and thus the probability then is even higher than 2 %. This introduces a significant amount of noise especially for Flush+Reload attacks on low-frequency events. Thus, the accuracy of Flush+Reload attacks can be improved significantly taking row hits into account for the cache hit threshold computation.

Table 7.2: Reverse engineered DRAM mapping on all platforms and configurations we analyzed via physical probing or via software analysis. These tables list the bits of the physical address that are XORed. For instance, for the entry (13, 17) we have $a_{13} \oplus a_{17}$.

(a) DDR3

CPU	Ch.	DIMM/Ch.	BA0	BA1	BA2	Rank	DIMM	Channel
Sandy Bridge	1	1	13, 17	14, 18	15, 19	16	-	-
Sandy Bridge [Sea15b]	2	1	14, 18	15, 19	16, 20	17	-	6
Ivy Bridge/Haswell	1	1	13, 17	14, 18	16, 20	15, 19	-	-
	1	2	13, 18	14, 19	17, 21	16, 20	15	-
	2	1	14, 18	15, 19	17, 21	16, 20	-	7, 8, 9, 12, 13, 18, 19
	2	2	14, 19	15, 20	18, 22	17, 21	16	7, 8, 9, 12, 13, 18, 19

(b) DDR4

CPU	Ch.	DIMM/Ch.	BG0	BG1	BA0	BA1	Rank	CPU	Channel
Skylake[†]	2	1	7, 14	15, 19	17, 21	18, 22	16, 20	-	8, 9, 12, 13, 18, 19
2x Haswell-EP (interleaved)	1	1	6, 22	19, 23	20, 24	21, 25	14	7, 17	-
	2	1	6, 23	20, 24	21, 25	22, 26	15	7, 17	8, 12, 14, 16, 18, 20, 22, 24, 26
2x Haswell-EP (non-interleaved)	1	1	6, 21	18, 22	19, 23	20, 24	13	-	-
	2	1	6, 22	19, 23,	20, 24	21, 25	14	-	7, 12, 14, 16, 18, 20, 22, 24, 26

(c) LPDDR2,3,4

CPU	Ch.	BA0	BA1	BA2	Rank	Channel
Qualcomm Snapdragon S4 Pro[†]	1	13	14	15	10	-
Samsung Exynos 5 Dual[†]	1	13	14	15	7	-
Qualcomm Snapdragon 800/820[†]	1	13	14	15	10	-
Samsung Exynos 7420[†]	2	14	15	16	8, 13	7, 12

[†] Software analysis only. Labeling of functions is based on results of other platforms.

Rowhammer. In a Rowhammer attack, an adversary tries to trigger bit flips in DRAM by provoking a high number of row switches. The success rate and efficiency of this attack benefit greatly from knowing the DRAM mapping, as we now demonstrate.

In order to cause row conflicts, one must alternately access addresses belonging to the same bank, but different row. The probability that 2 random addresses fulfill this criterion is 2^{-B}, where B is the total number of bank-addressing bits (this includes all bits for channel, rank, etc.). For instance, with the dual-channel DDR4 configuration shown in Figure 7.4c this probability is only $2^{-6} = 1/64$. By hammering a larger set of addresses, the probability of having at least two targeting the same bank increases. However, so does the time in between row switches, thus the success rate decreases.

The most efficient way of performing the Rowhammer attack is *double-sided hammering*. Here, one tries to cause bit flips in row n by alternatingly accessing the adjacent rows $n-1$ and $n+1$, which are most likely also adjacent in physical memory. The most commonly referenced implementation of the Rowhammer attack, by Seaborn and Dullien [SD15], performs double-sided hammering by making assumptions on, e.g., the position of the row-index bits. If these are not met, then their implementation does not find any bit flips. Also, it needs to test multiple address combinations as it does not use knowledge of the DRAM addressing functions. We tested their implementation on a Skylake machine featuring G.SKILL F4-3200C16D-16GTZB DDR4 memory at the highest possible refresh interval, yet even after 4 days of nonstop hammering, we did not detect any bit flips.

By using the DRAM addressing functions we can immediately determine whether two addresses map to the same bank. Also, we can very efficiently search for pairs allowing double-sided hammering. After taking the reverse-engineered addressing functions into account, we successfully caused bit flips on the same Skylake setup within minutes. Running the same attack on a Crucial DDR4-2133 memory module running at the default refresh interval, we observed the first bit flip after 16 seconds and subsequently observed on average one bit flip every 12 seconds.

One possible countermeasure against Rowhammer attacks is *target row refresh* (TRR), where rows with many accesses to its neighbors are selectively refreshed. Although the LPDDR4 standard includes TRR, the DDR4 standard does not. Still, some manufacturers include it in their products as a non-standard feature. For both DDR4 and LPDDR4, both the memory controller and the DRAM must support this feature in order to provide any protection. To the best of our knowledge, both our Haswell-EP test system and the Crucial DDR4-2133 memory module, with Micron DRAM chips, support TRR [Mic14; Int15]. However, we are still able to reproducibly trigger bit flips in this configuration.

8

Exploiting DRAM Addressing for Cross-CPU Attacks

The previous chapter shows that there is a clear and measurable timing difference between row hits and row conflicts. In fact, this difference is the basis for the software-based reverse-engineering approach.

As demonstrated by cache attacks, such a microarchitectural timing behavior can have powerful adversarial use. Unlike caches, however, DRAM and thus row buffers are shared even across physical CPUs. This potentially allows attacks in even more restrictive settings.

Contribution. In this chapter, we prove that this is indeed the case by presenting *DRAMA* attacks, a novel class of attacks that exploit the *DRAM Addressing*. In particular, they leverage that DRAM row buffers are a shared component even in multi-processor systems. Our attacks require that at least one memory module is shared between the attacker and the victim, which is the case even in the most restrictive settings. In these settings, attacker and victim cannot access the same memory cells, i.e., we do not circumvent system-level memory isolation. We do not make any assumptions on the cache, nor on the location of executing cores, nor on the availability of shared memory such as cross-VM memory deduplication.

First, we build a covert channel that achieves transmission rates of up to 2 Mbps, which is three to four orders of magnitude faster than previously presented memory-bus based channels. Second, we build a side-channel attack that allows to automatically locate and monitor memory accesses, e.g., user input or server requests, by performing template attacks.

Outline. In Section 8.1, we provide background information and performance figures of previously reported side channels on shared hardware. In Section 8.2, we build a high-speed cross-CPU DRAMA covert channel. In Section 8.3, we build a highly accurate cross-CPU DRAMA side-channel attack. We discuss countermeasures against our attack in Section 8.4. We conclude in Section 8.5.

8.1 Previous Shared-Hardware Exploits

Attacks exploiting hardware sharing can be grouped into two categories. In side-channel attacks, an attacker spies on a victim and extracts sensitive information such as cryptographic keys. In covert channels however, sender and receiver are actively cooperating to exchange information in a setting where they are not allowed to, e.g., across isolation boundaries.

Cache attacks. Covert and side channels using the CPU cache exploit the fact that cache hits are faster than cache misses. The methods Prime+Probe [Per05; Mau+15a; Liu+15a] and Flush+Reload [YF14; Ira+14; Ben+14] have been presented to either build covert channels or run side-channel attacks. The two methods work at a different granularity: Prime+Probe can spy on cache sets, while Flush+Reload has the finer granularity of a cache line but requires shared memory, such as shared libraries or memory deduplication.

Attacks targeting the last-level cache are cross-core, but require the sender and receiver to run on the same physical CPU. Gruss et al. [Gru+16] implemented cross-core covert channels using Prime+Probe and Flush+Reload as well as a new one, Flush+Flush, with the same protocol to normalize the results. The covert channel using Prime+Probe achieves 536 Kbps, Flush+Reload 2.3 Mbps, and Flush+Flush 3.8 Mbps. The cross-CPU cache attack by Irazoqui et al. [IES16] exploits cache coherency mechanisms and works across processors. It however requires shared memory.

An undocumented function maps physical addresses to the slices of the last-level cache. However, this function has been reverse engineered in previous work [Mau+15b; Inc+15; Yar+15], enhancing existing attacks and enabling attacks in new environments.

Memory and memory bus. Xiao et al. [Xia+13] presented a covert channel between VMs that exploits memory deduplication. In order to save memory, the hypervisor searches for identical pages in physical memory and merges them across VMs to a single read-only physical page. Writing to this page triggers a copy-on-write page fault, incurring a significantly higher latency than a regular write access. The authors built a covert channel that achieves up to 90 bps, and 40 bps on a system under memory pressure. Wu et al. [WXW15] proposed a bus-contention-based covert channel, which uses atomic memory operations locking the memory bus. This covert channel achieves a raw bandwidth of 38 Kbps between two VMs, with an effective capacity of 747 bps with error correction.

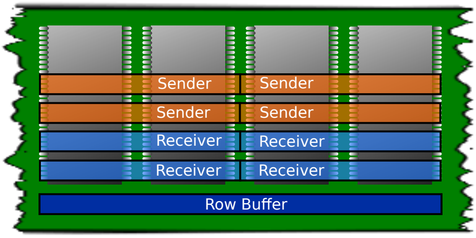


Figure 8.1: The sender occupies rows in a bank to trigger row conflicts. The receiver occupies rows in the same bank to observe these row conflicts.

8.2 A High-Speed Cross-CPU Covert Channel

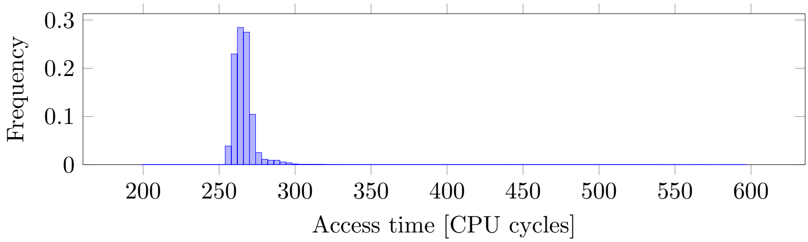
In this section, we present a first DRAMA attack, namely a high-speed cross-CPU covert channel that does not require shared memory. Our channel exploits the row buffer, which behaves like a directly-mapped cache. Unlike cache attacks, the only prerequisite is that two communicating processes have access to the same memory module.

