Countermeasures Against First Order Fault Attacks Using the Example of ring-TESLA

Technische Universität Darmstadt

Department of Computer Science

Cryptography and Computer Algebra

Bachelor-Thesis by Johannes Schreiber

October 2017

Affidavit

I herewith formally declare that I have written the submitted thesis independently. I did not use any outside support except for the quoted literature and other sources mentioned in the paper. I clearly marked and separately listed all of the literature and all of the other sources which I employed when producing this academic work, either literally or in content. This thesis has not been handed in or published before in the same or similar form. In the submitted thesis the written copies and the electronic version are identical in content.

Darmstadt, 10.10.2017

_____

(Johannes Schreiber)

Abstract

In recent years, many different cryptographic schemes have been invented which are thought to be difficult to attack, even by quantum computers. In the case of lattice cryptography, they are based on lattice problems that have been conjectured to be hard to solve. However, using brute force, mathematical cryptanalysis, or finding a back door are not the only ways to attack cryptographic primitives. In many cases it is possible to attack the hardware on which the cryptographic code runs by introducing faults into the calculations and gather secret information by analysing the output.

In this work we investigate fault attacks against the ring-TESLA signature scheme. While possible fault attacks have already been identified, we provide countermeasures and measure the performance drawback of this added security against hardware attacks. We consider first order fault attacks, and the countermeasures are implemented in software.

Contents

1 Introduction

Today, public key cryptography is at the heart of secure information exchange between computer systems. However, the invention of Shor's Algorithm 1994 puts all currently used schemes, such as RSA[1] , DSA, Elliptic Curve Cryptography, or the Diffie–Hellman key exchange in danger.

The rising threat that powerful quantum computers could in a few years be a reality calls for new cryptographic primitives. In recent years, many different schemes have been proposed, which can be classified as code-based, hash-based, lattice-based, isogeny-based, or multi variate. Lattice-based cryptographic primitives seem to be very promising in the sense that they have performance comparable to currently used schemes, and it looks like they are a candidate for fully homomorphic encryption. A few examples of lattice based signature schemes invented in recent years are GLP [13], BLISS [9], ring-TESLA [1], and NTRU-Sign [15].

We investigate ring-TESLA, because it has the fewest vulnerabilities against fault attacks. These vulnerabilities have all been identified in [6], with the exception of zeroing the secret polynomial s, which was discovered by the authors of this thesis, and the loop abort attack discovered by [11]. We differentiate between two major classes, skipping faults and zeroing faults. Our goal was to implement countermeasures against these attacks and measure how efficient the countermeasures are, both in terms of execution time and code size. To that end, we edited the original C implementation of ring-TESLA and used the GNU Compiler Collection (GCC) to generate the corresponding assembly code. We then analyzed the generated code by hand and reasoned why (or why not) it is secure against an attack. Some of these countermeasures, ideas, and performance measurements have been included in a conference paper [7].

While we are the first to implement countermeasures in ring-TESLA, there has already been a lot of work done for more widely used schemes such as RSA or AES. Some authors have also proposed generic countermeasures, which can be applied to any cryptographic scheme.

[1] Recently Bernstein et al. published an instantiation of RSA which can resist an attack by a quantum computer. This is achieved by choosing a key size of one terabyte, leading to encryption times on the order of 100 hours on a high performance machine, which is obviously not practical for day to day usage. [5]

In this work, we will first introduce the basics to discuss this topic: Chapter 2 is concerned with signature schemes, the Ring Learning with Error Problem, fault attacks, and the ring-TESLA scheme. In Chapter 3 we look at the fault attacks against ring-TESLA that have been found until now and provide countermeasures. In Chapter 4 we show the performance impact of implementing these countermeasures.

2 Preliminaries

2.1 Notation and Definitions

We denote the set of positive natural numbers by $\mathbb{N}_{>0}$ and define $\mathbb{B} = \{0, 1\}$. Let $n = 2^k$ for some $k \in \mathbb{N}$ and q be a prime with $q = 1 \pmod{2n}$. We denote the field $\mathbb{Z}/q\mathbb{Z}$ by $\mathbb{Z}_q$. We define $\mathcal{R} = \mathbb{Z}[x]/(x^n + 1)$, as well as $\mathcal{R}_q = \mathbb{Z}_q[x]/(x^n + 1)$ and $\mathcal{R}_{q,[B]} = \{\sum_{i=0}^{n-1} \alpha_i x^i \in \mathcal{R}_q \mid \alpha_i \in [-B, B]\}$ with $B \in \mathbb{N}$. The set of units of $\mathcal{R}_q$ is written as $\mathcal{R}_q^\times$. The euclidean norm of a vector $v \in \mathbb{R}^n$ is indicated by $\|v\|$. We denote the set of all vectors of length n with exactly ω ones and zeroes for all other components by $\mathbb{B}_{n,\omega} = \{v \in \mathbb{B}^n \mid \|v\|^2 = \omega\}$. We do not explicitly differentiate between a polynomial in $\mathcal{R}_q$ and its coefficient vector in $\mathbb{Z}_q^n$, because $\mathcal{R}_q$ is isomorphic to $\mathbb{Z}_q^n$.

We write $[c]_{2^d}$ with $d \in \mathbb{N}$ to denote the unique representative of c modulo 2^d in the set $(-2^{d-1}, 2^{d-1}]$. Intuitively, we can think of $[c]_{2^d}$ as the d least significant bits of c. Similarly we define $\lfloor \cdot \rceil_d : \mathbb{Z} \to \mathbb{Z}$, $c \mapsto (c - [c]_{2^d})/2^d$ and we think of $\lfloor c \rceil_d$ as the most significant bits of c (what remains after taking away the d least significant bits). We use $\lfloor v \rceil_{d,q}$ as an abbreviation for $\lfloor v \pmod{q} \rceil_d$.

The centered gaussian distribution $\mathcal{D}_\sigma$ over $\mathbb{Z}$ with standard deviation σ is defined as follows: For $z \in \mathbb{Z}$ the probability of z is given by $\rho_\sigma(z)/\rho(\mathbb{Z})$ with $\rho_\sigma(z) = \exp(-z^2/2\sigma^2)$ and $\rho_\sigma(\mathbb{Z}) = 1 + 2\sum_{z=1}^{\infty} \rho_\sigma(z)$. For a distribution $\mathcal{D}$ we write $d \leftarrow \mathcal{D}$ to denote sampling an element d from $\mathcal{D}$. Similarly, we write $v \leftarrow \mathcal{D}^n$ to denote sampling every component of the n-dimensional vector v from the probability distribution $\mathcal{D}$. The same notation will also be used to denote sampling each coefficient of a polynomial in $\mathcal{R}$. For a set S we write $s \leftarrow_\$ S$ to indicate we are sampling an element s uniformly random from S.

All these definitions are the same as in the original ring-TESLA paper [1].

Signature Schemes We only provide an informal definition. A signature scheme consists of three (probabilistic) algorithms: `KeyGen`, `Sign`, and `Verify`.

`KeyGen` takes the unary vector 1^λ with security parameter λ as input and

returns a signing key sk and a verifying key vk.

`Sign` takes a message μ to be signed and the signing key vk. It returns a signature σ for μ.

`Verify` takes a message μ, a possible signature σ, and the verifying key vk. The algorithm either accepts or rejects σ as a signature of μ.

The scheme needs to fulfill two properties: A valid signature is always accepted (correctness) and in polynomial time nobody can create a signature from a private key vk without knowing vk (security).

The Ring Learning with Errors Problem. We take the following definitions from [6].

