# Flush, Gauss, and Reload – A Cache Attack on the BLISS Lattice-Based Signature Scheme

Leon Groot Bruinderink[1], Andreas Hülsing[1], Tanja Lange[1], and Yuval Yarom[2]

[1] Department of Mathematics and Computer Science
Technische Universiteit Eindhoven, P.O. Box 513, 5600 MB Eindhoven, NL
`l.groot.bruinderink@tue.nl`, `andreas@huelsing.net`, `tanja@hyperelliptic.org`

[2] The University of Adelaide and NICTA
`yval@cs.adelaide.edu.au`

**Abstract.** We present the first side-channel attack on a lattice-based signature scheme, using the Flush+Reload cache-attack. The attack is targeted at the discrete Gaussian sampler, an important step in the Bimodal Lattice Signature Schemes (BLISS). After observing only 450 signatures with a perfect side-channel, an attacker is able to extract the secret BLISS-key in less than 2 minutes, with a success probability of 0.96. Similar results are achieved in a proof-of-concept implementation using the Flush+Reload technique with less than 3500 signatures.
We show how to attack sampling from a discrete Gaussian using CDT or Bernoulli sampling by showing potential information leakage via cache memory. For both sampling methods, a strategy is given to use this additional information, finalize the attack and extract the secret key. We provide experimental evidence for the idealized perfect side-channel attacks and the Flush+Reload attack on two recent CPUs.

**Keywords:** SCA, Flush+Reload, lattices, BLISS, discrete Gaussians.

## 1 Introduction

The possible advent of general purpose quantum computers will undermine the security of all widely deployed public key cryptography. Ongoing progress towards building such quantum computers recently motivated standardization bodies to set up programs for standardizing post-quantum public key primitives, focusing on schemes for digital signatures, public key encryption, and key exchange [7,18,23].

A particularly interesting area of post-quantum cryptography is lattice-based cryptography; there exist efficient lattice-based proposals for signatures, encryption, and key exchange [9,21,15,26,3,37,1] and several of the proposed schemes have implementations, including implementations in open source libraries [34].

This work was supported in part by the Commission of the European Communities through the Horizon 2020 program under project number 645622 PQCRYPTO. Permanent ID of this document: `da245c8568290e4a0f45c704cc62a2b8`.

While the theoretical and practical security of these schemes is under active research, security of implementations is an open issue.

In this paper we make a first step towards understanding implementation security, presenting the first side-channel attack on a lattice-based signature scheme. More specifically, we present a cache-attack on the Bimodal Lattice Signature Scheme (BLISS) by Ducas, Durmus, Lepoint, and Lyubashevsky from CRYPTO 2013 [9], attacking a research-oriented implementation made available by the BLISS authors at [8]. We present attacks on the two implemented methods for sampling from a discrete Gaussian and for both successfully obtain the secret signing key.

Note that most recent lattice-based signature schemes use noise sampled according to a discrete Gaussian distribution to achieve provable security and a reduction from standard assumptions. Hence, our attack might be applicable to many other implementations. It is possible to avoid our attack by using schemes which avoid discrete Gaussians at the cost of more aggressive assumptions [14].

**1.1. The attack target.** BLISS is the most recent piece in a line of work on identification-scheme-based lattice signatures, also known as signatures without trapdoors. An important step in the signature scheme is blinding a secret value in some way to make the signature statistically independent of the secret key. For this, a blinding (or noise) value $\mathbf{y}$ is sampled according to a discrete Gaussian distribution. In the case of BLISS, $\mathbf{y}$ is an integer polynomial of degree less than some system parameter $n$ and each coefficient is sampled separately. Essentially, $\mathbf{y}$ is used to hide the secret polynomial $\mathbf{s}$ in the signature equation $\mathbf{z} = \mathbf{y} + (-1)^b(\mathbf{s} \cdot \mathbf{c})$, where noise polynomial $\mathbf{y}$ and bit $b$ are unknown to an attacker and $\mathbf{c}$ is the challenge polynomial from the identification scheme which is given as part of the signature $(\mathbf{z}, \mathbf{c})$.

If an attacker learns the noise polynomials $\mathbf{y}$ for a few signatures, he can compute the secret key using linear algebra and guessing the bit $b$ per signature. Actually, the attacker will only learn the secret key up to the sign but for BLISS $-\mathbf{s}$ is also a valid secret key.

**1.2. Our contribution.** In this work we present a Flush+Reload attack on BLISS. We implemented the attack for two different algorithms for Gaussian sampling. First we attack the *CDT sampler with guide table*, as described in [29] and used in the attacked implementation as default sampler [8]. CDT is the fastest way of sampling discrete Gaussians, but requires a large table stored in memory. Then we also attack a rejection sampler, specifically the Bernoulli-based sampler that was proposed in [9], and also provided in [8].

On a high level, our attacks exploit cache access patterns of the implementations to learn a few coefficients of $\mathbf{y}$ per observed signature. We then develop mathematical attacks to use this partial knowledge of different $\mathbf{y}_j$s together with the public signature values $(\mathbf{z}_j, \mathbf{c}_j)$ to compute the secret key, given observations from sufficiently many signatures.

In detail, there is an interplay between requirements for the offline attack and restrictions on the sampling. First, restricting to cache access patterns that provide relatively precise information means that the online phase only allows to extract a few coefficients of $\mathbf{y}_j$ per signature. This means that trying all guesses for the bits $b$ per signature becomes a bottleneck. We circumvent this issue by only collecting coefficients of $\mathbf{y}_j$ in situations where the respective coefficient of $\mathbf{s} \cdot \mathbf{c}_j$ is zero as in these cases the bit $b_j$ has no effect.

Second, each such collected coefficient of $\mathbf{y}_j$ leads to an equation with some coefficients of $\mathbf{s}$ as unknowns. However, it turns out that for CDT sampling the cache patterns do not give exact equations. Instead, we learn equations which hold with high probability, but might be off by $\pm 1$ with non-negligible probability. We managed to turn the computation of $\mathbf{s}$ into a lattice problem and show how to solve it using the LLL algorithm [20]. For Bernoulli sampling we can obtain exact equations but at the expense of requiring more signatures.

We first tweaked the BLISS implementation to provide us with the exact cache lines used, modeling a perfect side-channel. For BLISS-I, designed for 128 bits of security, the attack on CDT needs to observe on average 441 signatures during the online phase. Afterwards, the offline phase succeeds after 37.6 seconds with probability 0.66. This corresponds to running LLL once. If the attack does not succeed at first, a few more signatures (on average a total of 446) are sampled and LLL is run with some randomized selection of inputs. The combined attack succeeds with probability 0.96, taking a total of 85.8 seconds. Similar results hold for other BLISS versions. In the case of Bernoulli sampling, we are given exact equations and can use simple linear algebra to finalize the attack, given a success probability of 1.0 after observing 1671 signatures on average and taking 14.7 seconds in total.

To remove the assumption of a perfect side-channel we performed a proof-of-concept attack using the Flush+Reload technique on a modern laptop. This attack achieves similar success rates, albeit requiring 3438 signatures on average for BLISS-I with CDT sampling. For Bernoulli sampling, we now had to deal with measurement errors. We did this again by formulating a lattice problem and using LLL in the final step. The attack succeeds with a probability of 0.88 after observing an average of 3294 signatures.

**1.3. Structure.** In Section 2, we give brief introductions to lattices, BLISS, and the used methods for discrete Gaussian sampling as well as to cache-attacks. In Section 3, we present two information leakages through cache-memory for CDT sampling and provide a strategy to exploit this information for secret key extraction. In Section 4, we present an attack strategy for the case of Bernoulli sampling. In Section 5, we present experimental results for both strategies assuming a perfect side-channel. In Section 6, we show that realistic experiments also succeed, using Flush+Reload attacks.

## 2 Preliminaries

This section describes the BLISS signature scheme and the used discrete Gaussian samplers. It also provides some background on lattices and cache attacks.

**2.1. Lattices.** We define a *lattice* $\Lambda$ as a discrete subgroup of $\mathbb{R}^n$: given $m \leq n$ linearly independent vectors $\mathbf{b}_1, \ldots, \mathbf{b}_m \in \mathbb{R}^n$, the lattice $\Lambda$ is given by the set

$\Lambda(\mathbf{b}_1, \ldots, \mathbf{b}_m)$ of all integer linear combinations of the $\mathbf{b}_i$'s:

$$\Lambda(\mathbf{b}_1, \ldots, \mathbf{b}_m) = \left\{ \sum_{i=1}^{m} x_i \mathbf{b}_i \mid x_i \in \mathbb{Z} \right\}.$$

We call $\{\mathbf{b}_1, \ldots, \mathbf{b}_m\}$ a basis of $\Lambda$ and define $m$ as the rank. We represent the basis as a matrix $\mathbf{B} = (\mathbf{b}_1, \ldots, \mathbf{b}_m)$, which contains the vectors $\mathbf{b}_i$ as column vectors. In this paper, we mostly consider full-rank lattices, i.e. $m = n$, unless stated otherwise. Given a basis $\mathbf{B} \in \mathbb{R}^{n \times n}$ of a full-rank lattice $\Lambda$, we can apply any unimodular transformation matrix $\mathbf{U} \in \mathbb{Z}^{n \times n}$ and $\mathbf{UB}$ will also be a basis of $\Lambda$. The LLL algorithm [20] transforms a basis $\mathbf{B}$ to its LLL-reduced basis $\mathbf{B}'$ in polynomial time. In an LLL-reduced basis the shortest vector $\mathbf{v}$ of $\mathbf{B}'$ satisfies $||\mathbf{v}||_2 \leq 2^{\frac{n-1}{4}}(|\det(\mathbf{B})|)^{1/n}$ and there are looser bounds for the other basis vectors. Here $||\cdot||_2$ denotes the Euclidean norm. Besides the LLL-reduced basis, NTL's [33] implementation of LLL also returns the unimodular transformation matrix $\mathbf{U}$, satisfying $\mathbf{UB} = \mathbf{B}'$.

