RUHR-UNIVERSITÄT BOCHUM

Side-Channel Attacks On Implementations Of Lattice-Based Cryptosystems

Julian Speith, Felix Haarmann

Seminarausarbeitung
June 26, 2016
Embedded Security Group - Prof. Dr.-Ing. Christof Paar

Abstract

Due to the progress in the developement of quantum computers, the need for pratical post-quantum cryptography is getting increrasingly stronger. Lattice-based cryptography is just one of a few candidates, that could replace the public-key algortithms used these days. However, it is the most promising one.

Though lattice-based cryptography has only came up in recent years, a lot of research has been done in the area. Most papers in recent years have focused on efficient implementations for some of the proposed lattice-based algorithms, without considering the possibility of side-channel attacks. Therefore, we summarize a few proposed side-channel attacks and countermeasures and evaluate their practicability and efficiency.

In that context, we present two masking schemes and their implementation for a ring-LWE encryption scheme. While just one of whom is sound against all first-order DPA attacks, it lacks in efficiency compared to the other one. The other one is much more efficient, but might still be vulnerable to some refined first-order DPA.

Furthermore, we describe a cache attack on the Gaussian sampling of the Bimodal Lattice Signature Scheme (BLISS).

Finally, two blinding techniques are shown, which can be used for any arbitrary algorithm, but are not proven to be secure against side-channel attacks. However, they do not add too many computations to the existing algorithms, meaning there will not be a big loss in terms of efficiency.

Contents

Acronyms

BLISS Bimodal Lattice Signature Scheme

DPA Differential Power Analysis

HO-DPA Higher Order Differential Power Analysis

LPR Lyubashevsky-Peikert-Regev

NTT Number Theoretic Transform

PRNG Pseudo Random Number Generator

ring-LWE Ring Learning with Errors Problem

1. Introduction

Despite the rapid progress in the development of quantum computers and the hereby increasingly urgent need for post-quantum cryptographic algorithms, no such algorithms has yet been standardized [CJL+16]. Current public-key cryptosystems like RSA, DHKE or even elliptic curve cryptography could easily be broken by a quantum computer, due to Shor's algorithm for prime factorization and discrete logarithms [Sho97]. As most of today's digital infrastructure depends (at least partially) on such public-key algorithms, the need for efficient and secure cryptography that can withstand the power of quantum computation is as high as never before.

Lattice-based cryptography is the most promising of all attempts in post-quantum cryptography, as its underlying mathematics are already well understood and reasonably efficient implementations of some of the proposed cryptographic schemes are available today. Our paper will give an overview over some selected lattice-based algorithms and their implementation in respect to their resistance to various side-channel attack techniques.

1.1. Related Work

This paper summarizes the content of several other papers, that have been published in recent years. Some of them are referred to below and we strongly recommend to take a look at them.

Shortly after the ring-LWE problem was introduced in [LPR12] in 2012, the authors of [RRVV15] laid the groundwork for masked implementations of one ring-LWE encryption scheme and refined it in [RdCR+16] by getting rid of the need for a masked decoder. Just a year after the ring-LWE encryption scheme was introduced, the authors of [DDLL13] proposed the BLISS signature scheme, which is as well based on the ring-LWE problem. A possible side-channel attack on a slightly altered version of that signature scheme was then shown in [BHLY16] in 2016, which might be prevented by the blinding techniques used by the authors of [Saa16].

1.2. Structure of this Paper

In Section 2 we will start with an explanation of our notation and give an overview over the mathematic background needed to understand this paper. This includes introducing the reader to the concept of *(ideal) lattices*, the *Ring Learning with Errors Problem (ring-LWE)*, *Discrete Gaussian Distributions*, a *ring-LWE Encryption Scheme* and the *BLISS Signature Scheme*. Additionally, we will give a short explanation of the side-channel attack terminology used throughout this paper.

Section 3 will deal with the ring-LWE encryption scheme and will be split into two parts, starting with the description of a masked implementation of the decryption function, including a masked decoder build upon a masked table lookup. The second part of this section will be an evaluation of the proposed implementation in respect to its soundness to first- and second-order side-channel attacks.

A different approach to masking of the ring-LWE encryption scheme will be presented in Section 4 of our paper, which will as well be split into a description of the proposed scheme and an evaluation. Furthermore, the second masking scheme will be compared to the first one in respect to efficiency and complexity.

Section 5 will discuss the `FLUSH+RELOAD` cache attacks on the Gaussian sampler used in the BLISS signature scheme. This part will start with a description of a perfect side channel attack on two Gaussian sampling algorithms, namely the cumulative distribution function (CDT sampling) and rejection sampling. This will be followed by an evaluation of the `FLUSH+RELOAD` attacks on an actual BLISS implementation, while running on modern CPUs.

Furthermore, in Section 6 we will be presenting two measures used for blinding polynomial multiplication and Gaussian sampling, which might help against the attacks described in Section 5.

Finally, Section 7 will summarize the content of our paper shortly and some conclusions will be drawn.

2. Theoretical Background

2.1. Notation

As we will only work with ideal lattices in our paper, all operations will be done within the ring $R_q = \mathbb{Z}_q[x]/(f(x))$ with $f(x)$ being an irreducible polynomial of degree n and all coefficients being reduced modulo q.

Polynomials will be written as bold lower case letters like $\mathbf{f}$. As we will use the *Number Theoretic Transform (NTT)* for efficient polynomial multiplication within a ring R_q, polynomials in the NTT domain will be written as $\tilde{\mathbf{f}}$. Vectors will be denoted with an arrow on top of them ($\vec{x}$), while matrices will be denoted by bold upper case letters ($\mathbf{A}$). The entries of a vector $\vec{x}$ will be called x_i, with i specifying the position within the vector starting at 0.

The notation for the l_p norm of a vector $\vec{x}$ will be $\|\vec{x}\|_p$, only with the exception of the l_2 norm, which will be referred to as $\|\vec{x}\| = \sqrt{\sum_i x_i^2}$.

2.2. Ideal Lattices

A lattice Λ is discrete subgroup of $\mathbb{R}^n$ that is defined as a set of $m \leq n$ linearly independent vectors $\vec{b}_1, ..., \vec{b}_m \in \mathbb{R}^n$ and is generated by all linear combinations of those $\vec{b}_i$'s with integer coefficients:

$$\Lambda(\vec{b}_1, ..., \vec{b}_m) = \left\{ \sum_{i=1}^{m} x_i \vec{b}_i \;\middle|\; x_i \in \mathbb{Z} \right\} \tag{2.1}$$

The set $\{\vec{b}_1, ..., \vec{b}_m\}$ of those vectors is called the basis of that lattice, which commonly is represented by a matrix $\mathbf{B} = (\vec{b}_1, ..., \vec{b}_m)$.

Furthermore, an ideal lattice is is lattice that corresponds to ideals in a ring R_q. From this it follows that we can deal with polynomials instead of matrices, which makes arithmetics used for cryptographic applications much more efficient. In our paper we will confine ourselves to those ideal lattices, as most of the current work in that area focuses around them. For more on the topic of ideal lattices, see [LPR12].

2.3. Ring Learning with Errors Problem

The Ring Learning with Errors Problem (ring-LWE) has been introduced in [LPR12] in 2012. There are two instances of the ring-LWE, a search one and a decisional one. For both of them, we can state that there is a given $\mathbf{s} \in R_q$. Next, an error polynomial $\mathbf{e}_i \in R_q$ is sampled from the discrete Gaussian distribution $\mathcal{N}_{\mathbb{Z}}^n(0, \sigma^2)$ and another polynomial $\mathbf{a}_i \in R_q$ is sampled uniformy at random. Finally, a polynomial $\mathbf{b}_i$ is computed as $\mathbf{b}_i = (\mathbf{a}_i \cdot \mathbf{s}) + \mathbf{e}_i$.

For the *search* ring-LWE problem, one needs to find the unknown polynomial $\mathbf{s}$ for a given pair $(\mathbf{a}_i, \mathbf{b}_i)$. The *decisional* ring-LWE problem is about distinguishing between pairs $(\mathbf{a}_i, \mathbf{b}_i)$ that have actually been computed the way described before and pairs $(\mathbf{a}_i, \mathbf{b}_i)$ that have been sampled uniformly at random in R_q.