Definition 1 (Ring Learning with Errors Distribution)**.** *Let $n, q \in \mathbb{N}_{>0}$, $s \in \mathcal{R}_q$, and χ be a distribution over $\mathcal{R}$. We define by $\mathcal{D}_{s,\chi}$ the R-LWE distribution which outputs $(a, as+e) \in \mathcal{R}_q \times \mathcal{R}_q$, where $a \leftarrow_{\$} \mathcal{R}_q$ and $e \leftarrow \chi$.*

Definition 2 (Decisional Ring Learning with Errors Problem)**.** *Let $n, q \in \mathbb{N}_{>0}$ and $n = 2^k$ for some $k \in \mathbb{N}_{>0}$ and χ be a distribution over $\mathcal{R}$. Given n, q, and samples $(a_1, b_1), \ldots, (a_m, b_m) \in \mathcal{R}_q \times \mathcal{R}_q$, the decisional ring learning with errors problem is to decide, whether $b_1, \ldots, b_m$ are chosen with the R-LWE distribution or whether they are chosen uniformly random over $\mathcal{R}_q$.*

There is also a search variant of R-LWE, where the problem is to find the polynomial s, however, the search variant can be reduced to the decision variant [16]. The R-LWE problem has been conjectured to be hard to solve, even by quantum computers.

2.2 Description of ring-TESLA

The ring-TESLA scheme was presented by Akleylek et al. [1]. It is a ring variant of the signature scheme TESLA by Alkim et al. [2], which in turn is based on a scheme first developed by Bai and Galbraith [3] and improved by Dagdelen et al. [8].

It is parametrized by the lattice dimension n, security parameter λ, discrete gaussian distribution $\mathcal{D}_\sigma$ with standard deviation σ and mean 0, modulus q and the integers $\kappa, \omega, d, B, L, U$. The condition $n > \kappa > \lambda$ has to be fulfilled. Two polynomials $a_1, a_2 \in \mathcal{R}_q^\times$ are needed as global constants and can be shared among all users. Furthermore we require a hash function $H : \mathbb{B}^* \to \mathbb{B}^\kappa$ and an encoding function $F : \mathbb{B}^\kappa \to \mathbb{B}_{n,\omega}$. More information about the encoding function can be found in [14].

Figure 2.1: Pseudocode of the algorithms for ring-TESLA

Algorithm 2.1: Sign

1 ***input***: $\mu; a_1, a_2, s, e_1, e_2$
2 ***output***: (z, c')
3
4 $y \leftarrow \mathcal{R}_{q,[B]}$
5 $v_1 \leftarrow a_1 y \pmod q$
6 $v_2 \leftarrow a_2 y \pmod q$
7 $c' \leftarrow H(\lfloor v_1 \rceil_{d,q}, \lfloor v_2 \rceil_{d,q}, \mu)$
8 $c \leftarrow F(c')$
9 $z \leftarrow y + sc$
10 $w_1 \leftarrow v_1 - e_1 c \pmod q$
11 $w_2 \leftarrow v_2 - e_2 c \pmod q$
12 ***If*** $[w_1]_{2^d}, [w_2]_{2^d} \notin \mathcal{R}_{2^d - L}$
13 $\vee z \notin \mathcal{R}_{B-U}$
14 $\vee \|w\|_\infty > \lfloor q/2 \rfloor - L$ $^{(*)}$
15 ***Restart*** at line 4
16 ***Return*** (z, c')

(∗) This check is not present in the original ring-TESLA paper. It was adapted from TESLA to eliminate a few cases where valid signatures would not verify.

Algorithm 2.2: KeyGen

1 ***input***: $1^\lambda, a_1, a_2$
2 ***output***: (sk, pk)
3
4 $s, e_1, e_2 \leftarrow D_\sigma^n$
5 ***If*** checkE$(e_1) = 0 \vee$ checkE$(e_2) = 0$
6 ***Restart*** at line 4
7 $t_1 \leftarrow a_1 s + e_1 \pmod q$
8 $t_2 \leftarrow a_2 s + e_2 \pmod q$
9 sk $\leftarrow (s, e_1, e_2)$
10 pk $\leftarrow (t_1, t_2)$
11 ***Return*** (sk, pk)

Algorithm 2.3: Verify

1 ***input***: $\mu; z, c'; a_1, a_2, t_1, t_2$
2
3 $c \leftarrow F(c')$
4 $w_1' \leftarrow a_1 z - t_1 c \pmod q$
5 $w_2' \leftarrow a_2 z - t_2 c \pmod q$
6 $c'' \leftarrow H(\lfloor w_1' \rceil_{d,q}, \lfloor w_2' \rceil_{d,q}, \mu)$
7 ***If*** $c' = c'' \wedge z \in \mathcal{R}_{B-U}$
8 ***Return*** 1
9 Else
10 ***Return*** 0

We now provide descriptions of the algorithms in Figure 2.1.

Key Generation. During key generation the secret polynomials $s, e_1, e_2 \in \mathcal{R}$ are sampled from the Gaussian distribution $\mathcal{D}_\sigma^n$. The polynomials e_1 and e_2 have to fulfil the property that the sum of their ω largest coefficients are less or equal than the rejection parameter L. This is checked in `checkE()` and the key generation is restarted if either polynomial does not satisfy this property. Then the signing key (s, e_1, e_2) and the verification key $(t_1, t_2) = (a_1 s + e_1, a_2 s + e_2)$ are returned. Calculating the signing key from the verification key directly would require solving the R-LWE problem with the two given samples (a_1, t_1) and (a_2, t_2).

Signing. The signing algorithm takes the message $\mu \in \mathbb{B}^*$ to be signed, the publicly known constants a_1, a_2, and the secret key (s, e_1, e_2). First, y is sampled uniformly random from $\mathcal{R}_{q,[B]}$. Then the polynomials v_1, v_2 are calculated as the products of the random polynomial y and the public constants a_1 and a_2 respectively, and hashed together with the message μ to produce the challenge c'. The response z is calculated as the sum of the random polynomial y and the product sc of the secret polynomial and the encoded challenge. This is characteristic for a Fiat-Shamir construction [12]. The algorithm returns a signature (z, c') of μ with $z \in \mathcal{R}$ and $c' \in \mathbb{B}^\kappa$.

Verification. The algorithm receives the message μ, a possible signature (z, c') of μ, the constants a_1, a_2 and the public key (t_1, t_2). First, it encodes the hash c' to a polynomial c. Then the values w_1', w_2' are calculated. If the signature is authentic, w_1', w_2' will have the same most significant bits as v_1, v_2 during signing. So if $c' = c'' \leftarrow H(\lfloor w_1' \rceil_{d,q}, \lfloor w_2' \rceil_{d,q}, \mu)$ and $z \in \mathcal{R}_{B-U}$ the algorithm accepts (z, c') as a signature of the message μ.

2.3 Fault Attacks and our Attacker Model

A (hardware) fault attack is an attack targeting a processor, which is executing cryptographic code. By introducing faults in the calculations, the device may output erroneous data, which can be analyzed in order to gather secret information. In the case of signature schemes, common goals for an attacker include – but are not limited to – getting access to the signing key, forging a signature, or convincing a victim that an invalid signature is valid. Common faults that have been shown to be possible are skipping faults and randomization faults [4]. During a skipping fault, a single machine code instruction is skipped. This can be achieved for example by "clock glitching", where the processor gets an irregular clock signal. Randomizing faults set variables or parts of the memory to random values. They can for example be introduced by heating the processor outside of its normal operating temperature range, or shining strong electromagnetic or ionizing radiation at it. Zeroing faults, where a variable is set to zero, can sometimes be achieved via skipping faults. Of course, physical access to the device is needed in all cases. In light of these and more possible attacks, key generation should happen on a secure machine, so this attack vector is somewhat unlikely. Because of this, it may be easier to extract the secret key from a smart card, because an attacker can have physical access to the device. Key verification is also of interest, because the smart card may have to verify the identity of whatever device it is communicating with.

