This report, appearing as an IACR ePrint Archive Record, is an extended version of the corresponding article that appears in the proceedings of FPS 2017.

Bounding the cache-side-channel leakage of lattice-based signature schemes using program semantics

Nina Bindel, Johannes Buchmann, Juliane Krämer,
Heiko Mantel, Johannes Schickel, and Alexandra Weber

Computer Science Department, TU Darmstadt, Germany
{nbindel, buchmann, jkraemer}@cdc.informatik.tu-darmstadt.de,
{mantel, schickel, weber}@mais.informatik.tu-darmstadt.de

September 27, 2017

Abstract

In contrast to classical signature schemes, such as RSA or ECDSA signatures, the lattice-based signature scheme ring-TESLA is expected to be resistant even against quantum adversaries. Due to a recent key recovery from a lattice-based implementation, it becomes clear that cache side channels are a serious threat for lattice-based implementations. In this article, we analyze an existing implementation of ring-TESLA against cache side channels. To reduce the effort for manual code inspection, we selectively employ automated program analysis. The leakage bounds we compute with program analysis are sound overapproximations of cache-side-channel leakage. We detect four cache-side-channel vulnerabilities in the implementation of ring-TESLA. Since two vulnerabilities occur in implementations of techniques common to lattice-based schemes, they are also interesting beyond ring-TESLA. Finally, we show how the detected vulnerabilities can be mitigated effectively.

1 Introduction

The threat posed by quantum computers to current public-key cryptography is known since Shor presented a quantum algorithm to solve the factorization and the discrete logarithm problem in polynomial time [31]. How serious this threat is taken became clear, e.g., when NIST announced to start a standardization process for quantum-resistant schemes beginning in fall 2017 [23].

A promising quantum-resistant, also called post-quantum, candidate to substitute current public-key cryptography is lattice-based cryptography that enjoys, among other things, strong security guarantees. However, security guarantees can be undermined by side-channel vulnerabilities at the implementation level. So far, this has happened mostly for implementations of classical cryptography, e.g., [2, 8, 26, 34]. However, first approaches to analyze lattice-based cryptography with respect to side-channel attacks are already made, e.g., [25, 27–29]. Recently, Groot Bruinderink et al. presented the first attack against a lattice-based signature scheme and broke the scheme BLISS [15] using cache side channels of the Gaussian sampling during the signature generation [17]. Although none of the existing lattice-based signature schemes (or their implementations) claim to be secure against side channels, the attack in [17] raises the question how lattice-based signature schemes can be implemented without cache-side-channel leakage. Furthermore, in the light of NIST's standardization process, it is important to analyze lattice-based implementations against cache side channels. The scheme ring-TESLA [3], which is one of the most efficient lattice-based signature

Proceedings version of this article to appear in: J. M. Fernandez et al. (Eds.): FPS 2017; LNCS TBA; pp. TBA;
© Springer International Publishing AG 2017; The final publication is available at Springer via http://dx.doi.org/TBA

schemes, seems to be a good candidate to be implemented without cache side channels: During signature generation, ring-TESLA does not use Gaussian sampling, the sampling method that was exploited in the attack on BLISS in [17].

In this work, we use program analysis to compute upper bounds on the cache-side-channel leakage of lattice-based implementations at the example of the signature generation in ring-TESLA[1]. More concretely, we follow an approach based on information theory and reachability analysis, which is implemented in CacheAudit [13]. Variants of CacheAudit were used to analyze multiple cryptographic implementations. CacheAudit 0.2 [13] was used to analyze PolarSSL AES and the eSTREAM Profile 1 portfolio (HC-128, Rabbit, Salsa20, and Sosemanuk). CacheAudit 0.2b was used in a systematic study of AES implementations [21]. Another extension of CacheAudit 0.2 was used on modular exponentiation from the libraries libgcrypt and OpenSSL [12]. In this article, we extend CacheAudit 0.2b to CacheAudit 0.2c and apply it to ring-TESLA. This is the first analysis of a post-quantum scheme using CacheAudit.

With CacheAudit 0.2c, we determine upper bounds on the leakage of a ring-TESLA implementation for four attacker models. The bounds are sound, i.e., conservative with respect to the attacker models. We obtain upper bounds between 2.6bit and 51.6bit of potential cache-side-channel leakage. By inspecting the code manually, we then identify vulnerable subroutines. We implement countermeasures in the vulnerable subroutines to mitigate the cache-side-channel leakage. Finally, we argue for the effectiveness of the mitigations. For two subroutines, the argument is completely automated by an analysis with CacheAudit 0.2c that reports 0bit leakage.

According to our code inspection, a potential for leakage in the signature computation remains, which is intrinsic to a method called rejection sampling. Rejection sampling is used by design in the most efficient lattice-based signature schemes, such as [3, 6, 7, 15]. We argue that the attacker cannot exploit this potential leakage to get information about the secret key. Therefore, we consider our resulting implementation of ring-TESLA to be resistant to the four types of cache-side-channel attacks we consider.

In summary, the contributions of this article are the following.

- We detect four cache-side-channel vulnerabilities in an existing implementation of ring-TESLA by code inspection, selectively supported by automatic program analysis.
- To mitigate the detected vulnerabilities, we augment the ring-TESLA implementation by side-channel countermeasures. We argue for the effectiveness of the countermeasures, again supported by selective program analysis.
- To automate parts of our analysis of the unmitigated and mitigated ring-TESLA implementation, we extend the analysis tool CacheAudit 0.2b to CacheAudit 0.2c. More concretely, we implement support for ten additional x86 instructions. The support can be used to analyze occurrences of these instructions in x86 binaries and is not limited to ring-TESLA.

The detection and mitigation of vulnerabilities not only hardens the ring-TESLA implementation against side-channel attacks. Multiple lattice-based primitives, such as key exchange protocols, encryption, and signature schemes, use techniques similar to the ones we analyze in ring-TESLA. In particular, rejection sampling and sparse multiplication, where we find two of the potential vulnerabilities, occur in ring-TESLA, as well as in other lattice-based primitives. Hence, our results also pave the way to make other lattice-based implementations more trustworthy using program analysis.

[1]We analyze an implementation of ring-TESLA despite an error that was detected in its security reduction, since we expect that reductions given for its predecessor TESLA [4, 6] will be applicable to ring-TESLA as well. Hence, we consider it to be a good candidate for practical applications that require post-quantum signatures.

Organization In Section 2, we provide background information on the signature scheme ring-TESLA, cache side channels, and CacheAudit. In Section 3, we describe the extension of CacheAudit 0.2b to CacheAudit 0.2c and necessary modifications on the implementations of ring-TESLA to make the application of CacheAudit 0.2c possible. Details on the practical analysis and results of the first application of CacheAudit 0.2c to ring-TESLA are shown in Section 4. A manual analysis of the potential cache-side-channel leakage in ring-TESLA is presented in Section 5. Our countermeasures and the analysis of their efficacy are explained in Section 6. We conclude this work in Section 7.

2 Preliminaries

2.1 Notation

For an integer $n \in \mathbb{N}$, we define $q \in \mathbb{N}$ to be a prime with $q = 1 \pmod{2n}$. We denote the finite field $\mathbb{Z}/q\mathbb{Z}$ with representatives in $[-q/2, q/2] \cap \mathbb{Z}$ by $\mathbb{Z}_q$. Furthermore, we define $\mathcal{R}_q = \mathbb{Z}_q[x]/\langle x^n + 1\rangle$ and $\mathcal{R}_{q,[B]} = \{\sum_{i=0}^{n-1} a_i x^i \mid a_i \in [-B, B] \cap \mathbb{Z}\}$ for $B \in [0, q/2] \cap \mathbb{Z}$ and $\mathbb{B}_{n,\omega} = \{\sum_{i=0}^{n-1} a_i x^i \mid a_i \in \{-1, 0, 1\}, \sum_{i=0}^{n-1} |a_i| = \omega\}$ for $\omega \in [0, n] \cap \mathbb{Z}$. All logarithms are in base 2. Let $\sigma \in \mathbb{R}_{>0}$. Let v be a polynomial, then $v \leftarrow_\sigma \mathcal{R}$ means sampling each coefficient of v with discrete Gaussian distribution with standard deviation σ and mean 0 over $\mathbb{Z}$. For a finite set S, we write $s \leftarrow_\$ S$ to indicate that an element s is sampled uniformly at random from S.

