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

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 e ort 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 e ectively.

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 [1]. 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 [2].

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., [3 6]. However, rst approaches to analyze lattice-based cryptography with respect to side-channel attacks are already made, e.g., [7 10]. Recently, Groot Bruinderink et al. presented the rst attack against a lattice-based signature scheme and broke the scheme BLISS [11] using cache side channels of the Gaussian sampling during the signature generation [12]. Although none of the existing lattice-based signature schemes (or their implementations) claim to be secure against side channels, the attack in [12] raises the question how lattice-based signature schemes can be implemented without cache-side-channel

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 [28]. 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 [29]. As the replacement strategy, we fix LRU.[4]

Configuration of ring-TESLA. We set the parameters of the ring-TESLA scheme to `PARAM_N` = n = 512, `PARAM_SIGMA` = = 48, `PARAM_Q` = q = 33550337,

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

Listing 4.1. Code of the subroutine `generate_c`

```
1  void generate_c(uint32_t *pos_list, unsigned char *c_bin){
2    int32_t c[PARAM_N]; int cnt =0; int pos; [...]
3    crypto_stream(r, R_LENGTH, nonce, c_bin);
4
5    for(i=0; i<PARAM_N; i++){ c[i] = 0;}
6    i=0;
7    while(i<PARAM_W){
8      pos = 0;
9      pos = (r[cnt]<<8) | (r[cnt+1]);
10     pos &= PARAM_N-1;
11     cnt += 2;
12     if (c[pos] == 0) { pos_list[i] = pos;  c[pos]=1;  i++; cnt++; } } }
```

Listing 4.2. Implementation of `sample_y`

```
1  //original
2  do {[...] if(val<0x7fffff) mat_y[i++] = val-PARAM_B; [...]} while(i<PARAM_N);
3
4  //adapted
5  for (i = 0; i < PARAM_N; ++i) { mat_y[i] = *(int *)(0x4) - PARAM_B; }
```

`PARAM_B` = B = 4194303, `PARAM_W` = $!$ = 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.[5]

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.

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

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

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 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.[6] 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

We manually analyze the signature generation `crypto_sign` to check if the potential leakage detected by program analysis corresponds to an actual concern. We identify substantiated threats of leakage to cache side channels (CSCs) in the routines `generate_c`, `test_w`, `test_rejection`, and `computeEc`.[7]. The following variables have to be kept secret during the execution of `crypto_sign`: the secret key `sk`, the randomness `vec_y`, the polynomials `vec_v1` and `vec_v2` to compute the hash value, and the polynomials `E1c`, `E2c`, and `Sc`. Furthermore, the hash value `c` and the representation of the corresponding encoded polynomial `pos_list` have to be kept secret until line 19 in Listing 3.1. In line 19, it

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

[7] The analysis of the other subroutines of `crypto_sign` can be found in the corresponding technical report under `http://eprint.iacr.org/2017/951`.

Listing 5.1. 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 5.2. 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; }
```

is decided whether the potential signature (computed in line 18) and `c` are returned and, hence, whether `c` and `pos_list` become public information (via the encoding function F, the values can be computed from each other), or whether 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 below.

Analysis of the subroutine `compute_Ec`*.* The implementation of the subroutine `compute_Ec` is given in Listing 5.1. The values that have to be kept secret during this computation are `sk`, `e`, `pos_list`, and `pos`. Most loops and branchings do not depend on any of the secret values. 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. 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.

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

Analysis of the subroutine `test_rejection`. The implementation of the subroutine `test_rejection` is given in Listing 5.2. The variable `poly_z` in Listing 5.2 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. This 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 5.2, is never true, then the value `0` is returned and the attacker gets a sequence of `PARAM_N` (or $2 \cdot$`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 5.2 holds true for some `i`$\in \{0; \ldots;$`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 $c = F(H([v_1]_M; [v_2]_M; \ ))$ 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 `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 [30, 31].

Analysis of the subroutine `test_w`. The implementation of the routine `test_w` is given in Listing 5.3. The values `poly_w`, `val`, and `left` in Listing 5.3 have to be kept secret. 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 5.3. 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`.

Analysis of the subroutine `generate_c`. The implementation of the subroutine `generate_c` is given in Listing 4.1. The values `pos_list`, `c`, and `pos` in Listing 4.1 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 [30, 31].

[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 5.3. Code of the subroutine `test_w`