2.4. Discrete Gaussian Distribution

The discrete Gaussian distribution with mean μ and standard deviation σ is denoted as $\mathcal{N}_{\mathbb{Z}}(\mu, \sigma^2)$. In this paper we will focus on zero-centered distributions $\mathcal{N}_{\mathbb{Z}}(0, \sigma^2)$ with a density function $\rho_\sigma(x)$ given by:

$$\rho_\sigma(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{x^2}{2\sigma^2}} \tag{2.2}$$

The probability function of discrete Gaussian distribution over $\mathbb{Z}$ is then defined as $D_\sigma(x) = \rho_\sigma(x)/\rho_\sigma(\mathbb{Z})$ with $\rho_\sigma(\mathbb{Z}) = \sum_{y=-\infty}^{\infty} \rho_\sigma(y)$. As we will sample entire vectors most of the time, we denote the discrete Gaussian distribution over $\mathbb{Z}^m$ as $\mathcal{N}_{\mathbb{Z}}^n(\mu, \sigma^2)$ and its probability function as $D_\sigma^m(\vec{x}) = \rho_\sigma(\vec{x})/\rho_\sigma(\mathbb{Z})^m$ with $\rho_\sigma(\vec{x})$ being defined as follows:

$$\rho_\sigma(\vec{x}) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{\|\mathbf{x}\|^2}{2\sigma^2}} \tag{2.3}$$

2.5. Cryptographic Algorithms

This Section will give a short overview of two cryptographic algorithms, that are based on ideal lattices and the ring-LWE problem. While the first algorithm can exclusively be used for encryption, the second one is a signature algorithm. For more information on the mathematical background of those algorithms, we would like to refer you to the cited papers.

2.5.1. Ring-LWE Encryption Scheme

In our paper we focus on the encryption scheme published in [LPR12], which will be refered to as *Lyubashevsky-Peikert-Regev (LPR)*. This scheme consists of three main operations: *Key Generation*, *Encryption* and *Decryption*. The globally known parameters of this scheme are (n, q, σ) and a polynomial $\mathbf{g}$. The dimension of the polynomial ring R_q is defined by n, while q is the modulus. The standard deviation of the discrete Gaussian distribution is given by σ.

Key Generation: In this step, the coefficients of the two polynomials $\mathbf{r}$ and $\mathbf{s}$ are sampled according to the discrete Gaussian distribution $\mathcal{N}_{\mathbb{Z}}^n(0, \sigma^2)$. Then the public key $\mathbf{p}$ is computed by $\mathbf{p} = \mathbf{r} - \mathbf{g} \cdot \mathbf{s}$. The resulting output is a key pair $(\mathbf{p}, \mathbf{s})$ with $\mathbf{p}$ being the public key and $\mathbf{s}$ being the secret key.

Encryption: The encryption phase takes a n-bit message $\mathbf{m}$ and the public key $\mathbf{p}$ as input. Initially, the message $\mathbf{m}$ is encoded as an element of the ring R_q by multiplying each of its bits by $q/2$ and is then denoted by $\mathbf{m}_{enc}$. In the following step, three error polynomials $\mathbf{e}_1$, $\mathbf{e}_2$ and $\mathbf{e}_3$ are sampled according to $\mathcal{N}_{\mathbb{Z}}^n(0, \sigma^2)$ and will be used as noise. The ciphertext then consists of two parts $(\mathbf{c}_1, \mathbf{c}_2)$, with $\mathbf{c}_1 = \mathbf{g} \cdot \mathbf{e}_1 + \mathbf{e}_2$ and $\mathbf{c}_2 = \mathbf{p} \cdot \mathbf{e}_1 + \mathbf{e}_3 + \mathbf{m}_{enc}$. The encryption algorithm returns the ciphertext $(\mathbf{c}_1, \mathbf{c}_2)$.

Decryption: For decryption, we start by computing $\mathbf{m}_{enc} = \mathbf{c}_1 \cdot \mathbf{s} + \mathbf{c}_2$. To decode $\mathbf{m}_{enc}$, we need a function $\textsc{Decode}(m_{enc,i})$ with $m_{enc,i}$ being an element of $\mathbf{m}_{enc}$. One possible function is given below:

$$\textsc{Decode}(x) = \begin{cases} 0, & \text{if } x \in (0, q/4) \cup (3q/4, q) \\ 1, & \text{if } x \in (q/4, 3q/4) \end{cases} \tag{2.4}$$

2.5.2. BLISS

The Bimodal Lattice Signature Scheme (BLISS) is a signature algorithm based on lattice cryptography, that was proposed in [DDLL13]. We will just povide the basic algortihms of the signature scheme, for the security proof and the motivation of the construction we would refer you to the original paper.

BLISS uses the following parameters: dimension n, modulus q and standard deviation σ. Furthermore, is makes use of a cryptographic hash function H, which outputs a n bit binary vector with weight κ. The density of the polynomials belonging to the secret key $\mathbf{S}$ is determined by the parameters d_1 and d_2 and d determines the length of the second signature component $\mathbf{z}_2^{\dagger}$.

Key Generation: The keys generated by Algorithm 2.5.2 are correct due to the following equation:

$$\mathbf{a}_1 \cdot \mathbf{s}_1 + \mathbf{a}_2 \cdot \mathbf{s}_2 = 2\mathbf{a}_q \cdot \mathbf{f} + (q-2) \cdot (2\mathbf{g}+1) \equiv 2(2\mathbf{g}+1) + (q-2)(2\mathbf{g}+1) \equiv q \; mod \; 2q \tag{2.5}$$

An attacker might validate a candidate for $\mathbf{s}_1 = \mathbf{f}$ by verifying the distributions of $\mathbf{f}$, $\mathbf{a}_q \cdot \mathbf{f} \equiv 2\mathbf{g} + 1 \bmod 2q$ and $\mathbf{a}_1 \cdot \mathbf{f} + \mathbf{a}_2 \cdot (\mathbf{a}_q \cdot \mathbf{f}) \equiv q \bmod 2q$.

Algorithm 1 BLISS Key Generation

Output: BLISS key pair $(\mathbf{A}, \mathbf{S})$ with public key $\mathbf{A} = (\mathbf{a}_1, \mathbf{a}_2) \in R_{2q}^2$ and secret key $\mathbf{S} = (\mathbf{s}_1, \mathbf{s}_2) \in R_{2q}^2$, such that $\mathbf{AS} = \mathbf{a}_1 \cdot \mathbf{s}_1 + \mathbf{a}_2 \cdot \mathbf{s}_2 \equiv q \bmod 2q$

1: Choose $\mathbf{f}, \mathbf{g} \in R_{2q}$ uniformly at random with exactly d_1 entries in $\{\pm 1\}$ and d_1 entries in $\{\pm 2\}$
2: $\mathbf{S} = (\mathbf{s}_1, \mathbf{s}_2) = (\mathbf{f}, 2\mathbf{g} + 1)$
3: **if** $\mathbf{f}$ violates certain conditions (see [DDLL13]) **then**
4: Restart
5: **end if**
6: $\mathbf{a}_q = (2\mathbf{g} + 1)/\mathbf{f} \bmod q$ (restart if $\mathbf{f}$ is not invertible)
7: **return** $(\mathbf{A}, \mathbf{S})$ with $\mathbf{A} = (2\mathbf{a}_q, q - 2) \bmod 2q$

Signature Generation: The signature algorithm given in Algo. 2.5.2 uses the parameters $p = \lfloor 2q/2^d \rfloor$ (the d highest order bits of $2q$) and a constant $\zeta = (q-2)^{-1} \bmod 2q$. In the following, the d highest order bits of a value x will be denoted as $\lfloor x \rceil_d$.

Algorithm 2 BLISS Signature Algorithm

Input: Message μ, public key $\mathbf{A} = (\mathbf{a}_1, q - 2)$, secret key $\mathbf{S} = (\mathbf{s}_1, \mathbf{s}_2)$
Output: Signature $(\mathbf{z}_1, \mathbf{z}_2^\dagger, \mathbf{c}) \in \mathbb{Z}_{2q}^n \times \mathbb{Z}_p^n \times \{0,1\}^n$

1: $\mathbf{y}_1, \mathbf{y}_2 \leftarrow \mathcal{N}_{\mathbb{Z}}^n(0, \sigma^2)$
2: $\mathbf{u} = \zeta \cdot \mathbf{a}_1 \cdot \mathbf{y}_1 + \mathbf{y}_2 \bmod 2q$
3: $\mathbf{c} = H(\lfloor \mathbf{u} \rceil_d) \bmod p$, μ
4: Choose a random bit b
5: $\mathbf{z}_1 = \mathbf{y}_1 + (-1)^b \mathbf{s}_1 \cdot \mathbf{c} \bmod 2q$
6: $\mathbf{z}_2 = \mathbf{y}_2 + (-1)^b \mathbf{s}_2 \cdot \mathbf{c} \bmod 2q$
7: Continue with a probability based on σ, $\|\mathbf{Sc}\|$, $\langle \mathbf{z}, \mathbf{Sc} \rangle$ (see [DDLL13]), else restart
8: $\mathbf{z}_2^\dagger = (\lfloor \mathbf{u} \rceil_d - \lfloor \mathbf{u} - \mathbf{z}_2 \rceil_d) \bmod p$
9: **return** $(\mathbf{z}_1, \mathbf{z}_2^\dagger, \mathbf{c})$

Signature Verification: Algorithm 2.5.2 shows the signature verification. We want to stress that reductions modulo $2q$ are done before reductions modulo p.

