To BLISS-B or not to be - Attacking strongSwan's Implementation of Post-Quantum Signatures

Peter Pessl
Graz University of Technology
peter.pessl@iaik.tugraz.at

Leon Groot Bruinderink
Technische Universiteit Eindhoven
l.groot.bruinderink@tue.nl

Yuval Yarom
University of Adelaide and Data61
yval@cs.adelaide.edu.au

ABSTRACT

In the search for post-quantum secure alternatives to RSA and ECC, lattice-based cryptography appears to be an attractive and efficient option. A particularly interesting lattice-based signature scheme is BLISS, offering key and signature sizes in the range of RSA moduli. A range of works on efficient implementations of BLISS is available, and the scheme has seen a first real-world adoption in strongSwan, an IPsec-based VPN suite. In contrast, the implementation-security aspects of BLISS, and lattice-based cryptography in general, are still largely unexplored.

At CHES 2016, Groot Bruinderink et al. presented the first side-channel attack on BLISS, thus proving that this topic cannot be neglected. Nevertheless, their attack has some limitations. First, the technique is demonstrated via a proof-of-concept experiment that was not performed under realistic attack settings. Furthermore, the attack does not apply to BLISS-B, an improved variant of BLISS and also the default option in strongSwan. This problem also applies to later works on implementation security of BLISS.

In this work, we solve both of the above problems. We present a new side-channel key-recovery algorithm against both the original BLISS and the BLISS-B variant. Our key-recovery algorithm draws on a wide array of techniques, including learning-parity with noise, integer programs, maximimum likelihood tests, and a lattice-basis reduction. With each application of a technique, we reveal additional information on the secret key culminating in a complete key recovery.

Finally, we show that cache attacks on post-quantum cryptography are not only possible, but also practical. We mount an asynchronous cache attack on the production-grade BLISS-B implementation of strongSwan. The attack recovers the secret signing key after observing roughly 6 000 signature generations.

KEYWORDS

lattice-based cryptography; side-channel analysis; signatures; cache attacks; learning parity with noise; lattice reduction

1 INTRODUCTION

Quantum computing might eventually break all widespread public-key cryptosystems. A recent estimate [26] states that quantum computers able to factor currently-used RSA moduli could be available as early as 2030. This outlook causes serious concerns and has lead to increased efforts in the search for post-quantum secure alternatives. Recently, the NSA issued an advisory stating that a shift to quantum-resistent cryptography is likely in the near future [30] and standardization bodies also started to look into this matter, as demonstrated by NIST's current call for proposals [28]. Modern post-quantum cryptography has also already seen (limited) real-world evaluation, e.g., the experiments with the NewHope [2] key-exchange by Google in their Chrome browser [11, 22].

Cryptography based on lattices has proven to be a particularly efficient candidate. For example, the Bimodal Lattice Signature Scheme (BLISS), which was proposed by Ducas, Durmus, Lepoint, and Lyubashevsky [14], offers key sizes in the range of current RSA moduli, with similar security levels. Additionally, it offers favorable runtime on a large set of platforms, ranging from FPGAs [39] to microcontrollers [29]. It has also seen adoption in the strongSwan IPsec-based VPN suite [45].

In contrast to the emerging real-world adoption and the large body of work targeting efficient implementation of lattice-based primitives, the implementation-security aspect is still a very open and under-explored topic. In 2016, Groot Bruinderink et al. [19] presented the first side-channel attack on BLISS. Their attack targets a noise vector which is sampled from the discrete Gaussian distribution and used to hide any information on the secret key in the signature. Dedicated algorithms, e.g., those proposed by the authors of BLISS, are used to sample from this distribution. By means of a cache attack on these samplers, Groot Bruinderink et al. are able to retrieve estimations of some elements of the noise vector. Using the signature and recovered noise elements of many signing operations, they then recover the secret key by means of a lattice reduction.

However, their attack has some shortcomings. First, in their proof-of-concept cache attack they target the "research-oriented" reference implementation of BLISS [1]. They also modified its code in order to achieve perfect synchronization of the attacker with the calls to the sampler. While this method demonstrates the existence and exploitability of the side-channel, it is not a realistic and practical setting.

Second, and maybe more importantly, their attack does not apply to BLISS-B, an improved version of BLISS proposed by Ducas [13]

[1] The reference implementation is available at http://bliss.di.ens.fr/

that accelerates the signing operation by a factor of up to 2.8, depending on the used parameter set. Due to its better performance, this new variant is used in strongSwan per default.