In cryptography, lattices are often defined via polynomials, e.g., to take advantage of efficient polynomial arithmetic. The elements in $R = \mathbb{Z}[x]/(x^n+1)$ are represented as polynomials of degree less than $n$. For each polynomial $f(x) \in R$ we define the corresponding vector of coefficients as $\mathbf{f} = (f_0, f_1, \ldots, f_{n-1})$. Addition of polynomials $f(x) + g(x)$ corresponds to addition of their coefficient vectors $\mathbf{f} + \mathbf{g}$. Additionally, multiplication of $f(x) \cdot g(x) \bmod (x^n + 1)$ defines a multiplication operation on the vectors $\mathbf{f} \cdot \mathbf{g} = \mathbf{gF} = \mathbf{fG}$, where $\mathbf{F}, \mathbf{G} \in \mathbb{Z}^{n \times n}$ are matrices, whose columns are the rotations of (the coefficient vectors of) $\mathbf{f}, \mathbf{g}$, with possibly opposite signs. Lattices using polynomials modulo $x^n + 1$ are often called *NTRU lattices* after the NTRU encryption scheme [15].

An *integer lattice* is a lattice for which the basis vectors are in $\mathbb{Z}^n$, such as the NTRU lattices just described. For integer lattices it makes sense to consider elements modulo $q$, so basis vectors and coefficients are taken from $\mathbb{Z}_q$. We represent the ring $\mathbb{Z}_q$ as the integers in $[-q/2, q/2)$. We denote the quotient ring $R/(qR)$ by $R_q$. When we work in $R_q = \mathbb{Z}_q[x]/(x^n + 1)$ (or $R_{2q}$), we assume $n$ is a power of 2 and $q$ is a prime such that $q \equiv 1 \bmod 2n$.

**2.2. BLISS.** We provide the basic algorithms of BLISS, as given in [9]. Details of the motivation behind the construction and associated security proofs are given in the original work. All arithmetic for BLISS is performed in $R$ and possibly with each coefficient reduced modulo $q$ or $2q$. We follow notation of BLISS and also use boldface notation for polynomials.

By $D_\sigma$ we denote the discrete Gaussian distribution with standard deviation $\sigma$. In the next subsection, we will zoom in on this distribution and how to sample from it in practice. The main parameters of BLISS are dimension $n$, modulus $q$ and standard deviation $\sigma$. BLISS uses a cryptographic hash function $H$, which outputs binary vectors of length $n$ and weight $\kappa$; parameters $d_1$ and $d_2$ determining the density of the polynomials forming the secret key; and $d$, determining the length of the second signature component.

**Algorithm 2.1** BLISS Key Generation

```
Output: A 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 $d_2$
   entries in $\{\pm 2\}$
2: $\mathbf{S} = (\mathbf{s}_1, \mathbf{s}_2) = (\mathbf{f}, 2\mathbf{g} + 1)$
3: if $\mathbf{S}$ violates certain bounds (details in [9]), then restart
4: $\mathbf{a}_q = (2\mathbf{g} + 1)/\mathbf{f} \bmod q$ (restart if $\mathbf{f}$ is not invertible)
5: return $(\mathbf{A}, \mathbf{S})$ where $\mathbf{A} = (2\mathbf{a}_q, q - 2) \bmod 2q$
```

Algorithm 2.1 generates correct keys because

$$\mathbf{a}_1 \cdot \mathbf{s}_1 + \mathbf{a}_2 \cdot \mathbf{s}_2 = 2\mathbf{a}_q \cdot \mathbf{f} + (q-2) \cdot (2\mathbf{g}+1) \equiv 2(2\mathbf{g}+1) + (q-2)(2\mathbf{g}+1) \equiv q \bmod 2q.$$

Note that when an attacker has a candidate for key $\mathbf{s}_1 = \mathbf{f}$, he can validate correctness by checking the distributions of $\mathbf{f}$ and $\mathbf{a}_q \cdot \mathbf{f} \equiv 2\mathbf{g} + 1 \bmod 2q$, and lastly verifying that $\mathbf{a}_1 \cdot \mathbf{f} + \mathbf{a}_2 \cdot (\mathbf{a}_q \cdot f) \equiv q \bmod 2q$, where $\mathbf{a}_q$ is obtained by halving $\mathbf{a}_1$.

Signature generation (Algorithm 2.2) uses $p = \lfloor 2q/2^d \rfloor$, which is the highest order bits of the modulus $2q$, and constant $\zeta = \frac{1}{q-2} \bmod 2q$. In general, with $\lfloor . \rceil_d$ we denote the $d$ highest order bits of a number. In Step 1 of Algorithm 2.2, two integer vectors are sampled, where each coordinate is drawn independently and according to the discrete Gaussian distribution $D_\sigma$. This is denoted by $\mathbf{y} \leftarrow D_{\mathbb{Z}^n,\sigma}$.

**Algorithm 2.2** BLISS Signature Algorithm

```
Input: Message $\mu$, public key $\mathbf{A} = (\mathbf{a}_1, q - 2)$, secret key $\mathbf{S} = (\mathbf{s}_1, \mathbf{s}_2)$
Output: A signature $(\mathbf{z}_1, \mathbf{z}_2^\dagger, \mathbf{c}) \in \mathbb{Z}_{2q}^n \times \mathbb{Z}_p^n \times \{0,1\}^n$ of the message $\mu$
1: $\mathbf{y}_1, \mathbf{y}_2 \leftarrow D_{\mathbb{Z}^n,\sigma}$
2: $\mathbf{u} = \zeta \cdot \mathbf{a}_1 \cdot \mathbf{y}_1 + \mathbf{y}_2 \bmod 2q$
3: $\mathbf{c} = H(\lfloor \mathbf{u} \rceil_d \bmod p, \mu)$
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 $\sigma$, $||\mathbf{Sc}||$, $\langle \mathbf{z}, \mathbf{Sc} \rangle$ (details in [9]), else restart
8: $\mathbf{z}_2^\dagger = (\lfloor \mathbf{u} \rceil_d - \lfloor \mathbf{u} - \mathbf{z}_2 \rceil_d) \bmod p$
9: return $(\mathbf{z}_1, \mathbf{z}_2^\dagger, \mathbf{c})$
```

In the attacks, we concentrate on the first signature vector $\mathbf{z}_1$, since $\mathbf{z}_2^\dagger$ only contains the $d$ highest order bits and therefore lost information about $\mathbf{s}_2 \cdot \mathbf{c}$; furthermore, $\mathbf{A}$ and $\mathbf{f}$ determine $\mathbf{s}_2$ as shown above. So in the following, we only consider $\mathbf{z}_1, \mathbf{y}_1$ and $\mathbf{s}_1$, and thus will leave out the indices.

In lines 5 and 6 of Algorithm 2.2, we compute $\mathbf{s} \cdot \mathbf{c}$ over $R_{2q}$. However, since secret $\mathbf{s}$ is sparse and challenge $\mathbf{c}$ is sparse and binary, the absolute value of

$||\mathbf{s}\cdot\mathbf{c}||_\infty \leq 5\kappa \ll 2q$, with $||\cdot||_\infty$ the $\ell_\infty$-norm. This means these computations are simply additions over $\mathbb{Z}$, and we can therefore model this computation as a vector-matrix multiplication over $\mathbb{Z}$:

$$\mathbf{s}\cdot\mathbf{c} = \mathbf{sC},$$

where $\mathbf{C} \in \{-1,0,1\}^{n\times n}$ is the matrix whose columns are the rotations of challenge $\mathbf{c}$ (with minus signs matching reduction modulo $x^n+1$). In the attacks we access individual coefficients of $\mathbf{s}\cdot\mathbf{c}$; note that the $j$th coefficient equals $\langle \mathbf{s}, \mathbf{c}_j\rangle$, where $\mathbf{c}_j$ is the $j$th column of $C$.

For completeness, we also show the verification procedure (Algorithm 2.3), although we do not use it further in this paper. Note that reductions modulo $2q$ are done before truncating and reducing modulo $p$.