We consider the following threat model: An attacker can introduce a single skipping fault with perfect precision at any point during code execution. In some cases we also consider the ability to induce two skipping

faults. The attacker can randomize any variables at any time, but they cannot set a variable to any predefined value or restrict its value to a predefined small range. They can however (by means of a skipping fault) set a variable to a value, which depends on the content of the memory, which will either contain values from previous computations, different processes, or may even be zero. Because of this, we also consider the possibility of zeroing faults, although our tests suggest this may only be possible on some architectures. Countermeasures can be applied to the hardware as well as on the software side, but in this work we only focus on the software. Here are some examples of possible software countermeasure:

- Redundant code execution: Perform the same computation multiple times, then make sure all results are equal. Of course, if you do the same work twice it is going to take twice as long, which is not very efficient.
- Additional correctness checks: Introduce additional checks to make sure, that all invariants of the code are true. For example, if during normal code execution a variable will never be zero, you can add an if-statement which makes sure the variable is nonzero even after a fault attack.
- Infectious computations: Introducing a fault progagates through the rest of the computation, rendering affected values unusuable.
- Branchless implementations: A branch (for example an if-statement) always depends on comparing two values and then jumping to a different part of the code based on the value of the zero flag or carry flag. When skipping the comparing instruction, there is a good chance that the body of the if-statement is skipped. In most cases, it is even possible to skip one of the jump instructions so that the desired branch is executed every time.

Out of these four, infectious computations and a branchless implementation seem to be the most promising options for ring-TESLA, because they are inexpensive and more resistant to higher order fault attacks. Hence, we will use these approaches to secure ring-TESLA, as described in the following chapter.

3 Fault Attacks against ring-TESLA and Countermeasures

The following vulnerabilities of ring-TESLA were identified in [6]. We name them here first, the exact explanations follow in the corresponding sections. To get a sense of the relevance of each variable, or at what point in the scheme these attacks can be mounted, we direct your attention to Figure 2.1.

- Zeroing the error polynomial e_1 or e_2 during key generation
- Skipping the addition of e_1 or e_2 during key generation
- Zeroing the random polynomial y during signing
- Skipping the rejection sampling during signing
- Zeroing the hash c during verification
- Skipping the hash comparison $c' = c''$ during verification
- There are no known vulnerabilities to randomization faults

Although not mentioned in [6], we discovered that ring-TESLA is also vulnerable to zeroing the secret polynomial s during key generation. Furthermore, the possibility to zero a polynomial with a loop abort fault has been described by Espitau et al. [11].

We received the C implementation from Sedat Akleylek and our modified version is available on `https://www.cdc.informatik.tu-darmstadt.de/en/cdc/staff/nina-bindel/`.

3.1 Skipping Faults

In some cases skipping faults lead to uninitialized values. The prominent example of this in ring-TESLA is skipping a function call which initializes a variable, for example when sampling a polynomial. Another possibility, which is more readily visible in assembly code, is the skipping of some instruction, which initializes a variable, such as a register. As an example

let us have a look at the instruction `movl $-1, %eax`: If it is skipped, the register `%eax` will not contain -1, but rather whatever value it held before the assignment.

We make the following assumptions about the distribution of these uninitialized values: In the case of registers, the value they take on depends on the previously executed code. In the case of more complex variables, such as polynomials, it depends on whatever was stored previously in that memory location on the heap. We also consider the possibility that the resulting variable is zero, as may be the case on a system where the entire memory is initialized to zero. However our experiments suggest that the probability of setting an entire polynomial to zero on a personal computer is low, although we have seen cases, where all but three coefficients were zero.

A generic countermeasure to prevent the skipping of a function call is to inline it by prefixing the definition with the `__attribute__((always_inline))` directive of GCC. That way the compiler will not generate a `call` instruction which can be skipped. This comes at no additional computational cost, but it will increase the size of the compiled binary, which is often a drawback when working with embedded systems with very small memory. Also it does not prevent fault attacks that can occur during the execution of the function or attacks that target branches which depend on the result of the function. For example when inlining a function `fun()`, the following code `if(fun() == 2)` can still be attacked by a skipping fault. We are not providing a more in depth analysis of this countermeasure, because in ring-TESLA GCC could not inline the critical functions.

3.1.1 Skipping the addition of the error polynomials during key generation

In ring-TESLA during key generation the public key is calculated to be $t_i = a_i s + e_i \pmod q$ with $i \in \{1, 2\}$. If e_i is not added, the public key will contain $t_i = a_i s$. Because a_i is public and a unit in $\mathcal{R}_q$, anyone can compute $s = a_i^{-1} t_i$ and thus obtain the secret polynomial s. Then the error polynomials $e_i = t_i - a_i s$ can be calculated and we have retrieved the signing key (s, e_1, e_2). Listing 3.1 shows the implementation for the calculation $a_i s + e_i \pmod q$ during key generation.

Listing 3.1: KeyGen: Polynomial multiplication and addition to compute the public key

```
poly_mul_fixed(poly_T1, poly_S, poly_a1);
poly_add(poly_T1, poly_T1, poly_E1);
poly_mul_fixed(poly_T2, poly_S, poly_a2);
poly_add(poly_T2, poly_T2, poly_E2);
. . .
compress_pk(pk, poly_T1, poly_T2);
```

If an attacker manages to skip the function call `poly_add(poly_T1, poly_T1, poly_E1)`, one half of the public key will be $t_1 = a_1 s$. A simple countermeasure against this is to introduce two new variables `poly_A1S, poly_A2S` to hold the results of the multiplication, as suggested by [6] and shown in Listing 3.2.

Listing 3.2: KeyGen: Countermeasure against skipping the addition, using new variables

```
poly poly_A1S, poly_A2S;

poly_mul_fixed(poly_A1S, poly_S, poly_a1);
poly_add(poly_T1, poly_A1S, poly_E1);
poly_mul_fixed(poly_A2S, poly_S, poly_a2);
poly_add(poly_T2, poly_A2S, poly_E2);
[...]
compress_pk(pk, poly_T1, poly_T2);
```

Now, if either addition is skipped, the value of t_1 or t_2 is uninitialized and the private and public key do not correspond to each other anymore. Signatures will most probably fail to verify, because if $t_1 = 0$ then $w_1' = a_1 z$ during verification, which is very unlikely to have the same most significant bits as $a_1 y$ during signing. If t_1 has random coefficients, it is still very unlikely and so the hash values c' and c'' will be different with very high probability. This leads to a rejection by the verifier.

We call this countermeasure *new_variable*.

3.1.2 Skipping the hash comparison during verification

The verification of ring-TESLA in part depends on the code in Listing 3.3, where `c` and `c_sig` represent the hash values c' and c''. Listing 3.4 shows the generated assembly of the same code (without compiler optimizations).

Listing 3.3: The last few lines of Verify: comparing the hashes c' and c''

```
1  if(memcmp(c, c_sig, 32))
2      return -1;
3
4  [...]
5
6  return 0;
```

Listing 3.4: Assembly code of the hash verification in Listing 3.3

```
1  leaq -112(%rbp), %rcx
2  leaq -48(%rbp), %rax
3  movl $32, %edx
4  movq %rcx, %rsi
5  movq %rax, %rdi
6  call _memcmp
7  testl %eax, %eax
8  je L139
9  movl $-1, %eax
10 jmp L140
11 L139:
12 [...]
13 movl $0, %eax
14 L140:
15 [...]
16 ret
```

An attacker who wants its victim to accept an invalid signature can attempt to force the verify function to return zero, which signifies a valid signature. This can be achieved by skipping the jump in line 10, so that `movl $0, %eax` in line 13 is executed independently of the result of `memcmp()`.