The attack target. The main operation in BLISS is to multiply the secret key $\mathbf{s}$ with a binary *challenge vector* $\mathbf{c}$ and add a *noise vector* $\mathbf{y}$ which is sampled at random from a discrete Gaussian distribution. The result $\mathbf{z} = \mathbf{y} + (-1)^b(\mathbf{s} \cdot \mathbf{c})$, where b is a random bit, together with the challenge vector $\mathbf{c}$ form the signature. Using the recovered values of $\mathbf{y}$ over many signatures, Groot Bruinderink et al. [19] construct a lattice from the challenge vectors such that $\mathbf{s}$ is part of the solution to the shortest vector problem in that lattice. This short vector is found using a lattice-basis reduction.

In BLISS-B, however, the secret $\mathbf{s}$ is multiplied with a ternary polynomial $\mathbf{c}' \in \{-1, 0, 1\}^n$ for which $\mathbf{c}' \equiv \mathbf{c} \bmod 2$. Still, only the binary version $\mathbf{c}$ is part of the signature and $\mathbf{c}'$ is undisclosed. Thus, the signs of the coefficients of the used challenge vectors are unknown and constructing the appropriate lattice to find $\mathbf{s}$ is infeasible for secure parameters. Note that this problem (or similar ones) are also present in other works on implementation attacks on the original BLISS, both for side-channel attacks [35] as well as fault attacks [9, 16]. Hence, one might be tempted to think of BLISS-B as a "free" side-channel countermeasure.

Our contributions. In this work we show that this is not the case. First, we present a new key-recovery attack that can, given side-channel information on the Gaussian samples in $\mathbf{y}$, recover the secret key $\mathbf{s}$. Apart from being applicable to BLISS-B, this new key recovery approach can also increase the efficiency (in the number of required side-channel measurements) of earlier attacks on the original BLISS [19, 35]. And second, we use this new key-recovery approach to mount an asynchronous cache attack on the BLISS implementation provided by strongSwan. Hence, we attack a real-world implementation under realistic settings.

Our key-recovery method consists of four steps:

- In the first step, we use side channels to gather information on the noise vector. We use these leaked values, together with known challenge vector elements, to construct a linear system of equations. However, the signs in this system are unknown. (Section 4.1)
- In the second step, we solve the above system. We circumnavigate the problem of unknown signs by using the fact that $-1 \equiv 1 \bmod 2$. That is, we first solve the linear system over the bits, i.e., in GF(2), instead of over the integers. Due to errors in the side channel the linear system may include some errors. Solving such a system is known as the Learning Parity with Noise (LPN) problem. We use an LPN solving algorithm to learn the parity of the secret key elements, i.e. to find $\mathbf{s} \bmod 2$ (Section 4.2).
- In some parameter sets (cf. Section 2.2), the key $\mathbf{s} \in \{0, \pm 1\}^n$ and thus the above already uniquely determines the magnitude of the coefficients. In others, however, the secret key can also have some coefficients with ± 2, which have parity zero. In the third step, we employ one of two heuristics (depending on the parameter set) to identify those, both exploit the magnitude of the coefficients of $\mathbf{s} \cdot \mathbf{c}'$. The first heuristic uses an Integer Programming solver. The second uses a Maximum Likelihood estimate. (Section 4.3)
- At this stage we know the magnitude of each of the coefficients of the secret key $\mathbf{s}$. In the fourth step, we finalize the attack and extract $\mathbf{s}$. We construct a Shortest Vector Problem (SVP) based on the public key and the known information about the secret key. We solve this problem using the BKZ lattice-reduction algorithm. (Section 4.4)

When using the idealized cache-attack presented by Groot Bruinderink et al. [19] and the BLISS-I parameter set, our new method can reduce the number of required signatures from 450 to 325. Furthermore, we also apply the key-recovery technique to an attack on the shuffling countermeasure by Pessl [35]. There, our attack reduces the number of required signatures by a factor of up to 22.

We then perform a cache attack on the BLISS-B implementation which is deployed as part of the strongSwan VPN software. Unlike Groot Bruinderink et al. [19], our adversary is asynchronous and runs in a different process than the victim. The adversary uses the Flush+Reload attack by Yarom and Falkner [49], combined with the amplification attack of Allan et al. [3]. Furthermore, we target a real-world implementation and not a research-oriented reference implementation. Consequently, our attack scenario is much more realistic. While strongSwan does not claim any side-channel security, our results still show that practical attacks on the BLISS family are feasible.

Outline. In Section 2, we recall BLISS, discrete Gaussians and sampling methods. Then, in Section 3 we discuss previous work on side channel analysis and countermeasures on BLISS. We then show our improved key-recovery attack in Section 4. We evaluate our new method in Section 5 by comparing it to earlier work. In Section 6, we perform a full attack on the BLISS implementation provided by strongSwan. Finally, we conclude in Section 7.