**Algorithm 2.3** BLISS Verification Algorithm

```
Input: Message $\mu$, public key $\mathbf{A} = (\mathbf{a}_1, q-2) \in R_{2q}^2$, signature $(\mathbf{z}_1, \mathbf{z}_2^\dagger, \mathbf{c})$
Output: Accept or reject the signature
1: if $\mathbf{z}_1, \mathbf{z}_2^\dagger$ violate certain bounds (details in [9]), then reject
2: accept iff $\mathbf{c} = H(\lfloor \zeta\cdot\mathbf{a}_1\cdot\mathbf{z}_1 + \zeta\cdot q\cdot\mathbf{c}\rceil_d + \mathbf{z}_2^\dagger \bmod p, \mu)$
```

**2.3. Discrete Gaussian distribution.** The probability distribution of a (centered) discrete Gaussian distribution is a distribution over $\mathbb{Z}$, with mean 0 and standard deviation $\sigma$. A value $x \in \mathbb{Z}$ is sampled with probability:

$$\frac{\rho_\sigma(x)}{\sum_{y=-\infty}^{\infty}\rho_\sigma(y)},$$

where $\rho_\sigma(x) = \exp\left(\frac{-x^2}{2\sigma^2}\right)$. Note that the sum in the denominator ensures that this is actually a probability distribution. We denote the denominator by $\rho_\sigma(\mathbb{Z})$.

To make sampling practical, most lattice-based schemes use three simplifications: First, a tail-cut $\tau$ is used, restricting the support of the Gaussian to a finite interval $[-\tau\sigma, \tau\sigma]$. The tail-cut $\tau$ is chosen such that the probability of a real discrete Gaussian sample landing outside this interval is negligible in the security parameter. Second, values are sampled from the positive half of the support and then a bit is flipped to determine the sign. For this the probability of obtaining zero in $[0, \tau\sigma]$ needs to be halved. The resulting distribution on the positive numbers is denoted by $D_\sigma^+$. Finally, the precision of the sampler is chosen such that the statistical distance between the output distribution and the exact distribution is negligible in the security parameter.

There are two generic ways to sample from a discrete Gaussian distribution: using the cumulative distribution function [25] or via rejection sampling [11]. Both these methods are deployed with some improvements which we describe next. These modified versions are implemented in [8]. We note that there are also other ways [10,31,30,5] of efficiently sampling discrete Gaussians.

**CDT sampling.** The basic idea of using the cumulative distribution function in the sampler, is to approximate the probabilities $p_y = \mathbb{P}[x \leq y|\ x \leftarrow D_\sigma]$, computed with $\lambda$ bits of precision, and save them in a large table. At sampling time, one samples a uniformly random $r \in [0, 1)$, and performs a binary search through the table to locate $y \in [-\tau\sigma, \tau\sigma]$ such that $r \in [p_{y-1}, p_y)$. Restricting to the non-negative part $[0, \tau\sigma]$ corresponds to using the probabilities $p_y^* = \mathbb{P}[|x| \leq y|\ x \leftarrow D_\sigma]$, sampling $r \in [0, 1)$ and locating $y \in [0, \tau\sigma]$. While this is the most efficient approach, it requires a large table. We denote the method that uses the approximate cumulative distribution function with tail cut and the modifications described next, as the *CDT sampling* method.

One can speed up the binary search for the correct sample $y$ in the table, by using an additional *guide table* $I$ [29,19,6]. The BLISS implementation we attack uses $I$ with 256 entries. The guide table stores for each $u \in \{0, \ldots, 255\}$ the smallest interval $I[u] = (a_u, b_u)$ such that $p_{a_u}^* \leq u/256$ and $p_{b_u}^* \geq (u+1)/256$. The first byte of $r$ is used to select $I[u]$ leading to a much smaller interval for the binary search. Effectively, $r$ is picked byte-by-byte, stopping once a unique value for $y$ is obtained. The CDT sampling algorithm with guide table is summarized in Algorithm 2.4.

**Algorithm 2.4** 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. Small table $I$ consisting of the 256 intervals

**Output:** Value $y \in [-\tau\sigma, \tau\sigma]$ sampled with probability according to $D_\sigma$

```
 1: pick a random byte $r$
 2: let $(I_{\min}, I_{\max}) = (a_r, b_r)$ be the left and right bounds of interval $I[r]$
 3: **if** $(I_{\max} - I_{\min} = 1)$:
 4:     generate a random sign bit $b \in \{0, 1\}$
 5:     **return** $y = (-1)^b I_{\min}$
 6: let $i = 1$ denote the index of the byte to look at
 7: pick a new random byte $r$
 8: **while** (1):
 9:     $I_z = \lfloor \frac{I_{\min} + I_{\max}}{2} \rfloor$
10:     **if** $(r > (i\text{th byte of } T[I_z]))$:
11:         $I_{\min} = I_z$
12:     **else if** $(r < (i\text{th byte of } T[I_z]))$:
13:         $I_{\max} = I_z$
14:     **else if** $(I_{\max} - I_{\min} = 1)$:
15:         generate a random sign bit $b \in \{0, 1\}$
16:         **return** $y = (-1)^b I_{\min}$
17:     **else**:
18:         increase $i$ by 1
19:         pick new random byte $r$
```

**Bernoulli sampling (Rejection sampling).** The basic idea behind rejection sampling is to sample a uniformly random integer $y \in [-\tau\sigma, \tau\sigma]$ and accept this sample with probability $\rho_\sigma(y)/\rho_\sigma(\mathbb{Z})$. For this, a uniformly random value $r \in [0, 1)$ is sampled and $y$ is accepted iff $r \leq \rho_\sigma(y)$. This method has two huge downsides: calculating the values of $\rho_\sigma(y)$ to high precision is expensive and the rejection rate can be quite high.

In the same paper introducing BLISS [9], the authors also propose a more efficient Bernoulli-based sampling algorithm. We recall the algorithms used (Algorithms 2.5, 2.6, 2.7), more details are given in the original work. We denote this method as *Bernoulli sampling* in the remainder of this paper.

**Algorithm 2.5** Sampling from $D^+_{K\sigma}$ for $K \in \mathbb{Z}$

```
Input: Target standard deviation $\sigma$, integer $K = \lfloor \frac{\sigma}{\sigma_2} + 1 \rfloor$, where $\sigma_2 = \frac{1}{2 \ln 2}$
Output: An integer $y \in \mathbb{Z}^+$ according to $D^+_{K\sigma_2}$
1: sample $x \in \mathbb{Z}$ according to $D^+_{\sigma_2}$
2: sample $z \in \mathbb{Z}$ uniformly in $\{0, \ldots, K-1\}$
3: $y \leftarrow Kx + z$
4: sample $b$ with probability $\exp\left(-z(z + 2Kx)/(2\sigma^2)\right)$
5: if $b = 0$ then restart
6: return $y$
```

**Algorithm 2.6** Sampling from $D_{K\sigma}$

```
Output: An integer $y \in \mathbb{Z}$ according to $D_{K\sigma_2}$
1: sample integer $y \leftarrow D^+_{K\sigma}$ (using Algorithm 2.5)
2: if $y = 0$ then restart with probability 1/2
3: generate random bit $b$ and return $(-1)^b y$
```

The basic idea is to first sample a value $x$, according to the binary discrete Gaussian distribution $D_{\sigma_2}$, where $\sigma_2 = \frac{1}{2 \ln 2}$ (Step 1 of Algorithm 2.5). This can be done efficiently using uniformly random bits [9]. The actual sample $y = Kx + z$, where $z \in \{0, \ldots, K-1\}$ is sampled uniformly at random and $K = \lfloor \frac{\sigma}{\sigma_2} + 1 \rfloor$, is then distributed according to the target discrete Gaussian distribution $D_\sigma$, by rejecting with a certain probability (Step 4 of Algorithm 2.5). The number of rejections in this case is much lower than in the original method. This step still requires computing a bit, whose probability is an exponential value. However, it can be done more efficiently using Algorithm 2.7, where $ET$ is a small table.

**2.4. Cache attacks.** The cache is a small bank of memory which exploits the temporal and the spatial locality of memory access to bridge the speed gap between the faster processor and the slower memory. The cache consists of *cache lines*, which, on modern Intel architectures, can store a 64-byte aligned *block* of memory of size 64 bytes.