But actually, Listing 3.3 looks like this:

Listing 3.5: Verify: comparing the hashes (without omitted code)

```
1  if(memcmp(c, c_sig, 32))
2      return -1;
3
4  *mlen = smlen - CRYPTO_BYTES;
5  memcpy(m, sm, *mlen);
6
7  return 0;
```

Line 4 and 5 in Listing 3.5 are responsible for copying the signed message into a memory location, which is accessed from outside the function for further use. We can safely rewrite that as shown in Listing 3.6, completely dropping the if-statement and making this part of the code a branchless implementation. There is not any real harm in returning a message and forged signature, if the verifier cannot be tricked into believing it is genuine.

Listing 3.6: Rewritten code for Listing 3.3 which is secure against a skipping attack

```
1 *mlen = smlen-CRYPTO_BYTES;
2 memcpy(m, sm, *mlen);
3
4 return memcmp(c,c_sig,32);
```

Listing 3.7: Generated assembly of Listing 3.6 line 4

```
1 [...]
2 leaq -112(%rbp), %rcx
3 leaq -48(%rbp), %rax
4 movl $32, %edx
5 movq %rcx, %rsi
6 movq %rax, %rdi
7 call _memcmp
8 [...]
9 ret
```

The function now still returns zero if the signature is valid and a nonzero value otherwise, albeit not -1. This stems from the fact that the `memcmp()` function returns 0 if the memory locations are equal. If they are unequal, the index of the first byte where they differ is returned. The disassembled code in Listing 3.7 shows that failing to call `memcmp()` returns the content of `%eax`, which is modified in line 3 and is unlikely to be zero. Because zero is the value that is returned to indicate a valid signature, it is unlikely that the verifier can be convinced that an invalid signature is valid, even when employing a fault attack.

We call this countermeasure *rewrite_branchless*.

3.1.3 Skipping the rejection sampling during signing

In [6] an attack is described, where skipping the rejection sampling during signing leaks a small amount of information about the secret. The authors did not provide an estimate how many faults would have to be induced, but a similar attack on NTRU needs about 8000 signatures [10]. Listing 3.8 shows the part of the code, where the attack could take place.

Listing 3.8: Last lines of Sign: Rejection sampling and returning the signature

```
1  while(1) {
2      [...]
3
4      // Rejection sampling
5      if (test_rejection(vec_y) != 0) {
6          continue;
7      }
8
9      //Copy the message into the signature package
10     for(i = 0; i < mlen; i++) {
11         sm[i] = m[i];
12     }
13
14     // Length of the output
15     *smlen = CRYPTO_BYTES + mlen;
16
17     // Put the signature into the output package
18     compress_sig(sm+mlen, c, vec_y);
19     return 0;
20 }
```

If the rejection sampling in line 2 would be skipped, every signature could leak a small amount of information about the secret key. To prevent this, we could apply rejection sampling again before the signature is compressed into its output package, but that is a somewhat inelegant solution, especially because we can imagine that an attacker that is powerful enough to induce a precise skipping fault in the first line could do it again a few lines down and thwart the countermeasure.

In Listing 3.8 the function `compress_sig()` in line 15 copies the hash `c` and the polynomial `vec_y` coefficient by coefficient into an array of bytes, so there is a possibility of applying rejection sampling a second time, but in a way that cannot easily be subverted. Listing 3.9 shows the implementation of `compress_sig()`. The function `fmodb_u()` (called in line 15) was added to the original code. It is comparable to `test_rejection()` in Listing 3.8 line

5, but modifies only a single coefficient: A coefficient a of the polynomial is turned into $a \pmod{B-U}$. Listing 3.10 shows the implementation of this modulo function.

Listing 3.9: Additional rejection sampling with `fmodb_u()` when compressing the signature into its output package

```
1  static void compress_sig(unsigned char *sm,
2                           unsigned char *c,
3                           double vec_z[PARAM_N])
4  {
5      int i,k;
6      int ptr =0;
7      int32_t t=0;
8
9      //store the hash value
10     for (i=0; i<32; i++){
11         sm[ptr++] =c[i];
12     }
13     for(i=0; i < PARAM_N; i++)
14     {
15         t = (int32_t) fmodb_u(vec_z[i]);
16         for(k=0;k<4;k++)
17             sm[ptr++] = ((t>>(8*(k))) & 0xff);
18     }
19 }
```

Listing 3.10: The rejection sampling function `fmodb_u()`

```
1  static double fmodb_u(double x) {
2      int modulus = PARAM_B - PARAM_U;
3      if(x < -modulus) {
4          return x + modulus;
5      } else if (x > modulus) {
6          return x - modulus;
7      } else {
8          return x;
9      }
10 }
```

Under normal circumstances, `fmodb_u()` is not going to alter the coefficients, because such a polynomial would have been rejected by the call to `test_rejection()` in line 5 of Listing 3.8. Suppose this first rejection sampling had been skipped. Now those coefficients whose absolute values are

bigger than $B - U$ are modified by `fmodb_u()` and prevented from leaking secret information. This results in an invalid signature, which will most likely not be accepted.

In order to circumvent this countermeasure, the call to `fmodb_u()` would have to be skipped in every iteration of the loop in Listing 3.9 line 15, which is executed n times[1]. If the entire loop is skipped, the signature will be empty, because this loop is responsible for copying the signature into the output array. This does not help the attacker either. The only information they get from that is the fact that they can induce a skipping fault.

We call this countermeasure *additional_rejection*.

3.2 Zeroing Faults

First, let us define a function, which we are going to use to prevent zeroing attacks. Let $z \in \mathbb{N}$, $p = \sum_{i=0}^{n-1} \alpha_i x^i$ and H be the Heaviside function with $H(0) = 0$. We define $|\cdot|^0_{<z}$ as follows:

$$
\begin{aligned}
|\cdot|^0_{<z} : \mathcal{R}_q &\to \mathbb{B} \\
p &\mapsto \begin{cases} 1, & \text{if } \sum_{i=0}^{n-1} H(\alpha_i) < z \\ 0, & \text{otherwise} \end{cases}
\end{aligned}
$$

So for a polynomial p the function $|p|^0_{<z}$ is 1, if the number of zero coefficients is strictly less than z, otherwise 0.

3.2.1 Zeroing the random polynomial during signing

Preventing first order fault attacks

Since ring-TESLA relies on a Fiat-Shamir construction [12], a random value (the 'commitment') has to be sampled during signing. Listing 3.11 shows the code for this, where `poly` is an array of type `double`.

Listing 3.11: The first lines of Sign: Sampling the random polynomial y

```
1 poly vec_y;
2 sample_y(vec_y);
```

Firstly, sampling (which in assembly is just a single `call` instruction) could be skipped. Secondly, the polynomial or parts of it could otherwise be

[1] In this instantiation of ring-TESLA $n = 512$

zeroed, for example by a loop abortion fault inside `sample_y()`, as described by Espitau et al. [11]. If $y = 0$ the signature will instantly leak the secret key, because then $z = sc$, c is known (because it is part of the signature) and s can be calculated. But even if only some coefficients of y are set to zero, an adversary only needs a few[2] faulty signatures and can derive the key from them [6]. There is a way to prevent a first order fault attack on this code, simply by checking if too many coefficients of y are zero, as shown in Listing 3.12. We show the code, which is responsible for counting the number of zero coefficients in Listing 3.14.