2.6. Side-Channel Attack Terminology

Side-channel attacks use leaked information from physical implementations of cryptographic algorithms to conclude secret information like encryption keys.

Algorithm 3 BLISS Verification Algorithm

Input: Message μ, public key $\mathbf{A} = (\mathbf{a}_1, q-2)$, signature $(\mathbf{z}_1, \mathbf{z}_2^\dagger, \mathbf{c})$

1: **if** $\mathbf{z}_1, \mathbf{z}_2^\dagger$ violate certain conditions (see [DDLL13]) **then**
2: Reject
3: **end if**
4: **if** $\mathbf{c} = H(\lfloor \zeta \cdot \mathbf{a}_1 \cdot \mathbf{z}_1 + \zeta \cdot q \cdot \mathbf{c} \rceil_d + \mathbf{z}_2^\dagger \bmod p,\ \mu)$ **then**
5: Accept
6: **end if**

Such leakage could e.g. be the power consumption during cryptographic operations. One type of attacks that makes use of leakages through power consumption is *Differential Power Analysis (DPA)*.

To perform a DPA, one needs to correlate a leakage with a prediction made on a special value that depends on both, the secret key and the plaintext. Such a value is called a *sensitive variable*. A common countermeasure to *DPA* is masking, where sensitives variables are randomly split into d shares. A masking approach with d shares is reffered to as a $(d-1)$-th order masking, as $d-1$ shares are picked randomly and the last share is computed in a way, that the combination of all d shares equals the shared sensitive variable. To defeat this kind of countermeasures, the class of *Higher Order Differential Power Analysis (HO-DPA)* has been introduced. To overcome $(d-1)$-th order masking, d-th order *DPA* is needed, which can be done by combining the leakage of d different signals that correspond to the d shares of the sensitive variable. Theoretically, higher order masking can always be defeated by HO-DPA. However, the difficulty of *HO-DPA* is growing exponentially due to noise effects. In practice, first order masking is most commonly used, thus there is a big focus on second order *DPA* in research.

3. Masking the Ring-LWE Encryption Scheme Using a Masked Decoder

Since most side-channel attacks focus on the decryption operation, this section will present an attempt to masking the decryption function of the *LPR ring-LWE* encryption scheme. This masking approach was originally proposed in [RRVV15], for more details we would refer you to that paper.

3.1. Implementation

We will start by giving the reader an overview of the general setup, before going into more detail about the masked decoding algorithms. We will make strong use of the *NTT* in this chapter. We recall, that our notation for polynomials in the *NTT* domain is $\tilde{\mathbf{f}}$. The *NTT* operation itself will be denoted as NTT(.), while its inverse operation will be written as INTT(.). We want to stress, that NTT(.) and INTT(.) are linear operations, as we will use this characteristic for our blinding technique.

3.1.1. Overview

This Subsection will cover a concise overview of the blinding technique proposed in [RRVV15]. For the sake of simplicity, the intermediate $\mathbf{m}_{enc}$ will be referred to as $\mathbf{a}$ in the following.

We start by splitting the secret key $\mathbf{s}$ into two shares $\mathbf{s}', \mathbf{s}'' \in R_q$ such that $\mathbf{s} = \mathbf{s}' + \mathbf{s}''$. Therefore, we choose all coefficients of $\mathbf{s}'$ uniformly at random and calculate $\mathbf{s}'' = \mathbf{s} - \mathbf{s}'$. In the *NTT* domain it follows that $\tilde{\mathbf{s}} = \tilde{\mathbf{s}}' + \tilde{\mathbf{s}}''$. Due to the linearity of INTT(.) and the multiplication, we can compute $\mathbf{a}$ as:

$$\mathbf{a} = \mathrm{INTT}(\tilde{\mathbf{s}} \cdot \tilde{\mathbf{c}}_1 + \tilde{\mathbf{c}}_2) = \mathrm{INTT}(\tilde{\mathbf{s}}' \cdot \tilde{\mathbf{c}}_1 + \tilde{\mathbf{c}}_2) + \mathrm{INTT}(\tilde{\mathbf{s}}'' \cdot \tilde{\mathbf{c}}_1) \tag{3.1}$$

This enables us to split the whole equation into two branches, calculating $\mathbf{m}'_{enc}$ and $\mathbf{m}''_{enc}$ in the following way:

$$\mathbf{a}' = \mathrm{INTT}(\tilde{\mathbf{s}}' \cdot \tilde{\mathbf{c}}_1 + \tilde{\mathbf{c}}_2),\ \mathbf{a}'' = \mathrm{INTT}(\tilde{\mathbf{s}}'' \cdot \tilde{\mathbf{c}}_1) \tag{3.2}$$

Those computations can be done on a arithmetic processor without any protection against side-channel attacks like *DPA*, as both branches are totally independent of our secret key $\mathbf{s}$.

However, the $\textsc{Decode}(a_i)$ function in the decryption stage of the *LPR* scheme is non-linear and cannot easily be split into two parts. For this reason, we will present a masked decoder in the next Subsection, that takes $\mathbf{a}'$ and $\mathbf{a}''$ as inputs to compute two shares $\mathbf{m}'$, $\mathbf{m}''$ of the decoded message $\mathbf{m}$ in a fairly efficient way.

3.1.2. Masked Decoder

This Section briefly describes a probabilistic masked decoder. We recall, that the i-th element of a is called a_i and the shares (a_i', a_i'') of such an element are chosen in a way, that $a_i' + a_i'' = a_i \pmod q$. To keep it simple, we will refer to an arbitrary a_i as a, the same follows for its shares.

For our masked decoder, we do not need to know the exact values of a' and a'' to compute $\textsc{Decode}(a)$. The following example will help us to lay down some rules for the decoder: Given (a', a'') with $0 < a' < q/4$ and $q/4 < a'' < q/2$. Then know that for $a = a' + a''$ it follows, that $q/4 < a < 3q/4$ and therefore $\textsc{Decode}(a) = 1$. We only need to know the most significant bits of a' and a'' to determine, which values they are bounded by.

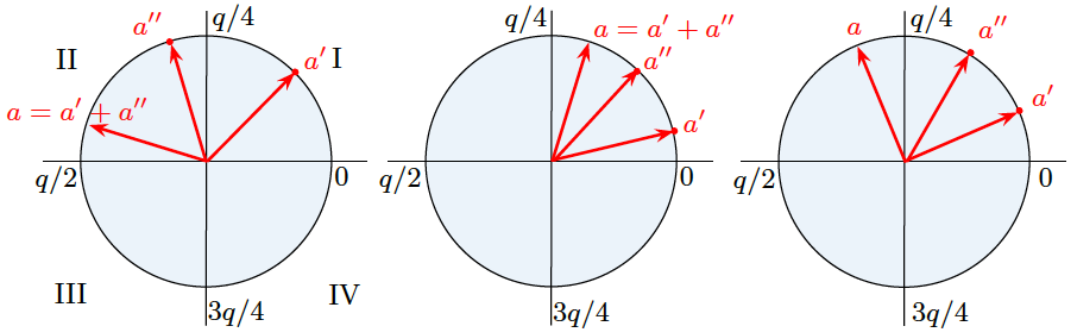

Figure 3.1.: The basic idea of our masked decoder. The circle represents elements in $\mathbb{Z}_q$. The first case shown allows us to conclude $\textsc{Decode}(a) = 1$, while we cannot make any guesses about the last two ones. [RRVV15]

Figure 3.1 shows our example from above on the left. We can use this knowledge to state a total of four rules, the first of whom is taken from our example:

- $0 < a' < q/4,\ q/4 < a'' < q/2 \implies a \in (q/4, 3q/4) \implies \textsc{Decode}(a) = 1$
- $q/2 < a' < 3q/4,\ 3q/4 < a'' < q \implies a \in (q/4, 3q/4) \implies \textsc{Decode}(a) = 1$

- $q/4 < a' < q/2,\ q/2 < a'' < 3q/4 \implies a \in (0, q/4) \cup (3q/4, q) \implies \textsc{Decode}(a) = 0$
- $3q/4 < a' < q,\ 0 < a'' < q/4 \implies a \in (0, q/4) \cup (3q/4, q) \implies \textsc{Decode}(a) = 0$

With swapping a' and a'' in the above rules, one can obtain another four rules. From the rules it follows, that we only need to know the quadrant of each a' an a'' to infer the output of $\textsc{Decode}(a)$. However, this does not work for all cases, as Figure 3.1 shows. We can actually only apply those rules in half of the possible cases.