2 PRELIMINARIES

In this section, we briefly describe background concepts required for the rest of the paper. These include lattices, the BLISS signature scheme [13, 14], the discrete Gaussian distribution and methods to sample from this distribution, and the Learning Parity with Noise (LPN) problem.

2.1 Lattices

A lattice Λ is a discrete subgroup of $\mathbb{R}^n$. When given m linearly independent vectors $\mathbf{b}_1, \ldots, \mathbf{b}_m \in \mathbb{R}^n$, the lattice $\Lambda(\mathbf{b}_1, \ldots, \mathbf{b}_m)$ contains all of the points that are integer linear combinations of the basis vectors:

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

We call $\mathbf{B} = (\mathbf{b}_1, \ldots, \mathbf{b}_m)$ the basis matrix of the lattice, with n the dimension and m the rank of the lattice. Lattice bases are not unique: for each full-rank basis $\mathbf{B} \in \mathbb{R}^{n \times n}$ of Λ, one can apply a unimodular matrix $\mathbf{U} \in \mathbb{Z}^{n \times n}$, such that $\mathbf{UB}$ is also a basis of Λ. There exist lattice-basis reduction algorithms that are aimed at finding a *good* basis, which consists of short and nearly orthogonal vectors. The most important of these algorithms are the LLL [23] as well as BKZ and its improved versions [12]. These algorithms output a new basis $\mathbf{B}'$ which satisfies certain conditions. Besides

outputting $\mathbf{B}'$, LLL and BKZ implementations (such as [43]) can also output $\mathbf{U}$ such that $\mathbf{B}' = \mathbf{UB}$.

For cryptographic purposes one often uses q-ary lattices. Simply speaking, for a vector $\mathbf{v} \in \Lambda$, all vectors $\mathbf{u}$ with $\mathbf{u} \equiv \mathbf{v} \bmod q$ are also in the lattice. In order to save memory and decrease execution time, the most efficient lattice-based cryptographic constructions introduce additional structure into the lattices they use. That is, they work with the polynomial ring $\mathcal{R}_q = \mathbb{Z}_q[x]/\langle x^n + 1\rangle$, with q being a prime and n a power of 2. An element $\mathbf{a} \in R_q$ can be described by its coefficient vector $\mathbf{a} = (a_0, \ldots, a_{n-1})$. Note that we will use boldface to interchangeably denote polynomials and their coefficient vectors. Addition of two polynomials $\mathbf{a}, \mathbf{b}$ is simply the componentwise addition mod q. Multiplication of two polynomials $\mathbf{a}, \mathbf{b} \in \mathcal{R}_q$ will be denoted by $\mathbf{a} \cdot \mathbf{b}$, and can be represented as a matrix-vector product, i.e., $\mathbf{a} \cdot \mathbf{b} = \mathbf{aB} = \mathbf{bA}$, where the columns of $\mathbf{A}, \mathbf{B} \in \mathbb{Z}_q^{n \times n}$ are negacyclic rotations of $\mathbf{a}$ and $\mathbf{b}$, respectively. The computation of the i-th coefficient of the product $\mathbf{a} \cdot \mathbf{b}$ can be written as $\langle \mathbf{a}, \mathbf{b}_i \rangle$, with $\mathbf{b}_i$ the i-th column of matrix $\mathbf{B}$.

2.2 Bimodal Lattice Signature Scheme (BLISS)

The most efficient instantiation of BLISS operates over the ring $\mathcal{R}_q$. Key generation for the improved version BLISS-B [13] is shown in Algorithm 1. During key generation, two polynomials $\mathbf{f}, \mathbf{g}$ with exactly $d_1 = \delta_1 n$ coefficients in $\{\pm 1\}$, $d_2 = \delta_2 n$ coefficients in $\{\pm 2\}$, and all remaining elements being 0, are sampled. n, δ_1, δ_2, and q are part of the parameter set.

Algorithm 1 BLISS-B Key Generation Algorithm

Output: Public key $\mathbf{A} \in \mathcal{R}_{2q}^2$, private key $\mathbf{S} \in \mathcal{R}_{2q}^2$

1: Choose random polynomials $\mathbf{f}, \mathbf{g}$ with d_1 entries in $\{\pm 1\}$ and d_2 entries in $\{\pm 2\}$ until $\mathbf{f}$ is invertible
2: $\mathbf{S} = (\mathbf{s}_1, \mathbf{s}_2) = (\mathbf{f}, 2\mathbf{g} + 1)$
3: $\mathbf{a}_q = \mathbf{s}_2 / \mathbf{s}_1 \bmod q$
4: **return** $(\mathbf{A}, \mathbf{S})$, with $\mathbf{A} = (2\mathbf{a}_q, q - 2) \bmod 2q$