8.2.1 Basic Concept

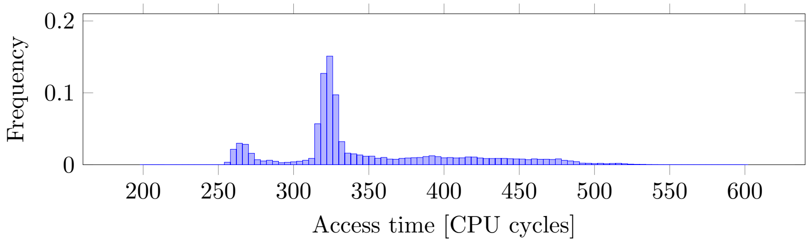
Our covert channel exploits timing differences caused by row conflicts. Sender and receiver occupy different rows in the same bank as illustrated in Figure 8.1. The receiver process continuously accesses a chosen physical address in the DRAM and measures the average access time over a few accesses. If the sender process now continuously accesses a different address in the same bank but in a different row, a row conflict occurs. This leads to higher average access times in the receiver process. Bits can be transmitted by switching the activity of the sender process in the targeted bank on and off. This timing difference is illustrated in Figure 8.2, an exemplary transmission is shown in Figure 8.3. The receiver process distinguishes the two values based on the mean access time. We assign a logic value of 0 to low access times (the sender is inactive) and a value of 1 to high access times (the sender is active).

Each (CPU, channel, DIMM, rank, bank) tuple can be used as a separate transmission channel. However, a high number of parallel channels leads to increased noise. Also, there is a strict limit on the usable bank parallelism. Thus, optimal performance is achieved when using only a subset of available tuples. Transmission channels are unidirectional, but the direction can be chosen for each one independently. Thus, two-way communication is possible.

To evaluate the performance of this new covert channel, we created a proof-of-concept implementation. We restrict ourselves to unidirectional communication, i.e., we have one dedicated sender and one dedicated receiver.



(a) Sender inactive on bank: sending a 0.



(b) Sender active on bank: sending a 1.

Figure 8.2: Timing differences between active and non-active sender (on one bank), measured on the Haswell i7 test system.

The memory access time is measured using `rdtsc`. The memory accesses are performed using `volatile` pointers. In order to cause a DRAM access for each request, data has to be flushed from the cache using `clflush`.

Determining channel, rank, and bank address. In an agreement phase, all parties need to agree on the set of (channel, DIMM, rank, bank) tuples that are used for communication. This set needs to be chosen only once, all subsequent communication can use the same set. Next, both the sender and the receiver need to find at least one address in their respective address space for each tuple. Note that some operating systems allow unprivileged resolution of virtual to physical addresses. In this case, finding such addresses is trivial.

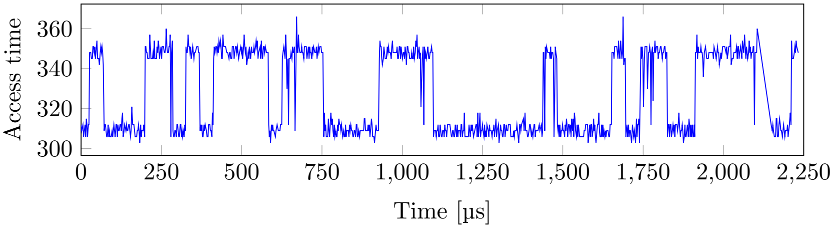


Figure 8.3: Covert channel transmission on one bank, cross-CPU and cross-VM on a Haswell-EP server. The time frame for one bit is 50 µs.

However, on Linux, which we used on our testing setup, unprivileged address resolution is not possible. Thus, we use the following approach. As observed in previous work [GBM15; GMM16], system libraries and the operating system assign 2 MB pages for arrays which are significantly larger than 2 MB. On these pages, the 21 lowest bits of the virtual address and the physical address are identical. Depending on the hardware setup, these bits can already be sufficient to fully determine bank, rank, and channel address. For this purpose, both processes request a large array. The start of this array is not necessarily aligned with a 2 MB border. Memory before such a border is allocated using 4 KB pages. We skip to the next 2 MB page border by choosing the next virtual address having the 21 lowest bits set to zero.

On systems that also use higher bits, an attacker can use the following approach, which we explain on the example of the mapping shown in Figure 7.4b. There an attacker cannot determine the BA2 bit by just using 2 MB pages. Thus, the receiving process selects addresses with chosen BA0, BA1, rank, and channel, but an unknown BA2 bit. The sender now accesses addresses for both possibilities of BA2, e.g., by toggling a_{17} between consecutive reads. Thus, only each second access in the sending process targets the correct bank. Due to bank parallelism this does not cause a notable performance decrease. Note however that this approach might not work if the number of unknown bank-address bits is too high.

In a virtualized environment, even a privileged attacker can retrieve only the *guest physical address*, which is further translated into the real physical address by the memory management unit. However, if the host system uses 1 GB pages for the second-level address translation (to improve efficiency), then the lowest 30 bits of the guest physical address are identical to the real physical address. Knowledge of these bits is sufficient on all systems we analyzed to use the full DRAM addressing functions.

Finally, the covert channel could also be built without actually reconstructing the DRAM addressing functions. Instead of determining the exact bank address, it can rely solely on the same-bank sets retrieved in Section 7.3.3. In an initialization phase, both sender and receiver perform the timing analysis and use it to build sets of same-bank addresses. Subsequently, the communicating parties need to synchronize their sets, i.e., they need to agree on which of them is used for transmission. This is done by sending predefined patterns over the channel. After that, the channel is ready for transmission. Thus, it can be established without having any information on the mapping function nor on the physical addresses.

Synchronization. In our proof-of-concept implementation, one set of bits (a data block) is transmitted for a fixed time span which is agreed upon before starting communication. Decreasing this period increases the raw bitrate, but it also increases the error rate, as shown in Figure 8.4.

For synchronizing the start of these blocks we employ two different mechanisms. If sender and receiver run natively, we use the wall clock as means of synchronization. Here blocks start at fixed points in time. If, however, sender and receiver run in two different VMs, then a common (or perfectly synchronized) wall clock is typically not available. In this case, the sender uses one of the transmission channels to transmit a clock signal which toggles at the beginning of each block. The receiver then recovers this clock and can thus synchronize with the sender.

We employ multiple threads for both the sender and receiver processes to achieve optimal usage of the memory bus. Thus, memory accesses are performed in parallel, increasing the performance of the covert channel.

8.2.2 Evaluation

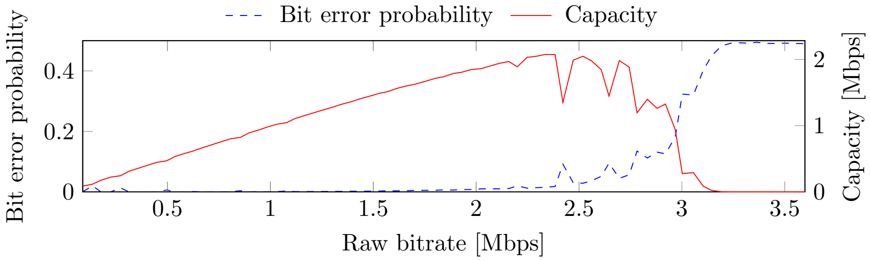
We evaluated the performance of our covert-channel implementation on two systems. First, we performed tests on a standard desktop PC featuring an Intel i7-4790 CPU with Haswell microarchitecture. It was equipped with 2 Kingston DDR3 KVR16N11/8 dual-rank 8 GB DIMMs in dual-channel configuration. The system was mostly idle during the tests, i.e., no other tasks were causing significant load on the system. The DRAM clock was set to its default of 800 MHz (DDR3-1600).

Furthermore, we also tested the capability of cross-CPU transmission on a server system. Our setup has two Intel Xeon E5-2630 v3 (Haswell-EP microarchitecture). It was equipped with a total of 4 Samsung M393A2G40DB0-CPB DDR4 registered ECC DIMMs. Each CPU was connected to two DIMMs in dual-channel configuration and NUMA was set to interleaved mode. The DRAM frequency was set to its maximum supported value (DDR4-1866).

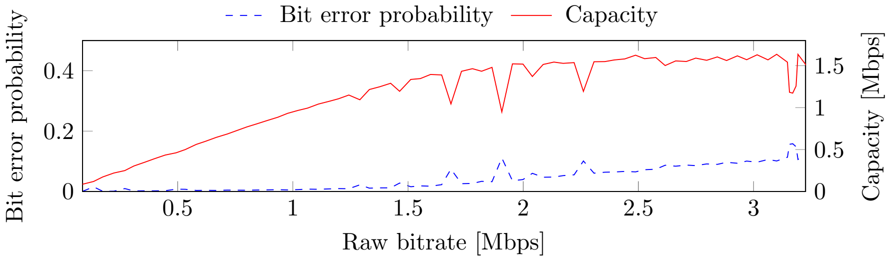
For both systems, we evaluated the performance in both a native scenario, i.e., both processes run natively, and in a cross-VM scenario. We transmit 8 bits per block (use 8 (CPU, channel, DIMM, rank, bank) tuples) in the covert channel and run 2 threads in both the sender and the receiver process. Every thread is scheduled to run on different CPU cores, and in the case of the Xeon system, sender and receiver run on different physical CPUs.

We tested our implementation with a large range of measurement intervals. For each one, we measure the raw channel capacity and the bit error probability. While the raw channel capacity increases proportionally to the reduction of the measurement time, the bit error rate increases significantly if the measurement time is too short. In order to find the best transmission rate, we use the channel capacity as our metric. When using the binary symmetric channel model, this metric is computed by multiplying the raw bitrate with $1 - \mathsf{H}(e)$, with e the bit error probability and $\mathsf{H}(e) = -e \cdot \log_2(e) - (1-e) \cdot \log_2(1-e)$ the binary entropy function.

Figure 8.4 shows the error rate depending on the raw bitrate for the case that both sender and receiver run natively. On our desktop setup (Figure 8.4a), the error probability stays below 1 % for bitrates of up to 2 Mbps. The channel capacity reaches up to 2.1 Mbps (raw bitrate of 2.4 Mbps, error probability of 1.8 %). Beyond this peak, the increasing error probability causes a decrease in the effective capacity. On our server setup (Figure 8.4b) the cross-CPU



(a) Desktop setup (Haswell)



(b) Server setup, cross-CPU (Haswell-EP)

Figure 8.4: Performance of our covert channel implementation (native).

communication achieves 1.2 Mbps with a 1 % error rate. The maximum capacity is 1.6 Mbps (raw 2.6 Mbps, 8.7 % error probability).