So, what happens if our (a', a'') does not match any rule? We simply need to refresh the splitting by computing $a' = a' + \Delta_1$ and $a'' = a'' - \Delta_1$ with $\Delta_i \in \mathbb{Z}_q$. From $(a' + \Delta_1) + (a'' - \Delta_1) = a' + a'' = a$ it follow, that a stays unchanged by that refresh. Now that we have a fresh pair (a', a''), we can again try to apply our rules from above. This process can be repeated until all shares have been decoded. Note, that a new Δ_i should be chosen for each iteration. About half of the a', a'' are decoded per iteration, so that the amount of decoded shares rises exponentially with the number of iterations. The authors of [RRVV15] propose a number of $N = 16$ iterations for a satisfactory result.

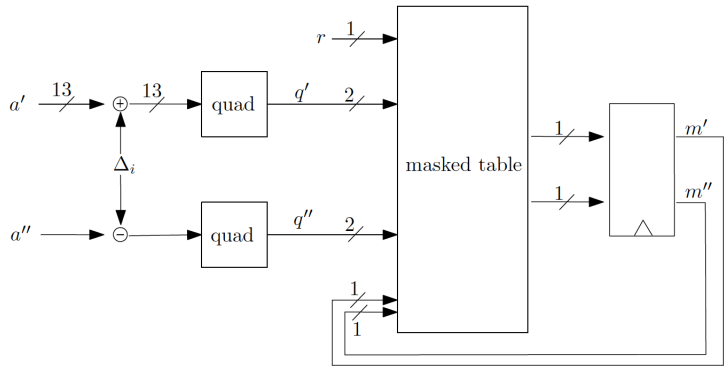

Figure 3.2.: Hardware implementation of the masked decoder. [RRVV15]

A possible hardware implementation of such a masked decoder is shown in Figure 3.2. The refreshing step is depicted on the left, using a different Δ_i in each iteration. The quadrant function used in the next step simply takes a refreshed share a' or a'' as an input and outputs two bits depending on the quadrant that the share belongs to. Next, a masked table is used to check the two bits against the rules we described above. Finally, the masked table function returns two one bit shares of the decoded message m. In our implementation the masked takes additional inputs, like a random bit r and the output of the last iteration (m'_{i-1}, m''_{i-1}). For more details on the masked table lookup, we would like to refer you to the paper of Oscar Reparaz et. al. [RRVV15].

3.2. Evaluation

Starting with efficiency, Reparaz et. al. showed that this implementation is at least 1.9x times better on a Virtex-2 FPGA than an unprotected high-speed elliptic curve scalar multiplier architecture introduced in [RRM12].

Furthermore, as both, the *LPR ring-LWE* encryption scheme and our masked decoder, are probabilistic, there will always be a chance for errors occurring during decoding. The global error rate of decoding rises significantly when using our masked decoder instead of a deterministic decoder. To offset this effect, we can adapt the number of iterations for the masked decoder. While for $N = 3$ iterations the global error rate is about 49 times larger than when using a deterministic decoder, $N = 16$ iterations yield a global error rate that is almost identical with the one of a deterministic decoder. Further improvement could be achieved by increasing the number of iterations again, but this would lead to a significantly higher cycle count and thus to a much more inefficient implementation.

For our evaluation in terms of side-channel attack soundness, we assume the attacker knows the details of our implementation and is aware of the rest of the key while guessing a subkey. Our evaluation will follow three steps: First, we perform a first-order key recovery attack with our source of randomness (*PRNG*) turned off. This attack will be successful, showing that our setting is correct. In the second step, we will turn on the PRNG and repeat the same attack, but it should not be successful in this case. Finally, we perform a second-order attack to confirm the correctness of the first two steps.

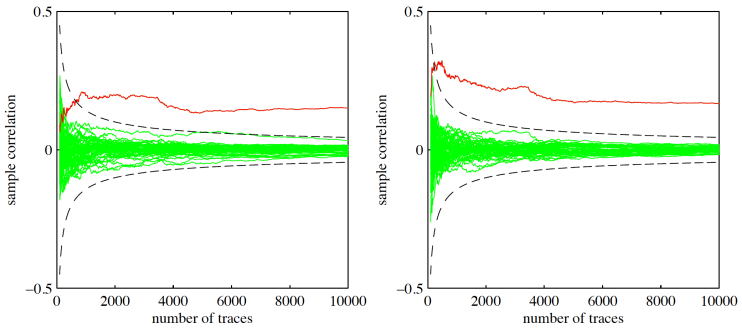

Figure 3.3.: *PRNG* is turned off. Graph shows the correlation coefficient increasing with the number of traces for the intermediates a'_0 (left) and a''_0 (right). The correct subkey is shown in red, all other guesses in green. [RRVV15]

For each of those steps, four different points that cover all relevant steps of the algorithm have been tested. The targets are a'_0, a''_0, the first input to the masked

decoder and the first output bit. Pearson's correlation coefficient has been used to compare our guesses with real measurements [BCO04].

***PRNG* off:** When the *Pseudo Random Number Generator (PRNG)* is turned off, sharing of $\mathbf{s}$ in $\mathbf{s}'$ and $\mathbf{s}''$ is deterministic. This translates to the masking being turned off. Figure 3.3 shows the correlation coefficient evolving with the number of traces. The attack seems to be successful starting at about a hundred traces.

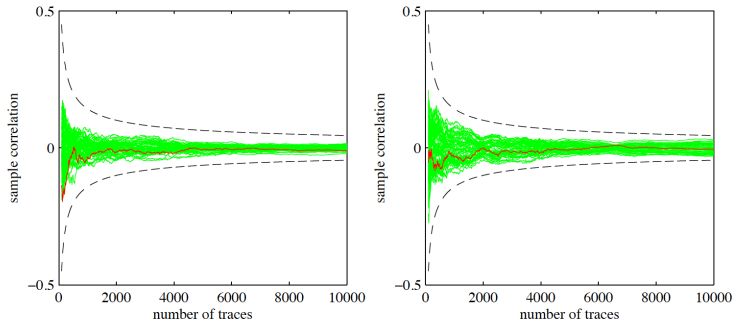

Figure 3.4.: Same as Figure 3.3, but with the *PRNG* turned on. It is no longer possible to identify the correct subkey within all guesses, meaning that the masking is successful. [RRVV15]

PRNG on: When the *PRNG* is turned on, the masking is effective. As we can see in Figure 3.4, the correct subkey can no longer be distinguished from all the other guesses, not even with an enormous amount of traces. This is what an attacker would see when conducting an first-order *DPA* on our masking scheme.

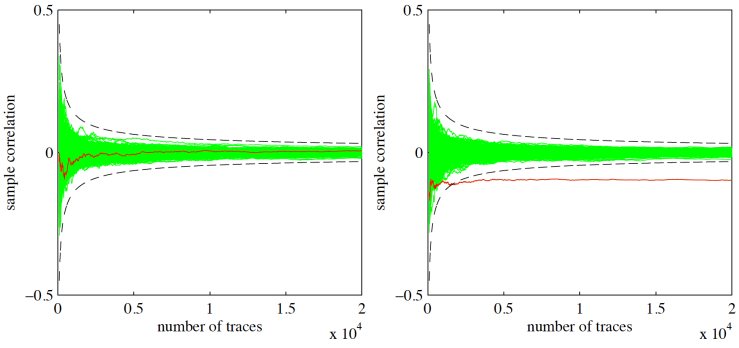

Figure 3.5.: On the left is the correlation for an increasing number of traces of a first-order attack with masking turned on. On the right we can see a successful second-order attack on our decoding scheme with masking turned on. [RRVV15]

Second-Order Attack: To confirm that we have used a sufficient number of traces in the first two steps, we perform a second-order attack on our masking scheme. In Figure 3.5 we can see, that the second-order attack starts to be successful at around 2000 traces. From this we conclude, that we carried out the first-order attack on the activated masking scheme correctly, as we are already successful with 2000 traces. We stress that an attacker would need a significantly higher number of traces and computations in reality, as we used a pretty friendly setting for our scenario.

4. Additively Homomorphic ring-LWE Masking

Less than a year after [RRVV15] (which has been described in the last Section) was published, Reparaz et. al. published a follow-up paper [RdCR+16] introducing a much easier approach for the masking of the ring-LWE encryption scheme.

In the following, the approach of said paper will be introduced and evaluated in terms of efficiency and side-channel attack resistance. In our description we again focus on the LPR scheme, though the techniques are also applicable for other ring-LWE encryption schemes.