The BLISS-B signing procedure is given in Algorithm 2. In the first step, two polynomials $\mathbf{y}_1, \mathbf{y}_2$ are sampled from a discrete Gaussian distribution D_σ. The challenge vector $\mathbf{c}$, used in the Fiat-Shamir transform [17], is computed by invoking a hash function H. This function returns a binary vector of length n and a Hamming weight of exactly κ. GreedySC (Algorithm 3) then computes the product $\mathbf{Sc}'$ for some ternary vector $\mathbf{c}'$ (this means $\mathbf{c}' \in \{-1, 0, +1\}^n$) that satisfies $\mathbf{c}' \equiv \mathbf{c} \mod 2$. Note that for the specific BLISS input $\mathbf{S} = (\mathbf{s}_1, \mathbf{s}_2) \in \mathcal{R}_{2q}^2$ in GreedySC, we have $m = 2n$ and $\mathbf{s}_i = \mathbf{S}_{1i}$ for $0 \leq i < n$ and $\mathbf{s}_i = \mathbf{S}_{2i}$ for $n \leq i < 2n$ where $\mathbf{S}_{1i}$ and $\mathbf{S}_{2i}$ are the negacyclic rotations of $\mathbf{s}_1$ and $\mathbf{s}_2$, respectively. The generated $\mathbf{c}'$ contains information on the secret key, hence it is kept secret and not output as part of the signature. GreedySC is not part of the first version of BLISS, which we will denote with BLISS-A. Instead, the product $\mathbf{Sc}$ is used directly in BLISS-A, i.e., $\mathbf{v}_1 = \mathbf{s}_1 \cdot \mathbf{c}$. Depending on a secret bit b, the outcome is then either added to or subtracted from the noise polynomials $\mathbf{y}_1, \mathbf{y}_2$. A final rejection-sampling step prevents any leakage of secret information. Parameters ζ, d, and p, are used for signature compression, but they are not relevant in the rest of this paper. The one exception is the following: due to

Algorithm 2 BLISS-B Signature Algorithm

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

1: $\mathbf{y}_1 \leftarrow D_\sigma^n$, $\mathbf{y}_2 \leftarrow D_\sigma^n$
2: $\mathbf{u} = \zeta \cdot \mathbf{a}_1 \cdot \mathbf{y}_1 + \mathbf{y}_2 \bmod 2q$
3: $\mathbf{c} = \mathrm{H}(\lfloor \mathbf{u} \rceil_d \bmod p || \mu)$
4: $(\mathbf{v}_1, \mathbf{v}_2) = \mathrm{GreedySC}(\mathbf{S}, \mathbf{c})$
5: Sample a uniformly random bit b
6: $(\mathbf{z}_1, \mathbf{z}_2) = (\mathbf{y}_1, \mathbf{y}_2) + (-1)^b (\mathbf{v}_1, \mathbf{v}_2)$
7: Continue with some probability $f(\mathbf{v}, \mathbf{z})$, restart otherwise (details in [13])
8: $\mathbf{z}_2^\dagger = (\lfloor \mathbf{u} \rceil_d - \lfloor \mathbf{u} - \mathbf{z}_2 \rceil_d) \bmod p$
9: **return** $(\mathbf{z}_1, \mathbf{z}_2^\dagger, \mathbf{c})$

Algorithm 3 GreedySC

Input: a matrix $\mathbf{S} \in \mathbb{Z}^{m \times n}$ and a binary vector $\mathbf{c} \in \mathbb{Z}^n$
Output: $\mathbf{v} = \mathbf{Sc}'$ for some $\mathbf{c}' \equiv \mathbf{c} \mod 2$

1: $\mathbf{v} = \mathbf{0} \in \mathbb{Z}^n$
2: **for** $i \in \mathcal{I}_c$ **do**
3: $\quad \zeta_i = \mathrm{sgn}(\langle \mathbf{v}, \mathbf{s}_i \rangle)$
4: $\quad \mathbf{v} = \mathbf{v} - \zeta_i \mathbf{s}_i$
5: **return** $\mathbf{v}$

rounding of the second signature vector $\mathbf{z}_2^\dagger$ (Line 8 of Algorithm 2) and the resulting loss of information on $\mathbf{s}_2 \cdot \mathbf{c}$, the attack in this paper, like previous works, only concentrates on $\mathbf{y}_1$. We will omit the index 1 of of $\mathbf{z}_1, \mathbf{y}_1$, and $\mathbf{s}_1$ in the next sections, and always imply it if not mentioned otherwise.