For the cross-core cross-VM scenario, we deployed two VMs which were configured to use 1 GB pages for second-stage address translation. We reach a maximum capacity of 309 kbps (raw 411 kbps, 4.1 % error probability) on our desktop system. The server setup (cross-CPU cross-VM) performs much better, we achieved a bitrate of 596 kbps with an error probability of just 0.4 %.

8.2.3 Comparison with State of the Art

We compare the bitrate of our DRAM covert channel with the normalized implementation of three cache covert channels by Gruss et al. [Gru+16]. For an error rate that is less than 1 %, the covert channel using Prime+Probe obtains 536 Kbps, the one using Flush+Reload 2.3 Mbps and the one using Flush+Flush 3.8 Mbps. With a capacity of up to 2 Mbps, our covert channel is within the same order of magnitude of current cache-based channels. However, unlike Flush+Reload and Flush+Flush, it does not require shared memory. Moreover, in contrast to our attack, these cache covert channels do not allow cross-CPU communication.

The work of Irazoqui et al. [IES16] focuses on cross-CPU cache-based side-channel attacks. They did not implement a covert channel, thus we cannot compare our performance with their cache attack. However, their approach also requires shared memory and thus it would not work in our attack setting.

The covert channel by Xiao et al. [Xia+13] using memory deduplication achieves up to 90 bps. However, due to security concerns, memory deduplication has been disabled in many cloud environments. The covert channel of Wu et al. [WXW15] using the memory bus achieves 746 bps with error correction. Our covert channel is therefore three to four orders of magnitude faster than state-of-the-art memory-based covert channels.

8.3 A Low-Noise Cross-CPU Side Channel

In this section, we present a second DRAMA attack, a highly accurate side-channel attack using DRAM addressing information. We again exploit the row buffer and its behavior similar to a directly-mapped cache. In this attack, the spy and the victim can run on separate CPUs and do not share memory, i.e., no access to shared libraries and no page deduplication between VMs. We mainly consider a local attack scenario where Flush+Reload cache attacks are not applicable due to the lack of shared memory. However, our side-channel attacks can also be applied in a cloud scenario with multi-tenant machines. There, one malicious user spies on other users through this side channel. The side channel achieves a timing accuracy that is comparable to Flush+Reload and a higher spatial accuracy than Prime+Probe. Thus, it can be used as a highly accurate alternative to Prime+Probe cache attacks in cross-core scenarios without shared memory.

8.3.1 Basic Concept

For the covert channel, an active sender caused row conflicts. In the side-channel attack, we infer the activity of a victim process by detecting *row hits* and *row conflicts* following our definitions from Section 7.2. For the attack to succeed, spy and victim need to have access to the same row in a bank, as illustrated in Figure 8.5. This is possible without shared memory due to the DRAM addressing functions.

Depending on the addressing functions, a single 4 KB page can map to multiple DRAM rows. As illustrated in Figure 8.6, in our Haswell-EP system the contents of a page are split over 8 DRAM rows (with the same row index, but different bank address). Conversely, a DRAM row contains content of at least two 4 KB pages, as the typical row size is 8 KB. More specifically, in our Haswell-EP setup a single row stores content for 16 different 4 KB pages, as again shown in Figure 8.6. The amount of memory mapping from one page to one specific row, e.g., 512 bytes in the previous case, is the achievable spatial accuracy of our attack. If none of the DRAM addressing functions uses low address bits ($a_0 - a_{11}$), the spatial accuracy is 4 KB, which is the worst case. However, if DRAM addressing functions (such as channel, BG0, CPU) use low address bits, better accuracy can be achieved, such as the 512 B for the server setup. On systems where 6 or more

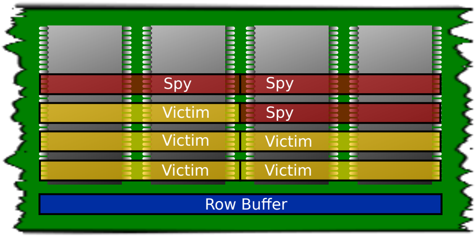


Figure 8.5: Victim and spy have memory allocated in the same DRAM row. By accessing this memory, the spy can determine whether the victim just accessed it.

low address bits are used, the spatial accuracy of the attack is 64 B and thus as accurate as a Flush+Reload cache side-channel attack.

Assuming that an attacker occupies at least one other 4 KB page that maps (in part) to the same bank and row, the attacker has established a situation as illustrated in Figure 8.5.

To run the side-channel attack on a private memory address t in a victim process, the attacker allocates a memory address p that maps to the same bank and the same row as the target address t. As shown in Figure 8.6, although t and p map to the same DRAM row, they belong to different 4 KB pages (i.e., no shared memory). The attacker also allocates a row conflict address $\bar{p}$ that maps to the same bank but a different row.

The side-channel attack then works in three steps:

1. Access the row conflict address $\bar{p}$
2. Wait for the victim to compute
3. Measure the access time on the targeted address p

If the measured timing is below a row-hit threshold (cf. the highlighted "row hit" region in Figure 7.1), the victim has just accessed t or another address in the target row. Thus, we can accurately determine when a specific non-shared memory location is accessed by a process running on another core or CPU. As p and $\bar{p}$ are on separate private 4 KB pages, they will not be prefetched and we can measure row hits without any false positives. By allocating all but one of the pages that map to a row, the attacker maximizes the spatial accuracy.

Based on this attack principle, we build a fully automated template attack (similar to cache template attacks [GSM15]) that triggers an event in the victim process running on the other core or CPU (e.g., by sending requests to a web interface or triggering user-interface events). For this attack we do not need to reconstruct the full addressing functions nor determine the exact bank address. Instead, we exploit the timing difference between row hits and row conflicts as shown in Figure 7.1.

To perform a DRAMA template attack, the attacker allocates a large fraction of memory, ideally in 4 KB pages. This ensures that some of the allocated

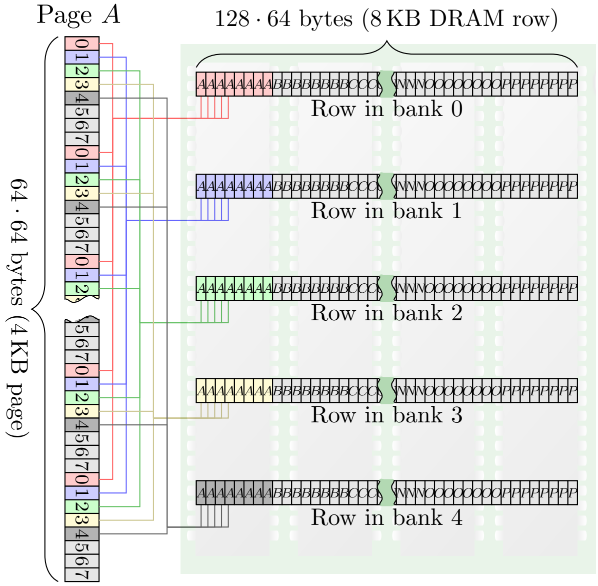


Figure 8.6: Mapping between a 4 KB page and an 8 KB DRAM row in the Haswell-EP setup. Banks are numbered $0-7$, pages are numbered $A-P$. Every eighth 64-byte region of a 4 KB page maps to the same bank in DRAM. In total 8 out of 64 regions ($= 512\,B$) map to the same bank. Thus, the memory of each row is divided among 16 different pages ($A-P$) that use memory from the same row. Occupying one of the pages $B-P$ is sufficient to spy on the eight 64-byte regions of page A in the same bank.

pages are placed in a row together with pages used by the victim. The attacker then profiles the entire allocated memory and records the row-hit ratio for each address.

False-positive detections are eliminated by running the profiling phase with different events. If an address has a high row-hit ratio for a single event, it can be used to monitor that event in the exploitation phase. After such an address has been found, all other remaining memory pages will be released and the exploitation phase is started.

8.3.2 Evaluation

We evaluated the performance of our side-channel attack in several tests. These tests were performed on a dual-core laptop with an Ivy Bridge Intel i5-3230M CPU with 2 Samsung DDR3-1600 dual-rank 4 GB DIMMs in dual-channel configuration.

The first test was a DRAMA template attack. The attack ran without any shared memory in an unprivileged user program. In this template attack we profiled access times on a private memory buffer while triggering keystrokes in

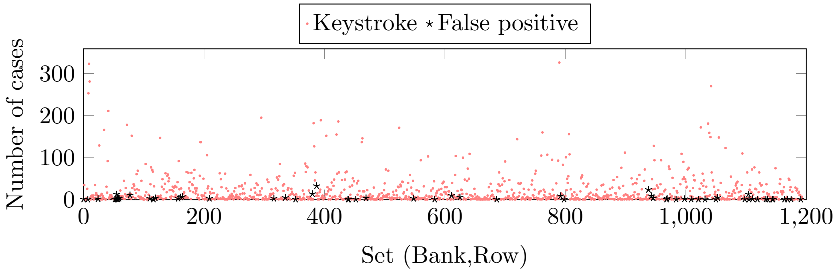


Figure 8.7: A DRAM template of the system memory with and without triggering keystrokes in the Firefox address bar. 1136 sets had row hits after a keystroke, 59 sets had false positive row hits (row hits without a keystroke), measured on our Ivy Bridge i5 test system.

the Firefox address bar. Figure 8.7 shows the template attack profile with and without keystrokes being triggered. While scanning a total of 7 GB of allocated memory, we found 1195 addresses that showed at least one row hit during the tests. 59 of these addresses had row hits independent of the event (false positives), i.e., these 59 addresses cannot be used to monitor keystroke events. For the remaining 1136 addresses we only had row hits after triggering a keystroke in the Firefox address bar. Out of these addresses, 360 addresses had more than 20 row hits. Any of these 360 addresses can be used to monitor keystrokes reliably. The time to find an exploitable address varies between a few seconds and multiple minutes. Sometimes the profiling phase does not find any exploitable address, for instance if there is no memory in one row with victim memory. In this case the attacker has to restart the profiling phase.