4.1. Implementation

For ring-LWE masking, we make use of the fact that the LPR encryption scheme is additively homomorphic. Thus for given ciphertexts $(\mathbf{c}_1, \mathbf{c}_2)$ and $(\mathbf{c}'_1, \mathbf{c}'_2)$, which are the encryption of two messages $\mathbf{m}$ and $\mathbf{m}'$ with $m_i, m'_i \in \{0, 1\}$ using the same public key $\mathbf{p}$, it follows that $(\mathbf{c}_1 + \mathbf{c}'_1, \mathbf{c}_2 + \mathbf{c}'_2)$ is the encryption of $\mathbf{m} \oplus \mathbf{m}'$. As a consequence, we can write down the following equation:

$$\mathit{decryption}(\mathbf{c}_1, \mathbf{c}_2) \oplus \mathit{decryption}(\mathbf{c}'_1, \mathbf{c}'_2) = \mathit{decryption}(\mathbf{c}_1 + \mathbf{c}'_1, \mathbf{c}_2 + \mathbf{c}'_2) \quad (4.1)$$

Now, we want to make use of the property of additive homomorphism for our masking scheme. This, again, focuses on the decryption function, as this is the part of the encryption scheme, where the secret key is used, which makes it a prime target for attackers.

To randomize the decryption of $(\mathbf{c}_1, \mathbf{c}_2)$, we need to follow three simple steps:

1. Generate a random message $\mathbf{m}'$ unknown to the adversary
2. Encrypt $\mathbf{m}'$ to $(\mathbf{c}'_1, \mathbf{c}'_2)$
3. Decrypt $(\mathbf{c}_1 + \mathbf{c}'_1, \mathbf{c}_2 + \mathbf{c}'_2)$ to receive $\mathbf{m} \oplus \mathbf{m}'$

The masked message returned by this approach is $(\mathbf{m}', \mathbf{m} \oplus \mathbf{m}')$, such that $\mathbf{m} = (\mathbf{m} \oplus \mathbf{m}') \oplus \mathbf{m}'$.

The advantage of this approach is, that no masked decoder is needed. For decoding, an unprotected decoder might be used without leaking any useful information for an attacker.

4.2. Evaluation

As we can precompute $\mathbf{m}'$ and $(\mathbf{c}'_1, \mathbf{c}'_2)$ due to the independency of $\mathbf{m}'$, the tradeoff in terms of efficiency seems to be pretty reasonably. Note that it was not possible to do any precomputations when using the masked decoder described in the previous Section.

Furthermore, the complexity of the implementation of this approach is much lower compared to the other approach, which needs a carefully implemented masked decoder. However, the error rate of the new scheme is about a hundred times higher than the ring-LWE encryption scheme with masking being turned off. This effect might be compensated by increasing the modulus q by one bit from 13 to 14 bits or by decreasing the standard deviation σ of the discrete Gaussian distribution.

Although, straightforward first-order DPA attacks do not immediately work on our masking approach, more refined ones are still possible. This is due to the fact, that the key itself is not masked. The masking is only making it harder for an attacker to model the power consumption for first-order DPA. Using the same technique as before, the authors of [RdCR+16] showed that while first-order DPA attacks on the decryption with the masks being turned off are easy to conduct, the same attack on the scheme with masking being turned on is not possible straight away. More details on this can be found in the corresponding paper.

5. Flush+Reload Cache Attack on Bliss

5.1. Gaussian Sampling

5.1.1. CDT Sampling

Using the cumulative distribution function in the sampler, we build a large table, in which we approximate the probabilities $p_y = \mathbb{P}[x \leq y | x \leftarrow D_\sigma]$ with λ Bits of precision. At sampling time, we generate a uniformly random $r \in [0, 1)$ and perform a binary search in the table to locate $y \in [-r\sigma, r\sigma]$, so $r \in [p_{y-1}, p_y)$. If restricted to the non-negative part $[0, r\sigma]$, the probabilities are $p_y^* = \mathbb{P}[|x| \leq y | x \leftarrow D_\sigma]$, sampling is still $r \in [0, 1)$, but $y \in [0, r\sigma]$ is located.

The binary search in this sampling method can take some time, so one can speed it up by using an additional *guide table I*. This table stores for example 256 entries consisting of intervals $I[u] = (a_u, b_u), u \in \{0, ...255\}$ such that $p_{a_u}^* \leq u/256$ and $p_{b_u}^* \geq (u+1)/256$. At sampling time, the first byte of r is then used to select the corresponding $I[u]$, which leads to a smaller interval to binary search. r is effectively picked byte-by-byte using the guide table approach. Algorithm 4 summarizes the guide table approach.

5.1.2. Rejection Sampling

Rejection Sampling basically accepts a sampled uniformly random integer $y \in [-r\sigma, r\sigma]$ with probability $p_\sigma(y)/p_\sigma(\mathbb{Z})$. This is done by sampling a uniformly random value $r \in [0, 1)$ and accepting the unformly random y if $r \leq p_\sigma(y)$. This procedure can be quite expensive, because $p_\sigma(y)$ has to be calculated to a high precision and even then the rejection rate may be quite high.

The authers of [DDLL13] which introduced BLISS proposed a more efficient rejection sampling algorithm, which will be used in the following chapters.

This algorithm reduces the amount of rejected samples significantly. It begins with sampling a value x according to the binary discrete Gaussian distribution D_{σ_2} with $\sigma_2 = \frac{1}{2ln2}$. Uniformly random bits can be used to do this efficiently.

Algorithm 4 CDT Sampling With Guide Table

Input: Big table $T[y]$ containing values p_y^* of the cumulative distribution function of the discrete Gaussian distribution (using only non-negative values), omitting the first byte. Samll table I consisting of the 256 intervals.

Output: Value $y \in [-r\sigma, r\sigma]$, sampled with probability according to D_σ

pick a random byte r
Let $(I_{min}, I_{max}) = (a_r, b_r)$ be the left and right bounds of interval $I[r]$
if $I_{max} - I_{min} = 1$ **then**
    generate a random sign bit $b \in \{0, 1\}$
    return $y = (-1)^b I_{min}$
end if
Let $i = 1$ denote the index of the byte to look at
Pick a new random byte r
while 1 **do**
    $I_z = \lfloor \frac{I_{min}+I_{max}}{2} \rfloor$
    if $r >$ (ith byte of $T[I_z]$) **then**
        $I_{min} = I_z$
    else if $r <$ (ith byte of $T[I_z]$) **then**
        $I_{max} = I_z$
    else if $I_{max} - I_{min} = 1$ **then**
        generate a random sign bit $b \in \{0, 1\}$
        return $y = (-1)^b I_{min}$
    else
        increase i by 1
        pick new random byte r
    end if
end while

$y = Kx+z, z \in \{0, ..., K-1\}$ is then uniformly random sampled and $K = \lfloor \frac{\sigma}{\sigma_2}+1 \rfloor$ is distributed according to the targeted discrete Gaussian distribution D_σ by rejecting when $b = \exp(z(z+2Kx)/(2^2)) = 0$ holds.

This last step still requires some computing because of the exponential value b, but the authors provided a more efficiant algorithm for this, too.

Algorithm 5 Sampling from $D^+_{K_\sigma}$ for $K \in \mathbb{Z}$

Input: Target standard deviation σ, integer $K = \lfloor \frac{\sigma}{\sigma_2} + 1$, where $\sigma_2 = \frac{1}{2ln2}$
Output: Integer $y \in \mathbb{Z}^+$ according to $D^+_{K_{\sigma_2}}$

1: sample $x \in \mathbb{Z}$ according to $D^+_{K_{\sigma_2}}$
2: sample $z \in \mathbb{Z}$ unformly in $\{0, ...K-1\}$
3: $y \leftarrow Kx + z$
4: sample b with probability $exp(-z(z+2Kx)/(2\sigma^2))$
5: **if** $b = 0$ **then**
6: **return** y
7: **end if**

Algorithm 6 Sampling from D_{K_σ}

Output: An integer $y \in \mathbb{Z}$ according to D_{K_σ}

1: Sample integer $y \leftarrow D^+_{K_\sigma}$ using algorithm 5
2: **if** $y = 0$ **then**
3: restart with probability 1/2
4: **end if**
5: generate random bit b and **return** $(-1)^b y$

5.2. The FLUSH+RELOAD Cache Attack

The attack used in [BHLY16] is the FLUSH+RELOAD cache attack. This attack abuses the fact, that modern CPUs feature multiple different cache levels, for example L1 cache. This is the fastest and smallest cache level and located closest to the cpu core. The next level, L2 cache, is bigger and slower than L1, L3 is even bigger and slower than L2, etc.