For completeness, we also present the verification algorithm in Algorithm 4. The verification algorithm is the same for both BLISS-A and BLISS-B. For a more thorough explanation, we refer to the original publications [13, 14].

Algorithm 4 BLISS Verification Algorithm

Input: Message μ, public key $\mathbf{A} = (\mathbf{a}_1, q - 2) \in \mathcal{R}_{2q}^2$, 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 [14]) **then** reject
2: accept iff $\mathbf{c} = H(\lfloor \zeta \cdot \mathbf{a}_1 \cdot \mathbf{z}_1 + \zeta \cdot q \cdot \mathbf{c} \rceil_d + \mathbf{z}_2^\dagger \bmod p, \mu)$

Ducas et al. [14] propose several parameter sets for different security levels. These remain unchanged for BLISS-B. We present the parameters relevant to this paper in Table 1.

2.3 Discrete Gaussians

The discrete Gaussian distribution with standard deviation σ and mean zero is denoted by D_σ. We denote a variable y sampled from this distribution D_σ with $y \leftarrow D_\sigma$. The probability of sampling a value x is given by $D_\sigma(x) = \rho_\sigma(x)/\rho_\sigma(\mathbb{Z})$, with $\rho_\sigma(x) =$

Table 1: BLISS Parameter Sets

Parameter Set	n	q	σ	δ_1, δ_2	κ
BLISS-0 (Toy)	256	7681	100	0.55, 0.15	12
BLISS-I	512	12289	215	0.3, 0	23
BLISS-II	512	12289	107	0.3, 0	23
BLISS-III	512	12289	250	0.42, 0.03	30
BLISS-IV	512	12289	271	0.45, 0.03	39

$\exp((-x^2)/(2\sigma^2))$ and the normalization constant $\rho_\sigma(\mathbb{Z})$. D_σ^n denotes an n-dimensional vector with elements independently sampled from D_σ. Compared to lattice-based public-key encryption [25], the standard deviation required for BLISS is relatively high. This makes the efficient implementation of samplers an especially taxing task.

CDT sampler. The inversion method (or CDT sampling) appears to be particularly efficient. In this method, one first precomputes and stores the (absolute value) cumulative distribution table (CDT), i.e., a table $T[y] = \Pr(|x| < y | x \leftarrow D_\sigma)$ for $y \in \{0, \ldots, \tau\sigma\}$ for some tail-cut τ. Then, a uniformly random $r \in [0, 1)$ is generated and the y satisfying $T[y] \leq r < T[y+1]$ is returned. Typically, a binary search is used to find the correct index in the table. A common method to speed up this search is to use so-called *guide-tables*, which narrow down the initial search range based on, e.g., the first byte of r. A byte-oriented version of this entire method is given in Algorithm 5, there $T_j[i]$ denotes the j-th byte of the entry at index i. For more details see, e.g., [39].

Algorithm 5 CDT Sampler using Guide Tables

Input: Guide table I, (absolute value) cumulative distribution table T
Output: A value y' sampled according to D_σ

Sample a uniformly random byte r_0
$[\min, \max] = I[r_0]$
$i = (\min + \max)/2$, $j = 0$, $k = 0$
while max-min > 1 **do**
  if $T_j[i] > r_j$ **then**
    $\min = i$, $i = (i + \max)/2$, $j = 0$
  else if $T_j[i] < r_j$ **then**
    $\max = i$, $i = (\min + i)/2$, $j = 0$
  else
    $j = j + 1$
    if $k < j$ **then**
      Sample uniformly random byte r_j, $k = j$
Sample a uniformly random bit s
if $s = 1$ **then return** $-i$
else return i

Bernoulli sampler (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 if $r \leq \rho_\sigma(y)$.

For the case of the discrete Gaussian distribution with high standard deviation, Ducas et al. [14] introduce a more efficient method called *Bernoulli sampler*. It uses the subroutine described in Algorithm 6 to sample a bit b from $\mathcal{B}(\exp(-x/f))$, i.e., the Bernoulli distribution $\mathcal{B}$ parametrized such that $\Pr(b = 1) = \exp(-x/f)$. The constant f depends on the standard deviation σ, while x varies. Pseudocode for the Bernoulli sampler appears in Algorithm 7.