After automatically switching to the exploitation phase we can monitor the exact timestamp of every keystroke in the address bar (but not the concretely pressed key). We verified empirically that row hits can be measured on the found addresses after keystrokes by triggering keystrokes by hand. Figure 8.8 shows an access time trace for an address found in a DRAMA template attack, while typing in the Firefox address bar. For every key the user presses, a low access time is measured. We found this address after less than 2 seconds. Over 80 seconds we measured no false positive row hits and when pressing 40 keys we measured no false negatives. During this test the system was entirely idle apart from the attack and the user typing in Firefox. In a real attack noise would introduce false negatives.

Comparison with cache template attacks. To compare DRAMA template attacks with cache template attacks, we performed two attacks on gedit. The first uses the result from a cache template attack in a DRAMA exploitation phase. The second is a modified cache template attack that uses the DRAMA side channel. Both attacks use shared memory to be able to compare them with cache template attacks. However, the DRAMA side-channel attack takes no advantage of shared memory in any attack.

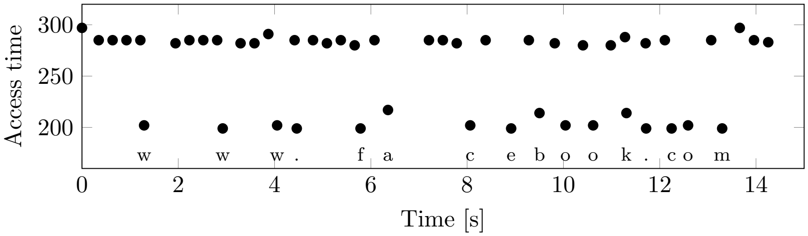


Figure 8.8: Exploitation phase on non-shared memory in a DRAMA template attack on our Ivy Bridge i5 test system. A low access time is measured when the user presses a key in the Firefox address bar. The typing gaps illustrate the low noise level.

In the first attack on gedit, we target tab open and tab close events. In an experiment over 120 seconds we opened a new tab and closed the new tab 50 times. The exploitable address in the shared library was found in a cache template attack. We computed the physical address and thus bank and row of the exploitable address using privileged operating services. Then we allocated large arrays to obtain memory that maps to the same row (and bank). This allows us to perform an attack that has only minimal differences to a Flush+Reload attack.

During this attack, our spy tool detected 1 false positive row hit and 1 false negative row hit. Running `stress -m 1` in parallel, which allocates and accesses large memory buffers, causes a high number of cache misses, but did not introduce a significant amount of noise. In this experiment the spy tool detected no false positive row hits and 4 false negative row hits. Running `stress -m 2` in parallel (i.e., the attacker's core is under stress) made any measurements impossible. While no false positive detections occurred, only 9 events were correctly detected. Thus, our attack is susceptible to noise especially if the attacker only gets a fraction of CPU time on its core.

In the second attack we compared the cache side channel and the DRAM side channel in a template attack on keystrokes in gedit. Figure 8.9 shows the number of cache hits and row hits over the virtual memory where the `gedit` binary is mapped. Row hits occur in spatial proximity to the cache hits and at shifted offsets due to the DRAM address mappings.

8.3.3 Comparison with State of the Art

We now compare DRAMA side-channel attacks with same-CPU cache attacks such as Flush+Reload and Prime+Probe, as well as with cross-CPU cache attacks [IES16]. Our attack is the first to enable monitoring non-shared memory cross-CPU with a reasonably high spatial accuracy and a timing accuracy that is comparable to Flush+Reload. This allows the development of new attacks on programs using dynamically allocated or private memory.

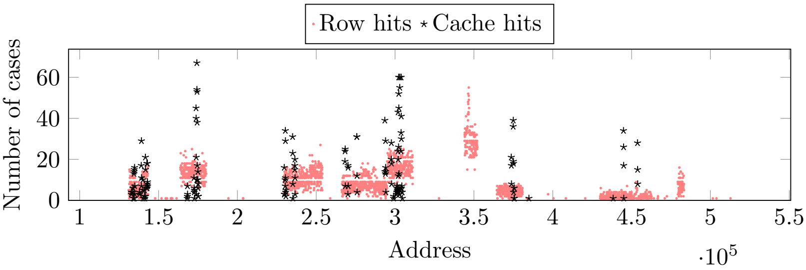


Figure 8.9: Comparison of a cache hits and row hits over the virtual memory where the `gedit` binary is mapped, measured on our Ivy Bridge i5 test system.

The spatial accuracy of the DRAMA side-channel attack is significantly higher than that of a Prime+Probe attack, which also does not necessitate shared memory, and only slightly lower than that of a Flush+Reload attack in most cases. Our Ivy Bridge i5 system has 8 GB DRAM and a 3 MB L3 cache that is organized in 2 cache slices with 2048 cache sets each. Thus, in a Prime+Probe attack 32768 memory lines map to the same cache set, whereas in our DRAMA side-channel attack, on the same system, only 32 memory lines map to the same row. The spatial accuracy strongly depends on the system. On our Haswell-EP system only 8 memory lines map to the same row whereas still 32768 memory lines map to the same cache set. Thus, on the Haswell-EP system the advantage of DRAMA side-channel attacks over Prime+Probe is even more significant.

For allocating memory lines that are in the same row as victim memory lines, it is necessary to allocate significantly larger memory buffers than in a cache attack like Prime+Probe. This is a clear disadvantage of DRAMA side-channel attacks. However, DRAMA side-channel attacks have a very low probability of false positive row hit detections, whereas Prime+Probe is highly susceptible to noise. Due to this noise, monitoring singular events using Prime+Probe is extremely difficult.

Irazoqui et al. [IES16] presented cache-based cross-CPU side-channel attacks. However, their work requires shared memory. Our approach works without shared memory. Not only does this allow cross-CPU attacks in highly restricted environments, but it also allows to perform a new kind of cross-core attack within one system.

8.4 Countermeasures

Defending against row buffer attacks is a difficult task. Making the corresponding DRAM operations constant time might introduce unacceptable performance degradation. However, as long as the timing difference exists and can be measured, the side channel cannot be closed.

Our attack implementations use the unprivileged `clflush` instruction in order to cause a DRAM access with every memory request. Thus, one countermeasure might be to restrict said operation. However, this requires architectural changes and an attacker can still use eviction as a replacement. The additional memory accesses caused by eviction could make our row-buffer covert channel impractical. However, other attacks such as the fully automated reverse engineering or our row-hit side-channel attack are still possible. Restricting the `rdtsc` instruction would also not prevent an attack as other timing sources can be used as a replacement.

To prevent cross-VM attacks on multi-CPU cloud systems, the cloud provider could schedule each VM on a dedicated physical CPU and only allow access to CPU-local DRAM. This can be achieved by using a non-interleaved NUMA configuration and assigning pages to VMs carefully. This approach essentially splits a multi-CPU machine into independent single-CPU systems, which leads to a loss of many of its advantages.

Saltaformaggio et al. [SXZ13] presented a countermeasure to the memory bus-based covert channel of Wu et al. It intercepts atomic instructions that are responsible for this covert channel, so that only cores belonging to the attacker's VM are locked, instead of the whole machine. This countermeasure is not effective against our attacks as they do not rely on atomic instructions.

Finally, our attack could be detected due to the high number of cache misses. However, it is unclear whether it is possible to distinguish our attacks from non-malicious applications.

8.5 Conclusion

In this chapter, we presented two methods to reverse engineer the mapping of physical memory addresses to DRAM channels, ranks, and banks. One uses physical probing of the memory bus, the other runs entirely in software and is fully automated. We ran our method on a wide range of architectures, including desktop, server, and mobile platforms.

Based on the reverse-engineered functions, we demonstrated DRAMA (DRAM addressing) attacks. This novel class of attacks exploits the DRAM row buffer that is a shared resource in single and multi-processor systems. This allows our attacks to work in the most restrictive environments, i.e., across processors and without any shared memory. We built a covert channel with a capacity of 2 Mbps, which is three to four orders of magnitude faster than memory-bus-based channels in the same setting. We demonstrated a side-channel template attack automatically locating and monitoring memory accesses, e.g., user input, server requests. This side-channel attack is as accurate as recent cache attacks like Flush+Reload, while requiring no shared memory between the victim and the spy. Finally, we show how to use the reverse-engineered DRAM addressing functions to improve existing attacks, such as Flush+Reload and Rowhammer. Our work enables practical Rowhammer attacks on DDR4.

We emphasize the importance of reverse engineering microarchitectural components for security reasons. Before we reverse engineered the DRAM address mapping, the DRAM row buffer was transparent to the operating system and other software. Only by reverse engineering we made this shared resource visible and were able to identify it as a powerful side channel.

9

Conclusions

Side-channel attacks aim at moving targets. Cryptographic algorithms as well as (general-purpose) hardware never stop evolving, which creates a constant influx of new challenges for both attackers and defenders. In this thesis, we extended the understanding of possible attacks in two main directions.

First, we showed new side-channel attacks on lattice-based cryptography, which is a possible contender for replacing current public-key cryptosystems. A common theme in all proposed attacks is the exploitation of some algebraic structure or feature not seen in more classic and established constructions. Lattice basis reductions allow to combine public information (the public key) with side-channel leakage and inferred partial knowledge of the private key. We further exploited the difficulty in high-precision sampling from Gaussian distributions, the sparsity of private keys, and vastly different distributions of certain intermediate variables. Another key point is the linearity of many involved operations allowing efficient gathering and combination of side-channel information across multiple algorithm invocations. Finally, many of the attacks target key-independent subroutines; thus Differential Power Analysis (DPA) is not applicable there. All this clearly shows that defending solely against "classic" side-channel techniques such as DPA is not enough and that new attack techniques enabled by novel algorithms and their algebraic structures have to be considered and to be understood.

Second, we presented a new microarchitectural side-channel attack capable of targeting even multi-processor systems, a common class of machines for cloud and server applications. Our DRAMA techniques highlight the dangers of hardware sharing and that disabling it after the fact, e.g., by assigning physical CPUs exclusively, is not trivial and can leave open other attack paths.