If the CPU has to access a memory address, it looks for the corresponding memory block in higher levels first. If the block is found, a *cache hit*, the data is accessed. But in case of a *cache miss*, it starts to look for the memory block on lower cache levels down to the system memory. If the corresponding block is found in lower levels, the processor *evicts* a cache line in the higher memory

Algorithm 7 Sampling a bit with probability $exp(-x/(2\sigma^2))$ for $x \in [0, 2^\ell =$

Input: $x \in [0, 2^\ell)$ an integer in binary form $x = x_{\ell-1}...x_0$. Table ET with precomputed values $ET[i] = exp(-x/(2\sigma^2))$ for $0 \leq i \leq \ell - 1$
Output: A bit b with probability $exp(-x/(2\sigma^2))$ of being 1

```
1: for i = ℓ − 1 do
2:     if x_i = 1 then
3:         Sample A_i with probability ET[i]
4:         if A_i = 0 then return 0
5:         end if
6:     end if
7: end for
8: return 1
```

and places the found memory line there, to speed up future access on the same memory block.

The higher access times on lower cache levels are exploited by cache timing attacks like the FLUSH+RELOAD attack. The attacker has to use the same memory as the victim to apply this kind of attack. He can monitor the state of the cache and use the timing differences to check which memory blocks are cached and which addresses were accessed by the victim. If done correctly, the attacker can deduce the cache lines of the victims table access, which limits the possible chosen values.

The FLUSH+RELOAD attack abuses the x86_64 instruction `clflush` to evict a memory block from cache, before the victim executes his algorithm. After the victim is done with his memory access, he measures the time to access the memory block again. If the victim accessed the memory block, the block will be in a fast cache level and the access time will be low. If it was not accessed, the CPU has to load the memory block from a lower cache level and the access will be much slower. The attacker then knows, if the flushed memory block was accessed or not.

5.2.1. Attacking CDT Sampling

The CDT sampling algorithm with an interval table I and a table with actual values T like in algorithm 4 can leak information based on cache hits and cache misses.

A cache attack as described above can only yield the cache lines of index u of the interval table and the cache line of index I_z of the look-up in T. This leaves a range of values for $|y_i|$, additionally the sign of $y_i$is not stored in the table. But the combination of both table look-ups can yield precise information, too. Two

possible combinations of table accesses are *intersection* and *last-jump*.
The intersection cache weakness intersects knowledge about index u in I and the access $T[I_z]$. Sometimes values in the range of intervalls are largly not overlapping with the range of values from $T[I_z]$, so the intersection narrows down the possible values. If, for example, the sample $|y_i|$ has to be in the set $S_1 = \{0, 1, 2, 3, 4, 5, 6, 7, 8\}$ according to the cache line $I[u]$, but $T[I_z]$ reveals $|y_i|$ must be in $S_2 = \{8, 9, 10, 11, 12, 13, 14, 15\}$, we can learn $|y_i| \in S_1 \cap S_2 = \{7, 8\}$.
Another abusable weakness is called *last-jump* and can be used if elements of an interval $I[u]$ are spread over two cache lines of T. For example interval $I[u] = 5, 6, 7, 8, 9$ is divided over cache-lines $T_1 = \{0, 1, 2, 3, 4, 5, 6, 7\}$ and $T_2 = \{8, 9, 10, 11, 12, 13, 14, 15\}$. A binary search will start at the value 7 in the always accessed cache-line T_1, but cache-line T_2 is only accessed, if $|y_i| \in \{8, 9\}$.
[BHLY16] restricted the patterns even more:

- To limit the maximum possible error of $|y_i|$ to at most 1, only the weaknesses are used, in which the number of candidates for $|y_i|$ is 2.
- Because adjacent intervals can partially overlap, $I[u] \cap I[v] \neq 0$ for $u = v + 1$, for certain parts of $I[u]$, the outcome of the sample is biased. The second restriction considers only cache weaknesses for which one of the two candidates is much more likely to be sampled.

  $$\mathbb{P}[|y_i| = \gamma_1 | y_i \in \{\gamma_1, \gamma_2\} \subset I[u]] \gg \mathbb{P}[|y_i| = \gamma_2 | y_i \in \{\gamma_1, \gamma_2\} \subset I[u]]$$

  Values γ_1 with $\mathbb{P}[X = \gamma_1 | X \in \{\gamma_1, \gamma_2\} \subset I[u]] = 1 - \alpha$ with small α are wanted. Together with the first restriction, a matched access pattern is almost always γ_1.
- To learn the sign of $|y_i|$, only cache patterns with $|y_i| > \beta \cdot \mathbb{E}[\langle s, c \rangle], \beta \geq 1$ are used. By looking at the sign of z_i the sign of y_i can be learned, because

  $$sign(y_i) \neq sign(z_i) \iff \langle s, c \rangle > (y_i + z_i)$$

Choosing different values α and β is flexible, but might lead to no matching access patterns or let other parts fail. The paper described $\alpha \leq 0.1$ had at least one usable cache access pattern for every parameter set. β was not used in the experiments.

The attack assumes a cache access pattern reveals if $y_i \in \{\gamma_1, \gamma_2\}$ for $i = 0, ..., n - 1$ of polynomial y and $\mathbb{P}[y_i = \gamma_1] = 1 - \alpha$ with small α.
The victim then has to generate N signatures $(z_j, c_j), j = 1, ..., N$, which are collected by the attacker and the corresponding cache information for the noise polynomial y_i. When a cache weakness is found, the attacker has to solve the equation

$$z_{ji} = y_{ji} + (-1)^{b_j} \langle s, c_j i \rangle$$

Unknown values are b_j and s. Because of restrictions above, if $z_{ji=\gamma_1}$, the attacker adds $\xi_k = c_{ji}$ to a list of good vectors. With enough vectors $\xi_k = c_{ji}; 0 \leq i \leq n-1, 1 \leq j \leq N, 1 \leq k \leq n$ the matrix $L \in \{-1, 0, 1\}^{n \times n}$ can be formed. The column vectors ξ_k satisfy $sL = v$, where v is a unknown, but short, vector with norm about $\sqrt{\alpha n}$ in the lattice spanned by the rows of l. A lattice reduction algorithm, like LLL, can search for v and output a uni-modular matrix U with $UL = L'$. Every row of U (or its rotations) is tested if it is a candidate for $s = f$ by checking against the public key.

Because this last step is not always successful, not only n but $m > n$ vectors ξ_k are collected and a random subset forms the input for LLL.

ALGORITHMUS EINRCKUNGEN

Algorithm 8 Cache attack on BLISS with CDT Sampling

Input: Access to cache memory of a victim with a key-pair (A, S). BLISS input parameters n, σ, q, κ with $\kappa < K$. Access to signature polynomials $(z_1, z_2^\dagger, c)$ produced using S. Victim uses CDT sampling with tables T, I for noise polynomial y. Cache weakness that allows to determine if coefficient $y_i \in \{\gamma_1, \gamma_2\}$ of y, and when this is the case, the value of y_i, is biased towards γ_1

Output: Secret key S

Let $k = 0$ be the number of vectors gained so far and let $M = []$ be an empty list of vectors
while $k < m$ (m vectors before randomising LLL) **do**
    Collect signature $(z_1, z_2^\dagger, c)$ together with cache information for each coefficient y_i of noise polynomial y
    for $i = 1, ..., n$ **do**
        if $y_i \in \{\gamma_1, \gamma_2\}$ (according to cache information), and $z_{1i} = \gamma_1$ **then**
            add vector $\xi_k = c_i$ as a column to M and set $k = k + 1$
        end if
    end for
end while
while 1 **do**
    Choose random subset of n vectors from M and construct matrix L whose columns are those vectors from M
    Perform LLL basis reduction on L to get $UL = L'$, where U is a unimodular transformation matrix and L' is LLL reduced.
    for all $J = 1, ..., n$ **do**
        check if row u_j of U has the same distribution as f and if $(a_1/2) \cdot u_j \bmod 2q$ has the same distribution as $2g + 1$. Lastly verify if $a_1 \cdot u_j + a_2 \cdot (a_2/2) \cdot u_j \equiv q \bmod 2q$
        return $S = (u_j, (a_1/2) \cdot u_j \bmod 2q)$ if this is the case
    end for
end while

5.2.2. Attacking Rejection Sampling

When rejection sampling is used in the BLISS signature scheme, the side channel has to decide, if there was a table acces in the ET table. The attack exploits the small size of the ET table, which leaks very precise information about the sampling process.

Depending on the bit i of input x, $ET[i]$ is accessed, if $x = 0$, no table look-up is performed. If the attacker detects this, he knows $z = 0$ is the sampled value in step 2 in algorithm 5. In this case the attacker can assume $y \in \{0, \pm K, \pm 2K, ...\}$ for usable cache access patterns.