Listing 3.12: Sign: Sampling with countermeasure 'count_zeroes'

```
1 poly vec_y;
2 sample_y(vec_y);
3 if (count_zeroes(vec_y) > 8) {
4     // restart key generation
5 }
```

Listing 3.13: Disassembly of line 3 in Listing 3.12

```
1 leaq -4176(%rbp), %rax
2 movq %rax, %rdi
3 call _count_zeroes
4 cmpl $8, %eax
5 jg L134
```

Listing 3.14: The function `count_zeroes()`: Counting how many coefficients of the polynomial are zero

```
1 int count_zeroes(poly p) {
2     int zeroes = 0;
3     for (int i = 0; i < PARAM_N; i++) {
4         if (p[i] == 0.0) {
5             zeroes++;
6         }
7     }
8     return zeroes;
9  }
```

To make the attack more difficult, one might be tempted to disallow any zero coefficients at all, because the chances of a coefficient being zero are only 1 in 4,194,303. But this would change the distribution too much and might invalidate the security proof. We chose to forbid any polynomials with 8 or more zero coefficients, because that rejects less than 1 in 2^{128} polynomials, meaning the change in the probability distribution is negligable, as shown in the following:

[2] If the attacker can set 12 bytes to zero during each attack, they need 384 signatures to recover the signing key, as shown in [6].

The polynomial y in ring-TESLA has 512 integer coefficients, which are sampled uniformly random from $[-2097151, 2097151]$. Let z be the number of zero coefficients in y. We denote the probability of a single coefficient being zero by $p = \frac{1}{2 \times 2097151 + 1}$. Then the probability $Pr(z > 8)$ of y having more than 8 zero coefficients is

$$1 - Pr(z \leq 8) = 1 - \sum_{i=0}^{8} \binom{512}{i} p^i (1-p)^{512-i} = 1.54 \times 10^{-41} < 2^{-128}.$$

However, this does not stop a more powerful attacker, which is able to induce two or more precise skipping faults, from recovering the secret key. Line 5 in Listing 3.13 shows the jump instruction, which restarts the signature generation. Simply skipping this instruction with a second fault would lead to a successful attack.

We call this countermeasure *count_zeroes*.

Preventing second order fault attacks

We use the previously defined function $|\cdot|^0_{<8}$, which returns 1, if there are strictly less than 8 coefficients which are zero, otherwise 0.

Algorithm 3.1: Original Sign pseudocode

1 $y \leftarrow \mathcal{R}_{q,[B]}$
2 [...]
3 $z \leftarrow y + sc$
4 [...]
5 ***Return*** (z, c')

Algorithm 3.2: Sign pseudocode, which is secured against first order attacks

1 $y \leftarrow \mathcal{R}_{q,[B]}$
2 [...]
3 $\lambda \leftarrow |y|^0_{<8}$
4 $z \leftarrow y + \lambda sc$
5 [...]
6 ***Return*** (z, c')

Listing 3.17: Sign: sampling y and computing $z = y + sc$

```
1 sample_y(vec_y);
2 [...]
3 poly_add(vec_y, vec_y, Sc);
4 [...]
5 return 0;
```

Listing 3.18: Sign: Secured computation of $z = y + sc$

```
1 sample_y(vec_y);
2 [...]
3 long lambda = check_zeroes(vec_y);
4 poly_mul_constant(lSc, Sc, lambda);
5 poly_add(vec_y, vec_y, lSc);
6 [...]
7 return 0;
```

To circumvent the countermeasure in Listing 3.18, the adversary could skip line 1 and somehow make `lambda` be 1 in order to return a faulty signature. Skipping line 3, 4 or 5 would result in returning an uninitialized or random value instead of a faulty signature, which does not reveal anything about the secret key. If the adversary only attacks line 1, the function will return $z = 0$ by design, which does not leak information about the secret s.

However, in order for that countermeasure to work, `check_zeroes()` must resist a single fault attack, which is not trivial to achieve. Let us first examine the insecure implementation in Listing 3.19.

Listing 3.19: Implementation of $|\cdot|^0_{<8}$ that is not secure against first order attacks

```
1  long check_zeroes(poly p) {
2      int zeroes = 0;
3      for (int i = 0; i < PARAM_N; i++) {
4          if (p[i] == 0.0) {
5              zeroes++;
6          }
7      }
8      if (zeroes > 8) {
9          return 0;
10     } else {
11         return 1;
12     }
13 }
```

There is an inherent problem with counting zeroes like that: No matter what follows the for-loop, if the loop is skipped, we think we have no zero coefficients. In that case `check_zeroes()` would return 1, even though y might actually have too many zeroes. Also, if-statements can be skipped, because they always compile to some variation of a compare and a jump instruction.

It is better to count the nonzero coefficients and do not use if-statements. Let us look at the code in Listing 3.20. We rewrote the critical if-statement in assembly, doubling each instruction so that skipping a single one has no effect on the computation. The compiler replaces `%0` with the input and `%1` with the output register. The keyword `volatile` makes sure that the rest of the assembly code is not optimized. This ensures the redundant statements and our protection against first order fault attacks are preserved and not removed by the compiler's optimization. The inline assembly code makes sure that 1 is returned if the number of nonzero coefficients is between 505 and 512. These values arise, because the polynomials have 512 coefficients and we allowed strictly less than 8 of those to be zero.

Listing 3.20: A way of checking the coefficients of y that is more robust against fault attacks

```
1  int check_zeroes(poly p) {
2      int nonzeroes = 0;
3      for (int i = 0; i < PARAM_N; i++) {
4          if (p[i] != 0.0) {
5              nonzeroes++;
6          }
7      }
8
9      asm volatile (
10         "cmpl $504, %0;"
11         "cmpl $504, %0;"
12         "setg %%bl;"
13         "setg %%bl;"
14         "cmpl $513, %0;"
15         "cmpl $513, %0;"
16         "setl %%bh;"
17         "setl %%bh;"
18         "andb %%bh, %%bl;"
19         "andb %%bh, %%bl;"
20         "movzbl %%bl, %1;"
21         "movzbl %%bl, %1;"
22     :"=r"(nonzeroes)
23     :"r"(nonzeroes)
24     :"%ebx"
25     );
26
27     return nonzeroes;
28 }
```

If an attacker can only induce one fault, they could skip the initialization in line 2, or skip some instruction inside the loop, which would essentially be a randomizing fault. Because the probability is very low that the resulting value of `nonzeroes` will be in the range $[505, 512]$, the attack has a low probability of success.

We call this countermeasure *zero_signature*.

3.2.2 Zeroing the secret or error polynomial during key generation

An attacker could also attempt to recover s from the public key. Normally, this is an instance of R-LWE and will be hard to solve, however, if part of the

public key would be $t_1 = a_1 s$, it becomes very easy. This could be achieved by zeroing the error polynomial e_1 during key generation. One possibility is skipping the call to `sample_gauss_poly()` in Listing 3.21, which will leave e_1 uninitialized and potentially zero on some architectures. Alternatively they could induce a loop aborting fault in Listing 3.24 line 7, by skipping the instruction `jne L18`. This has been conjectured to be easier, because it is easier to detect the repeating pattern of the loop [11]. Both methods can also be applied to zero e_2 and s, which are sampled in the same way.