Outlook

While this thesis uncovers new attack paths for multi-processor systems and implementations of lattice-based cryptography, there are still many open questions and further potential for side-channel attacks. Especially the recent rapid development of post-quantum cryptography, driven by the NIST call for proposals, has lead to an increased need for side-channel evaluations and secure implementation techniques. We now give some more concrete examples of possible future research directions in this field.

Single-trace attacks on lattice-based cryptography. Many lattice-based key-exchange schemes use only ephemeral, i.e., one-time use, secrets. This property makes them a challenging target for side-channel attacks, as multi-trace attacks like DPA, but also the attack presented in Chapter 4 are inherently prevented. However, single-trace attacks still have to be considered, as demonstrated by Primas et al. [PPM17]. This attack targets the number-theoretic transform (NTT) and combines side-channel leakage across the entire algorithm using belief propagation. However, it requires a powerful attacker capable of building a large number of side-channel templates. An interesting question is if this attack can be made more practical and can be mounted by less powerful adversaries. Some very recent work [GRO18; GGS18] using belief propagation in side-channel attacks, albeit in the context of AES, gives potential directions for improvements.

More general fault attacks. The fault attack presented in Chapter 6 targets deterministic signature schemes and furthermore requires that the same message is signed twice. The susceptibility of non-deterministic signatures to certain fault models was also already analyzed by Bindel et al. [BBK16]. Extending the applicability of fault attacks to key-exchange schemes (those that use long-term secrets) is an interesting open problem. One way to achieve practical attacks might be the adaptation of statistical fault attacks [Fuh+13; Dob+18a] to the lattice scenario.

Analysis of new schemes. Due to the ongoing NIST call, there is no shortage of new schemes and implementation techniques to be analyzed for side-channel vulnerabilities. Many features, such as the use of the NTT, are common to many proposals. Hence, advances in this direction can be applied to a large number of schemes.

Some other techniques are unique to a small number of schemes, but can still make for interesting targets. Noteworthy examples are the complex trapdoor sampling employed by the Falcon signature scheme [Fou+17] or the use of the Learning with Rounding problem in, e.g., Round5 [Bha+18].

Bibliography

[ADP18] Martin R. Albrecht, Amit Deo, and Kenneth G. Paterson. *Cold Boot Attacks on Ring and Module LWE Keys Under the NTT.* Cryptology ePrint Archive, Report 2018/672. To appear at CHES 2018. 2018.

[Adv13] Advanced Micro Devices. *BIOS and Kernel Developer's Guide (BKDG) for AMD Family 15h Models 00h-0Fh Processors.* `http://support.amd.com/TechDocs/42301_15h_Mod_00h-0Fh_BKDG.pdf`. 2013.

[Ajt96] Miklós Ajtai. "Generating Hard Instances of Lattice Problems." In: *Electronic Colloquium on Computational Complexity (ECCC)* 3.7 (1996).

[Akl+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: *AFRICACRYPT*. Vol. 9646. LNCS. Springer, 2016, pp. 44–60.

[Alb+18] Martin R. Albrecht, Christian Hanser, Andrea Höller, Thomas Pöppelmann, Fernando Virdia, and Andreas Wallner. *Learning with Errors on RSA Co-Processors.* Cryptology ePrint Archive, Report 2018/425. 2018.

[Alk+16] Erdem Alkim, Léo Ducas, Thomas Pöppelmann, and Peter Schwabe. "Post-quantum Key Exchange – A New Hope." In: *USENIX Security Symposium*. USENIX Association, 2016, pp. 327–343.

[Alk+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: *PQCrypto*. Vol. 10346. LNCS. Full version available at `https://ia.cr/2015/755`. Springer, 2017, pp. 143–162.

[All+16] Thomas Allan, Billy Bob Brumley, Katrina E. Falkner, Joop van de Pol, and Yuval Yarom. "Amplifying side channels through performance degradation." In: *ACSAC*. ACM, 2016, pp. 422–435.

[Amb+18] Christopher Ambrose, Joppe W. Bos, Björn Fay, Marc Joye, Manfred Lochter, and Bruce Murray. "Differential Attacks on Deterministic Signatures." In: *CT-RSA*. Vol. 10808. LNCS. Springer, 2018, pp. 339–353.

[Ava+17] Roberto Avanzi, Joope Bos, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, John M. Schank, Peter Schwabe, Gregor Seiler, and Damien Stehlé. *CRYSTALS-Kyber*. Submission to the NIST Post-Quantum Cryptography Standardization [NISc]. `https://pq-crystals.org/kyber`. 2017.

[Bar+06] Hagai Bar-El, Hamid Choukri, David Naccache, Michael Tunstall, and Claire Whelan. "The Sorcerer's Apprentice Guide to Fault Attacks." In: *Proceedings of the IEEE* 94.2 (2006), pp. 370–382.

[Bar+18] Gilles Barthe, Sonia Belaïd, Thomas Espitau, Pierre-Alain Fouque, Benjamin Grégoire, Mélissa Rossi, and Mehdi Tibouchi. "Masking the GLP Lattice-Based Signature Scheme at Any Order." In: *EUROCRYPT (2)*. Vol. 10821. LNCS. Springer, 2018, pp. 354–384.

[BBK16] Nina Bindel, Johannes A. Buchmann, and Juliane Krämer. "Lattice-Based Signature Schemes and Their Sensitivity to Fault Attacks." In: *FDTC*. IEEE Computer Society, 2016, pp. 63–77.

[BDL97] Dan Boneh, Richard A. DeMillo, and Richard J. Lipton. "On the Importance of Checking Cryptographic Protocols for Faults (Extended Abstract)." In: *EUROCRYPT*. Vol. 1233. LNCS. Springer, 1997, pp. 37–51.

[Bel+15] Sonia Belaïd, Jean-Sébastien Coron, Pierre-Alain Fouque, Benoît Gérard, Jean-Gabriel Kammerer, and Emmanuel Prouff. "Improved Side-Channel Analysis of Finite-Field Multiplication." In: *CHES*. Vol. 9293. LNCS. Springer, 2015, pp. 395–415.

[Ben+14] 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: *CHES*. Vol. 8731. LNCS. Springer, 2014, pp. 75–92.

[Ber] Daniel J. Bernstein. *CAESAR: Competition for Authenticated Encryption: Security, Applicability, and Robustness*. `https://competitions.cr.yp.to/caesar.html`.

[Ber+11] Daniel J. Bernstein, Niels Duif, Tanja Lange, Peter Schwabe, and Bo-Yin Yang. "High-Speed High-Security Signatures." In: *CHES*. Vol. 6917. LNCS. Springer, 2011, pp. 124–142.

[Ber05] Daniel J. Bernstein. *Cache-timing attacks on AES*. Preprint available at `http://cr.yp.to/antiforgery/cachetiming-20050414.pdf`. 2005.

[BFG14] Sonia Belaïd, Pierre-Alain Fouque, and Benoît Gérard. "Side-Channel Analysis of Multiplications in GF(2^{128}) - Application to AES-GCM." In: *ASIACRYPT (2)*. Vol. 8874. LNCS. Springer, 2014, pp. 306–325.

[Bha+18] Sauvik Bhattacharya, Oscar Garcia-Morchon, Thijs Laarhoven, Ronald Rietman, Markku-Juhani O. Saarinen, Ludo Tolhuizen, and Zhenfei Zhang. *Round5*. Submission to the NIST Post-Quantum Cryptography Standardization [NISc]. Merger of proposals Round2 and Hila5. `https://round5.org/`. 2018.

[Bin+17a] 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 [NISc]. `https://qtesla.org`. 2017.

[Bin+17b] Nina Bindel, Johannes A. Buchmann, Juliane Krämer, Heiko Mantel, Johannes Schickel, and Alexandra Weber. “Bounding the Cache-Side-Channel Leakage of Lattice-Based Signature Schemes Using Program Semantics.” In: *FPS*. Vol. 10723. LNCS. Springer, 2017, pp. 225–241.

[BKW03] Avrim Blum, Adam Kalai, and Hal Wasserman. “Noise-tolerant learning, the parity problem, and the statistical query model.” In: *J. ACM* 50.4 (2003), pp. 506–519.

[BL17] Daniel J. Bernstein and Tanja Lange. *Post-quantum cryptography—dealing with the fallout of physics success.* Cryptology ePrint Archive, Report 2017/314. 2017.

[BLP08] Daniel J. Bernstein, Tanja Lange, and Christiane Peters. “Attacking and Defending the McEliece Cryptosystem.” In: *PQCrypto*. Vol. 5299. LNCS. Springer, 2008, pp. 31–46.

[Bos+15] 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*. IEEE Computer Society, 2015, pp. 553–570.

[BP16] Alessandro Barenghi and Gerardo Pelosi. “A Note on Fault Attacks Against Deterministic Signature Schemes.” In: *IWSEC*. Vol. 9836. LNCS. Springer, 2016, pp. 182–192.

[Bra16] Matt Braithwaite. *Experimenting with Post-Quantum Cryptography.* `https://security.googleblog.com/2016/07/experimenting-with-post-quantum.html`. July 2016.

[Buc+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: *SAC*. Vol. 8282. LNCS. Springer, 2013, pp. 402–417.

[Bye+10] Ben Byer, Hector Martin Cantero, Segher Boessenkool, and Sven Peter. *PS3 Epic Fail.* 27th Chaos Communication Congress. `https://events.ccc.de/congress/2010/Fahrplan/events/4087.en.html`. 2010.

[CC98] Anne Canteaut and Florent Chabaud. "A New Algorithm for Finding Minimum-Weight Words in a Linear Code: Application to McEliece's Cryptosystem and to Narrow-Sense BCH Codes of Length 511." In: *IEEE Trans. Information Theory* 44.1 (1998), pp. 367–378.

[CG90] J.T. Coffey and R.M. Goodman. "Any code of which we cannot think is good." In: *Information Theory, IEEE Transactions on* 36.6 (1990), pp. 1453–1461.

[CN11] Yuanmi Chen and Phong Q. Nguyen. "BKZ 2.0: Better Lattice Security Estimates." In: *ASIACRYPT*. Vol. 7073. LNCS. Springer, 2011, pp. 1–20.

[CRR02] Suresh Chari, Josyula R. Rao, and Pankaj Rohatgi. "Template Attacks." In: *CHES*. Vol. 2523. LNCS. Springer, 2002, pp. 13–28.

