Special Session: Hampering fault attacks against lattice-based signature schemes—countermeasures and their efficiency

Nina Bindel
Technische Universität Darmstadt
Germany
nbindel@cdc.tu-darmstadt.de

Juliane Krämer
Technische Universität Darmstadt
Germany
jakraemer@cdc.tu-darmstadt.de

Johannes Schreiber
Technische Universität Darmstadt
Germany

ABSTRACT
Research on physical attacks on lattice-based cryptography has seen some progress in recent years and first attacks and countermeasures have been described. In this work, we perform an exhaustive literature review on fault attacks on lattice-based encryption and signature schemes. Based on this, we provide a complete overview of suggested countermeasures and analyze which of the proposed attacks can prevented by respective countermeasures. Moreover, we show for selected countermeasures how they affect the runtime of the protected operations.

KEYWORDS
lattice-based cryptography, fault attacks, side channel analysis

1 INTRODUCTION
Due to their high efficiency and their strong security properties, lattice-based primitives are a very promising post-quantum candidate to replace currently used public key cryptography. While the security of lattice-based schemes has been deeply analyzed mathematically, only little effort has been spent on the analysis with respect to implementation attacks. First results about powerful attacks exist, but there is no comprehensive work on countermeasures that helps software developers of lattice-based primitives to secure the implementations against physical attacks.

In this work, we provide such an overview for active physical attacks, i.e., fault attacks. Based on all publications on fault attacks on lattice-based signature [2, 5, 9] and encryption schemes [8, 10, 11], we provide a complete overview of suggested countermeasures1 for first-order fault attacks. Thus, we assume that an attacker can only induce a single fault, and he cannot circumvent a cm with an additional fault. We analyze which fault attacks can be prevented by the found countermeasures and which signature scheme can be secured with them. A summary of the proposed countermeasures and the attacks they can prevent is given in Table 1. Moreover, we describe for selected countermeasures the impact that they have on the runtime of the protected operations. To this end we implement four countermeasures of the lattice-based signature scheme ring-TELSA [1].

2 PRELIMINARIES
For \( q \in \mathbb{N} \) a prime, we denote by \( \mathbb{Z}_q \) the finite field \( \mathbb{Z}/q\mathbb{Z} \). Furthermore, we define the rings \( R = \mathbb{Z}[x]/(x^n + 1) \) and \( R_q = \mathbb{Z}_q[x]/(x^n + 1) \). For a finite set \( S \), we write \( s \leftarrow S \) to indicate that an element \( s \) is sampled uniformly at random from \( S \).

For the description of the lattice-based signature schemes GLP [6], BLISS [3], PassSign [7], and the NTRU-like instantiation of GPV [4], we refer to the original works or to [2, 5]. We briefly describe the scheme ring-TELSA [1] since we explain countermeasures for this scheme in Section 4. The secret key \( sk \) is a tuple of polynomials \( s, e_1, e_2 \) with small coefficients; the public key \( vk \) consists of \( a_1, a_2 \leftarrow R_q, b_1 = a_1s + e_1 \) (mod \( q \)), and \( b_2 = a_2s + e_2 \) (mod \( q \)).

The signature of a message \( \mu \) is \((z, c')\) with \( z = y + sc, y \) a random vector, \( c \) an encoding polynomial of the hash value \( c' \) that depends on the most significant bits of \( ay, zc, y \) and \( \mu \). Before returning the signature, rejection sampling is applied. For verification of the signature \((c', z)\), the size of \( z \) and the equality of \( c' \) and the hash of \( \mu \) and the most significant bits of \( a_1z - b_1c \) is checked.

3 OVERVIEW OF COUNTERMEASURES
We provide an exhaustive overview of all countermeasures (cm) from the literature about fault attacks on lattice-based signature and encryption schemes and summarize our findings in Table 1. In the table we consider the following attacks during key generation (KG), sign (S), and verify (V): (1) Randomization of the secret (KG), [2, Sec. III-A]; (2) Zeroing of secret (KG), [2, Sec. V-A]; (3) Skip of addition (KG), [2, Sec. IV-A2]; (4) Skip of addition (S), [2, Sec. IV-B2]; (5) Zeroing of randomness (S), [2, Sec. V-B]; (6) Zeroing of hash polynomial (V), [2, Sec. V-D]; (7) Skip of addition (KG), [2, Sec. IV-A2]; (8) Skip of rejection sampling, [2, Sec. IV-B1]; (9) Skip of correctness check (V), [2, Sec. IV-C1]; (10) Skip of size check (V), [2, Sec. IV-C2]; (11) Zeroing of hash value (S), [2, Sec. V-C]; (12) Loop-abort against Fiat-Shamir signatures (S), [5, Sec. 3]; (13) Loop-abort against Hash-and-Sign signatures (S), [5, Sec. 4].