Listing 3.21: During KeyGen: Sampling the error polynomial e_1

```
1 do
2 {
3     sample_gauss_poly(poly_E1);
4 }
5 while(check_E(poly_E1) != 0);
```

Listing 3.22: KeyGen: Generated assembly code for sampling e_1

```
1 L54:
2 [...]
3 call _sample_gauss_poly
4 [inlined code of check_E()]
5 ja L54
```

Listing 3.23: sample.c: Gaussian sampling function

```
1  void sample_gauss_poly(poly x)
2  {
3    unsigned int j;
4    double gauss;
5
6      for(j=0; j<PARAM_N; j++)
7      {
8        gauss = fmodq(sample_gauss());
9        x[j] = gauss;
10     }
11 }
```

Listing 3.24: sample.c: Disassembled sampling function

```
1 _sample_gauss_poly:
2 [...]
3 L18:
4 call _sample_gauss
5 [...]
6 cmpq %rbx, %rbp
7 jne L18
8 [...]
9 ret
```

We present two possible countermeasures in the following.

Additional sampling to prevent a key from being zero

One countermeasure which falls under the category of redundant code execution works as follows.

Listing 3.25: KeyGen: sampling the error polynomial multiple times

```
1 sample_gauss_poly(poly_E1);
2 do {
3     sample_gauss_poly(poly_E1);
4 } while(check_E(poly_E1) != 0);
```

Listing 3.25 shows the sampling of e_1 during key generation. Line 1 was added to the original code. Even if the attacker can skip the sampling of e_1 in line 1, it is performed a second time in line 3, this time making sure that e_1 satisfies the necessary conditions. If the attacker skipped only the sampling in line 3, then e_1 would already contain some nonzero values. However, in this case signatures may not always verify, because the polynomial sampled in line 1 may not satisfy the condition for e_1 that is necessary for signatures to verify correctly. But either way, $t_1 \neq a_1 s$ and the public key does not leak the secret s. The same things apply to e_2. The obvious downside to this is that gaussian sampling is expensive, so the execution time of the key generation algorithm will increase by almost a third. We refer to Section 4.2 for exact numbers.

We call this countermeasure *sample_twice*.

Setting the public key to zero in case of a fault attack

We can also prevent this zeroing attack by checking the coefficients of each polynomial, just like in the case of zeroing y during signing. First, we determine how many coefficients we can allow to be zero, without changing the probability distribution too much:

The probability p of a single coefficient being zero is $p = \rho_\sigma(0)/\rho_\sigma(\mathbb{Z})$ with $\rho_\sigma(0) = \exp(0) = 1$ and $\rho_\sigma(\mathbb{Z}) = 1 + 2\sum_{z=1}^{\infty} \rho_\sigma(z) = 75.1988$. This works out to $p = 0.013298085$. Let z be the number of zero coefficients in a ring-TESLA polynomial. Then the probability $Pr(z > 62)$ of more than 62 coefficients being zero is

$$1 - Pr(z \leq 62) = 1 - \sum_{i=0}^{62} \binom{512}{i} p^i (1-p)^{512-i} = 7.75 \times 10^{-40} < 2^{-128}.$$

We can now introduce the following countermeasure to the signature scheme, which sets the public key to zero in case of a fault attack. For the purpose of legibility we use the shorthand z_p instead of $|p|^0_{<62}$ with $p \in \mathcal{R}_q$.

Algorithm 3.3: Original KeyGen: Sampling the polynomials and computation of the public key

1 $s, e_1, e_2 \leftarrow D_\sigma^n$
2 [...]
3 $t_1 \leftarrow a_1 s + e_1 \pmod q$
4 $t_2 \leftarrow a_2 s + e_2 \pmod q$
5 [...]

Algorithm 3.4: KeyGen: Countermeasure against zeroing the polynomials

1 $s, e_1, e_2 \leftarrow D_\sigma^n$
2 [...]
3 $t_1 \leftarrow z_{e_1} \cdot z_s \cdot (a_1 s + e_1) \pmod q$
4 $t_2 \leftarrow z_{e_2} \cdot z_s \cdot (a_2 s + e_2) \pmod q$
5 [...]

If either e_1, e_2, or s in Algorithm 3.4 contain too many zero coefficients then either t_1, t_2, or both are zero respectively. We now provide the implementation of the above pseudocode in Listing 3.28.

Listing 3.28: KeyGen with countermeasure against zeroing in line 6–10

```
1  poly_mul_fixed(poly_A1S,poly_S,poly_a1);
2  poly_add(poly_T1,poly_A1S,poly_E1);
3  poly_mul_fixed(poly_A2S,poly_S,poly_a2);
4  poly_add(poly_T2,poly_A2S,poly_E2);
5
6  int zs = check_zeroes_62(poly_S);
7  int ze1 = check_zeroes_62(poly_E1);
8  int ze2 = check_zeroes_62(poly_E2);
9  poly_mul_constant(poly_T11, poly_T1, zs*ze1);
10 poly_mul_constant(poly_T22, poly_T2, zs*ze2);
11
12 compress_pk(pk, poly_T11, poly_T22);
13 compress_sk(sk, poly_S, poly_E1, poly_E2);
```

Listing 3.29: Implementation of $|\cdot|^0_{<62}$ that can resist a single fault attack

```
1  int check_zeroes_62(poly p) {
2      int nonzeroes = 0;
3
4      for (int i = 0; i < PARAM_N; i++) {
5          if (p[i] != 0.0) {
6              nonzeroes++;
7          }
8      }
9
10     asm volatile (
11             "cmpl $450, %0;"
12             "cmpl $450, %0;"
13             "setg %%bl;"
14             "setg %%bl;"
15             "cmpl $513, %0;"
16             "cmpl $513, %0;"
17             "setl %%bh;"
18             "setl %%bh;"
19             "andb %%bh, %%bl;"
20             "andb %%bh, %%bl;"
21             "movzbl %%bl, %1;"
22             "movzbl %%bl, %1;"
23     :"=r"(nonzeroes)
24     :"r"(nonzeroes)
25     :"%ebx"
26     );
27
28     return nonzeroes;
29 }
```

Note: Due to what looks like a bug in GCC 7.1.0, we actually had to prefix the entire function definition with the `__attribute__((optimize("O0")))` directive, which turns off all optimization for this function. Otherwise it would in some cases not return 0 or 1, but rather a random integer.
Even a second fault would not suffice for a successful attack, because the attacker has to get the value of `poly_T1` into `poly_T11` and the same is true for `poly_T2` and `poly_T22`. For that, the product `zs*ze1` in Listing 3.28 line 9 would have to be 1, which requires both `zs` and `ze1` to be 1 or -1. Because `check_zeroes_62()` can resist a fault attack, this is rather difficult in our model.

There is one disadvantage to this approach: Zeroing s leads to $s = t_1 = t_2 = 0$ (because $s = 0$, so $|s|^0_{<62} = 0$ and thus both t_1 and t_2 are multiplied

by 0), which makes it trivial for anyone to forge signatures that can be verified with the public key $(t_1, t_2) = (0, 0)$: For any message μ the forger creates the signature $(0, H(0, 0, \mu))$. The verifier computes $w'_1 = a_1 z - t_1 c = a_1 \cdot 0 - 0 \cdot c = 0$ and similarly $w'_2 = 0$. In that case $c' = c''$ and the verifier believes they are looking at a genuine signature. This makes it necessary to forbid the public key $(0, 0)$. We did not implement this in the code of ring-TESLA, because such a check would inevitable require an if-statement, which can readily be attacked with a skipping fault. We believe that for additional security this check should be applied when transferring keys or saving them on a key server.

We call this countermeasure *zero_key*.

3.2.3 Zeroing the hash during verification

The following attack is described in [6]: Suppose Eve wants to pretend to be Alice during a communication with Bob. So she has to forge a signature (z, c') for a message μ and make Bob believe, that message was signed by Alice. In order to achieve this, she selects a random $z \in \mathcal{R}_{B-U}$ and calculates $c' = H(\lfloor a_1 z \rceil_{d,q}, \lfloor a_2 z \rceil_{d,q}, \mu)$. She then sends the signature (z, c') to Bob. When Bob verifies the signature, Eve mounts a fault attack and zeroes the hash $c \leftarrow F(c')$, for example by skipping the function call $F()$.