[DG14] Nagarjun C. Dwarakanath and Steven D. Galbraith. "Sampling from discrete Gaussians for lattice-based cryptography on a constrained device." In: *Appl. Algebra Eng. Commun. Comput.* 25.3 (2014), pp. 159–180.

[Dob+16] Christoph Dobraunig, Maria Eichlseder, Thomas Korak, Victor Lomné, and Florian Mendel. "Statistical Fault Attacks on Nonce-Based Authenticated Encryption Schemes." In: *ASIACRYPT (1)*. Vol. 10031. LNCS. 2016, pp. 369–395.

[Dob+18a] Christoph Dobraunig, Maria Eichlseder, Hannes Groß, Stefan Mangard, Florian Mendel, and Robert Primas. *Statistical Ineffective Fault Attacks on Masked AES with Fault Countermeasures*. Cryptology ePrint Archive, Report 2018/357. 2018.

[Dob+18b] Christoph Dobraunig, Maria Eichlseder, Thomas Korak, Stefan Mangard, Florian Mendel, and Robert Primas. *Exploiting Ineffective Fault Inductions on Symmetric Cryptography*. Cryptology ePrint Archive, Report 2018/071. To appear at CHES 2018. 2018.

[Duc+13] Léo Ducas, Alain Durmus, Tancrède Lepoint, and Vadim Lyubashevsky. "Lattice Signatures and Bimodal Gaussians." In: *CRYPTO (1)*. Vol. 8042. LNCS. Springer, 2013, pp. 40–56.

[Duc+17] Leo Ducas, Tancrede Lepoint, Vadim Lyubashevsky, Peter Schwabe, Gregor Seiler, and Damien Stehle. *CRYSTALS – Dilithium: Digital Signatures from Module Lattices*. Cryptology ePrint Archive, Report 2017/633. 2017.

[Duc14] Léo Ducas. *Accelerating Bliss: the geometry of ternary polynomials*. Cryptology ePrint Archive, Report 2014/874. 2014.

[ECR] ECRYPT. *eSTREAM: the ECRYPT Stream Cipher Project*. `http://www.ecrypt.eu.org/stream/`.

[Esp+16] Thomas Espitau, Pierre-Alain Fouque, Benoît Gérard, and Mehdi Tibouchi. "Loop-Abort Faults on Lattice-Based Fiat-Shamir and Hash-and-Sign Signatures." In: *SAC*. Vol. 10532. LNCS. Springer, 2016, pp. 140–158.

[Esp+17] Thomas Espitau, Pierre-Alain Fouque, Benoît Gérard, and Mehdi Tibouchi. "Side-Channel Attacks on BLISS Lattice-Based Signatures: Exploiting Branch Tracing against strongSwan and Electromagnetic Emanations in Microcontrollers." In: *CCS*. ACM, 2017, pp. 1857–1874.

[Fou+17] Pierre-Alain Fouque, Jeffrey Hoffstein, Paul Kirchner, Vadim Lyubashevsky, Thomas Pornin, Thomas Prest, Thomas, Ricosset, Gregor Seiler, William Whyte, and Zhenfei Zhang. *Falcon.* Submission to the NIST Post-Quantum Cryptography Standardization [NISc]. `https://falcon-sign.info/`. 2017.

[Fuh+13] Thomas Fuhr, Éliane Jaulmes, Victor Lomné, and Adrian Thillard. "Fault Attacks on AES with Faulty Ciphertexts Only." In: *FDTC*. IEEE Computer Society, 2013, pp. 108–118.

[FW10] M. Fernandez and S. Williams. "Closed-Form Expression for the Poisson-Binomial Probability Density Function." In: *Aerospace and Electronic Systems, IEEE Transactions on* 46.2 (Apr. 2010), pp. 803–817.

[GBM15] Daniel Gruss, David Bidner, and Stefan Mangard. "Practical Memory Deduplication Attacks in Sandboxed Javascript." In: *ESORICS (1)*. Vol. 9326. LNCS. Springer, 2015, pp. 108–122.

[Ge+18] Qian Ge, Yuval Yarom, David Cock, and Gernot Heiser. "A survey of microarchitectural timing attacks and countermeasures on contemporary hardware." In: *J. Cryptographic Engineering* 8.1 (2018), pp. 1–27.

[GGS18] Qian Guo, Vincent Grosso, and François-Xavier Standaert. *Modeling Soft Analytical Side-Channel Attacks from a Coding Theory Viewpoint*. Cryptology ePrint Archive, Report 2018/498. 2018.

[GJL14] Qian Guo, Thomas Johansson, and Carl Löndahl. "Solving LPN Using Covering Codes." In: *ASIACRYPT (1)*. Vol. 8873. LNCS. Springer, 2014, pp. 1–20.

[GLP06] Benedikt Gierlichs, Kerstin Lemke-Rust, and Christof Paar. "Templates vs. Stochastic Methods." In: *CHES*. Vol. 4249. LNCS. Springer, 2006, pp. 15–29.

[GLP12] Tim Güneysu, Vadim Lyubashevsky, and Thomas Pöppelmann. "Practical Lattice-Based Cryptography: A Signature Scheme for Embedded Systems." In: *CHES*. Vol. 7428. LNCS. Springer, 2012, pp. 530–547.

[GMM16] Daniel Gruss, Clémentine Maurice, and Stefan Mangard. “Rowhammer.js: A Remote Software-Induced Fault Attack in JavaScript.” In: *DIMVA*. Vol. 9721. LNCS. Springer, 2016, pp. 300–321.

[GMO01] Karine Gandolfi, Christophe Mourtel, and Francis Olivier. “Electromagnetic Analysis: Concrete Results.” In: *CHES*. Vol. 2162. LNCS Generators. Springer, 2001, pp. 251–261.

[Gro+16] 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: *CHES*. Vol. 9813. LNCS. Full version available at: `http://ia.cr/2016/300`. Springer, 2016, pp. 323–345.

[GRO18] Joey Green, Arnab Roy, and Elisabeth Oswald. *A Systematic Study of the Impact of Graphical Models on Inference-based Attacks on AES*. Cryptology ePrint Archive, Report 2018/671. 2018.

[Gru+16] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. “Flush+Flush: A Fast and Stealthy Cache Attack.” In: *DIMVA*. Vol. 9721. LNCS. Springer, 2016, pp. 279–299.

[GS15] Gabriel Goller and Georg Sigl. “Side Channel Attacks on Smartphones and Embedded Devices Using Standard Radio Equipment.” In: *COSADE*. Vol. 9064. LNCS. Springer, 2015, pp. 255–270.

[GSM15] Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. “Cache Template Attacks: Automating Attacks on Inclusive Last-Level Caches.” In: *USENIX Security Symposium*. USENIX Association, 2015, pp. 897–912.

[GST14] Daniel Genkin, Adi Shamir, and Eran Tromer. “RSA Key Extraction via Low-Bandwidth Acoustic Cryptanalysis.” In: *CRYPTO (1)*. Vol. 8616. LNCS. Springer, 2014, pp. 444–461.

[HKP15] Mohamed Hassan, Anirudh M. Kaushik, and Hiren D. Patel. “Reverse-engineering embedded memory controllers through latency-based analysis.” In: *RTAS*. IEEE Computer Society, 2015, pp. 297–306.

[HLS18] Andreas Hülsing, Tanja Lange, and Kit Smeets. “Rounded Gaussians - Fast and Secure Constant-Time Sampling for Lattice-Based Crypto.” In: *Public Key Cryptography (2)*. Vol. 10770. LNCS. Springer, 2018, pp. 728–757.

[Hof+14] Jeffrey Hoffstein, Jill Pipher, John M. Schanck, Joseph H. Silverman, and William Whyte. “Practical Signatures from the Partial Fourier Recovery Problem.” In: *ACNS*. Vol. 8479. LNCS. Springer, 2014, pp. 476–493.

[How+18] James Howe, Tobias Oder, Markus Krausz, and Tim Güneysu. *Standard Lattice-Based Key Encapsulation on Embedded Devices*. Cryptology ePrint Archive, Report 2018/686. To appear at CHES 2018. 2018.

[HS13] Michael Hutter and Jörn-Marc Schmidt. “The Temperature Side Channel and Heating Fault Attacks.” In: *CARDIS*. Vol. 8419. LNCS. Springer, 2013, pp. 219–235.

[Hua+12] Rei-Fu Huang, Hao-Yu Yang, Mango Chia-Tso Chao, and Shih-Chin Lin. “Alternate hammering test for application-specific DRAMs and an industrial case study.” In: *DAC*. ACM, 2012, pp. 1012–1017.

[IES16] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. “Cross Processor Cache Attacks.” In: *AsiaCCS*. ACM, 2016, pp. 353–364.

[Inc+15] Mehmet Sinan Inci, Berk Gulmezoglu, Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. *Seriously, get off my cloud! Cross-VM RSA Key Recovery in a Public Cloud*. Cryptology ePrint Archive, Report 2015/898. 2015.

[Int15] Intel Corporation. *Intel® Xeon® Processor E5 v3 Product Family – Processor Specification Update*. 330785-009US. Aug. 2015.

[Ira+14] Gorka Irazoqui, Mehmet Sinan Inci, Thomas Eisenbarth, and Berk Sunar. “Wait a Minute! A fast, Cross-VM Attack on AES.” In: *RAID*. Vol. 8688. LNCS. Springer, 2014, pp. 299–319.

[Jaf07] Joshua Jaffe. “A First-Order DPA Attack Against AES in Counter Mode with Unknown Initial Counter.” In: *CHES*. Vol. 4727. LNCS. Springer, 2007, pp. 1–13.

[Kar+18] Angshuman Karmakar, Sujoy Sinha Roy, Oscar Reparaz, Frederik Vercauteren, and Ingrid Verbauwhede. “Constant-time Discrete Gaussian Sampling.” In: *IEEE TC* (2018).

[Kel18] Julian Kelly. *A Preview of Bristlecone, Google’s New Quantum Processor*. `https://research.googleblog.com/2018/03/a-preview-of-bristlecone-googles-new.html`. 2018.

[Kim+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*. IEEE Computer Society, 2014, pp. 361–372.

[KJJ99] Paul C. Kocher, Joshua Jaffe, and Benjamin Jun. “Differential Power Analysis.” In: *CRYPTO*. Vol. 1666. LNCS. Springer, 1999, pp. 388–397.