We emphasize that all countermeasures have to be implemented carefully to ensure that they work as expected. While this is true

---

1To the best of our knowledge, no attacks on lattice-based key exchange protocols have been published so far.
in general, it is especially important for those cms where a constraint on the effectiveness is shown in Table 1. We believe that these constraints are necessary for the respective cm to be effective. For instance, the cm loop-counter, which we think can be transferred to several attacks, requires that the vulnerable operation is implemented with a for loop. Regarding the attack skip of addition, this means that the addition of polynomials has to be processed coefficient by coefficient, with a for loop indexing the coefficients. We use the following constraints in the table: (15) attacks prevented in at least 50% of the times; (16) if addition is implemented as a for loop; (17) if check is implemented as a for loop; (18) if zeroing fault is an early abort of a for loop.

sk-correctness: To evaluate the correctness of the secret key during key generation is to use correctness checks or comparisons [2].

dummy-value: To ensure that the correct secret key is used during sign, a "dummy" value is computed and multiplied with the values that should be returned. If no fault occurs, the dummy value is equal to one, otherwise it randomizes the signature. In [2] this is described for the GLP scheme. A similar idea for a different scheme is also mentioned in [9].

new-variable: A way to circumvent skipping faults addressing the addition or multiplication is to define a new variable to save the resulting sum [2].

pk-correctness: To prevent returning a leaking public key, advanced correctness checks should be implemented. dummy-value: To ensure that the correct secret key is used during sign, a "dummy" value is computed and multiplied with the values that should be returned. If no fault occurs, the dummy value is equal to one, otherwise it randomizes the signature. In [2] this is described for the GLP scheme. A similar idea for a different scheme is also mentioned in [9].

Table 1: Overview on countermeasures (cm); in column “prevented attacks” ✭ denotes that we find that the respective cm also prevents the corresponding attack, otherwise cm was originally proposed; in column “schemes vulnerable to attacks”, ● denotes that the attack was originally proposed against the respective scheme.

<table>
<thead>
<tr>
<th>Countermeasure</th>
<th>Prevented attacks</th>
<th>Schemes vulnerable to attacks</th>
<th>Constraints</th>
</tr>
</thead>
<tbody>
<tr>
<td>sk-correctness</td>
<td>(1) (2)</td>
<td>GLP BLISS ring- Pass- GPV- TESLA Sign NTRU</td>
<td></td>
</tr>
<tr>
<td>pk-correctness</td>
<td>(3)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>dummy-value</td>
<td>(1)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>new-variable</td>
<td>(3) (4) (5)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>add-secret-to-random</td>
<td>(7)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>if-true</td>
<td>(9) (10) (11)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>loop-counter</td>
<td>(13)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>zero-counting</td>
<td>(2) (5) (6)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>redund-comp.</td>
<td>(9)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>verify-in-sign</td>
<td>(4)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>parity-bits</td>
<td>(1) (2) (3)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>error-det.-codes</td>
<td>(8)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>add-rej.-sampling</td>
<td>(9)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>return-comparison</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The Efficiency of Countermeasures

We implemented the countermeasure (cm) new-variable to prevent skipping the addition of the error during the key generation. In Listing 1, we show the original, vulnerable key generation algorithm of ring-TESLA next to our implemented cm. To protect against zeroing attacks against the randomness poly vec_y in the signature generation of ring-TESLA, we implement the cm...
Hampering fault attacks against LBSS  
CODES/ISSS ’17 Companion, October 15–20, 2017, Seoul, Republic of Korea

LISTING 1: Parts of the key generation algorithm of ring-TESLA with and without countermeasures against skipping the addition of the error polynomial.

Listing 2: Sampling during signature generation with check if zeroing fault was introduced and implementation of count_zeros() .

Listing 3: Changed subroutines compress_sig() and implementation of fmodb_u() .

zero-counting, see Listing 2. Because vec_y has 512 integer coefficients, sampled uniformly random over [−2097151, 2097151], the probability of a polynomial having more than eight zero coefficients is less than 2−128. Hence, we consider the change in the probability distribution to be negligible. We implement the cm additional-rejection-sampling within the compression of the signature. The function fmodb_u() was added to the original code. We show the modification of compress_sig() and implementation of fmodb_u() in Listing 3. During the verification algorithm of ring-TESLA, the input value c_sig and the computed hash polynomial c are compared. If they are equal, memcmp() returns 0, otherwise it returns a positive integer (which results in rejected the signature). An attacker who wants his victim to accept an invalid signature, must force the verification algorithm to return 0, which signifies a valid signature. This can be achieved by a skipping attack. Considering only first-order attacks, we can prevent the skipping attack by implementing the cm return-comparison as in Listing 4. The function still returns 0 if the signature is valid and, by definition of memcmp(), a nonzero value otherwise (albeit not -1).

We benchmarked the implementations of ring-TESLA with and without countermeasures. The benchmarks were recorded on a 4.0 GHz Intel Core i7-6700k. For compilation we used GCC 7.1.0. The benchmarks are given in clock cycles and are averaged over 2,000 runs for the key generation and 20,000 runs for signature generation and verification. Since compiler optimizations might remove some of the countermeasures, we report two benchmarks – with and without the optimization −Ofast – in Table 2.

ACKNOWLEDGMENTS

This work has been supported by the the German Research Foundation (DFG) as part of project P1 within the CRC 1119 CROSSING.

REFERENCES