```
static int test_w(poly poly_w)
{ int i; int64_t left, right, val;
  for(i=0; i<PARAM_N; i++){
    val = (int64_t) poly_w[i];
    val = val % PARAM_Q;
    if (val < 0){ val = val + PARAM_Q;}
   left = val;
   left = left % (1<<(PARAM_D));
   left -= (1<<PARAM_D)/2;
   left++;
   right = (1<<(PARAM_D-1))-PARAM_REJECTION;
   if (abs(left) > right){ return -1; } }
  return 0; }
```

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 routines `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 3. There are, indeed, non-zero bounds for all three unmitigated routines.

	Unmitigated routines				Mitigated routines			
	acc	*accd*	*trace*	*time*	*acc*	*accd*	*trace*	*time*
`test_w`	31	31	**49152**	19.3	0	0	0	0
`test_rejection`	**31**	**31**	10.1	10.1	0	0	0	0
`computeEc`	0	0	**20**	5.9	0	0	**19**	4.4
`crypto_sign`	12.9	2.6	**51.6**	9.5	8.1	1.6	**48.6**	9.0

Table 3. Leakage bounds [bit]

```
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]; } } }
```

We adapt the routines, as shown in the above listing, 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 [32] - a program transformation to mitigate timing side channels, which performs rather well in practical evaluation [33]. 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.

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 3. 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 [16, 17], 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 3 lists also 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 even lower than the reported leakage bounds. We expect in particular the leakage bounds for *trace* and *time* to be quite 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 [11, 15, 19] and in key exchange protocols [34]. Sparse multiplication also occurs in many lattice-based schemes [11, 19, 35]. 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. Shor, P.W.: Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer. SIAM J. Comput. **26**(5) (1997) 1484–1509
2. National Institute of Standards and Technology (NIST): Post-quantum project. `https://pqcrypto2016.jp/data/pqc2016_nist_announcement.pdf` (2016) [Online; accessed 05/23/2017].
3. Yarom, Y., Genkin, D., Heninger, N.: CacheBleed: A Timing Attack on OpenSSL Constant Time RSA. In: CHES. (2016) 346–367
4. Acıiçmez, O., Ç. K. Koç: Trace-driven cache attacks on AES. Cryptology ePrint Archive, Report 2006/138
5. Osvik, D.A., Shamir, A., Tromer, E.: Cache Attacks and Countermeasures: The Case of AES. In: CT-RSA. (2006) 1–20
6. Bernstein, D.J.: Cache-timing attacks on AES. Technical report, University of Illinois at Chicago (2005)
7. Oder, T., Schneider, T., Pöppelmann, T., Güneysu, T.: Practical CCA2-Secure and Masked Ring-LWE Implementation. Cryptology ePrint Archive, Report 2016/1109
8. Pessl, P.: Analyzing the Shu□ing Side-Channel Countermeasure for Lattice-Based Signatures. Cryptology ePrint Archive, Report 2017/033
9. Saarinen, M.J.O.: Arithmetic coding and blinding countermeasures for lattice signatures. Cryptology ePrint Archive, Report 2016/276
10. Reparaz, O., Roy, S.S., Vercauteren, F., Verbauwhede, I.: A masked ring-LWE implementation. Cryptology ePrint Archive, Report 2015/724
11. Ducas, L., Durmus, A., Lepoint, T., Lyubashevsky, V.: Lattice Signatures and Bimodal Gaussians. In: CRYPTO. (2013) 40–56