Algorithm 6 Sampling a bit from $\mathcal{B}(\exp(-x/(2\sigma^2)))$ for $x \in [0, 2^\ell)$

Input: $x \in [0, 2^\ell)$ an integer in binary form $x = x_{\ell-1} \ldots x_0$. Precomputed table E with $E[i] = \exp(-2^i/(2\sigma^2))$ for $0 \leq i < \ell$
Output: A bit b from $\mathcal{B}(\exp(-x/(2\sigma^2)))$

for $i = \ell - 1$ downto 0 **do**
  if $x_i = 1$ **then**
    sample bit A_i from $\mathcal{B}(E[i])$
    if $A_i = 0$ **then return** 0
return 1

Algorithm 7 Bernoulli Sampler

Input: Standard deviation σ, integer $K = \lfloor \frac{\sigma}{\sigma_2} + 1 \rfloor$ with $\sigma_2^2 = \frac{1}{2 \ln 2}$
Output: A value y' sampled according to D_σ

Sample $x \in \mathbb{Z}$ according to $D_{\sigma_2}^+$ (details in [14])
Sample $z \in \mathbb{Z}$ uniformly in $\{0, \ldots, K-1\}$
Set $y = Kx + z$
sample b from $\mathcal{B}(\exp(-z(z + 2Kx)/(2\sigma^2)))$ using Algorithm 6
if $b = 0$ **then** restart
if $y = 0$ **then** restart with probability $1/2$
Sample uniformly random bit s and **return** $(-1)^s y$

2.4 Learning Parity with Noise (LPN)

We now recall the Learning Parity with Noise (LPN) problem, whose search version appears in Definition 2.1.

Definition 2.1 (Learning Parity with Noise). Let $\mathbf{k} \in \mathrm{GF}(2^n)$ and $\epsilon \in (0, 0.5)$ be a constant noise rate. Then, given ν vectors $\mathbf{a}_i \in \mathrm{GF}(2^n)$ and noisy observations $b_i = \langle \mathbf{a}_i, \mathbf{k} \rangle + e_i$, the $\mathbf{a}_i$ sampled uniformly, and the e_i sampled from the Bernoulli distribution with parameter ϵ, find $\mathbf{k}$.

The most efficient algorithms aimed at solving this problem are based on the work of Blum, Kalai, and Wasserman [10]. Later work then modified and improved the BKW algorithm [20, 24]. While these algorithms run in sub-exponential time, they tend to require a large number of LPN samples as well as a lot of memory. A different approach is to view LPN as decoding a random linear code over the binary field GF(2). While this second approach runs in exponential time, it typically offers a negligible memory consumption and lower sample requirements.

LPN is a well-researched problem and is used as a basis for cryptographic constructions [37]. Furthermore, LPN solving algorithms have also been used in side-channel attacks on binary-field multiplication [4, 5, 36]. The extension of this problem from the binary field GF(2) to a prime field GF(q) is known as Learning with Errors (LWE) [41] and is a major cornerstone in lattice-based cryptography.

3 SIDE-CHANNEL ATTACKS ON BLISS

In this section, we briefly describe previous work on side-channel attacks on BLISS. We start with an introduction to cache-based side channel attacks. We then discuss the attack of Groot Bruinderink et al. [19] on the BLISS algorithm, followed by the technique Pessl [35] uses to overcome the shuffling protection of Saarinen [42].

3.1 Cache Attacks

To bridge the speed gap between the faster processor and the slower memory, modern processor architectures employ multiple *caches* which store data that the processor predicts a program might use in the future. While the cache does not change the logical behavior of programs, it does affect their execution time. For the past 15 years, it has been known that timing variations due to the cache state can leak secret information about the execution of the program [7, 32, 46]. Over the years, many attacks that exploit the cache state have been designed. For a survey of cache and other microarchitectural attacks, see Ge et al. [18].

The Flush+Reload Attack. In this work we use the Flush+Reload attack by Yarom and Falkner [49]. Flush+Reload exploits read-only memory sharing, which is commonly used for sharing library code in modern operating systems. The attack consists of two phases. In the *flush* phase, the attacker evicts the contents of a monitored address from the cache. On Intel processors this is typically achieved using the `clflush` instruction. The attacker then waits a bit before performing the *reload* phase of the attack. In the reload phase, the attacker reads the contents of the monitored memory address, while measuring the time it takes to perform the read. If the victim has accessed the monitored memory between the flush and the reload phases, the contents of the address will be cached and the attacker's read will be fast. Otherwise, the memory address will not be in the cache and the read will be slow.

By repeatedly interleaving the flush and the reload phases of multiple locations, the attacker can create a trace of the victim uses of the monitored locations over time. When the victim access patterns depend on secret data, the attacker can use the trace to recover the data. Flush+Reload has been used to attack RSA [49], AES [21], ECDSA [6, 38, 48], as well as non-cryptographic software [31, 50].