**Algorithm 2.7** 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 $x = x_{\ell-1} \dots x_0$. Table ET with precomputed values $\mathrm{ET}[i] = \exp(-2^i/(2\sigma^2))$ for $0 \leq i \leq \ell - 1$
Output: A bit $b$ with probability $\exp(-x/(2\sigma^2))$ of being 1
1: for $i = \ell - 1$ to 0:
2:     if $x_i = 1$ then
3:         sample $A_i$ with probability $\mathrm{ET}[i]$.
4:         if $A_i = 0$ then return 0
5: return 1
```

In a typical processor there are several cache *levels*. At the top level, closest to the execution core, is the *L1 cache*, which is the smallest and the fastest of the hierarchy. Each successive level (L2, L3, etc.) is bigger and slower than the preceding level.

When the processor accesses a memory address it looks for the block containing the address in the L1 cache. In a *cache hit*, the block is found in the cache and the data is accessed. Otherwise, in a *cache miss*, the search continues on lower levels, eventually retrieving the memory block from the lower levels or from the memory. The cache then *evicts* a cache line and replaces its contents with the retrieved block, allowing faster future access to the block.

Because cache misses require searches in lower cache levels, they are slower than cache hits. Cache timing attacks exploit this timing difference to leak information [2,27,24,13,22]. In a nutshell, when an attacker uses the same cache as a victim, victim memory accesses change the state of the cache. The attacker can then use the timing variations to check which memory blocks are cached and from that deduce which memory addresses the victim has accessed. Ultimately, the attacker learns the cache line of the victim's table access: a range of possible values for the index of the access.

In this work we use the Flush+Reload attack [36,13]. A Flush+Reload attack uses the `clflush` instruction of the x86-64 architecture to evict a memory block from the cache. The attacker then lets the victim execute before measuring the time to access the memory block. If during its execution the victim has accessed an address within the block, the block will be cached and the attacker's access will be fast. If, however, the victim has not accessed the block, the attacker will reload the block from memory, and the access will take much longer. Thus, the attacker learns whether the victim accessed the memory block during its execution. The Flush+Reload attack has been used to attack implementations of RSA [36], AES [13,17], ECDSA [35,28] and other software [38,12].

## 3 Attack 1: CDT Sampling

This section presents the mathematical foundations of our cache attack on the CDT sampling. We first explain the phenomena we can observe from cache misses and hits in Algorithm 2.4 and then show how to exploit them to derive the secret signing key of BLISS using LLL. Sampling of the first noise polynomial $\mathbf{y} \in D_{\mathbb{Z}^n,\sigma}$ is done coefficientwise. Similarly the cache attack targets coefficients $y_i$ for $i = 0, \ldots, n-1$ independently.

**3.1. Weaknesses in cache.** Sampling from a discrete Gaussian distribution using both an interval table $I$ and a table with the actual values $T$, might leak information via cache memory. The best we can hope for is to learn the cache-lines of index $r$ of the interval and of index $I_z$ of the table lookup in $T$. Note that we cannot learn the sign of the sampled coefficient $y_i$. Also, the cache line of $T[I_z]$ always leaves a range of values for $|y_i|$. However, in some cases we can get more precise information combining cache-lines of table lookups in both tables. Here are two observations that narrow down the possibilities:

**Intersection:** We can intersect knowledge about the used index $r$ in $I$ with the knowledge of the access $T[I_z]$. Getting the cache-line of $I[r]$ gives a range of intervals, which is simply another (bigger) interval of possible values for sample $|y_i|$. If the values in the range of intervals are largely non-overlapping with the range of values learned from the access to $T[I_z]$, then the combination gives a much more precise estimate. For example: if the cache-line of $I[r]$ reveals that sample $|y_i|$ is in set $S_1 = \{0, 1, 2, 3, 4, 5, 7, 8\}$ and the cache-line of $T[I_z]$ reveals that sample $|y_i|$ must be in set $S_2 = \{7, 8, 9, 10, 11, 12, 13, 14, 15\}$, then by intersecting both sets we know that $|y_i| \in S_1 \cap S_2 = \{7, 8\}$, which is much more precise information.

**Last-Jump:** If the elements of an interval $I[r]$ in $I$ are divided over two cache-lines of $T$, we can sometimes track the search for the element to sample. If a small part of $I[r]$ is in one cache-line, and the remaining part of $I[r]$ is in another, we are able to distinguish if this small part has been accessed. For example, interval $I[r] = \{5, 6, 7, 8, 9\}$ is divided over two cache-lines of $T$: cache-line $T_1 = \{0, 1, 2, 3, 4, 5, 6, 7\}$ and line $T_2 = \{8, 9, 10, 11, 12, 13, 14, 15\}$. The binary search starts in the middle of $I[r]$, at value 7, which means line $T_1$ is always accessed. However, only for values $\{8, 9\}$ also line $T_2$ is accessed. So if both lines $T_1$ *and* $T_2$ are accessed, we know that sample $|y_i| \in \{8, 9\}$.

We will restrict ourselves to only look for cache access patterns that give even more precision, at the expense of requiring more signatures:

1. The first restriction is to only look at cache weaknesses (of type Intersection or Last-Jump), in which the number of possible values for sample $|y_i|$ is two. Since we do a binary search within an interval, this is the most precision one can get (unless an interval is unique): after the last comparisons (table lookup in $T$), one of two values will be returned. This means that by picking either of these two values we limit the error of $|y_i|$ to at most 1.
2. The probabilities of sampling values using CDT sampling with guide table $I$ are known to match the following probability requirement :

$$\sum_{r=0}^{255} \mathbb{P}[X = x \mid X \in I[r]] = \frac{\rho_\sigma(x)}{\rho_\sigma(\mathbb{Z})}. \tag{1}$$

Due to the above condition, it is possible that adjacent intervals are partially overlapping. That is, for some $r, s$ we have that $I[r] \cap I[s] \neq \emptyset$. In practice, this only happens for $r = s + 1$, meaning adjacent intervals might overlap. For example, if the probability of sampling $x$ is greater than $1/256$, then $x$ has to be an element in at least two intervals $I[r]$. Because of this, it is possible that for certain parts of an interval $I[r]$, there is a biased outcome of the sample.

The second restriction is to only consider cache weaknesses for which additionally one of the two values is significantly more likely to be sampled, i.e., if $|y_i| \in \{\gamma_1, \gamma_2\} \subset I[r]$ is the outcome of cache access patterns, then we further insist on

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

So we search for values $\gamma_1$ so that $\mathbb{P}[|y_i| = \gamma_1 \mid |y_i| \in \{\gamma_1, \gamma_2\} \subset I[r]] = 1 - \alpha$ for small $\alpha$, which also matches access patterns for the first restriction. Then, if we observe a matching access pattern, it is safe to assume the outcome of the sample is $\gamma_1$.

3. The last restriction is to only look at cache-access patterns, which reveal that $|y_i|$ is larger than $\beta \cdot \mathbb{E}[\langle \mathbf{s}, \mathbf{c} \rangle]$, for some constant $\beta \geq 1$, which is an easy calculation using the distributions of $\mathbf{s}, \mathbf{c}$. If we use this restriction in our attack targeted at coefficient $y_i$ of $\mathbf{y}$, we learn the sign of $|y_i|$ by looking at the sign of coefficient $z_i$ of $\mathbf{z}$, since:

$$\operatorname{sign}(y_i) \neq \operatorname{sign}(z_i) \leftrightarrow \langle \mathbf{s}, \mathbf{c} \rangle > (y_i + z_i)$$

So by requiring that $|y_i|$ must be larger than the expected value of $\langle \mathbf{s}, \mathbf{c} \rangle$, we expect to learn the sign of $y_i$. We therefore omit the absolute value sign in $|y_i|$ and simply write that we learn $y_i \in \{\gamma_1, \gamma_2\}$, where the $\gamma$'s took over the sign of $y_i$ (which is the same as the sign of $z_i$).

There is some flexibility in these restrictions, in choosing parameters $\alpha, \beta$. Choosing these parameters too restrictively, might lead to no remaining cache-access patterns, choosing them too loosely makes other parts fail.

In the last part of the attack described next, we use LLL to calculate short vectors of a certain (random) lattice we create using BLISS signatures. We noticed that LLL works very well on these lattices, probably because the basis used is sparse. This implies that the vectors are already relatively short and orthogonal. The parameter $\alpha$ determines the shortness of the vector we look for, and therefore influences if an algorithm like LLL finds our vector. For the experiments described in Section 5, we required $\alpha \leq 0.1$. This made it possible for every parameter set we used in the experiments to always have at least one cache-access pattern to use.

Parameter $\beta$ influences the probability that one makes a huge mistake when comparing the values of $y_i$ and $z_i$. However, for the parameters we used in the experiments, we did not find recognizable cache-access patterns which correspond to small $y_i$. This means, we did not need to use this last restriction to reject certain cache-access patterns.

**3.2. Exploitation.** For simplicity, we assume we have one specific cache access pattern, which reveals if $y_i \in \{\gamma_1, \gamma_2\}$ for $i = 0, \ldots, n-1$ of polynomial $\mathbf{y}$, and if this is the case, $y_i$ has probability $(1-\alpha)$ to be value $\gamma_1$, with small $\alpha$. In practice however, there might be more than one cache weakness, satisfying the above requirements. This would allow the attacker to search for more than one cache access pattern done by the victim. For the attack, we assume the victim is creating $N$ signatures[3] $(\mathbf{z}_j, \mathbf{c}_j)$ for $j = 1, \ldots, N$, and an attacker is gathering these signatures with associated cache information for noise polynomial $\mathbf{y}_j$. We assume the attacker can search for the specific cache access pattern, for which he can determine if $y_{ji} \in \{\gamma_1, \gamma_2\}$. For the cases revealed by cache access patterns, the attacker ends up with the following equation:

$$z_{ji} = y_{ji} + (-1)^{b_j} \langle \mathbf{s}, \mathbf{c}_{ji} \rangle, \tag{2}$$

where the attacker knows coefficient $z_{ji}$ of $\mathbf{z}_j$, rotated coefficient vectors $\mathbf{c}_{ji}$ of challenge $\mathbf{c}_j$ (both from the signatures) and $y_{ji} \in \{\gamma_1, \gamma_2\}$ of noise polynomial $\mathbf{y}_j$ (from the side-channel attack). Unknowns to the attacker are bit $b_j$ and $\mathbf{s}$.

If $z_{ji} = \gamma_1$, the attacker knows that $\langle \mathbf{s}, \mathbf{c}_{ji} \rangle \in \{0, 1, -1\}$. Moreover, with high probability $(1-\alpha)$ the value will be 0, as by the second restriction $y_{ji}$ is biased to be value $\gamma_1$. So if $z_{ji} = \gamma_1$, the attacker adds $\xi_k = \mathbf{c}_{ji}$ to a list of *good* vectors. The restriction $z_{ji} = \gamma_1$ means that the attacker will in some cases not use the information in Equation (2), although he knows that $y_{ji} \in \{\gamma_1, \gamma_2\}$.

When the attacker collects enough of these vectors $\xi_k = \mathbf{c}_{ji}; 0 \le i \le n-1, 1 \le j \le N, 1 \le k \le n$, he can build a matrix $\mathbf{L} \in \{-1, 0, 1\}^{n \times n}$, whose columns are the $\xi_k$'s. This matrix satisfies:

$$\mathbf{sL} = \mathbf{v} \tag{3}$$

for some unknown but short vector $\mathbf{v}$. The attacker does not know $\mathbf{v}$, so he cannot simply solve for $\mathbf{s}$, but he does know that $\mathbf{v}$ has norm about $\sqrt{\alpha n}$, and lies in the lattice spanned by the rows of $\mathbf{L}$. He can use a lattice reduction algorithm, like LLL, on $\mathbf{L}$ to search for $\mathbf{v}$. LLL also outputs the unimodular matrix $\mathbf{U}$ satisfying $\mathbf{UL} = \mathbf{L}'$. The attack tests for each row of $\mathbf{U}$ (and its rotations) whether it is sparse and could be a candidate for $\mathbf{s} = \mathbf{f}$. As stated before, correctness of a secret key guess can be verified using the public key.

This last step does not always succeed, just with high probability. To make sure the attack succeeds, this process is randomized. Instead of collecting exactly $n$ vectors $\xi_k = \mathbf{c}_{ji}$, we gather $m > n$ vectors, and pick a random subset of $n$ vectors as input for LLL. While we do not have a formal analysis of the success probability, experiments (see Section 5) confirm that this method works and succeeds in finding the secret key (or its negative) in few rounds of randomization.

A summary of the attack is given in Algorithm 3.1.

[3] Here $\mathbf{z}_j$ refers to the first signature polynomial $\mathbf{z}_{j1}$ of the $j$th signature $(\mathbf{z}_{j1}, \mathbf{z}_{j2}^{\dagger}, \mathbf{c}_j)$.

**Algorithm 3.1** Cache-attack on BLISS with CDT Sampling