12. Groot Bruinderink, L., Hülsing, A., Lange, T., Yarom, Y.: Flush, Gauss, and Reload – A Cache Attack on the BLISS Lattice-Based Signature Scheme. In: CHES. (2016) 323–345
13. Akleylek, S., Bindel, N., Buchmann, J., Krämer, J., Marson, G.A.: An E□-cient Lattice-Based Signature Scheme with Provably Secure Instantiation. In: AFRICACRYPT. (2016) 44–60

14. Alkim, E., Bindel, N., Buchmann, J., Dagdelen, Ö., Eaton, E., Gutoski, G., Krämer, J., Pawlega, F.: Revisiting TESLA in the quantum random oracle model. In: PQCrypto. (2017) 143–162
15. Bai, S., Galbraith, S.D.: An Improved Compression Technique for Signatures Based on Learning with Errors. In: CT-RSA. (2014) 28–47
16. Doychev, G., Köpf, B., Mauborgne, L., Reineke, J.: CacheAudit: A Tool for the Static Analysis of Cache Side Channels. ACM TISSEC **18**(1) (2015) 4:1–4:32
17. Mantel, H., Weber, A., Köpf, B.: A Systematic Study of Cache Side Channels across AES Implementations. In: ESSoS. (2017) 213–230
18. Doychev, G., Köpf, B.: Rigorous Analysis of Software Countermeasures against Cache Attacks. In: PLDI. (2017) 406–421
19. Barreto, P.S.L.M., Longa, P., Naehrig, M., Ricardini, J.E., Zanon, G.: Sharper Ring-LWE Signatures. Cryptology ePrint Archive, Report 2016/1026
20. Abel, A., Reineke, J.: Reverse engineering of cache replacement policies in Intel microprocessors and their evaluation. In: ISPASS. (2014) 141–142
21. Smith, G.: On the Foundations of Quantitative Information Flow. In: FOSSACS. (2009) 288–302
22. Shannon, C.E.: A Mathematical Theory of Communication. Bell System Technical Journal **27**(3) (1948) 379–423, 623–656
23. Köpf, B., Smith, G.: Vulnerability Bounds and Leakage Resilience of Blinded Cryptography under Timing Attacks. In: CSF. (2010) 44–56
24. Alvim, M.S., Chatzikokolakis, K., Palamidessi, C., Smith, G.: Measuring Information Leakage using Generalized Gain Functions. In: CSF. (2012) 265–279
25. Yarom, Y., Falkner, K.: FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack. In: USENIX Security. (2014) 719–732
26. Cousot, P., Cousot, R.: Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In: POPL. (1977) 238–252
27. Doychev, G.: Commit f063813faa548da9bfb11dea9□6fe39c0f11626: Adding support for CDQ and NEG instructions. `https://github.com/cacheaudit/cacheaudit/commit/f063813faa548da9bfb11dea9ff6fe39c0f11626` (2016) [Online; accessed 05/23/2017].
28. Degenbaev, U.: Formal Specification of the x86 Instruction Set Architecture. PhD thesis, Universität des Saarlandes (2012)
29. Intel Corporation: Intel® 64 and IA-32 Architectures Optimization Reference Manual. Order Number: 248966-032 (2016)
30. Ducas, L., Nguyen, P.Q.: Learning a Zonotope and More: Cryptanalysis of NTRUSign Countermeasures. In: ASIACRYPT. (2012) 433–450
31. Nguyen, P.Q., Regev, O.: Learning a Parallelepiped: Cryptanalysis of GGH and NTRU Signatures. In: EUROCRYPT. (2006) 271–288
32. Molnar, D., Piotrowski, M., Schultz, D., Wagner, D.: The Program Counter Security Model: Automatic Detection and Removal of Control-Flow Side Channel Attacks. In: ICISC. (2005) 156–168
33. Mantel, H., Starostin, A.: Transforming Out Timing Leaks, More or Less. In: ESORICS. (2015) 447–467
34. Zhang, J., Zhang, Z., Ding, J., Snook, M., Dagdelen, Ö.: Authenticated key exchange from ideal lattices. In: EUROCRYPT. (2015) 719–751
35. Buchmann, J., Göpfert, F., Güneysu, T., Oder, T., Pöppelmann, T.: High-Performance and Lightweight Lattice-Based Public-Key Encryption. In: IoTPTS. (2016) 2–9