So the attacker knows the coefficients $y_i \in \{0, \pm K, \pm 2K, ...\}, i \in \{0, ..., n-1\}$ of the noise polynomial y. Because anyone can check if $max|\langle s, c\rangle| \leq \kappa < K$ with the public parameters, y_i can be determined completely with the signature vector z. With N more signatures $(z_j, c_j), j = 1, ..., N$, the attacker can search for $y_{ji} \in \{0, \pm K, \pm 2K, ...\}$ (y_{ji} means the ith coefficient of y_j). If the attacker additionally sees, that $z_{ji} = y_{ji}$, he knows $\langle s, c_{ji}\rangle = 0$. Such vector is a *good vector* for use in the attack and $\zeta_k = c_{ji}$ is saved for later (some known y_{ji} will be discarded, if they dont satisfy the necessary requirements). With n of these vectors $\xi_k = c_{ji}; 0 \leq i \leq n-1, 1 \leq j \leq N, 1 \leq k \leq n$ a matrix $L \in \{-1, 0, 1\}^{nxn}$ can be formed. The column vectors ξ_k satisfy $sL = 0$ (0 is the all-zero vector) and most likely the only dependency of ξ_k is introduced by s, so s is the only kernel vector. There is no need to randomize this process, because the all-zero vector is used.

EINRCKUNGEN NEU

Algorithm 9 Cache attack on BLISS with Rejection Sampling

Input: Access to cache memory of a victim with a key-pair (A, S). BLISS input parameters n, σ, q, κ with $\kappa < K$. Access to signatures $(z_1, z_2^\dagger, c)$ produced using S. Victim uses rejection sampling with small exponential table to sample noise polynomial y

Output: Secret key S

1: Let $k = 0$ be the number of vectors gained so far and let $M = []$ be an empty list of vectors
2: **while** $k < n$ **do**
3: Collect signature $(z_1, z_2^\dagger, c)$ together with cache information for each coefficient y_i of polynomial y
4: **for** $i = 1, ..., n$ **do**
5: **if** $y_i \in \{0, \pm K, \pm 2K, ...\}$ (according to cache information), and $z_{1i} = y_i$ **then**
6: add coefficient vector $\xi_k = c_i$ as a column to M and set $k = k + 1$
7: **end if**
8: **end for**
9: **end while**
10: Form a matrix M from the columns in M. Calculate kernel space of M. This gives a matrix $U \in \mathbb{Z}^{\ell \times n}$ such that $UM = 0$ where 0 is the all-zero matrix.
11: **for** $j = 1, ..., \ell$ (assume $\ell = 1$) **do**
12: check if row u_j of U has the same distribution as f and if $(a_1/2) \cdot u_j \bmod 2q$ has the same distribution as $2g + 1$. Lastly verify if $a_1 \cdot u_j + a_2 \cdot (a_2/2) \cdot u_j \equiv q \bmod 2q$
13: **return** $S = (u_j, (a_1/2) \cdot u_j \bmod 2q)$ if this is the case
14: **end for**
15: Remove a random entry from M, put $k = k - 1$, goto step 2

5.3. Attacking Sampling Algorithms with a Perfect Side Channel

The paper [BHLY16] provided experimental results for a perfect side-channel attack using the procedure explained. This requires the attacker to get every cache line of every table look-up while computing y in CDT and rejection sampling algorithms. The victim is assumed to sign random messages and the signatures are collected by the attacker. Cache lines are 64 byte and each element is 8 byte. To fulfill this requirements, the authors of [BHLY16] modified the *research oriented* C++ implementation published by the BLISS authors [DDLL]. NTL was used for LLL reductions and kernel calculations.

5.3.1. Perfect Side Channel Attack on CDT Sampling

A perfect side channel yields the attacker the values $\lfloor u/8 \rfloor$ and $\lfloor I_z/8 \rfloor$ of the table accesses for $I[u]$ and $T[I_z]$, the full cache line for a specific value.

EINSCHRNKUNGEN DER PARAMETER

For each BLISS parameter set at least one usable cache weakness was found, which could be abused. The attacker then collects m coefficient vectors c_j and runs LLL up to $t = 2(m - n) + 1$ times searching for s. A bigger value t is not likely to have better success probabilities, because the randomly constructed lattices have overlapping base vectors, so the authors of the paper considered a experiment failed after this number of tries. Each experiment (with different parameters and different sizes of m) was performed 1000 times to measure the success probability $p_{successs}$, the average number of required signatures $\bar{N}$ to get m usable challenges and the average length of v, if one was found. The expected number of needed signatures is:

$$\mathbb{E}[N] = \frac{m}{n \cdot \mathbb{P}[CP] \cdot \mathbb{P}[\langle s_q, c \rangle = 0]}$$

where the event CP means a usable cache access pattern for a coordinate of y.

We can see from the results in table 5.1, that the average number of signatures $\bar{N}$ needed to collect m usable columns depends more on the paramter σ and not much on n, because the number of usable cache weaknesses varies with that value QUELLE. The experimental results show a much better succes probabilty $p_{success}$ for a small increase of m, so an attacker can reach probabilities close to 1.0, if e simply collects enough usable signatures ([BHLY16] suggests picking $m \simeq 2n$).

Parameter Set	m	$p_{success}$	$\|\|v\|\|_2^2$	$\bar{N}$	$\mathbb{E}[N]$	Offline Time
	256	0.690	10	2537	2518	1.9s
BLISS-0	257	0.841	10	2547	2528	2.9s
$n = 256$	258	0.886	10	2565	2538	3.5s
$\sigma = 100$	259	0.903	10	2671	2548	4.0s
$\kappa = 12$	260	0.943	10	2580	2558	4.5s
	261	0.943	10	2596	2568	4.6s
	512	0.655	29	441	450	37.6s
BLISS-I	513	0.809	29	442	451	60.0s
$n = 512$	514	0.881	29	442	452	71.3s
$\sigma = 215$	515	0.925	29	443	453	73.9s
$\kappa = 23$	516	0.950	29	446	454	81.3s
	517	0.961	29	446	455	85.8s
	512	0.478	33	2021	2020	37.5s
BLISS-II	513	0.675	34	2023	2024	72.1s
$n = 512$	514	0.772	34	2030	2028	95.6s
$\sigma = 107$	515	0.818	35	2033	2032	110.4s
$\kappa = 23$	516	0.870	35	2033	2036	117.5s
	517	0.897	35	2041	2040	122.0s
	512	0.855	23	945	930	42.2s
BLISS-III	513	0.950	23	946	932	51.6s
$n = 512$	514	0.975	23	951	934	55.9s
$\sigma = 250$	515	0.987	24	954	935	55.9s
$\kappa = 30$	516	0.987	24	952	937	55.8s
	517	0.996	24	957	939	54.4s
	512	0.617	35	1206	1189	46.2s
BLISS-IV	513	0.817	36	1209	1191	75.3s
$n = 512$	514	0.885	36	1211	1194	88.4s
$\sigma = 271$	515	0.932	36	1215	1196	93.7s
$\kappa = 39$	516	0.947	36	1216	1198	102.4s
	517	0.955	36	1217	1201	104.4s

Table 5.1.: Experimental results of an attack on BLISS with CDT sampling using a perfect side channel and various parameter sets

5.3.2. Perfect Side Channel Attack on Rejection Sampling

The perfect side channel tells the attacker, if there was an table look-up in the table ET. The attack explained in section 5.2.2 can be applied and requires $m = n$ challenges c_i for the kernel calculation, because no randomization is needed. By checking the cache lines of the small part of the table, the attacker can learn if any element has been accessed in ET. Only 100 experiments were

performed in [BHLY16], because for all parameter sets $p_{success} = 1.0$ holds.

The expected numbers of signatures needed is:

$$\mathbb{E}[N] = ((\frac{1}{p_\sigma(\mathbb{Z})} \sum_{x=-r\sigma}^{r\sigma} p_\sigma(xK)) * \mathbb{P}[\langle s_1, c\rangle = 0])^{-1}$$

with $K = \lfloor \frac{\sigma}{\sigma_2} + 1 \rfloor$ and tail-cut $r \geq 1$. The average number $\bar{N}$ of required signatures is heavily dependent on the value σ, because xK is more likely to be sampled for small σ.