Side-Channel Amplification. Because Flush+Reload only monitors victim accesses between the flush and the reload phases, accesses that occur during these phases may be missed, resulting in false negatives. The timing of victim accesses is mostly independent of the attacker's activity. Consequently, increasing the wait between these phases reduces the probability of false negatives, albeit at the cost of reduced temporal resolution. To mitigate the effects of the reduced temporal resolution, Allan et al. [3] suggest slowing down the victim. They demonstrate that by repeatedly evicting frequently-used code from the cache, they are able to slow programs down by a factor of up to 150. The combination of Flush+Reload and the Allen et al. attack has been used for attacks on ECDSA [3, 33] and DSA [34]

3.2 A Cache Attack on BLISS

At CHES 2016, Groot Bruinderink et al. [19] presented the first side-channel attack on BLISS. Their cache attack targets the Gaussian sampling component, they describe attacks on both the CDT sampler (using Guide Tables) and the Bernoulli sampler described in the previous section. Remember that we are omitting the index of the vectors, as the attack only uses knowledge on $\mathbf{z}_1$ and $\mathbf{y}_1$. Recall also that the i'th coefficient of a signature vector can be written as $z_i = y_i + (-1)^b \langle \mathbf{s}, \mathbf{c}_i \rangle$. The vectors $\mathbf{y}$ and bit b are unknown to the attacker, as is the secret vector $\mathbf{s}$. Note that in this section we consider previous work on the original BLISS scheme (BLISS-A), so $\mathbf{c} \in \{0,1\}^n$.

The core idea of the cache-attack of Groot Bruinderink et al. is to exploit knowledge learned through cache-timing to gather information on the table lookups performed during Gaussian sampling. From this information, it is possible to derive a precise estimate of some of the coefficients y_i of $\mathbf{y}$, and consequently recover the secret key. For both previously mentioned samplers, they performed an evaluation using ideal adversaries and practical experiments using the Flush+Reload attack technique. We now describe their attacks on the two samplers.

Attacking the CDT sampler. The CDT sampler uses two tables (CDT table T and interval table I). Accessing these tables can leak the accessed cache-line. This information, in turn, leaks a range of possible values for y_i. Groot Bruinderink et al. describe two approaches to estimate y_i more precisely than naively using these leaked ranges. The first approach is to intersect the ranges of possible values learned from each table. The second approach is to track down the binary search steps done in the sampling procedure by looking at multiple accesses in table T.

As the search step in table T is a binary search, one of two adjacent values is returned after the last table-lookup. This means that a sample y_i can only be determined up to an uncertainty of ± 1. However, in general there is a bias in the value that is returned, as the targeted distribution is a discrete Gaussian and not uniform. If this bias is large enough, Groot Bruinderink et al. guess the returned value to be the more likely one.

Another obstacle is that they do not get the sign of y_i, but only know $|y_i|$ from the accessed cache-lines. However, they use the knowledge of the corresponding coefficient z_i of the signature vector $\mathbf{z}$. It is possible to derive the sign from z_i, as $\langle \mathbf{s}, \mathbf{c}_i \rangle$ is small and thus the sign of y_i will most likely be the sign of z_i.

After the above procedure, they have approximate knowledge on y_i. However, bit b of the signature is still unknown. Instead of guessing or recovering the value of this bit for each signature, they only uses samples where, with a high probability, $z_i = y_i$. In these samples one has that $\langle \mathbf{s}, \mathbf{c}_i \rangle = 0$ (w.h.p.). After collecting enough samples, they use the challenge vectors $\mathbf{c}_i$ that satisfy the above restrictions to construct a matrix $\mathbf{L}$ such that $\mathbf{sL} \approx \mathbf{0}$ is a small vector in the lattice spanned by $\mathbf{L}$. They then use the LLL lattice-reduction algorithm [23] on $\mathbf{L}$ to find a small lattice basis. With a high probability, the secret key $\mathbf{s}$ is part of the unimodular transformation matrix retrieved from LLL. The correctness of the key can be verified by matching against the known public key.

Attacking the Bernoulli sampler. The Bernoulli sampler uses the table E which stores (high precision) exponential values required to do rejection steps. As this table is only accessed for every set bit of input x (Line 2 in Algorithm 6), no table access is done in the case that input $x = 0$. This only happens when input z to the Bernoulli sampler (Line 4 in Algorithm 7) is zero, leading to a small subset of possible values $y_i \in \{0, \pm K, \pm 2K, \dots\}$. As K is in general large, this can lead to a complete retrieval of y_i by also using knowledge of the corresponding signature coefficient z_i. By again restricting to the cases when $y_i = z_i$, Groot Bruinderink et al. used the challenge vectors $\mathbf{c}_i$ to construct a matrix $\mathbf{L}$ such that $\mathbf{sL} = \mathbf{0}$. The secret vector $\mathbf{s}$ can then be found by calculating the (integer left) kernel of $\mathbf{L}$.