2.2 Description of ring-TESLA

The signature scheme ring-TESLA is parametrized by n, ω, d, B, q, U, L, κ, σ, by the hash function $H : \{0,1\}^* \to \{0,1\}^\kappa$, and by the encoding function $F : \{0,1\}^\kappa \to \mathbb{B}_{n,\omega}$, see Figure 1. For detailed information about the system parameters and the encoding function F, we refer to the original work [3]. For $c \in \mathbb{Z}$, we denote by $[c]_L$ the unique representative of c in $(-2^{d-1}, 2^{d-1}] \cap \mathbb{Z}$ such that $c = [c]_L$ modulo 2^d and define $[\cdot]_M : \mathbb{Z} \to \mathbb{Z}, c \mapsto (c - [c]_L)/2^d$. The operators $[\cdot]_L$ and $[\cdot]_M$ correspond to the *least* or *most* significant bits, respectively. We extend the definitions to polynomials by applying $[\cdot]_L$ and $[\cdot]_M$ to each coefficient of the polynomial.

The secret key sk is a tuple of three polynomials $s, e_1, e_2 \leftarrow_\sigma \mathcal{R}$ where the entries of e_1 and e_2 have to be small enough for the scheme to be correct; the public key pk consists of the polynomials $a_1, a_2 \leftarrow_\$ \mathcal{R}_q$, $b_1 = a_1 s + e_1 \pmod{q}$, and $b_2 = a_2 s + e_2 \pmod{q}$. We depict the algorithm to generate a signature (c', z) for message μ in Figure 1. For verification of the signature (c', z), it is checked that $z \in \mathcal{R}_{q,[B-U]}$ and that c' equals $H([a_1 z - b_1 c]_M, [a_2 z - b_2 c]_M, \mu)$, with $c = F(c')$. The parameters proposed for ring-TESLA [3] are currently not supported by a security reduction, since in November 2016 an error was detected in the existing reduction. However, we expect

---

$\mathsf{Sign}(\mu; a_1, a_2, s, e_1, e_2)$:

1 $y \leftarrow_\$ \mathcal{R}_{q,[B]}$
2 $v_1 \leftarrow a_1 y \pmod{q}$
3 $v_2 \leftarrow a_2 y \pmod{q}$
4 $c' \leftarrow H([v_1]_M, [v_2]_M, \mu)$
5 $c \leftarrow F(c')$
6 $z \leftarrow y + sc$
7 $w_1 \leftarrow v_1 - e_1 c \pmod{q}$
8 $w_2 \leftarrow v_2 - e_2 c \pmod{q}$
9 If $[w_1]_L, [w_2]_L \notin \mathcal{R}_{q,[2^d-L]} \vee z \notin \mathcal{R}_{q,[B-U]}$:
10 Restart
11 Return (z, c')

Figure 1: Specification of the scheme ring-TESLA [3].

that existing security reductions for ring-TESLA's predecessor TESLA [4,6] are applicable to ring-TESLA as well[2]. Our modifications described in Section 3.1 and 4 do not depend on the *values* of the parameters. Hence, our results can be applied to other parameter sets of ring-TESLA as long as all values can be represented by the data type int.[3]

2.3 Cache side channels

A cache is a small piece of memory that stores selected entries from main memory for quick access by the Central Processing Unit (CPU). If the CPU accesses a memory entry, the access can lead to a cache hit (if the entry is stored in the cache) or to a cache miss (if the entry is not stored in the cache). Inside the cache, the memory entries are stored in sections called cache lines. The sequence of cache lines in a cache is partitioned into cache sets. In a k-way set-associative cache, each cache set consists of k cache lines. A cache has a strategy for replacing entries if the cache is full. A popular strategy is to replace the least recently used entry (LRU strategy). Variants of LRU are used, e.g., in Intel processors [1].

A cache-side-channel vulnerability exists if the interaction between a program and the cache depends on secret information, e.g., on a cryptographic key. In this case, an attacker, observing aspects of this interaction, might learn secret information. Attacks on cryptographic implementations have exploited secret-dependence in the trace of cache hits and misses [2], the time taken for cache hits and misses [8], and the final cache state of an execution [26].

2.4 Leakage bounds on cache side channels

In this article, we follow an approach based on information theory and reachability analysis to compute upper bounds on the cache-side-channel leakage of ring-TESLA (compiled to an x86 binary). Let Obs^a be the set of possible observations an attacker a can make about a single run of an x86 binary. Then $\log_2 |Obs^a|$ is an upper bound on the leakage of the binary with respect to min-entropy [32] and Shannon entropy [30] by [19, Theorem 1] and [5, Theorem 5.3].

We consider the following attacker models $a \in \{acc, accd, trace, time\}$ [13].

accd generalizes techniques like Evict+Time and Prime+Probe [26]. More concretely, it captures attackers who can determine the number of memory blocks in each cache set in the final cache state after a program execution.

acc captures attackers who can determine the position of each memory block in the final cache state, inspired by techniques like Flush+Reload [33].

trace captures trace-based attackers who can determine the trace of cache hits and misses that occur during one program execution. For instance, the trace-based attack in [2] uses such traces of hits and misses.

time models time-based attackers who can observe the running time of one program execution. Actual running times, as used in attacks like [8], are modeled by the amount of cache hits and cache misses that occur.

The possible observations under an attacker model can be computed by reachability analysis [13]. Let $\mathcal{D}$ be the set of the possible states during an execution and let $\mathrm{upd}_{\mathcal{D}} : \mathcal{D} \to \mathcal{D}$ model the concrete

[2]Security properties of schemes over standard lattices (like TESLA) often hold for corresponding schemes over ideal lattices (like ring-TESLA), e.g., the security reduction from [15] holds for the standard-lattice variant and for the ideal-lattice variant.

[3]If some values could not be represented by the type int, we would have to change the data types from int to long int. Still, we expect that most of the analysis would be similar.

semantics of x86 instructions, i.e., how the execution of instructions updates the state. The possible attacker observations depend on the states that an execution can reach according to $\text{upd}_{\mathcal{D}}$.

Instead of implementing a reachability analysis from scratch, we extend the existing tool CacheAudit 0.2b [21] – a version of CacheAudit [13]. CacheAudit performs a reachability analysis using abstract interpretation [10].

For an abstract reachability analysis, an abstract domain $\overline{\mathcal{D}}$ is defined, which abstracts from details of the concrete execution that are not relevant for the analysis. An abstraction function and a concretization function are defined to convert states between $\mathcal{D}$ and $\overline{\mathcal{D}}$. To represent executions in the abstract domain $\overline{\mathcal{D}}$, an abstract semantics $\text{upd}_{\overline{\mathcal{D}}} : \overline{\mathcal{D}} \to \overline{\mathcal{D}}$ is defined. To allow a transfer of analysis results from the abstract domain to the concrete domain, the abstract semantics $\text{upd}_{\overline{\mathcal{D}}}$ should be sound with respect to the concrete semantics $\text{upd}_{\mathcal{D}}$, i.e., it should overapproximate the set of reachable states in an execution.

CacheAudit uses multiple abstract domains [13]. The position of a memory block in the cache is abstracted by a set of possible positions. The values of registers and memory entries are abstracted by sets of possible values.