```
Input: Access to cache memory of a victim with a key-pair $(\mathbf{A}, \mathbf{S})$. Input parameters
    $n, \sigma, q, \kappa$ of BLISS. Access to signature polynomials $(\mathbf{z}_1, \mathbf{z}_2^\dagger, \mathbf{c})$ produced using $\mathbf{S}$.
    Victim uses CDT sampling with tables $T, I$ for noise polynomials $\mathbf{y}$. Cache weakness
    that allows to determine if coefficient $y_i \in \{\gamma_1, \gamma_2\}$ of $\mathbf{y}$, and when this is the case,
    the value of $y_i$ is biased towards $\gamma_1$
Output: Secret key $\mathbf{S}$
 1: let $k = 0$ be the number of vectors collected so far and let $M = []$ be an empty list
    of vectors
 2: while $(k < m)$:        // collect $m$ vectors $\xi_k$ before randomizing LLL
 3:     collect signature $(\mathbf{z}_1, \mathbf{z}_2^\dagger, \mathbf{c})$, together with cache information for each coeffi-
        cient $y_i$ of noise polynomial $\mathbf{y}$
 4:     for each $i = 0, \ldots, n-1$:
 5:         if $y_i \in \{\gamma_1, \gamma_2\}$ (determined via cache information) and $z_{1i} = \gamma_1$:
 6:           add vector $\xi_k = \mathbf{c}_i$ to $M$ and set $k = k + 1$
 7: while (1):
 8:     choose random subset of $n$ vectors from $M$ and construct matrix $\mathbf{L}$ whose
        columns are those vectors from $M$
 9:     perform LLL basis reduction on $\mathbf{L}$ to get: $\mathbf{UL} = \mathbf{L}'$, where $\mathbf{U}$ is a unimodular
        transformation matrix and $\mathbf{L}'$ is LLL reduced
10:     for each $j = 1, \ldots, n$:
11:         check if row $\mathbf{u}_j$ of $\mathbf{U}$ has the same distribution as $\mathbf{f}$ and if $(\mathbf{a}_1/2) \cdot$
            $\mathbf{u}_j \bmod 2q$ has the same distribution as $2\mathbf{g}+1$. Lastly verify if $\mathbf{a}_1 \cdot \mathbf{u}_j +$
            $\mathbf{a}_2 \cdot (\mathbf{a}_1/2) \cdot \mathbf{u}_j \equiv q \bmod 2q$
12:         return $\mathbf{S} = (\mathbf{u}_j, (\mathbf{a}_1/2) \cdot \mathbf{u}_j \bmod 2q)$ if this is the case
```

## 4 Attack 2: Bernoulli Sampling

In this section, we discuss the foundations and strategy of our second cache attack on the Bernoulli-based sampler (Algorithms 2.5, 2.6, and 2.7). We show how to exploit the fact that this method uses a small table ET, leaking very precise information about the sampled value.

**4.1. Weaknesses in cache.** The Bernoulli-sampling algorithm described in Section 2.3 uses a table with exponential values $\mathrm{ET}[i] = \exp(-2^i/(2\sigma^2))$ and inputs of bit-size $\ell = O(\log K)$, which means this table is quite small. Depending on bit $i$ of input $x$, line 3 of Algorithm 2.7 is performed, requiring a table look-up for value $\mathrm{ET}[i]$. In particular when input $x = 0$, no table look-up is required. An attacker can detect this event by examining cache activity of the sampling process. If this is the case, it means that the sampled value $z$ equals 0 in Step 2 of Algorithm 2.5. The possible values for the result of sampling are $y \in \{0, \pm K, \pm 2K, \ldots\}$. So for some cache access patterns, the attacker is able to determine if $y \in \{0, \pm K, \pm 2K, \ldots\}$.

**4.2. Exploitation.** We will use the same methods as described in Section 3.2, but now we know that for a certain cache access pattern the coefficient $y_i \in \{0, \pm K, \pm 2K, \ldots\}$, $i = 0, \ldots, n-1$, of the noise polynomial $\mathbf{y}$. If $\max |\langle \mathbf{s}, \mathbf{c} \rangle| \leq \kappa < K$, (which is something anyone can check using the public parameters and which holds for typical implementations), we can determine $y_i$ completely using the knowledge of signature vector $\mathbf{z}$. When more signatures[4] $(\mathbf{z}_j, \mathbf{c}_j); j = 1, \ldots, N$ are created, the attacker can search for the specific access pattern and verify whether $y_{ji} \in \{0, \pm K, \pm 2K, \ldots\}$, where $y_{ji}$ is the $i$'th coefficient of noise polynomial $\mathbf{y}_j$.

If the attacker knows that $y_{ji} \in \{0, \pm K, \pm 2K, \ldots\}$ and it additionally holds that $z_{ji} = y_{ji}$, where $z_{ji}$ is the $i$'th coefficient of signature polynomial $\mathbf{z}_j$, he knows that $\langle \mathbf{s}, \mathbf{c}_{ji} \rangle = 0$. If this is the case, the attacker includes coefficient vector $\zeta_k = \mathbf{c}_{ji}$ in the list of *good* vectors. Also for this attack the attacker will discard some known $y_{ji}$ if it does not satisfy $z_{ji} = y_{ji}$.

Once the attacker has collected $n$ of these vectors $\xi_k = \mathbf{c}_{ji}; 0 \leq i \leq n-1, 1 \leq j \leq N, 1 \leq k \leq n$, he can form a matrix $\mathbf{L} \in \{-1, 0, 1\}^{n \times n}$, whose columns are the $\xi_k$'s, satisfying $\mathbf{sL} = 0$, where $0$ is the all-zero vector. With very high probability, the $\xi_k$'s have no dependency other than introduced by $\mathbf{s}$. This means $\mathbf{s}$ is the only kernel vector. Note the subtle difference with Equation (3): we do not need to randomize the process, because we know the right-hand side is the all-zero vector. The attack procedure is summarized in Algorithm 4.1.

**Algorithm 4.1** Cache-attack on BLISS with Bernoulli sampling