[KLS18] Eike Kiltz, Vadim Lyubashevsky, and Christian Schaffner. “A Concrete Treatment of Fiat-Shamir Signatures in the Quantum Random-Oracle Model.” In: *EUROCRYPT (3)*. Vol. 10822. LNCS. Springer, 2018, pp. 552–586.

[Kni17] Will Knight. *IBM Raises the Bar with a 50-Qubit Quantum Computer*. `https://www.technologyreview.com/s/609451/ibm-raises-the-bar-with-a-50-qubit-quantum-computer/`. 2017.

[Knu98] Donald E. Knuth. “Seminumerical Algorithms.” In: 3rd. Vol. 2. The Art of Computer Programming. Addison-Wesley, 1998. Chap. 3, pp. 145–146.

[Koc+18] Paul Kocher, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. *Spectre Attacks: Exploiting Speculative Execution*. `https://meltdownattack.com`. 2018.

[Koc96] Paul C. Kocher. “Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems.” In: *CRYPTO*. Vol. 1109. LNCS. Springer, 1996, pp. 104–113.

[Lan16] Adam Langley. *CECPQ1 results*. `https://www.imperialviolet.org/2016/11/28/cecpq1.html`. Nov. 2016.

[LF06] Éric Levieil and Pierre-Alain Fouque. “An Improved LPN Algorithm.” In: *SCN*. Vol. 4116. LNCS. Springer, 2006, pp. 348–359.

[Lip+18] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. *Meltdown*. `https://meltdownattack.com`. 2018.

[Liu+15a] 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*. IEEE Computer Society, 2015, pp. 605–622.

[Liu+15b] Zhe Liu, Hwajeong Seo, Sujoy Sinha Roy, Johann Großschädl, Howon Kim, and Ingrid Verbauwhede. “Efficient Ring-LWE Encryption on 8-Bit AVR Processors.” In: *CHES*. Vol. 9293. LNCS. Springer, 2015, pp. 663–682.

[LLL82] A. K. Lenstra, H. W. Lenstra, and L. Lovász. “Factoring polynomials with rational coefficients.” In: *Mathematische Annalen* 261.4 (1982), pp. 515–534.

[LP11] Richard Lindner and Chris Peikert. “Better Key Sizes (and Attacks) for LWE-Based Encryption.” In: *CT-RSA*. Vol. 6558. LNCS. Springer, 2011, pp. 319–339.

[LPR13] Vadim Lyubashevsky, Chris Peikert, and Oded Regev. “A Toolkit for Ring-LWE Cryptography.” In: *EUROCRYPT*. Vol. 7881. LNCS. Springer, 2013, pp. 35–54.

[Lyu+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 [NISc]. `https://pq-crystals.org/dilithium`. 2017.

[Lyu09] Vadim Lyubashevsky. “Fiat-Shamir with Aborts: Applications to Lattice and Factoring-Based Signatures.” In: *ASIACRYPT*. Vol. 5912. LNCS. Springer, 2009, pp. 598–616.

[Mar14] Matteo Mariantoni. *Building a superconducting quantum computer – Invited Talk in PQCrypto 2014*. https://www.youtube.com/watch?v=wWHAs--HA1c. Oct. 2014.

[Mau+15a] Clémentine Maurice, Christoph Neumann, Olivier Heen, and Aurélien Francillon. "C5: Cross-Cores Cache Covert Channel." In: *DIMVA*. Vol. 9148. LNCS. Springer, 2015, pp. 46–64.

[Mau+15b] Clémentine Maurice, Nicolas Le Scouarnec, Christoph Neumann, Olivier Heen, and Aurélien Francillon. "Reverse Engineering Intel Last-Level Cache Complex Addressing Using Performance Counters." In: *RAID*. Vol. 9404. LNCS. Springer, 2015, pp. 48–65.

[MH15] Amir Moradi and Gesine Hinterwälder. "Side-Channel Security Analysis of Ultra-Low-Power FRAM-Based MCUs." In: *COSADE*. Vol. 9064. LNCS. Springer, 2015, pp. 239–254.

[Mic14] Micron. *DDR4 SDRAM*. https://www.micron.com/~/media/documents/products/data-sheet/dram/ddr4/4gb_ddr4_sdram.pdf. Retrieved on February 17, 2016. 2014.

[Moo17] Dustin Moody. *The Ship has Sailed - The NIST Post-Quantum Crypto "Competition"*. Invited Talk at ASIACRYPT 2017. https://csrc.nist.gov/CSRC/media//Projects/Post-Quantum-Cryptography/documents/asiacrypt-2017-moody-pqc.pdf. 2017.

[MOP07] Stefan Mangard, Elisabeth Oswald, and Thomas Popp. *Power analysis attacks – revealing the secrets of smart cards*. Springer, 2007.

[MW17] Daniele Micciancio and Michael Walter. "Gaussian Sampling over the Integers: Efficient, Generic, Constant-Time." In: *CRYPTO (2)*. Vol. 10402. LNCS. Springer, 2017, pp. 455–485.

[NISa] NIST. *AES Development - Cryptographic Standards and Guidelines*. https://csrc.nist.gov/projects/cryptographic-standards-and-guidelines/archived-crypto-projects/aes-development.

[NISb] NIST. *Lightweight Cryptography*. https://csrc.nist.gov/Projects/Lightweight-Cryptography.

[NISc] NIST. *Post-Quantum Cryptography Standardization*. https://csrc.nist.gov/projects/post-quantum-cryptography/post-quantum-cryptography-standardization.

[NISd] NIST. *SHA-3 Project - Hash Functions*. https://csrc.nist.gov/projects/hash-functions/sha-3-project.

[NSA16] NSA/IAD. *CNSA Suite and Quantum Computing FAQ*. https://www.iad.gov/iad/library/ia-guidance/ia-solutions-for-classified/algorithm-guidance/cnsa-suite-and-quantum-computing-faq.cfm. Jan. 2016.

[NSA72] NSA. *TEMPEST: A Signal Problem*. 1972.

[Ode+18] Tobias Oder, Tobias Schneider, Thomas Pöppelmann, and Tim Güneysu. “Practical CCA2-Secure and Masked Ring-LWE Implementation.” In: *TCHES* 2018.1 (2018), pp. 142–174.

[OPG14] Tobias Oder, Thomas Pöppelmann, and Tim Güneysu. “Beyond ECDSA and RSA: Lattice-based Digital Signatures on Constrained Devices.” In: *DAC*. ACM, 2014, 110:1–110:6.

[Ore+15] Yossef Oren, Vasileios P. Kemerlis, Simha Sethumadhavan, and Angelos D. Keromytis. “The Spy in the Sandbox: Practical Cache Attacks in JavaScript and their Implications.” In: *ACM Conference on Computer and Communications Security*. ACM, 2015, pp. 1406–1418.

[OST06] Dag Arne Osvik, Adi Shamir, and Eran Tromer. “Cache Attacks and Countermeasures: The Case of AES.” In: *CT-RSA*. Vol. 3860. LNCS. Springer, 2006, pp. 1–20.

[Par+14] Kyungbae Park, Sanghyeon Baeg, ShiJie Wen, and Richard Wong. “Active-Precharge Hammering on a Row Induced Failure in DDR3 SDRAMs under 3x nm Technology.” In: *IIRW*. 2014, pp. 82–85.

[PB17] Cesar Pereida García and Billy Bob Brumley. “Constant-Time Callees with Variable-Time Callers.” In: *USENIX Security Symposium*. USENIX Association, 2017, pp. 83–98.

[PBY16] Cesar Pereida García, Billy Bob Brumley, and Yuval Yarom. “”Make Sure DSA Signing Exponentiations Really are Constant-Time”.” In: *ACM Conference on Computer and Communications Security*. ACM, 2016, pp. 1639–1650.

[PDG14] Thomas Pöppelmann, Léo Ducas, and Tim Güneysu. “Enhanced Lattice-Based Signatures on Reconfigurable Hardware.” In: *CHES*. Vol. 8731. LNCS. Springer, 2014, pp. 353–370.

[Per05] Colin Percival. *Cache Missing for Fun and Profit*. `http://daemonology.net/hyperthreading-considered-harmful/`. 2005.

[Pie12] Krzysztof Pietrzak. “Cryptography from Learning Parity with Noise.” In: *SOFSEM*. Vol. 7147. LNCS. Springer, 2012, pp. 99–114.

[Pod+17] Damian Poddebniak, Juraj Somorovsky, Sebastian Schinzel, Manfred Lochter, and Paul Rösler. *Attacking Deterministic Signature Schemes using Fault Attacks*. Cryptology ePrint Archive, Report 2017/1014. 2017.

[POG15] Thomas Pöppelmann, Tobias Oder, and Tim Güneysu. “High-Performance Ideal Lattice-Based Cryptography on 8-Bit ATxmega Microcontrollers.” In: *LATINCRYPT*. Vol. 9230. LNCS. Springer, 2015, pp. 346–365.

[Por13] T. Pornin. *Deterministic Usage of the Digital Signature Algorithm (DSA) and Elliptic Curve Digital Signature Algorithm (ECDSA)*. RFC 6979. `https://tools.ietf.org/html/rfc6979`. 2013.

[PQ03] Gilles Piret and Jean-Jacques Quisquater. “A Differential Fault Attack Technique against SPN Structures, with Application to the AES and KHAZAD.” In: *CHES*. Vol. 2779. LNCS. Springer, 2003, pp. 77–88.

[Pra62] E. Prange. “The use of information sets in decoding cyclic codes.” In: *Information Theory, IRE Transactions on* 8.5 (1962), pp. 5–9.

[Pro] GNU Project. *GLPK (GNU Linear Programming Kit)*. `https://www.gnu.org/software/glpk/`.

[PSY15] Joop van de Pol, Nigel P. Smart, and Yuval Yarom. “Just a Little Bit More.” In: *CT-RSA*. Vol. 9048. LNCS. Springer, 2015, pp. 3–21.

[QS01] Jean-Jacques Quisquater and David Samyde. “ElectroMagnetic Analysis (EMA): Measures and Counter-Measures for Smart Cards.” In: *E-smart*. Vol. 2140. LNCS. Springer, 2001, pp. 200–210.