3 Enabling the automatic analysis of ring-TESLA

3.1 Integer implementation of ring-TESLA

We obtained the implementation of ring-TESLA [3] from the authors. The original implementation makes use of floating point operations. CacheAudit 0.2b, however, cannot analyze floating point operations and can therefore not be used to analyze the original implementation directly. An extension of CacheAudit 0.2b to support floating point instructions is out of scope for this article. Changing from floating point to integer operations does not affect the security of the signature scheme, since all operations during the signature generation (cf. Figure 1) are over $\mathbb{Z}_q$. Moreover, it is a step towards a ring-TESLA implementation for devices without floating point unit, e.g., embedded devices. We replaced all floating point instructions by integer instructions. The resulting implementation can be analyzed with CacheAudit 0.2c (our extension of CacheAudit 0.2b).

Listing 3.1 displays the parts of the signature generation function `crypto_sign` that are most important for our analysis, leaving out variable declarations.

3.2 Extension of CacheAudit 0.2b

The implementation of the scheme ring-TESLA is the first implementation of post-quantum cryptography (and of lattice-based cryptography) that is analyzed with CacheAudit. The implementation of ring-TESLA contains x86 instructions that are not supported by CacheAudit 0.2b. We extended CacheAudit 0.2b to CacheAudit 0.2c by adding support for these instructions.

To add support for additional x86 instructions to CacheAudit 0.2b, the underlying abstract semantics $\text{upd}_{\overline{\mathcal{D}}}$ must be extended. We implemented abstract semantics for the instructions in the ring-TESLA binary that are unsupported in CacheAudit 0.2b. Table 1 lists the opcodes (unique identifiers) and mnemonics (human-readable descriptions) of the instructions that we added.[4]

We illustrate the process of extending $\text{upd}_{\overline{\mathcal{D}}}$ at the example of the instruction `Bsr` (Bit scan reverse), which takes the operands `dst` and `src`. The concrete semantics $\text{upd}_{\mathcal{D}}$ of `Bsr dst src` is to compute the index of the most significant bit that is set, i.e., non-zero, in `src` [11]. If such a bit exists, its index is written to `dst` and the zero flag is set to 0. Otherwise, the zero flag is set to 1.

[4]The instructions 0xF7/3 and 0x99 were integrated independently but concurrently into a different version of CacheAudit by Doychev [14].

Listing 3.1: Signature generation in `crypto_sign`

```
[...]
while(1) {
  sample_y(vec_y);
  poly_mul_fixed(vec_v1, vec_y, poly_a1);
  poly_mul_fixed(vec_v2, vec_y, poly_a2);
  random_oracle(c, vec_v1, vec_v2, m, mlen);
  generate_c(pos_list, c);

  computeEc(E1c, sk+sizeof(int)*PARAM_N, pos_list);
  poly_sub(vec_v1,vec_v1, E1c);
  if (test_w(vec_v1) != 0){ continue; }

  computeEc(E2c, sk+sizeof(int)*PARAM_N*2, pos_list);
  poly_sub(vec_v2,vec_v2, E2c);
  if (test_w(vec_v2) != 0){ continue; }

  computeEc(Sc, sk, pos_list);
  poly_add(vec_y, vec_y, Sc);
  if (test_rejection(vec_y) != 0){ continue; }

  for(i=0; i<mlen; i++){ sm[i]=m[i]; }
  *smlen = CRYPTO_BYTES + mlen;
  compress_sig(sm+mlen, c, vec_y);
  return 0; }
```

Type	Opcodes (and mnemonics) of additional instructions
Arithmetic	13 (`Adc`), 1B (`Sbb`), 6B (`Imul`), F7/3 (`Neg`), F7/4 (`Mul`), F7/5 (`Imul`)
Bit string	0FBD (`Bsr`), 99 (`Cdq`)
Move	0F9C (`Setl`), 0F9F (`Setg`)

Table 1: Additional instructions for ring-TESLA in CacheAudit 0.2c

To support `Bsr`, we extended the parser, the internal instruction representation, and the abstract semantics in CacheAudit 0.2b. We extended the parser to create a `Bsr` instruction in the internal representation when it encounters the opcode `0FBD`. We implemented the abstract semantics of `Bsr` by a function `bsr` in the module `valAD`. The function consists of roughly 100 lines of OCaml code.

The function `bsr` operates on sets of potential values for `dst` and `src` and returns a map from possible resulting status flag combinations to the resulting values of registers and memory entries, for which the flag combinations can occur. For each possible value of `src`, we proceed according to the formalization of `Bsr` by Degenbaev [11]. We check whether the value consists only of zeros. In this case, we add a binding (mapping a flag combination to register and memory values) to the resulting map, in which the zero flag is 1 and the value of `dst` is unchanged. Otherwise, we first compute the number of leading zeros by divide and conquer, where we check recursively whether the first half of each non-zero prefix contains bits that are set to 1. The index of the most significant set bit is 64 minus the number of leading zeros. In this case, we add a binding to the resulting map, in which the zero flag is 0 and `dst` contains the computed index.

Our implementation for the other instructions follows the same pattern of parsing and abstract semantics, reusing existing support for similar instructions (e.g., with the same mnemonic) in CacheAudit 0.2b when possible.

4 Detection of potential leakage

We use CacheAudit 0.2c to analyze the signature generation in ring-TESLA for potential leakage of the secret key. We assume that the random number generator is secure and analyze the remaining computation with a few adaptations that allow a meaningful analysis with CacheAudit 0.2c. In the following, we provide details on the configurations of CacheAudit and ring-TESLA, details of our adaptations, and the results of our analysis.

Configuration of CacheAudit. We configure CacheAudit to use a 32kByte, 8-way set-associative data cache with a cache line size of 64Byte. This cache configuration is, e.g., used in the first level cache of the Intel Skylake architecture [18]. As the replacement strategy, we fix LRU.[5]

Configuration of ring-TESLA. We set the parameters of the ring-TESLA scheme to `PARAM_N` $= n = 512$, `PARAM_SIGMA` $= \sigma = 48$, `PARAM_Q` $= q = 33550337$, `PARAM_B` $= B = 4194303$, `PARAM_W` $= \omega = 19$, `PARAM_D` $= d = 23$, and `PARAM_U` $= U = 2848$. We analyze the function `crypto_sign` from the file `sign.c` in a 32-bit x86 binary of the ring-TESLA implementation. To this end, we use a wrapper function that calls `crypto_sign` with an uninitialized secret key, uninitialized message, uninitialized signature buffer, the message size 59 (as in the ring-TESLA test suite), and a pointer to `smlen` to store the length of the signed message including the signature. By leaving the secret key and message uninitialized, we treat them as secret input in our analysis.[6]

We compiled the ring-TESLA sources and our wrapper with *gcc* version 4.8.4, using `-static` for static linking, `-m32` to target an Intel i386 CPU architecture, and `-fno-stack-protector` to avoid insertion of code for overflow protection.

Adaptation of ring-TESLA. CacheAudit 0.2c does not support memory accesses that could refer to any possible address, e.g., in the analysis of loop counters that are advanced only under certain conditions and used to index array accesses. This occurs in the ring-TESLA routines `generate_c` and `sample_y`.

Listing 4.1 shows the implementation of the function `generate_c`. It uses rejection sampling to generate random values for the parameter array `pos_list`. The loop counter is only increased if the generated value is not rejected. To allow a meaningful analysis, we remove the check that rejects if the same value would occur twice in `pos_list` (highlighted in gray). That is, we overapproximate the possible values of `pos_list`. This manual overapproximation of the semantics preserves the validity of analysis results because it cannot decrease the number of possible attacker observations.

Listing 4.2 shows our adaptation of a loop in `sample_y`. Again, the loop counter advances only if the random number generated in the current iteration satisfies certain criteria. To allow a meaningful analysis with CacheAudit 0.2c, we remove the check of the random number and assign an uninitialized value that overapproximates the possible range to each entry in `mat_y`. With this adaptation, our analysis uses a safe overapproximation of the values that `sample_y` can return, but assumes that `sample_y` itself, i.e., the random number generator, does not have any cache-side-channel leakage.