```
Input: Access to cache memory of victim with a key-pair $(\mathbf{A}, \mathbf{S})$. Input parameters
    $n, \sigma, q, \kappa$ of BLISS, with $\kappa < K$. Access to signatures $(\mathbf{z}_1, \mathbf{z}_2^\dagger, \mathbf{c})$ produced using $\mathbf{S}$.
    Victim uses Bernoulli sampling with the small exponential table to sample noise
    polynomial $\mathbf{y}$
Output: Secret key $\mathbf{S}$
 1: let $k = 0$ be the number of vectors gained so far and let $M = []$ be an empty list
    of vectors
 2: while($k < n$):
 3:     collect signature $(\mathbf{z}_1, \mathbf{z}_2^\dagger, \mathbf{c})$ together with cache information for each coeffi-
        cient $y_i$ of noise polynomial $\mathbf{y}$
 4:     for each $i = 1, \ldots, n$ do:
 5:         if $y_i \in \{0, \pm K, \pm 2K, ..\}$ (according to cache information), and $z_{1i} = y_i$
            then add coefficient vector $\xi_k = \mathbf{c}_i$ as a column to $M$ and set $k = k + 1$
 6: form a matrix $\mathbf{L}$ from the columns in $M$. Calculate kernel space of $\mathbf{L}$. This gives a
    matrix $\mathbf{U} \in \mathbb{Z}^{\ell \times n}$ such that $\mathbf{UL} = 0$, where $0$ is the all-zero matrix
 7: for each $j = 1, \ldots, \ell$ do:        // we expect $\ell = 1$
 8:     check if row $\mathbf{u}_j$ of $\mathbf{U}$ has the same distribution as $\mathbf{f}$ and if $(\mathbf{a}_1/2) \cdot$
        $\mathbf{u}_j$ mod $2q$ has the same distribution as $2\mathbf{g} + 1$. Lastly verify if $\mathbf{a}_1 \cdot \mathbf{u}_j +$
        $\mathbf{a}_2 \cdot (\mathbf{a}_1/2) \cdot \mathbf{u}_j \equiv q \bmod 2q$
 9:     return $\mathbf{S} = (\mathbf{u}_j, (\mathbf{a}_1/2) \cdot \mathbf{u}_j \bmod 2q)$ if this is the case
10: remove a random entry from $M$, put $k = k - 1$, goto step 2
```

**4.3. Possible extensions.** One might ask why we not always use the knowledge of $y_{ji}$, since we can completely determine its value, and work with a non-zero

[4] Again, $\mathbf{z}_j$ refers to the first signature polynomial $\mathbf{z}_{j1}$ of the $j$th signature $(\mathbf{z}_{j1}, \mathbf{z}_{j2}^\dagger, \mathbf{c}_j)$.

right-hand side. Unfortunately, bits $b_j$ from Equation 2 of the signatures are unknown. This means an attacker has to use a linear solver $2^N$ times, where $N$ is the number of required signatures (grouping columns appropriately if they come from the same signature). For large $N$ this becomes infeasible and $N$ is typically on the scale of $n$. By requiring that $z_{ji} = y_{ji}$, we remove the unknown bit $b_j$ from the Equation (2).

Similar to the first attack, an attacker might also use vectors $\xi_k = \mathbf{c}_{ji}$, where $\langle \mathbf{s}, \mathbf{c}_{ji} \rangle \in \{-1, 0, 1\}$, in combination with LLL and possibly randomization. This approach might help if fewer signatures are available, but the easiest way is to require exact knowledge, which comes at the expense of needing more signatures, but has a very fast and efficient offline part. Section 6.3 deals with this approximate information.

## 5 Results with a Perfect Side-Channel

In this section we provide experimental results, where we assume the attacker has access to a perfect side-channel: no errors are made in measuring the table accesses of the victim. We apply the attack strategies discussed in the previous two sections and show how many signatures are required for each strategy.

**5.1. Attack setting.** Sections 3 and 4 outline the basic ideas behind cache attacks against the two sampling methods for noise polynomials $\mathbf{y}$ used in the target implementation of BLISS. We now consider the following idealized situation: the victim is signing random messages and an attacker collects these signatures. The attacker knows the exact cache-lines of the table look-ups done by the victim while computing the noise vector $\mathbf{y}$. We assume cache-lines have size 64 bytes and each element is 8 bytes large (type LONG). To simplify exposition, we assume the cache-lines are divided such that element $i$ of any table is in cache-line $\lfloor i/8 \rfloor$.

Our test machine is an AMD FX-8350 Eight-Core CPU running at 4.1 GHz. We use the *research oriented* C++ implementation of BLISS, made available by the authors on their webpage [8]. Both of the analyzed sampling methods are provided by the implementation, where the tables $T, I$ and ET are constructed dependent on $\sigma$. We use the NTL library [33] for LLL and kernel calculations.

The authors of BLISS [9] proposed several parameter sets for the signature scheme (see full version [4, Table A.1]). We present attacks against all combinations of parameter sets and sampling methods; the full results of the perfect side-channel attacks are given in the full version [4, Appendix B].

**5.2. CDT sampling.** When the signing algorithm uses CDT sampling as described in Algorithm 2.4, the perfect side-channel provides the values of $\lfloor r/8 \rfloor$ and $\lfloor I_z/8 \rfloor$ of the table accesses for $r$ and $I_z$ in tables $I$ and $T$. We apply the attack strategy of Section 3.

We first need to find cache-line patterns, of type intersection or last-jump, which reveal that $|y_i| \in \{\gamma_1, \gamma_2\}$ and $\mathbb{P}[|y_i| = \gamma_1 |\ |y_i| \in \{\gamma_1, \gamma_2\}] = 1 - \alpha$ with $\alpha \leq 0.1$. One way to do that is to construct two tables: one table that lists elements $I[r]$, that belong to certain cache-lines of table $I$, and one table that lists the accessed elements $I_z$ inside these intervals $I[r]$, that belong to certain cache-lines of table $T$. We can then brute-force search for all cache weaknesses of type intersection or last-jump. For example, in BLISS-I the first eight elements of $I$ (meaning $I[0], \ldots, I[7]$) belong to the first cache-line of $I$, but for the elements in $I[7] = \{7, 8\}$, the sampler accesses element $I_z = 8$, which is part of the second cache-line of $T$. This is an intersection weakness: if the first cache-line of $I$ is accessed and the second cache-line of $T$ is accessed, we know $y_i \in \{7, 8\}$. Similarly, one can find last-jump weaknesses, by searching for intervals $I[r]$ that access multiple cache-lines of $T$. Once we have these weaknesses, we need to use the biased restriction with $\alpha \leq 0.1$. This can be done by looking at all bytes except the first of the entry $T[I_z]$ (this is already used to determine interval $I[r]$). If we denote the integer value of these 7 bytes by $T[I_z]_{\text{byte}\neq 1}$, then we need to check if $T[I_z]$ has property

$$(T[I_z])_{\text{byte}\neq 1}/(2^{56} - 1) \leq \alpha$$

(or $(T[I_z])_{\text{byte}\neq 1}/(2^{56} - 1) \geq (1 - \alpha)$). If one of these properties holds, then we have $y_i \in \{I_z - 1, I_z\}$ and $\mathbb{P}[|y_i| = I_z| \; |y_i| \in \{I_z - 1, I_z\}] = 1 - \alpha$ (or with $I_z$ and $I_z - 1$ swapped). For each set of parameters we found at least one of these weaknesses using the above method (see the full version [4, Table B.1] for the values).

We collect $m$ (possibly rotated) coefficient vectors $\mathbf{c}_j$ and then run LLL at most $t = 2(m - n) + 1$ times, each time searching for $\mathbf{s}$ in the unimodular transformation matrix using the public key. We consider the experiment failed if the secret key is not found after this number of trials; the randomly constructed lattices have a lot of overlap in their basis vectors which means that increasing $t$ further is not likely to help. We performed 1000 repetitions of each experiment (different parameters and sizes for $m$) and measured the success probability $p_{\text{succ}}$, the average number of required signatures $\overline{N}$ to retrieve $m$ usable challenges, and the average length of $\mathbf{v}$ if it was found. The expected number of required signatures $\mathbb{E}[N]$ is also given, as well as the running time for the LLL trials. This expected number of required signatures can be computed as:

$$\mathbb{E}[N] = \frac{m}{n \cdot \mathbb{P}[\text{CP}] \cdot \mathbb{P}[\langle \mathbf{s}_1, \mathbf{c} \rangle = 0]},$$

where CP is the event of a usable cache-access pattern for a coordinate of $\mathbf{y}$.

From the results (given in the full version [4, Table B.2]) we see that, although BLISS-0 is a toy example (with security level $\lambda \leq 60$), it requires the largest average number $\overline{N}$ of signatures to collect $m$ columns, i.e., before the LLL trials can begin. This illustrates that the cache-attack depends less on the dimension $n$, but mainly on $\sigma$. For BLISS-0 with $\sigma = 100$, there is only one usable cache weakness with the restrictions we made.

For all cases, we see that a small increase of $m$ greatly increases the success probability $p_{\text{succ}}$. The experimental results suggest that picking $m \approx 2n$ suffices to get a success probability close to 1.0. This means that one only needs more signatures to always succeed in the offline part.