Parameter Set	m	$p_{success}$	$\bar{N}$	$\mathbb{E}[N]$	Offline Time
BLISS-0	256	1.0	1105	1102	0.8s
BLISS-I	512	1.0	1671	1694	14.7s
BLISS-II	512	1.0	824	839	14.4s
BLISS-III	512	1.0	3018	2970	16.0s
BLISS-IV	512	1.0	4223	4154	18.1S

Table 5.2.: Experimental results of an attack on BLISS with rejection sampling using a perfect side channel

5.4. Evaluation

6. Blinding Countermeasures

Blinding is a countermeasure commonly used to prevent side-channel attacks like *DPA* [KJJ99]. It is used to add additional randomness to mathematical operations in a way, that the attacker cannot easily draw conclusions from his observations. This Section summarizes two blinding countermeasures presented in [Saa16], which appear to be of special interest for ring-LWE cryptosystems. While the first countermeasure will be an approach to blinding of polynomial multiplication within a ring R_q, the second one will be a blinding countermeasure for Gaussian sampling. All those techniques are believed to help against the attacks we described in 5, yet this has not been verified.

6.1. Blinding Polynomial Multiplication

There are two pretty types of blinding for polynomial multiplications, the first of whom is the multiplication of each polynomial with a constant. For two polynomials $\mathbf{f}, \mathbf{g} \in R_q$ and constants $a, b \in \mathbb{Z}_q$ the blinding operation and the inverse operation look as follows:

$$\mathbf{h} = a\mathbf{f} \cdot b\mathbf{g} \tag{6.1}$$

$$\mathbf{f} \cdot \mathbf{g} = (ab)^{-1}\mathbf{h} \tag{6.2}$$

The second type would be circularly shifting the coefficients in each of the polynomials. As a polynomial can be written as $\mathbf{f} = \sum_{i=0}^{n-1} f_i x^i$ with f_i being the i-th coefficient of the polynomial $\mathbf{f}$, a shift by j positions would be equal to the following computation:

$$x^j \mathbf{f} = \sum_{i=0}^{n-1} f_i x^{i+j} = \sum_{i=0}^{n-1} f_{i-j} x^i \tag{6.3}$$

Both of those blinding operation can be combined within one function, which will be called PolyBlind$(\mathbf{v}, s, c)$ from now on. This function works on the coefficient vectors of polynomials of degree n and is given in Algorithm 10.

The inverse operation can be denoted by PolyBlind$(\vec{v'}, -s, c^{-1})$. Due to the isometries of the ring R_q, the multiplication of two polynomials (here: their coefficient vectors) can be blinded using the PolyBlind function in the following way:

Algorithm 10 PolyBlind

Input: coefficient vector $\vec{v}$, number of shifts s, constant c
Output: blinded coefficient vector $\vec{v'}$

1: **for** $i = 0, ..., n - s - 1$ **do**
2: $\quad v'_i = cv_{i+s} \bmod q$
3: **end for**
4: **for** $i = n - s, ..., n - 1$ **do**
5: $\quad v'_i = q - cv_{i+s-n} \bmod q$
6: **end for**
7: **return** $\vec{v'}$

$$\vec{f'} = \text{PolyBlind}(\vec{f}, r, a) \text{ with } r \in_R 0, ..., n-1 \text{ and } a \in_R \mathbb{Z}_q$$
$$\vec{g'} = \text{PolyBlind}(\vec{g}, s, b) \text{ with } s \in_R 0, ..., n-1 \text{ and } b \in_R \mathbb{Z}_q$$
$$\vec{h'} = \vec{f'} \cdot \vec{g'}$$
$$\vec{h} = \text{PolyBlind}(\vec{h'}, -(r+s), (ab)^{-1})$$

6.2. Blinding Gaussian Sampling

As with the blinding of polynomial multiplication in the last subsection, there are two pretty easy ways to blind the coefficient vectors during the process of Gaussian sampling. We will again give a short description of both of them and present a function, that combines both methods.

We define a function VectorSample(n, σ), that samples and returns a vector according to the discrete Gaussian distribution $\mathcal{N}^n_{\mathbb{Z}}(0, \sigma^2)$. A naive implementation of this function could lead to leakage of information to an attacker using e.g. DPA. This has been done in the cache attack from [BHLY16] we described in Section 5.

The first approach to blinding would be to randomly shuffle the elements in the coefficient vector. The function VectorShuffle$(\vec{x})$ is doing exactly that, so that VectorShuffle(VectorSample(n, σ)) would increase security to a certain extend. For the second approach we need to take a short detour through probability theory. For two Gaussian distributions $X = \mathcal{N}^n_{\mathbb{Z}}(\mu_X, \sigma^2_X)$ and $Y = \mathcal{N}^n_{\mathbb{Z}}(\mu_Y, \sigma^2_Y)$ it holds that their sum is equal to $X + Y = \mathcal{N}^n_{\mathbb{Z}}(\mu_X + \mu_Y, \sigma^2_X + \sigma^2_Y)$. As we focus on zero-centered distributions, the center does not change for us. For the standard deviation it follows, that $\sigma_{X+Y} = \sqrt{\sigma^2_x + \sigma^2_Y}$. Algorithm 11 is one possible way to make use of those characteristics of Gaussian distributions. It also makes use of the VectorShuffle$(\vec{x})$ function to increase overall security. Another, very similar approach, has been described in [PDG14].

Algorithm 11 VectorBlindSample

Input: length of the vector n, number of iterations m, standard deviation σ
Output: sampled vector $\vec{x}$

1: $\mathbf{x} = \mathbf{0}$
2: **for** $i = 1, ..., m$ **do**
3: $\quad \vec{x} = \vec{x} + \mathcal{N}_{\mathbb{Z}}^{n}(0, (\frac{1}{\sqrt{m}}\sigma)^2)$
4: $\quad \vec{x} = \text{VectorShuffle}(\vec{x})$
5: **end for**
6: **return** $\vec{x}$

7. Conclusion

Conclude your thesis and discuss your results...

List of Figures

List of Tables

List of Algorithms

A. Bibliography

[BCO04] Eric Brier, Christophe Clavier, and Francis Olivier. *Correlation Power Analysis with a Leakage Model*, pages 16–29. Springer Berlin Heidelberg, 2004.

[BHLY16] Leon Groot Bruinderink, Andreas Hlsing, Tanja Lange, and Yuval Yarom. Flush, gauss, and reload – a cache attack on the bliss lattice-based signature scheme. Cryptology ePrint Archive, Report 2016/300, 2016.

[CJL+16] Lily Chen, Stephen Jordan, Yi-Kai Liu, Dustin Moody, Rene Peralta, Ray Perlner, and Daniel Smith-Tone. Report on post-quantum cryptography. Technical Report NIST IR 8105, National Institute of Standards and Technology (NIST), February 2016.

[DDLL] Lúo Ducas, Alain Durmus, Tancrde Lepoint, and Vadim Lyubashevsky. Bliss: Bimodal lattice signature schemes. http://bliss.di.ens.fr/.

[DDLL13] Léo Ducas, Alain Durmus, Tancrde Lepoint, and Vadim Lyubashevsky. Lattice signatures and bimodal gaussians. Cryptology ePrint Archive, Report 2013/383, 2013.

[KJJ99] Paul C. Kocher, Joshua Jaffe, and Benjamin Jun. Differential power analysis. In *Advances in Cryptology - CRYPTO '99, 19th Annual International Cryptology Conference, Santa Barbara, California, USA, August 15-19, 1999, Proceedings*, pages 388–397, 1999.

[LPR12] Vadim Lyubashevsky, Chris Peikert, and Oded Regev. On ideal lattices and learning with errors over rings. Cryptology ePrint Archive, Report 2012/230, 2012.

[PDG14] Thomas Pöppelmann, Léo Ducas, and Tim Güneysu. Enhanced lattice-based signatures on reconfigurable hardware. Cryptology ePrint Archive, Report 2014/254, 2014.

[RdCR+16] Oscar Reparaz, Ruan de Clercq, Sujoy Sinha Roy, Frederik Vercauteren, and Ingrid Verbauwhede. *Post-Quantum Cryptography: 7th International Workshop, PQCrypto 2016, Fukuoka, Japan, February 24-26, 2016, Proceedings*, chapter Additively Homomorphic Ring-LWE Masking, pages 233–244. Springer International Publishing, 2016.

[RRM12] Chester Rebeiro, Sujoy Sinha Roy, and Debdeep Mukhopadhyay. *Pushing the Limits of High-Speed $GF(2^m)$ Elliptic Curve Scalar Multiplication on FPGAs*, pages 494–511. Springer Berlin Heidelberg, 2012.

[RRVV15] Oscar Reparaz, Sujoy Sinha Roy, Frederik Vercauteren, and Ingrid Verbauwhede. A masked ring-lwe implementation. Cryptology ePrint Archive, Report 2015/724, 2015.

[Saa16] Markku-Juhani O. Saarinen. Arithmetic coding and blinding countermeasures for ring-lwe. Cryptology ePrint Archive, Report 2016/276, 2016.

[Sho97] Peter W. Shor. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM J. Comput.*, 26(5):1484–1509, October 1997.