The function `crypto_sign` contains a potentially infinite while loop, on which CacheAudit 0.2b does not terminate within reasonable time. To make the analysis of this loop feasible, we fix the

[5] We also investigated FIFO (first in first out) replacement. The leakage bounds (on the unmitigated implementation) are less than 10bit lower than under LRU.

[6] Treating the message as secret is overly conservative in a signature scenario. We investigated the effect of fixing the message to all '0's and obtained the same leakage bounds as for an uninitialized message in the unmitigated implementation.

Listing 4.1: Code of the subroutine `generate_c`

```
void generate_c(uint32_t *pos_list, unsigned char *c_bin){
  int32_t c[PARAM_N]; int cnt =0; int pos; [...]
  crypto_stream(r, R_LENGTH, nonce, c_bin);

  for(i=0; i<PARAM_N; i++){ c[i] = 0;}
  i=0;
  while(i<PARAM_W){
    pos = 0;
    pos = (r[cnt]<<8) | (r[cnt+1]);
    pos &= PARAM_N-1;
    cnt += 2;
    if (c[pos] == 0){ pos_list[i] = pos; c[pos]=1; i++; cnt++; } } }
```

Listing 4.2: Implementation of `sample_y`

```
//original
do {[...] if(val<0x7fffff) mat_y[i++] = val-PARAM_B; [...]} while(i<PARAM_N);

//adapted
for (i = 0; i < PARAM_N; ++i) { mat_y[i] = *(int *)(0x4) - PARAM_B; }
```

Attacker model	*acc*	*accd*	*trace*	*time*
Leakage in bit	12.9	2.6	51.6	9.5

Table 2: Upper bounds on the leakage of the signature generation

number of iterations while keeping the source code in the loop body unchanged. More concretely, we fixed the number of iterations to two to account for the effect of more than one iteration.

Note that, we use the modifications described in this section only for the initial automatic analysis of ring-TESLA. In the detailed manual inspection in Section 5, we use the unmodified integer implementation.

Analysis Results We obtain the leakage bounds listed in Table 2.[7] The bounds lie between 2.6bit and 51.6bit for the different attacker models. One run of the adapted ring-TESLA leaks at most 2.6bit to attackers under *accd*, at most 9.5bit to attackers under *time*, at most 12.9bit to attackers under acc, and at most 51.6bit to attackers under *trace*. In the remainder of this article we investigate whether these non-zero leakage bounds are substantiated by concrete threats and how the leakage bounds can be reduced by mitigating concrete threats.

5 Manual analysis of the potential leakage

In this section, we manually analyze the signature generation `crypto_sign` to check if the potential leakage detected by program analysis corresponds to an actual concern. Therefore, we analyze all subroutines used in `crypto_sign` that get secret values as input or compute secret values, e.g., `random_y` does not depend on secret information but the output value has to be kept secret. We identify substantiated threats of leakage to cache side channels (CSCs) in the routines `generate_c`, `test_w`, `test_rejection`, and `computeEc`.

[7] Throughout the article, we round bounds up to one decimal place and truncate the bounds to $3 * \text{PARAM\_N} * 32\text{bit} = 49152\text{bit}$, i.e., the maximum size of the key.

The following variables have to be kept secret during the execution of `crypto_sign`:

- the secret key `sk`; the secret key consists of three secret polynomials s, e_1, e_2, which are sometimes extracted from `sk`, e.g., Listing 5.11, line 4.
- the randomness `vec_y`; learning information about the randomness might give information about secrets, e.g., in Listing 3.1, line 18 the input `vec_y` of `poly_add(vec_y, vec_y, Sc)` corresponds to the randomness y in the pseudo code (cf. Figure 1), `Sc` corresponds to sc, and the output value `vec_y` corresponds to z. Hence, information about y might give information about sc since $z = y + sc$ is returned as part of the signature.
- the polynomials `vec_v1` and `vec_v2` to compute the hash value; `vec_v1` corresponds to the product of `vec_y` and `poly_a1` (similarly for `vec_v2`). Hence, learning information about `vec_v1` or `vec_v2` might give information about `vec_y`, which is also secret as explained above; the polynomials `poly_a1` and `poly_a2` are publicly known.
- the hash value `c` and the coefficient vector of the corresponding encoded polynomial `pos_list` have to be kept secret until line 19 in Listing 3.1. In line 19 it is decided whether the potential signature (computed in line 18) is returned together with the value `c` (and hence `c` and `pos_list` become public information; via the encoding function F the values can be computed from each other) or if all computed values are discarded. An attacker should not learn the values of the discarded polynomials, e.g., `c` or `pos_list`. If the attacker learns values of `pos_list` or `c` there exists a potential attack as described in the analysis of the subroutines `test_rejection` and `test_w`.
- the polynomials `E1c`, `E2c`, and `Sc`; those values correspond to the polynomials e_1c, e_2c, and sc, respectively, in the pseudo code, cf. Figure 1. Hence, information about those values might yield information about the secret polynomials e_1, e_2, or s.

First we present the subroutines that are not vulnerable to CSC attacks according to our manual analysis and give our reasoning. Afterwards, we describe potential CSCs in the subroutines `generate_c`, `test_w`, `test_rejection`, and `compute_Ec`. Finally, based on the analysis of the subroutines we discuss the vulnerability of the whole routine `crypto_sign`.

5.1 Analysis of the subroutine `sample_y`

The implementation of the subroutine `sample_y` is given in Listing 5.1. The function is called once in the sign algorithm in line 3 via `sample_y(vec_y)`. The result `vec_y` (corresponding to `poly mat_y` in Listing 5.1) of the subroutine has to be kept secret.

The use of the cache in this subroutine does not depend on the values of `mat_y` for the following reason. The only values loaded in the cache are `pos`, `buf`, `val`, and potentially the elements of `mat_y` that are changed during the routine. However, each of those values is loaded into the cache only once during the run of the subroutine (and is not overwritten by any other value, hence it stays in the cache during the run of the subroutine) since those elements are small enough to fit in the cache at the same time. The number of accesses to the individual values is not constant due to the branchings in line 8 and line 14. However, it depends only on the number of tries it takes to obtain valid values for `pos` and `val`, which is not secret.

Two more subroutines are called during `sample_y`, namely `fastrandombytes` and `randombytes`. Both do not leak any information about the value `mat_y` as we explain shortly next.

The subroutine `fastrandombytes` is given in Listing 5.2; there are no branchings, loops, or memory accesses that depend on secret values and hence all cache hits and misses are the same

Listing 5.1: Code of the subroutine `sample_y`

```
1  void sample_y(poly mat_y)
2  { int32_t val;
3    unsigned char buf[3*PARAM_N+68];
4    int pos=0, i=0;
5
6    fastrandombytes(buf,3*PARAM_N+68);
7    do
8    { if(pos == 3*PARAM_N+66){
9        fastrandombytes(buf,3*PARAM_N+68);
10       pos = 0;
11     }
12     val  = (*(int32_t *)(buf+pos)) & 0x7fffff;
13
14     if(val < 0x7fffff){ mat_y[i++] = val-PARAM_B;}
15     pos+=3;
16   }
17   while(i< PARAM_N);
18 }
```

Listing 5.2: Code of the subroutine `fastrandombytes`

```
1  void fastrandombytes(unsigned char *r, unsigned long long rlen)
2  { unsigned long long n=0;
3    int i;
4    if(!init){
5      randombytes(key, crypto_stream_KEYBYTES);
6      init = 1;
7    }
8    crypto_stream(r,rlen,nonce,key);
9
10   for(i=0;i<8;i++){ n ^= ((unsigned long long)nonce[i]) << 8*i;}
11   n++;
12   for(i=0;i<8;i++){ nonce[i] = (n >> 8*i) & 0xff;}
13 }
```

independently of the computed/sampled values. Furthermore, the call to `crypto_stream` does not pass any secret information to `crypto_stream` since `rlen` is a public parameter and `key` is random. Hence, the subroutine `fastrandombytes` does not have CSCs.