**5.3. Bernoulli sampling.** When the signature algorithm uses Bernoulli sampling from Algorithm 2.6, a perfect side-channel determines if there has been a table access in table ET. Thus, we can apply the attack strategy given in Section 4. We require $m = n$ (possibly rotated) challenges $\mathbf{c}_i$ to start the kernel calculation. We learn whether any element has been accessed in table ET, e.g., by checking the cache-lines belonging to the small part of the table. We performed only 100 experiments this time, since we noticed that $p_{\text{succ}} = 1.0$ for all parameter sets with a perfect side-channel. This means that the probability that $n$ random challenges $\mathbf{c}$ are linearly independent is close to 1.0. We state the average number $\overline{N}$ of required signatures in the full version [4, Table B.3]. This time, the expected number is simply:

$$\mathbb{E}[N] = \left( \left( \frac{1}{\rho_\sigma(\mathbb{Z})} \sum_{x=-\lfloor \tau\sigma/K \rfloor}^{\lfloor \tau\sigma/K \rfloor} \rho_\sigma(xK) \right) \cdot \mathbb{P}[\langle \mathbf{s}_1, \mathbf{c} \rangle = 0] \right)^{-1}$$

for $K = \lfloor \frac{\sigma}{\sigma_2} + 1 \rfloor$ and tail-cut $\tau \geq 1$. Note that the number of required signatures is smaller for BLISS-II than for BLISS-I. This might seem surprising as one might expect it to increase or be about the same as BLISS-I because the dimensions and security level are the same for these two parameter sets. However, $\sigma$ is chosen a lot smaller in BLISS-II, which means that also value $K$ is smaller. This influences $\overline{N}$ significantly as the probability to sample values $xK$ is larger for small $\sigma$.

## 6 Proof-of-Concept Implementation

So far, the experimental results were based on the assumption of a perfect side-channel: we assumed that we would get the cache-line of every table look-up in the CDT sampling and Bernoulli sampling. In this section, we reduce the assumption and discuss the results of more realistic experiments using the Flush+Reload technique.

When moving to real hardware some of the assumptions made in Section 5 no longer hold. In particular, allocation does not always ensure that tables are aligned at the start of cache lines and processor optimizations may pre-load memory into the cache, resulting in false positives. One such optimization is the *spatial prefetcher*, which pairs adjacent cache lines into 128-byte chunks and prefetches a cache line if an access to its pair results in a cache miss [16].

**6.1. Flush+Reload on CDT sampling.** Due to the spatial prefetcher, Flush+Reload cannot be used consistently to probe two paired cache lines. Consequently, to determine access to two consecutive CDT table elements, we must use a pair that spans two unpaired cache lines. In the full version [4, Table C.3], we show that when the CDT table is aligned at 16 bytes, we can always find such a pair for BLISS-I. Although this is not a proof that our attack works in all scenarios, i.e. for all $\sigma$ and all offsets, it would also not be a solid defence to pick exactly those scenarios for which our attack would not work, e.g., because $\alpha$ could be increased.

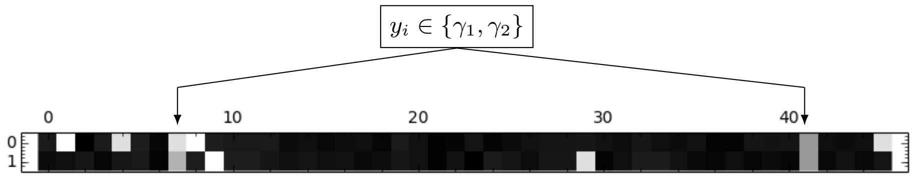

Fig. 6.1: Visualization of Flush+Reload measurements of table look-ups for BLISS-I using CDT sampling with guide table $I$. Two locations in memory are probed, denoted in the vertical axis by $0, 1$, and they represent two adjacent cache-lines. For interval $I[51] = [54, 57]$, there is a last-jump weakness for $\{\gamma_1, \gamma_2\} = \{55, 56\}$, where the outcome of $|y_i|$ is biased towards $\gamma_1 = 55$ with $\alpha = 0.0246$. For each coordinate (the horizontal axis), we get a response time for each location we probe: dark regions denote a long response time, while lighter regions denote a short response time. When both of the probed locations give a fast response, it means the victim accessed both cache-lines for sampling $y_i$. In this case the attacker knows that $|y_i| \in \{55, 56\}$; here for $i = 8$ and $i = 41$.

The attack was carried out on an HP Elite 8300 with an i5-3470 processor. running CentOS 6.6. Before sampling each coordinate $y_i$, for $i = 0, \ldots, n-1$, we flush the monitored cache lines using the `clflush` instruction. After sampling the coordinate, we reload the monitored cache lines and measure the response time. We compare the response times to a pre-defined threshold value to determine whether the cache lines were accessed by the sampling algorithm.

A visualization of the Flush+Reload measurements for CDT sampling is given in Figure 6.1. Using the intersection and last-jump weakness of the CDT sampler in cache-memory, we can determine which value is sampled by the victim by probing two locations in memory. To reduce the number of false positives, we focus on one of the weaknesses (given in the full version [4, Table B.1]) as a target for the Flush+Reload. This means that the other weaknesses are not detected and we need to observe more signatures than with a perfect side-channel, before we collect enough columns to start with the offline part of the attack.

We executed 50 repeated attacks against BLISS-I, probing the last-jump weakness for $\{\gamma_1, \gamma_2\} = \{55, 56\}$. We completely recovered the private key in 46 out of the 50 cases. On average we require 3438 signatures for the attack, to collect $m = 2n = 1024$ equations. We tried LLL five times after the collection and considered the experiment a failure if we did not find the secret key in these five times. We stress that this is not the optimal strategy to minimize the number of required signatures or to maximize the success probability. However, it is an indication that this proof-of-concept attack is feasible.

**6.2. Other processors.** We also experimented with a newer processor (Intel core i7-5650U) and found that this processor has a more aggressive prefetcher. In particular, memory locations near the start and the end of the page are more likely to be prefetched. Consequently, the alignment of the tables within the page can affect the attack success rate. We find that in a third of the locations within a page the attack fails, whereas in the other two thirds it succeeds with probabilities similar to those on the older processor. We note that, as demonstrated in the full version [4, Table B.1], there are often multiple weaknesses in the CDT. While some weaknesses may fall in unexploitable memory locations, others may still be exploitable.

**6.3. Flush+Reload on Bernoulli sampling.** For attacking BLISS using Bernoulli sampling, we need to measure if table ET has been accessed at all. Due to the spatial prefetcher we are unable to probe all of the cache lines of ET. Instead, we flush all cache lines containing ET before sampling and reload only even cache lines after the sampling. Flushing even cache lines is required for the Flush+Reload attack. We flush the odd cache lines to trigger the spatial prefetcher, which will prefetch the paired even cache lines when the sampling accesses an odd cache line. Thus, flushing all of the cache lines gives us a complete coverage of the table even though we only reload half of the cache lines.

Since we do not get error-free side-channel information, we are likely to collect some $\mathbf{c}$ with $\langle \mathbf{s}, \mathbf{c}_i \rangle \neq 0$ as columns in $\mathbf{L}$. Instead of computing the kernel (as in the idealized setting) we used LLL (as in CDT) to handle small errors and we gathered more than $n$ columns and randomized the selection of $\mathbf{L}$.

We tested the attack on a MacBook air with the newer processor (Intel core i7-5650U) running Mac OS X El Capitan. We executed 50 repeated attacks against BLISS-I, probing three out of the six cache lines that cover the ET table. We completely recovered the private key in 44 of these samples. On average we required 3294 signatures for the attack to collect $m = n + 100 = 612$ equations. The experiment is considered a failure if we did not find the secret key after trying LLL five times.

**6.4. Conclusion.** Our proof-of-concept implementation demonstrates that in many cases we can overcome the limitations of processor optimizations and perform the attack on BLISS. The attack, however, requires a high degree of synchronization between the attacker and the victim, which we achieve by modifying the victim code. For a similar level of synchronization in a real attack scenario, the attacker will have to be able to find out when each coordinate is sampled. One possible approach for achieving this is to use the attack of Gullasch et al. [13] against the Linux Completely Fair Scheduler. The combination of a cache attack with the attack on the scheduler allows the attacker to monitor each and every table access made by the victim, which is more than required for our attacks.

## 7 Discussion of Candidate Countermeasures

In this paper we presented cache attacks on two different discrete Gaussian samplers. In the following we discuss some candidate countermeasures against our specific attacks but note that other attacks might still be possible. A standard countermeasure against cache-attacks are constant-time accesses.