[Reg05] Oded Regev. “On lattices, learning with errors, random linear codes, and cryptography.” In: *STOC*. ACM, 2005, pp. 84–93.

[Rep+15] Oscar Reparaz, Sujoy Sinha Roy, Frederik Vercauteren, and Ingrid Verbauwhede. “A Masked Ring-LWE Implementation.” In: *CHES*. Vol. 9293. LNCS. Springer, 2015, pp. 683–702.

[Ris+09] Thomas Ristenpart, Eran Tromer, Hovav Shacham, and Stefan Savage. “Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds.” In: *ACM Conference on Computer and Communications Security*. ACM, 2009, pp. 199–212.

[Roy+14a] Sujoy Sinha Roy, Oscar Reparaz, Frederik Vercauteren, and Ingrid Verbauwhede. *Compact and Side Channel Secure Discrete Gaussian Sampling*. Cryptology ePrint Archive, Report 2014/591. 2014.

[Roy+14b] Sujoy Sinha Roy, Frederik Vercauteren, Nele Mentens, Donald Donglong Chen, and Ingrid Verbauwhede. “Compact Ring-LWE Cryptoprocessor.” In: *CHES*. Vol. 8731. LNCS. Springer, 2014, pp. 371–391.

[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.” In: *J. Cryptographic Engineering* 8.1 (2018), pp. 71–84.

[Sam+18] Niels Samwel, Lejla Batina, Guido Bertoni, Joan Daemen, and Ruggero Susella. “Breaking Ed25519 in WolfSSL.” In: *CT-RSA*. Vol. 10808. LNCS. Springer, 2018, pp. 1–20.

[SB18] Niels Samwel and Lejla Batina. “Practical Fault Injection on Deterministic Signatures: The Case of EdDSA.” In: *AFRICACRYPT*. Vol. 10831. LNCS. Springer, 2018, pp. 306–321.

[Sch15] Bruce Schneier. *NSA Plans for a Post-Quantum World*. `https://www.schneier.com/blog/archives/2015/08/nsa_plans_for_a.html`. Aug. 2015.

[SD15] Mark Seaborn and Thomas Dullien. *Test DRAM for bit flips caused by the rowhammer problem.* https://github.com/google/rowhammer-test. Retrieved on July 27, 2015. 2015.

[Sea15a] Mark Seaborn. *Exploiting the DRAM rowhammer bug to gain kernel privileges.* http://googleprojectzero.blogspot.com/2015/03/exploiting-dram-rowhammer-bug-to-gain.html. Retrieved on June 26, 2015. Mar. 2015.

[Sea15b] Mark Seaborn. *How physical addresses map to rows and banks in DRAM.* http://lackingrhoticity.blogspot.com/2015/05/how-physical-addresses-map-to-rows-and-banks.html. Retrieved on July 20, 2015. May 2015.

[Sha16] Adi Shamir. *Financial Cryptography: Past, Present, and Future.* Invited Talk at Financial Cryptography 2016. Summary available at https://www.lightbluetouchpaper.org/2016/02/22/financial-cryptography-2016. Feb. 2016.

[Sho] Victor Shoup. *NTL: A Library for doing Number Theory.* http://www.shoup.net/ntl/.

[Sho99] Peter W. Shor. “Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer.” In: *SIAM Review* 41.2 (1999), pp. 303–332.

[SHS16] Hermann Seuschek, Johann Heyszl, and Fabrizio De Santis. “A Cautionary Note: Side-Channel Leakage Implications of Deterministic Signature Schemes.” In: *CS2@HiPEAC*. ACM, 2016, pp. 7–12.

[Ste88] Jacques Stern. “A method for finding codewords of small weight.” In: *Coding Theory and Applications.* Vol. 388. LNCS. Springer, 1988, pp. 106–113.

[str] strongSwan. *Bimodal Lattice Signature Scheme (BLISS) - strongSwan.* https://wiki.strongswan.org/projects/strongswan/wiki/BLISS.

[SXZ13] Brendan Saltaformaggio, Dongyan Xu, and Xiangyu Zhang. “BusMonitor: A Hypervisor-Based Solution for Memory Bus Covert Channels.” In: *EuroSec*. 2013.

[SZM17] Johanna Sepúlveda, Andreas Zankl, and Oliver Mischke. “Cache attacks and countermeasures for NTRUEncrypt on MPSoCs: Post-quantum resistance for the IoT.” In: *SoCC*. IEEE, 2017, pp. 120–125.

[tea16] The FPLLL development team. “fplll, a lattice reduction library.” https://github.com/fplll/fplll. 2016.

[Tex] Texas Instruments. *MSP432P401R LaunchPad.* http://www.ti.com/tool/msp-exp432p401r.

[Tsu+03] Yukiyasu Tsunoo, Teruo Saito, Tomoyasu Suzaki, Maki Shigeri, and Hiroshi Miyauchi. "Cryptanalysis of DES Implemented on Computers with Cache." In: *CHES*. Vol. 2779. LNCS. Springer, 2003, pp. 62–76.

[Val00] A. Valembois. "Fast soft-decision decoding of linear codes, stochastic resonance in algorithms." In: *Information Theory, 2000. Proceedings. IEEE International Symposium on.* 2000, pp. 91–.

[WXW15] Zhenyu Wu, Zhang Xu, and Haining Wang. "Whispers in the Hyper-Space: High-Bandwidth and Reliable Covert Channel Attacks Inside the Cloud." In: *IEEE/ACM Trans. Netw.* 23.2 (2015), pp. 603–615.

[Xia+13] Jidong Xiao, Zhang Xu, Hai Huang, and Haining Wang. "Security implications of memory deduplication in a virtualized environment." In: *DSN*. IEEE Computer Society, 2013, pp. 1–12.

[Xia+16] Yuan Xiao, Xiaokuan Zhang, Yinqian Zhang, and Radu Teodorescu. "One Bit Flips, One Cloud Flops: Cross-VM Row Hammer Attacks and Privilege Escalation." In: *USENIX Security Symposium*. USENIX Association, 2016, pp. 19–35.

[Yar+15] Yuval Yarom, Qian Ge, Fangfei Liu, Ruby B. Lee, and Gernot Heiser. *Mapping the Intel Last-Level Cache*. Cryptology ePrint Archive, Report 2015/905. 2015.

[Yar16] Yuval Yarom. *Mastik: A Micro-Architectural Side-Channel Toolkit*. `http://cs.adelaide.edu.au/~yval/Mastik/Mastik.pdf`. Sept. 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.

[YF14] Yuval Yarom and Katrina Falkner. "FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack." In: *USENIX Security Symposium*. USENIX Association, 2014, pp. 719–732.

[Zha+12] Yinqian Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. "Cross-VM side channels and their use to extract private keys." In: *ACM Conference on Computer and Communications Security*. ACM, 2012, pp. 305–316.

[Zha+14] Yinqian Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. "Cross-Tenant Side-Channel Attacks in PaaS Clouds." In: *ACM Conference on Computer and Communications Security*. ACM, 2014, pp. 990–1003.

About the Author

Author information as of October 2018.

Personal Information

Name:	Peter Peßl
Date of birth:	September 23th, 1987
Place of birth:	Graz, Austria

Education

- **06/2014 – present:** Doctoral studies, Graz University of Technology, Austria.
- **03/2011 – 4/2014:** Master studies in Information and Computer Engineering (Telematik), Graz University of Technology, Austria.
- **10/2007 – 02/2011:** Bachelor studies in Information and Computer Engineering (Telematik), Graz University of Technology, Austria.

Professional and Academic Experience

- **06/2014 – present:** Research assistant, Institute for Applied Information Processing and Communications (IAIK), Graz University of Technology, Austria.

Other Activities

- **Teaching:** Lectures *Introduction to Information Security* and *Embedded Security*, supervised multiple Master's and Bachelor's theses
- **Reviews:** Member of the CHES 2019 Program Committee and Subreviewer for multiple conferences (CHES, ASIACRYPT, CT-RSA, CARDIS,...)

Author's Publications

Author's publications as of October 2018. For publications, an internationalized spelling of the author's name is used (Pessl). All publications were accepted on first submission.

Publications Used in this Thesis

Peter Pessl and Stefan Mangard. "Enhancing Side-Channel Analysis of Binary-Field Multiplication with Bit Reliability." In: *CT-RSA*. vol. 9610. LNCS. Springer, 2016, pp. 255–270.

Peter Pessl, Daniel Gruss, Clémentine Maurice, Michael Schwarz, and Stefan Mangard. "DRAMA: Exploiting DRAM Addressing for Cross-CPU Attacks." In: *USENIX Security Symposium*. USENIX Association, 2016, pp. 565–581.

Peter Pessl. "Analyzing the Shuffling Side-Channel Countermeasure for Lattice-Based Signatures." In: *INDOCRYPT*. vol. 10095. LNCS. 2016, pp. 153–170.

Peter Pessl, Leon Groot Bruinderink, and Yuval Yarom. "To BLISS-B or not to be: Attacking strongSwan's Implementation of Post-Quantum Signatures." In: *CCS*. ACM, 2017, pp. 1843–1855.

Leon Groot Bruinderink and Peter Pessl. "Differential Fault Attacks on Deterministic Lattice Signatures." In: *TCHES* 2018.3 (2018), pp. 21–43.

Further Contributions

Peter Pessl and Michael Hutter. "Pushing the Limits of SHA-3 Hardware Implementations to Fit on RFID." in: *CHES*. vol. 8086. LNCS. Springer, 2013, pp. 126–141.
Contribution: I am the first author and provided all technical contributions.

Peter Pessl and Michael Hutter. "Curved Tags - A Low-Resource ECDSA Implementation Tailored for RFID." in: *RFIDSec*. Vol. 8651. LNCS. Springer, 2014, pp. 156–172.
Contribution: I am the first author and provided all technical contributions.

Robert Primas, Peter Pessl, and Stefan Mangard. "Single-Trace Side-Channel Attacks on Masked Lattice-Based Encryption." In: *CHES*. vol. 10529. LNCS. Springer, 2017, pp. 513–533.
Contribution: This publication is the result of Robert Primas' Master's Thesis, which I supervised. Apart from that, I contributed the initial idea, some of the technical contributions, and wrote large pieces of the paper.