The subroutine `randombytes` is given in Listing 5.3. There are also no CSCs in this subroutines since (the only) branchings in line 3 and line 4 do not depend on secret (but on random) information.

5.2 Analysis of the subroutine `poly_mul_fixed`

The implementation of the subroutine `poly_mul_fixed` is given in Listing 5.4. In the signature algorithm in Listing 3.1 the function is called twice in lines 4 and 5 via `poly_mul_fixed(vec_v1,vec_y, poly_a1)` and via `poly_mul_fixed(vec_v2,vec_y,poly_a2)`, respectively. The values `vec_v1`, `vec_y`, and `poly_a1` (resp., `vec_v2`, `vec_y`, and `poly_a2`) correspond to the variables `result`, `x`, and `a` in Listing 5.4.

The input value that has to be kept secret is `x` (corresponding to `vec_y` in both calls). Later on in the subroutine also the value `result` has to be kept secret. The only condition depending on `x` is in line 4, Listing 5.4. However, this does not lead to a CSC since the used values in the then-branch only depend on `result` and `x` — the two values the if-condition depends on. Hence, by a cache hit the attacker just learns that `results != x` before, which does not give any additional information about the values of the secret. There are no further loops, conditions, or access indices depending

Listing 5.3: Code of the subroutine `randombytes`

```
1  void randombytes(unsigned char *x,unsigned long long xlen)
2  { while (xlen > 0)
3    { if (!outleft){
4        if (!++in[0]) if (!++in[1]) if (!++in[2]) ++in[3];
5        surf();
6        outleft = 8;
7      }
8      *x = out[--outleft];
9      ++x;
10     --xlen;
11   }
12 }
```

Listing 5.4: Code of the subroutine `poly_mul_fixed`

```
1  void poly_mul_fixed(poly  result, const poly  x, const poly  a)
2  { unsigned int i;
3
4    if (result != x){
5      for(i = 0; i < PARAM_N; i++){ result[i] = x[i];}
6    }
7    mul_coefficients(result, psis);
8    bitrev_vector(result);
9    ntt(result, omegas);
10
11   for(i=0; i<PARAM_N; i++){ result[i] = fmodq(result[i] * a[i]);}
12
13   bitrev_vector(result);
14   ntt(result, omegas_inv);
15   mul_coefficients(result, psis_inv);
16 }
```

on `x` or `result`, but they only depend on publicly known constants. However, there are four more calls to subroutines with input `result`: `ntt`, `fmodq`, `mul_coefficients`, and `bitrev_vector`. All four subroutines have control flow and memory accesses that are independent of secret values. The operations computed do not depend on the values of `result` but only on constants or public parameters such as `PARAM_N` or `PARAM_Q`. Hence, the subroutine `poly_mul_fixed` is not vulnerable to CSC attacks.

5.3 Analysis of the subroutine `random_oracle`

The implementation of the subroutine `random_oracle` is given in Listing 5.5. The function is called once in the sign algorithm in line 6 via `random_oracle(c,vec_v1,vec_v2,m,mlen)`. The input values correspond to the variables `c_bin`, `v1`, `v2`, `m`, and `mlen` in Listing 5.5.

The input values that have to be kept secret are `v1` and `v2` (they correspond to `vec_v1` and `vec_v2`, respectively, in the call). Due to the following reasons they are not leaked through a CSC. The control flow and memory accesses outside the call to `crypto_hash_sha512`, are independent of the value of the secrets, e.g., every coefficient of `v1` is computed modulo `PARAM_Q` (corresponding to the modulus q in the pseudo code) independently of whether the value is already in the interval $(-q/2, q/2]$ or not. Furthermore, the subroutine `compress_v` has control flow and memory accesses independent of secret information as can be seen in Listing 5.6. Also the subroutine `crypto_hash_sha512` does not contain secret-dependent branches, loops, or access indices. We do not display the (very long) source code here. Overall, `random_oracle` is not vulnerable to CSCs.

Listing 5.5: Code of the subroutine `random_oracle`

```
1  void random_oracle(unsigned char *c_bin,  poly v1, poly v2, const unsigned char
2  *m, unsigned long long mlen)
3  { int32_t t1[PARAM_N],t2[PARAM_N];
4    unsigned long long i;
5    unsigned char buf[2*PARAM_N+mlen];
6    compress_v(t1, v1);
7    compress_v(t2, v2);
8    for(i=0; i<PARAM_N; i++){ buf[i] = t1[i];}
9    for(i=0; i<PARAM_N; i++){ buf[i+PARAM_N] = t2[i];}
10   for(i=0; i<mlen; i++){ buf[i+2*PARAM_N] = m[i];}
11
12   crypto_hash_sha512(c_bin, buf, mlen+2*PARAM_N);
13 }
```

Listing 5.6: Code of the subroutine `compress_v`

```
1  static void compress_v(int32_t t[PARAM_N], poly v)
2  { int i;
3    for(i=0;i<PARAM_N;i++){
4      int32_t c = v[i] % PARAM_Q;
5      t[i] = ((int64_t)(v[i]-c))>>PARAM_D;
6    }
7  }
```

5.4 Analysis of the subroutines `poly_sub` and `poly_add`

The implementations of the subroutines `poly_sub` and `poly_add` are given in Listing 5.7 and Listing 5.8, respectively. The functions are called three times during the signature generation: via `poly_sub(vec_v1,vec_v1, E1c)` in line 10, in line 15 via `poly_sub(vec_v2,vec_v2,E2c)`, and via `poly_add(vec_y,vec_y,Sc)` in line 20. In each of these function calls, the input values correspond to the variables `result`, `x`, and `y` in Listing 5.7 and 5.8. The two subroutines are exactly the same except for one summand in line 4 in both of the listings, hence we analyze them together.

Both `result` and `y` have to be kept secret. The two subroutines consist of a for-loop depending on the publicly known value `PARAM_N` and calls to the subroutine `fmodq`. No branchings, loops, or access indices depend on secret values (in the routines `poly_sub` and `poly_add`, and the subroutine `fmod`, given in Listing 5.9). Hence, there are no CSCs in those two subroutines.

5.5 Analysis of the subroutine `compress_sig`

The implementation of the subroutine `compress_sig` is given in Listing 5.10. The function is called once in the sign algorithm in line 28 via `compress_sig(sm+mlen,c,vec_y)`.

In this subroutine the values `c` and `vec_y` have to be kept secret. There are neither branchings nor loops or access indices depending on the secret values, but all operations are computed exactly

Listing 5.7: Code of the subroutine `poly_sub`

```
1  void poly_sub(poly  result, const poly  x, const poly  y)
2  { unsigned int i;
3    for(i = 0; i < PARAM_N; i++){
4      result[i] = fmodq((int64_t)(x[i] + (PARAM_Q-y[i]))<<39);}
5  }
```

Listing 5.8: Code of the subroutine `poly_add`

```
1  void poly_add(poly  result, const poly  x, const poly  y)
2  { unsigned int i;
3    for(i = 0; i < PARAM_N; i++){
4      result[i] = fmodq((int64_t)(x[i] + y[i])<<39);}
5  }
```

Listing 5.9: Code of the subroutine `fmodq`

```
1  static int32_t fmodq(int64_t x)
2  {   int64_t u = (x * 206175203327);
3      u &= (((int64_t)1<<39)-1);
4      u *= PARAM_Q;
5      u += x;
6      return u>>39;
7  }
```

the same way for different secret values. Hence, there are no CSCs in `compress_sig`.