Constant-time table accesses, meaning accessing *every element of the table for every coordinate* of the noise vector, were also discussed (and implemented) by Bos et al. [3] for key exchange. This increased the number of table accesses by about two orders of magnitude. However, in the case of signatures the tables are much larger than for key exchange: a much larger standard deviation for the discrete Gaussian distribution is required. For 128 bits of security, a standard deviation $\sigma = 8/\sqrt{2\pi} \approx 3.19$ suffices for key exchange, resulting in a table size of 52 entries. In contrast, BLISS-I uses a standard deviation of $\sigma = 215$, resulting in a table size of 2580 entries. It therefore seems that this countermeasure induces significant overhead for signatures: at least as much as for the key exchange. It might be the case that constant-time accesses to a certain part of the table is already sufficient as a countermeasure against our attack, but it is unclear how to do this precisely. One might think that constant-time accesses to table $I$ in the CDT sampler is already sufficient as a countermeasure. In this case, the overhead is somewhat smaller, since $I$ contains 256 entries. However, the last-jump weakness only uses the knowledge of accesses in the $T$ table, which is still accessible in that case.

In the case of the Bernoulli-based sampler, doing constant-time table accesses does not induce that much overhead: the size of table ET is about $\ell \approx 2 \log K$. This means swapping line 2 and 3 of Algorithm 2.7 might prevent our attack as all elements of ET are always accessed. Note that removing line 4 of Algorithm 2.7 (and returning 0 or 1 at the end of the loop) does not help as a countermeasure. It does make the sampler constant-time, but we do not exploit that property. We exploit the fact that table accesses occur, depending on the input.

A concurrent work by Saarinen [32] discusses another candidate countermeasure: the VectorBlindSample procedure. The VectorBlindSample procedure basically samples $m$ vectors of discrete Gaussian values with a smaller standard deviation, shuffles them in between, and adds the results. The problem of directly applying our attack is that we need side-channel information of all summands for a coefficient. The chances for this are quite small. However, it does neither mean that other attacks are not possible nor that it is impossible to adapt our attack.

## 8 Acknowledgements

The authors would like to thank Daniel J. Bernstein and Léo Ducas for fruitful discussions and suggestions.

## References

1. E. Alkim, L. Ducas, T. Pöppelmann, and P. Schwabe. Post-quantum key exchange – a new hope. IACR Cryptology ePrint Archive 2015/1092, 2015.
2. Daniel J. Bernstein. Cache-timing attacks on AES, 2005. Preprint available at http://cr.yp.to/antiforgery/cachetiming-20050414.pdf.

3. J. W. Bos, C. Costello, M. Naehrig, and D. Stebila. Post-quantum key exchange for the TLS protocol from the ring learning with errors problem. In *S&P 2015*, pages 553–570. IEEE Computer Society, 2015.
4. Leon Groot Bruinderink, Andreas Hülsing, Tanja Lange, and Yuval Yarom. Flush, Gauss, and Reload - A cache attack on the BLISS lattice-based signature scheme. IACR Cryptology ePrint Archive 2016/300, 2016.
5. J. A. Buchmann, D. Cabarcas, F. Göpfert, A. Hülsing, and P. Weiden. Discrete Ziggurat: A time-memory trade-off for sampling from a Gaussian distribution over the integers. In T. Lange, K. E. Lauter, and P. Lisonek, editors, *SAC 2013*, volume 8282 of *LNCS*, pages 402–417. Springer, 2014.
6. H.-C. Chen and Y. Asau. On generating random variates from an empirical distribution. *AIIE Transactions*, 6(2):163–166, 1974.
7. L. Chen, Y.-K. Liu, S. Jordan, D. Moody, R. Peralta, R. Perlner, and D. Smith-Tone. Report on post-quantum cryptography. NISTIR 8105, Draft, February 2016.
8. L. Ducas, A. Durmus, T. Lepoint, and V. Lyubashevsky. BLISS: Bimodal Lattice Signature Schemes, 2013. http://bliss.di.ens.fr/.
9. L. Ducas, A. Durmus, T. Lepoint, and V. Lyubashevsky. Lattice Signatures and Bimodal Gaussians. In R. Canetti and J. A. Garay, editors, *CRYPTO 2013, Part I*, volume 8042 of *LNCS*, pages 40–56. Springer, 2013.
10. N. C. Dwarakanath and S. D. Galbraith. Sampling from discrete Gaussians for lattice-based cryptography on a constrained device. *Appl. Algebra Eng. Commun. Comput.*, 25(3):159–180, 2014.
11. C. Gentry, C. Peikert, and V. Vaikuntanathan. Trapdoors for hard lattices and new cryptographic constructions. In C. Dwork, editor, *STOC 2008*, pages 197–206. ACM, 2008.
12. D. Gruss, R. Spreitzer, and S. Mangard. Cache template attacks: Automating attacks on inclusive last-level caches. In J. Jung and T. Holz, editors, *USENIX Security 15*, pages 897–912. USENIX Association, 2015.
13. D. Gullasch, E. Bangerter, and S. Krenn. Cache games – bringing access-based cache attacks on AES to practice. In *S&P 2011*, pages 490–505. IEEE Computer Society, 2011.
14. T. Güneysu, V. Lyubashevsky, and T. Pöppelmann. Practical lattice-based cryptography: A signature scheme for embedded systems. In E. Prouff and P. Schaumont, editors, *CHES 2012*, volume 7428 of *LNCS*, pages 530–547. Springer, 2012.
15. J. Hoffstein, J. Pipher, and J. H. Silverman. NTRU: A ring-based public key cryptosystem. In J. Buhler, editor, *ANTS-III*, volume 1423 of *LNCS*, pages 267–288. Springer, 1998.
16. Intel Corporation. *Intel 64 and IA-32 Architectures Optimization Reference Manual*, April 2012.
17. G. Irazoqui, M. S. Inci, T. Eisenbarth, and B. Sunar. Wait a minute! a fast, cross-VM attack on AES. In A. Stavrou, H. Bos, and G. Portokalidis, editors, *RAID 2014*, volume 8688 of *LNCS*, pages 299–319. Springer, 2014.
18. ETSI Quantum-Safe Cryptography (QSC) ISG. Quantum-safe cryptography. ETSI working group http://www.etsi.org/technologies-clusters/technologies/quantum-safe-cryptography, 2015.
19. P. L'Ecuyer. Non-uniform random variate generations. In *International Encyclopedia of Statistical Science*, pages 991–995. Springer, 2011.
20. A. K. Lenstra, H. W. Lenstra Jr., and L. Lovász. Factoring polynomials with rational coefficients. *Mathematische Annalen*, 261(4):515–534, 1982.

21. R. Lindner and C. Peikert. Better key sizes (and attacks) for LWE-based encryption. In A. Kiayias, editor, *CT-RSA 2011*, volume 6558 of *LNCS*, pages 319–339. Springer, 2011.
22. F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee. Last-level cache side-channel attacks are practical. In *S&P 2015*, pages 605–622. IEEE Computer Society, 2015.
23. NSA. NSA Suite B Cryptography. NSA website, `https://www.nsa.gov/ia/programs/suiteb_cryptography/`, 2015.
24. D. A. Osvik, A. Shamir, and E. Tromer. Cache attacks and countermeasures: The case of AES. In D. Pointcheval, editor, *CT-RSA 2006*, volume 3860 of *LNCS*, pages 1–20. Springer, 2006.
25. C. Peikert. An efficient and parallel Gaussian sampler for lattices. In T. Rabin, editor, *CRYPTO 2010*, volume 6223 of *LNCS*, pages 80–97. Springer, 2010.
26. C. Peikert. Lattice cryptography for the internet. In M. Mosca, editor, *PQCrypto 2014*, volume 8772 of *LNCS*, pages 197–219. Springer, 2014.
27. Colin Percival. Cache missing for fun and profit. In *BSDCan 2005*, 2005.
28. J. van de Pol, N. P. Smart, and Y. Yarom. Just a little bit more. In K. Nyberg, editor, *CT-RSA 2015*, volume 9048 of *LNCS*, pages 3–21. Springer, 2015.
29. T. Pöppelmann, L. Ducas, and T. Güneysu. Enhanced lattice-based signatures on reconfigurable hardware. In L. Batina and M. Robshaw, editors, *CHES 2014*, volume 8731 of *LNCS*, pages 353–370. Springer, 2014.
30. T. Pöppelmann and T. Güneysu. Towards practical lattice-based public-key encryption on reconfigurable hardware. In T. Lange, K. E. Lauter, and P. Lisonek, editors, *SAC 2013*, volume 8282 of *LNCS*, pages 68–85. Springer, 2014.
31. S. S. Roy, F. Vercauteren, and I. Verbauwhede. High precision discrete Gaussian sampling on FPGAs. In T. Lange, K. E. Lauter, and P. Lisonek, editors, *SAC 2013*, volume 8282 of *LNCS*, pages 383–401. Springer, 2014.
32. M.-J. O. Saarinen. Arithmetic coding and blinding countermeasures for ring-LWE. *IACR Cryptology ePrint Archive 2016/276*, 2016.
33. V. Shoup. NTL: A library for doing number theory, 2015. `http://www.shoup.net/ntl/`.
34. strongSwan. strongSwan 5.2.2 released, January 2015. `https://www.strongswan.org/blog/2015/01/05/strongswan-5.2.2-released.html`.
35. Y. Yarom and N. Benger. Recovering OpenSSL ECDSA nonces using the Flush+Reload cache side-channel attack. IACR Cryptology ePrint Archive 2014/140, 2014.
36. Y. Yarom and K. Falkner. Flush+Reload: a high resolution, low noise, L3 cache side-channel attack. In K. Fu and J. Jung, editors, *USENIX Security 2014*, pages 719–732. USENIX Association, 2014.
37. J. Zhang, Z. Zhang, J. Ding, M. Snook, and Ö. Dagdelen. Authenticated key exchange from ideal lattices. In E. Oswald and M. Fischlin, editors, *EUROCRYPT 2015, Part II*, volume 9057 of *LNCS*, pages 719–751. Springer, 2015.
38. Y. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart. Cross-Tenant side-channel attacks in PaaS clouds. In *CCS'14*, pages 990–1003. ACM, 2014.