Now Bob computes $w'_1 = a_1 z - t_1 c = a_1 z - t_1 \cdot 0 = a_1 z$ and similarly $w'_2 = a_2 z$. He then computes $c'' = H(\lfloor w'_1 \rceil_{d,q}, \lfloor w'_2 \rceil_{d,q}, \mu) = H(\lfloor a_1 z \rceil_{d,q}, \lfloor a_2 z \rceil_{d,q}, \mu)$. This means that $c' = c''$ and $z \in \mathcal{R}_{B-U}$ and he believes that the signature is valid and was indeed created by Alice.

However, this attack only works in theory, in the current implementation it is not actually possible to set $c = 0$, even when intending to do so: The polynomial c is defined to have exactly $\omega = 11$ coeffcients which are one, and the rest is zero. In code this is implemented as `uint32_t pos_list[PARAM_W]`, which encodes the polynomial $p(x) = \sum_{i=0}^{n-1} \alpha_i x^i$ where $\alpha_i = 1$ if i is contained in the array `pos_list`.

This means that if we set `pos_list = {0,. . . ,0}`, in theory we get $p(x) = 1$, not $p(x) = 0$. In practice this is an ill-formed polynomial and the multiplication $t_1 c$ via `computeEc()` returns neither 0 nor t_1, but some polynomial whose coefficients are close to t_1. So $w'_1 \neq a_1 z$, $w'_2 \neq a_2 z$, and thus $c' \neq c''$. Bob will not accept the signature. Because of this, we do not propose or implement a countermeasure.

4 Efficiency of the countermeasures

The following benchmarks were recorded on a 4.0 GHz Intel Core i7-6700k. GCC 7.1.0 was used for compilation. The time it took to run the code is given in clock cycles, code sizes are given in number of instructions. The code was optimized using the following compiler flags: `-msse2avx -mfma -march=corei7-avx -Ofast -fomit-frame-pointer`.

4.1 Execution time of the algorithms

Figure 4.1 shows the execution time for each algorithm of ring-TESLA. The execution time was averaged over 2000 runs for the key generation and 20,000 runs for the other two algorithms. Note that even when averaging over 20,000 runs, the execution times for Verify vary up to around 2000 cycles. The difference is even bigger for KeyGen, often differing around 500,000 cycles.

Figure 4.2 shows a histogram of how the actual execution times of the Verify algorithm are distributed. The time it takes to verify two messages can differ up to 4000 cycles. In theory, verifying signatures for messages of the same length should always take the same amount of time, so the multimodal distribution of verification times is likely due to scheduling of the operating system and varying system load. Figure 4.3 shows that the execution time of KeyGen varies more than an order of magnitude, which is due to gaussian sampling.

Figure 4.1: Execution times of ring-TESLA's algorithms in clock cycles

	KeyGen	Sign	Verify
Cycles	33,429,812	335,735	71,299

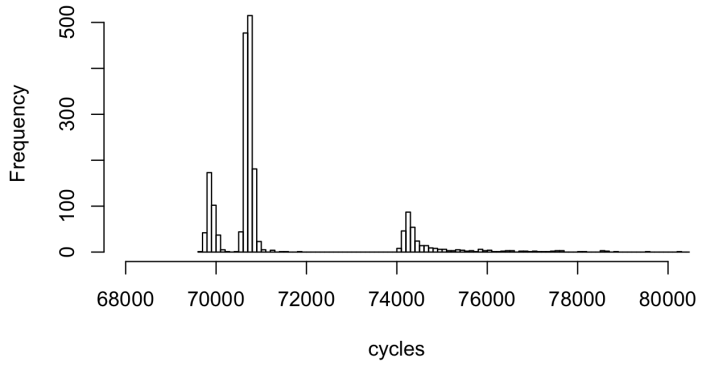

Figure 4.2: Distribution of the execution time for 2000 runs of Verify (without a handful of outliers that took up to 120,000 cycles)

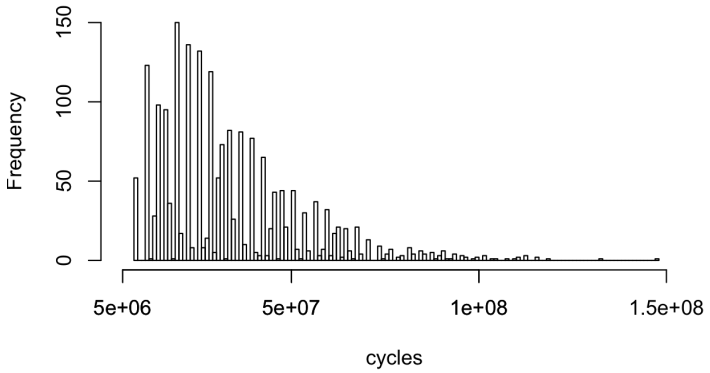

Figure 4.3: Distribution of the execution time for 2000 runs of KeyGen

4.2 Performance impact of the countermeasures

The change in code size was calculated by compiling all relevant files with `gcc -S -Ofast` with and without the countermeasure and counting the difference in the number of instructions. The change in execution time was measured by calling the function in Listing 4.1 right before and after the code that was timed and calculating the difference between the return values. The function was provided to us together with the original ring-TESLA code.

Figure 4.4 shows the overhead that the countermeasures introduce. We write "~ 10" if the change in execution time or file size is on the order of around 10 *instructions*. All other execution times are given in *cycles*. They are averaged over 100 runs and rounded to one or two significant digits. The numbers in brackets are the subsections in which we introduced the respective countermeasures.

We do not employ the countermeasure *sample_twice*, because it is fairly inefficient (due to gaussian sampling) and less secure against higher order attacks than *zero_key*. Similarly, we do not use *count_zeroes*, because *zero_signature* is more secure against higher order attacks. Of the remaining countermeasures the most expensive one in absolute terms is *zero_key* in KeyGen, which takes around 3000 clock cycles. However, the natural fluctuation of the execution time of KeyGen is many orders of magnitude bigger, so the drawback of introducing this countermeasure is negligable. The most expensive countermeasure in relative terms is *zero_signature*, which only increases the execution time by around 0.45%, which again is well within the expected natural variation. When looking at the number of cycles the countermeasures *count_zeroes, zero_signature* and *zero_key* take, we noticed that they are slightly lower than what we would expect, given the number of polynomial coefficients that have to be checked or multiplied. This is most likely due to the heavy compiler optimization.

A fully functioning code snippet, which creates and verifies a signature compiles to an executable of 95 kilobytes size. The addition of at most 300 instructions per countermeasure will not be a problem on most architectures, however on some embedded systems with very small memory this may be a disadvantage.

Figure 4.4: Increase in execution time (given in clock cycles) and binary size (given in machine instructions) for each countermeasure

Countermeasure	Section	Algorithm	Time (cyc.)	Size (instr.)
new_variable	3.1.1	KeyGen	~ 10	~ 10
rewrite_branchless	3.1.2	Verify	~ 10	~ 10
additional_rejection	3.1.3	Sign	1100	241
count_zeroes	3.2.1	Sign	400[1]	167
zero_signature	3.2.1	Sign	1500	237
zero_key	3.2.2	KeyGen	3000[2]	286
sample_twice	3.2.2	KeyGen	9,000,000[3]	~ 10

Listing 4.1: Function used to count the number of CPU cycles

```
1 long long cpucycles(void)
2 {
3     unsigned long long result;
4     asm volatile(".byte 15;.byte 49;shlq $32,%%rdx;orq %%rdx,%%rax"
5     : "=a" (result) :: "%rdx");
6     return result;
7 }
```

[1]The time is for one execution of the `count_zeroes()` function. Due to rejection sampling, this function is usually executed two to four times.