5.6 Analysis of the subroutine `compute_Ec`

The implementation of the subroutine `compute_Ec` is given in Listing 5.11. The function is called three times in sign: line 9 via `computeEc(E1c, sk+sizeof(int)*PARAM_N, pos_list)`, line 13 via `computeEc(E2c, sk+sizeof(int)*PARAM_N*2, pos_list)`, and line 17 via `computeEc(Sc, sk, pos_list)`. The input values correspond to `Ec`, `sk`, and `pos_list[PARAM_W]` in Listing 5.11. The values that have to be kept secret during this computation are `sk`, and, as part of `sk`, also `e`, cf. line 4 in Listing 5.11, `pos_list`, and `pos`. Most loops and branchings do not depend on any of the secret values. However, the for-loops in line 9 and line 10 in Listing 5.11 do depend on `pos_list`. The first for-loop runs for `j=0,...,pos_list[i]-1` and the second for-loop runs for `j=pos_list[i],...,PARAM_N`. In both loop bodies values are written in `Ec[j]`, i.e., independently of the values in `pos_list`, all entries of `Ec` are changed. Hence, no bits are leaked because of cache hits/misses depending on `Ec[j]`. However, there might be a possible leakage of `pos` (and hence of the values in the secret `pos_list`) because of the cache hits/misses depending on `e`. In both loop bodies values are read from `e` (namely, either `e[j+PARAM_N-pos]` or `e[j-pos]`) such that in both loops together all entries of `e` are read. However, leakage arises from the chronological order of cache hits and misses. We illustrate the leakage using an example: The array `e` consists of `PARAM_N` many entries of type int, i.e., each entry of `e` is represented in 32bit. Since one cache line is 64Byte (cf. Section 4), 16 entries (depending on the alignment in the memory) of `e` fit into one cache line.

Listing 5.10: Code of the subroutine `compress_sig`

```
1  static void compress_sig(unsigned char *sm, unsigned char *c, poly vec_z)
2  { int i,k;
3    int ptr =0;
4    int32_t t=0;
5
6    for (i=0; i<32; i++){ sm[ptr++] =c[i];}
7    for(i=0; i<PARAM_N; i++){
8       t = (int32_t)vec_z[i];
9       for(k=0;k<3;k++){ sm[ptr++] = ((t>>(8*(k))) & 0xff);}
10 }
```

Listing 5.11: Code of the subroutine `computeEc`

```
1  static void computeEc(poly Ec, const unsigned char *sk, const
2  uint32_t pos_list[PARAM_W]) {
3    int i,j, pos, * e;
4    e = (int*)sk;
5    for(i=0;i<PARAM_N;i++){ Ec[i] = 0;}
6
7    for(i=0;i<PARAM_W;i++){
8      pos = pos_list[i];
9      for(j=0;j<pos;j++){ Ec[j] += e[j+PARAM_N - pos];}
10     for(j=pos;j<PARAM_N;j++){ Ec[j] -= e[j-pos];} } }
```

Listing 5.12: Code of the subroutine `test_rejection`

```
1  static int test_rejection(poly poly_z) {
2    int i;
3    for(i=0; i<PARAM_N; i++){
4      if(poly_z[i]<-(PARAM_B-PARAM_U)||poly_z[i]>(PARAM_B-PARAM_U)){return 1;}}
5    return 0; }
```

Let `pos`=14. Then, under the trace-driven attacker model, an attacker sees one cache miss (on element `e[PARAM_N-14]`) and 13 cache hits (on elements `e[PARAM_N-13]`,...,`e[PARAM_N-1]`) during the loop in line 9.[8] However, two more entries of `e` are also already loaded in the cache, namely `e[PARAM_N-16]` and `e[PARAM_N-15]`. Thus, in the second loop in line 10, the attacker sees cache hits on those two elements. He might, hence, be able to determine the value of `pos` from the distribution of the hits for the considered cache line over the loops.

5.7 Analysis of the subroutine `test_rejection`

The implementation of the subroutine `test_rejection` is given in Listing 5.12. The function is called once in the sign algorithm in line 19, Listing 3.1, via `test_rejection(vec_y)`. The value `vec_y` corresponds to the variable `poly_z` in Listing 5.12 that has to be kept secret. The subroutine `test_rejection` consists of a for-loop that loops independently of the secret over `i=0,...,PARAM_N`. Within the for-loop, there is a secret-dependent if-condition that leads to a CSC vulnerability.

Assume a strong trace-driven attacker model, i.e., the attacker has a sequence of occurred cache hits and misses. Assume furthermore that `poly_z` is already loaded in the cache before the if-condition is evaluated.[9] When the if-condition in line 4, Listing 5.12, is never true, then the value `0` is returned and the attacker gets a sequence of `PARAM_N` (or 2·`PARAM_N` — depending on the compilation) hits. This essentially means that all coefficients of `poly_z` are in the interval $[-B+U, B-U]$ and, hence, the corresponding signature is compressed and returned (cf. Listing 3.1). Next, we consider the other case, i.e., the absolute value of at least one of the coefficients of `poly_z` is larger than $B-U$. That means that the if-condition in Listing 5.12 holds true for some `i`$\in \{$`0`, ...,`PARAM_N`$\}$. Hence, `1` is returned in the `i`-th iteration and the attacker gets a sequence of only `PARAM_N`$-$`i` hits. Hence, the attacker knows the exact index of the coefficient that violated the if-condition. Assume the attacker also knows the values in the array `pos_list` (which corresponds to the polynomial $c = F(H([v_1]_M, [v_2]_M, \mu))$ in Figure 1) from another cache-side-channel vulnerability. Then the attacker might know which coefficients of the secret s contributed to the `i`-th, large coefficient of

[8] To simplify our explanation we assume that the corresponding cache line ends with the last entry of `e`, i.e., the cache line starts with `e[PARAM_N-16]` and ends with `e[PARAM_N-1]`.

[9] Our arguments hold also true if we assume that `poly_z` is not loaded in the cache. In the ring-TESLA implementation, it is already loaded in line 18 in Listing 3.1.

Listing 5.13: Code of the subroutine `test_w`

```
1  static int test_w(poly poly_w)
2  { int i; int64_t left, right, val;
3    for(i=0; i<PARAM_N; i++){
4      val = (int64_t) poly_w[i];
5      val = val % PARAM_Q;
6      if (val < 0){ val = val + PARAM_Q;}
7     left = val;
8     left = left % (1<<(PARAM_D));
9     left -= (1<<PARAM_D)/2;
10    left++;
11    right = (1<<(PARAM_D-1))-PARAM_REJECTION;
12    if (abs(left) > right){ return -1; } }
13   return 0; }
```

`poly_z`. If an attacker learns the exact position `i` and the corresponding `pos_list` for many different values to the same secret key s, then the attacker might receive enough information about the size of the entries in s to successfully break the scheme via a learning-the-parallelepiped-attack [16, 24].

5.8 Analysis of the subroutine `test_w`

The implementation of the routine `test_w` is given in Listing 5.13. The function is called twice in the sign algorithm in line 11 with `test_w(vec_v1)` and in line 15 with `test_w(vec_v2)`. The values `vec_v1` and `vec_v2` have to be kept secret; they correspond to `poly_w` in Listing 5.13. Furthermore, the values of `val` and `left` have to be kept secret since they consists of partial information of `poly_w`. The CSC vulnerability is similar to the channel described previously for `test_rejection`. In the subroutine `test_w`, the CSC vulnerability comes from the early abortion depending on `left` in line 12 of Listing 5.13. When the if-condition in line 12 holds for some `i` and the corresponding `abs(left)`, `-1` is immediately returned and a trace-driven attacker might learn the exact index `i`.

5.9 Analysis of the subroutine `generate_c`

The implementation of the subroutine `generate_c` is given in Listing 4.1; it is called in the sign algorithm in line 7 via `generate_c(pos_list,c)`. The values `pos_list` and `c` correspond to the variables `pos_list` and `c_bin`, respectively, in Listing 4.1. The values `pos_list`, `c`, and `pos` have to be kept secret. There are no branchings or loops depending on the secret value, except for one if-condition on `c[pos]` in line 12 of Listing 4.1. If a cache with no-write-allocate policy is used, the values `c[i]` are not cached in line 5. Hence, an attacker might be able to find out which elements `c[i]` are cached in line 12 and to learn information about the values of `pos`. Together with the vulnerability in `test_rejection`, an attacker might be able to successfully break the scheme via a learning-the-parallelepiped-attack [16, 24].

5.10 Combined analysis of the overall signature generation

The most important parts of the implementation of `crypto_sign` are depicted in Listing 3.1. In the signature generation, most operations, branchings, or loops are independent of the *value* of the secret. Exceptions are the branchings in lines 11, 15, and 19 in Listing 3.1: They depend on secret values and, hence, the length of the observed trace of cache hits and misses depends on the branches that are taken. What does this mean from a cryptographic viewpoint? Assuming the subroutine `test_w` does not leak any bit, then the attacker does not learn more information about the secret if

he knows whether or not the condition in line 11 holds. The attacker would just learn that `vec_v1` does not fulfill the conditions needed for a valid signature. However, the attacker does not learn why exactly the condition was not fulfilled (the attacker does not learn the index on which the if-condition failed). Furthermore, since the value `vec_v1` depends on `vec_y` and the value `vec_y` is discarded if the if-condition in line 11 does not hold, the attacker does not get any additional information about the secret he did not know before. The same explanation also holds for the branchings in line 15 and line 19.

In summary, this means that there exists a potential leakage that we probably cannot get rid of, but it does not affect the security of the signature scheme as long as we do not have leakages in `test_w` and `test_rejection` or `generate_c`.

6 Mitigation of the vulnerabilities

6.1 Adaptation of vulnerable routines

In Section 5, we identified substantiated threats of CSC leakage in the four subroutines `test_w`, `test_rejection`, `computeEc`, and `generate_c`. Since the leakage in `generate_c` is only a concern in combination with the leakage in `test_w` and `test_rejection`, it suffices to analyze and mitigate the leakage in `test_w`, `test_rejection`, and `computeEc`. We analyze `test_w`, `test_rejection`, and `computeEc` individually with CacheAudit 0.2c to obtain leakage bounds on the unmitigated implementations. The leakage bounds are listed in Table 3a. There are, indeed, non-zero bounds for all three unmitigated routines.

We adapt the routines, as shown in Listing 6.1 to mitigate the leakage. In `test_rejection`, we collect the result, i.e., whether 0 or 1 is returned, in an auxiliary variable `res` and return it after `PARAM_N` iterations, instead of returning early in case of a failed test. In `test_w`, we also collect the result, i.e., whether 0 or 1 is returned, in an auxiliary variable `res`, instead of returning early in case of failure. Furthermore, we replace the branching on `val` by an assignment that masks the value by the branching condition. The idea to mask assignments by branching conditions comes from conditional assignment [22] - a program transformation to mitigate timing side channels, which performs rather well in practical evaluation [20]. In `computeEc`, we add preloading of the variable `e` to ensure that the sequence of cache hits and misses does not depend on the secret-dependent order of accesses (under the assumption that no process interferes with the cache during the ring-TESLA execution).

By code inspection, the modifications should remove the CSC leakage in the three routines. In the following, we investigate this with CacheAudit.

	acc	*accd*	*trace*	*time*
`test_w`	31	31	**49152**	19.3
`test_rejection`	**31**	**31**	10.1	10.1
`computeEc`	0	0	**20**	5.9

(a) Unmitigated routines

	acc	*accd*	*trace*	*time*
`test_w`	0	0	0	0
`test_rejection`	0	0	0	0
`computeEc`	0	0	**19**	4.4

(b) Mitigated routines

Table 3: Leakage bounds [bit]

Listing 6.1: Mitigations added to ring-TESLA

```
int test_rejection(poly poly_z) {
  int i; int res; res = 0;
  for(i=0; i<PARAM_N; i++){
     res |= (poly_z[i] < -(PARAM_B-PARAM_U));
     res |= (poly_z[i] > (PARAM_B-PARAM_U)); }
  return res; }

int test_w(poly poly_w) { [...]
  for(i=0; i<PARAM_N; i++) {
    val = poly_w[i]; val = val % PARAM_Q;
    val += (((unsigned int)val & 0x80000000) >> 31)*PARAM_Q;
    left = val; left = left % (1<<(PARAM_D));
    left -= (1<<PARAM_D)/2; left++;
    right = (1<<(PARAM_D-1))-PARAM_REJECTION;
    res |= (abs(left) - right > 0); }
  return -res; }

void computeEc([...]) { [...]
  for(i=0;i<PARAM_N;i++) Ec[i] = 0;
  for(i=0;i<PARAM_N;i++) tmp = e[i];
  for(i=0;i<PARAM_W;i++) {
    pos = pos_list[i];
    for(j=0;j<pos;j++) { Ec[j] += e[j+PARAM_N - pos]; }
    for(j=pos;j<PARAM_N;j++) { Ec[j] -= e[j-pos]; } } }
```

	acc	*accd*	*trace*	*time*
Unmitigated	12.9	2.6	**51.6**	9.5
Mitigated	8.1	1.6	**48.6**	9.0

Table 4: Leakage bounds [bit] for `crypto_sign`

6.2 Analysis of the effectiveness of the mitigations

We analyze the mitigated routines `test_w`, `test_rejection`, and `computeEc` using CacheAudit 0.2c and obtain the leakage bounds listed in Table 3b. For `test_w` and `test_rejection`, we obtain the leakage bound 0bit for all attacker models. Thus, we effectively removed the potential leakage. For `computeEc`, we obtain 0bit leakage bounds for *acc* and *accd* and non-zero leakage bounds, namely 19bit and 4.4bit, for *trace* and *time*, respectively. The bounds computed with CacheAudit are provable upper bounds, but not necessarily tight. The preloading of `e` should remove the leakage from `computeEc`, because it makes the caching of `e` independent of secrets. Since CacheAudit was able to recognize preloading as effective in other cases [13,21], it is interesting why CacheAudit 0.2c does not yield a 0bit leakage bound in this case. The investigation and fine-tuning of the analysis precision is an interesting direction for future work.

Table 4 lists the leakage bounds we obtain on `crypto_sign` with and without our countermeasures. All four leakage bounds are reduced by our countermeasures. The highest reduction is achieved for the *acc* leakage bound. The *acc* leakage bound is reduced by 4.8bit to 8.1bit.

Based on our manual inspection of the individual routines in ring-TESLA, there are two possible sources for the remaining potential leakage reported by CacheAudit 0.2c. One source is `generate_c`, where, as discussed above, the remaining leakage is harmless. The second source is the rejection sampling in `crypto_sign`. This matches the fact that the leakage bounds are non-zero. Note that,

since we compute upper bounds on the leakage based on overapproximation, the actual leakage of the implementation could be much lower than the leakage bounds. We expect in particular the leakage bounds for *trace* and *time* to be too conservative because CacheAudit 0.2c was not able to recognize the preloading countermeasure in the implementation of `computeEc` and the results of `computeEc` are propagated further through the implementation. Nevertheless, the bounds show that the CSC leakage of ring-TESLA to *acc*, *accd*, *trace*, and *time* is rather low. For *acc*, *accd*, and *time*, it even lies below 10bit.

Note that, the leakage bounds refer to information about the secret key, i.e., about the three polynomials that all together are saved in roughly 38,400bit. By construction of the learning with errors problem (LWE) — the underlying hardness assumption of ring-TESLA— the potential leakage of at most 49bit of the secret key does not immediately translate to the bit-hardness of LWE (resp., the bit-security of ring-TESLA).