[2]Due to the apparent bug in GCC 7.1.0, mentioned in Section 3.2.2, we had to turn off the compiler optimization for the code used in this countermeasure. This does not affect optimization of the rest of the code. The actual non-optimized execution time is around 4100 cycles.

[3]This is the approximate time it takes to sample all three polynomials a second time.

5 Conclusion

In this work, we presented countermeasures against first order fault attacks on ring-TESLA. The vulnerabilities were described by [6]. We analyzed the generated assembly code of the C implementation of ring-TESLA, to show how an attacker might actually perform these attacks. We distinguished zeroing and skipping faults, and presented countermeasures for each vulnerability. For each countermeasure we showed why it is successful in hampering an attack. In some cases we also presented countermeasures that can resist higher order attacks. Lastly, we measured how each countermeasure increases the execution time and binary size of the signature scheme.

Of the countermeasures we implemented the following could be applied to other signatures schemes as well:

- Define new variables to hold the result of every arithmetic operation. This leads to uninitinialized values, if the operation is skipped. However you have to make sure, that a random or zero result does not easily permit a security breach.
- The function `check_zeroes()` was a valuable asset in this work. It can be easily adapted to resist high order fault attacks, which is due to instruction level redundancy. However, not all instructions can be executed multiple times without changing the output, for example `add`, or `mul`.
- Inlining functions prevents skipping the corresponding function call. However they do not prevent attacks on the inlined code that have the same effect, for example a loop abortion fault.
- If-statements should be avoided, especially when they check the correctness of a signature, key, or security critical invariant. That is because they can always be attacked by a simple skipping fault. Loops can also be problematic, so for every loop you have to consider what happens if it is skipped.

However, when implementing a countermeasure against a fault attack, we always have to make sure that the countermeasure itself does not introduce a new vulnerability. Except for sampling a polynomial multiple times, all

countermeasures we investigated are very efficient, both in terms of executable size as well as execution time.

We only provided countermeasures against the faults described in [6]. However the authors only analyzed possible fault attacks against the high level code of ring-TESLA, not fault attacks against low-level mathematic operations, such as the polynomial multiplication or addition. Whether or not this code contains vulnerabilities is something that could be investigated in the future. Furthermore it would be interesting to investigate higher order fault attacks, since we only covered these in some cases.

Acknowledgement

Thanks to Nina Bindel for supervising this thesis, providing valuable feedback, and helping me stay on track. Thanks to Sedat Akleylek for providing details about the ring-TESLA signature scheme, my parents for the in-depth discussions about C and assembly code, my mother for proofreading, my father for helping me solve the optimization bug, and Benjamin Rosswinkel for helping me understand the underlying mathematics and for proofreading.

Bibliography

[1] Sedat Akleylek, Nina Bindel, Johannes Buchmann, Juliane Krämer, and Giorgia Azzurra Marson. An efficient lattice-based signature scheme with provably secure instantiation. Cryptology ePrint Archive, Report 2016/030, 2016. http://eprint.iacr.org/2016/030.

[2] Erdem Alkim, Nina Bindel, Johannes A. Buchmann, and Özgür Dagdelen. TESLA: tightly-secure efficient signatures from standard lattices. *IACR Cryptology ePrint Archive*, 2015:755, 2015.

[3] Shi Bai and Steven D. Galbraith. An improved compression technique for signatures based on learning with errors. Cryptology ePrint Archive, Report 2013/838, 2013. http://eprint.iacr.org/2013/838.

[4] Josep Balasch. Introduction to fault attacks. https://www.cosic.esat.kuleuven.be/summer_school_sardinia_2015/slides/Balasch.pdf.

[5] Daniel J Bernstein, Nadia Heninger, Paul Lou, and Luke Valenta. Post-quantum rsa. In *International Workshop on Post-Quantum Cryptography*, pages 311–329. Springer, 2017.

[6] Nina Bindel, Johannes Buchmann, and Juliane Krämer. Lattice-based signature schemes and their sensitivity to fault attacks. Cryptology ePrint Archive, Report 2016/415, 2016. http://eprint.iacr.org/2016/415.

[7] Nina Bindel, Juliane Krämer, and Johannes Schreiber. Special Session: Hampering fault attacks against lattice-based signature schemes - countermeasures and their efficiency. In *International Conference on Hardware/Software Codesign and System Synthesis- CODES+ISSS'17, Seoul, South Korea, October 2017, Proceedings*. ACM, 2017.

[8] Özgür Dagdelen, Rachid El Bansarkhani, Florian Göpfert, Tim Güneysu, Tobias Oder, Thomas Pöppelmann, Ana Helena Sánchez, and Peter Schwabe. High-speed signatures from standard lattices. In *Progress in Cryptology - LATINCRYPT 2014 - Third International Conference on Cryptology and Information Security in Latin America,*

Florianópolis, Brazil, September 17-19, 2014, Revised Selected Papers, pages 84–103, 2014.

[9] Léo Ducas, Alain Durmus, Tancrède Lepoint, and Vadim Lyubashevsky. Lattice signatures and bimodal gaussians. In *Advances in Cryptology - CRYPTO 2013 - 33rd Annual Cryptology Conference, Santa Barbara, CA, USA, August 18-22, 2013. Proceedings, Part I*, pages 40–56, 2013.

[10] Léo Ducas and Phong Q. Nguyen. Learning a zonotope and more: Cryptanalysis of ntrusign countermeasures. In *Advances in Cryptology - ASIACRYPT 2012 - 18th International Conference on the Theory and Application of Cryptology and Information Security, Beijing, China, December 2-6, 2012. Proceedings*, pages 433–450, 2012.

[11] Thomas Espitau, Pierre-Alain Fouque, Benoît Gérard, and Mehdi Tibouchi. Loop abort faults on lattice-based fiat-shamir & hash'n sign signatures. *IACR Cryptology ePrint Archive*, 2016:449, 2016.

[12] Amos Fiat and Adi Shamir. How to prove yourself: Practical solutions to identification and signature problems. In *Advances in Cryptology - CRYPTO '86, Santa Barbara, California, USA, 1986, Proceedings*, pages 186–194, 1986.

[13] Tim Güneysu, Vadim Lyubashevsky, and Thomas Pöppelmann. Practical lattice-based cryptography: A signature scheme for embedded systems. In *Cryptographic Hardware and Embedded Systems - CHES 2012 - 14th International Workshop, Leuven, Belgium, September 9-12, 2012. Proceedings*, pages 530–547, 2012.

[14] Tim Güneysu, Vadim Lyubashevsky, and Thomas Pöppelmann. Practical lattice-based cryptography: A signature scheme for embedded systems. In *Proceedings of the 14th International Conference on Cryptographic Hardware and Embedded Systems*, CHES'12, pages 530–547, Berlin, Heidelberg, 2012. Springer-Verlag.

[15] Jeffrey Hoffstein, Nick Howgrave-Graham, Jill Pipher, Joseph H. Silverman, and William Whyte. NTRUSIGN: digital signatures using the NTRU lattice. In *Topics in Cryptology - CT-RSA 2003, The Cryptographers' Track at the RSA Conference 2003, San Francisco, CA, USA, April 13-17, 2003, Proceedings*, pages 122–140, 2003.

[16] Oded Regev. The learning with errors problem (invited survey). In *Proceedings of the 25th Annual IEEE Conference on Computational Complexity, CCC 2010, Cambridge, Massachusetts, June 9-12, 2010*, pages 191–204, 2010.