7 Conclusion

In this article, we analyzed an implementation of the lattice-based signature scheme ring-TESLA for cache-side-channel vulnerabilities. We identified four routines in the implementation that are vulnerable through cache side channels. Two of these routines, a rejection sampling and a signature validity check, use a secret-dependent number of iterations. One routine is a sparse polynomial multiplication that traverses one polynomial in a secret-dependent order. We modified these functions to ensure a constant number of iterations and secret-independent caching of the polynomial. By modifying these functions, we also eliminated the possibility to exploit the fourth vulnerability. For the modified ring-TESLA implementation, we obtained low upper bounds on the leakage to four attacker models, using program analysis.

Our results show that implementations of rejection sampling and sparse multiplication should be inspected for side channels with particular care. While these techniques are not very common in classical cryptography like RSA, they play a significant role in post-quantum cryptography. Rejection sampling occurs, for instance, in multiple lattice-based signature schemes [6, 7, 15] and in key exchange protocols [35]. Sparse multiplication also occurs in many lattice-based schemes [7,9,15]. Overall, the implementation and analysis of post-quantum cryptography poses additional challenges compared to classical cryptography.

Acknowledgements We thank the anonymous reviewers for their helpful suggestions and Sedat Akleylek for contributing to our modifications of the original ring-TESLA implementation. This work has been partially funded by the DFG as part of projects P1 and E3 within the CRC 1119 CROSSING.

References

[1] A. Abel and J. Reineke. Reverse engineering of cache replacement policies in intel microprocessors and their evaluation. In *ISPASS*, pages 141–142, 2014.

[2] O. Acıiçmez and Ç. K. Koç. Trace-driven cache attacks on aes. Cryptology ePrint Archive, Report 2006/138.

[3] S. Akleylek, N. Bindel, J. Buchmann, J. Krämer, and G. A. Marson. An Efficient Lattice-Based Signature Scheme with Provably Secure Instantiation. In *AFRICACRYPT*, pages 44–60, 2016.

[4] E. Alkim, N. Bindel, J. Buchmann, Ö. Dagdelen, E. Eaton, G. Gutoski, J. Krämer, and F. Pawlega. Revisiting TESLA in the quantum random oracle model. In *PQCrypto*, pages 143–162, 2017.

[5] M. S. Alvim, K. Chatzikokolakis, C. Palamidessi, and G. Smith. Measuring Information Leakage using Generalized Gain Functions. In *CSF*, pages 265–279, 2012.

[6] S. Bai and S. D. Galbraith. An Improved Compression Technique for Signatures Based on Learning with Errors. In *CT-RSA*, pages 28–47, 2014.

[7] P. S. L. M. Barreto, P. Longa, M. Naehrig, J. E. Ricardini, and G. Zanon. Sharper Ring-LWE Signatures. Cryptology ePrint Archive, Report 2016/1026.

[8] D. J. Bernstein. Cache-timing attacks on AES. Technical report, University of Illinois at Chicago, 2005.

[9] J. Buchmann, F. Göpfert, T. Güneysu, T. Oder, and T. Pöppelmann. High-Performance and Lightweight Lattice-Based Public-Key Encryption. In *IoTPTS*, pages 2–9, 2016.

[10] P. Cousot and R. Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *POPL*, pages 238–252, 1977.

[11] U. Degenbaev. *Formal Specification of the x86 Instruction Set Architecture*. PhD thesis, Universität des Saarlandes, 2012.

[12] G. Doychev and B. Köpf. Rigorous Analysis of Software Countermeasures against Cache Attacks. In *PLDI*, pages 406–421, 2017.

[13] G. Doychev, B. Köpf, L. Mauborgne, and J. Reineke. CacheAudit: A Tool for the Static Analysis of Cache Side Channels. *ACM TISSEC*, 18(1):4:1–4:32, 2015.

[14] Goran Doychev. Commit f063813faa548da9bfb11dea9ff6fe39c0f11626: Adding support for CDQ and NEG instructions. `https://github.com/cacheaudit/cacheaudit/commit/f063813faa548da9bfb11dea9ff6fe39c0f11626`, 2016. [Online; accessed 05/23/2017].

[15] L. Ducas, A. Durmus, T. Lepoint, and V. Lyubashevsky. Lattice Signatures and Bimodal Gaussians. In *CRYPTO*, pages 40–56, 2013.

[16] L. Ducas and P. Q. Nguyen. Learning a Zonotope and More: Cryptanalysis of NTRUSign Countermeasures. In *ASIACRYPT*, pages 433–450, 2012.

[17] L. Groot Bruinderink, A. Hülsing, T. Lange, and Y. Yarom. Flush, Gauss, and Reload – A Cache Attack on the BLISS Lattice-Based Signature Scheme. In *CHES*, pages 323–345, 2016.

[18] Intel Corporation. Intel® 64 and IA-32 Architectures Optimization Reference Manual. Order Number: 248966-032, 2016.

[19] B. Köpf and G. Smith. Vulnerability Bounds and Leakage Resilience of Blinded Cryptography under Timing Attacks. In *CSF*, pages 44–56, 2010.

[20] H. Mantel and A. Starostin. Transforming Out Timing Leaks, More or Less. In *ESORICS*, pages 447–467, 2015.

[21] H. Mantel, A. Weber, and B. Köpf. A Systematic Study of Cache Side Channels across AES Implementations. In *ESSoS*, pages 213–230, 2017.

[22] D. Molnar, M. Piotrowski, D. Schultz, and D. Wagner. The Program Counter Security Model: Automatic Detection and Removal of Control-Flow Side Channel Attacks. In *ICISC*, pages 156–168, 2005.

[23] National Institute of Standards and Technology (NIST). Post-quantum project. `https://pqcrypto2016.jp/data/pqc2016_nist_announcement.pdf`, 2016. [Online; accessed 05/23/2017].

[24] P. Q. Nguyen and O. Regev. Learning a Parallelepiped: Cryptanalysis of GGH and NTRU Signatures. In *EUROCRYPT*, pages 271–288, 2006.

[25] T. Oder, T. Schneider, T. Pöppelmann, and T. Güneysu. Practical CCA2-Secure and Masked Ring-LWE Implementation. Cryptology ePrint Archive, Report 2016/1109.

[26] D. A. Osvik, A. Shamir, and E. Tromer. Cache Attacks and Countermeasures: The Case of AES. In *CT-RSA*, pages 1–20, 2006.

[27] P. Pessl. Analyzing the Shuffling Side-Channel Countermeasure for Lattice-Based Signatures. Cryptology ePrint Archive, Report 2017/033.

[28] O. Reparaz, S. S. Roy, F. Vercauteren, and I. Verbauwhede. A masked ring-LWE implementation. Cryptology ePrint Archive, Report 2015/724.

[29] M.-J. O. Saarinen. Arithmetic coding and blinding countermeasures for lattice signatures. Cryptology ePrint Archive, Report 2016/276.

[30] C. E. Shannon. A Mathematical Theory of Communication. *Bell System Technical Journal*, 27(3):379–423, 623–656, 1948.

[31] Peter W. Shor. Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer. *SIAM J. Comput.*, 26(5):1484–1509, 1997.

[32] G. Smith. On the Foundations of Quantitative Information Flow. In *FOSSACS*, pages 288–302, 2009.

[33] Y. Yarom and K. Falkner. FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack. In *USENIX Security*, pages 719–732, 2014.

[34] Y. Yarom, D. Genkin, and N. Heninger. CacheBleed: A Timing Attack on OpenSSL Constant Time RSA. In *CHES*, pages 346–367, 2016.

[35] J. Zhang, Z. Zhang, J. Ding, M. Snook, and Ö. Dagdelen. Authenticated key exchange from ideal lattices. In *EUROCRYPT*, pages 719–751, 2015.