3.3 The Shuffling Countermeasure and Analysis

Saarinen [42] proposed to use shuffling to protect implementations of BLISS against the above attack. Instead of (fully) protecting the sampler itself, he proposes to sample a vector and to randomly permute it. This breaks the connection between sampling time and index in the signature and hence prevents the above attack. Concretely, he proposes to generate m Gaussian vectors with smaller standard deviation $\sigma' = \sigma/\sqrt{m}$, to shuffle all vectors independently, and to add them to get a vector from the desired distribution. Alternatively, one can also combine this idea with the sampler of Pöppelmann et al. [39], i.e., choose some k, set $\sigma' = \sigma/\sqrt{1+k^2}$, and then compute $\mathbf{y}', \mathbf{y}'' \leftarrow D^n_{\sigma'}$, $\mathbf{y} = k \cdot \text{Shuffle}(\mathbf{y}') + \text{Shuffle}(\mathbf{y}")$. Due to the smaller σ', this sampling approach also drastically reduces the size of lookup tables required for (more efficient) CDT sampling.

Pessl [35] later analyzed this countermeasure. He found that due to the vastly different distributions of the added variables in $\mathbf{z} = \mathbf{y} + (-1)^b(\mathbf{s} \cdot \mathbf{c})$ ($\mathbf{y}$ distributed according to D^n_σ and large $\sigma > 0$, $\mathbf{s} \cdot \mathbf{c}$ as the product of two small polynomials) one can say that $\mathbf{z} \approx \mathbf{y}$ and thus it is possible to reassign some Gaussian samples. Namely, if one is given a Gaussian sample $y \in \mathbf{y}$, then it is possible to check for *proximity* to all z_i. If only one z_i is close to y, then it is possible to reassign y to index i. Note that this approach works mostly for outliers, i.e., for samples that are in the tail of the discrete Gaussian distribution.

After having reassigned a sufficient number of samples that match with high probability, e.g., $\Pr(z_i \sim y) > 0.99$, it is possible to perform the key-recovery of Groot Bruinderink et al. Under the assumption that bit b (Line 5 of Algorithm 2) is recoverable with SCA, Pessl needs to observe 260 000 signatures for key recovery. If this assumption is not met, then only samples that fulfill $z_i = y_i$ can be used, which increases the number of signatures to 1 550 000.

3.4 Limitations of Previous Attacks

Both previous side-channel attacks on BLISS have certain limitations and caveats. As already stated above, due to the unknown bit b, which is potentially different for each signature, Groot Bruinderink et al. [19] only use samples where $z_i = y_i$ and thus $\langle \mathbf{s}, \mathbf{c}_i \rangle = 0$ (with high probability). This, however, only holds in roughly 15% of all samples (cf. Figure 4) and thus a lot of information is discarded. As the attack on shuffling by Pessl [35] uses the same method for key recovery, this limitation holds there as well. By finding a method to use all samples for the attack, the number of required signatures could drop drastically.

A second and more severe limitation is that the previous attacks do not apply to the improved BLISS-B signature scheme. Groot Bruinderink et al. recover the key by solving a (possibly erroneous) linear system $\mathbf{sL} \approx \mathbf{0}$, where $\mathbf{L}$ consists of the used challenge vectors $\mathbf{c}_i$. However, the GreedySC algorithm, which was added with BLISS-B, performs a multiplication of $\mathbf{s}$ with some unknown ternary $\mathbf{c}' \equiv \mathbf{c} \mod 2$, with $\mathbf{c}' \in \{-1, 0, 1\}^n$. In simple terms, the signs of the coefficients in $\mathbf{c}'$ (and thus also in the resulting lattice basis $\mathbf{L}'$) are unknown. Hence, a straight-forward solving of $\mathbf{sL}' \approx \mathbf{0}$ is not possible anymore.

On the practicality of previous attacks. A third limitation of the attack of Groot Bruinderink et al. [19] is the question of practicality. The attack targets an academic implementation that is not used in any "real-world" applications. Furthermore, the attack is synchronous. To achieve this, Groot Bruinderink et al. modify the code of the BLISS implementation in order to interleave the phases of the Flush+Reload attack with the Gaussian sampler. In practice, it is not clear if an attacker can achieve such a level of synchronization without modifying the source, and an adversary that can modify the source can access the secret key directly without needing to resort to side channel attacks. Consequently, while Groot Bruinderink et al. show a proof-of-concept, their attack falls short of being practical.

We now present a new key-recovery technique that resolves the issues discussed in this section. That is, it works even in the case of BLISS-B and can reduce the number of required signatures by using all recovered samples. Furthermore, in Section 6 we give results on our improvements on the practicality of previous attack, i.e. the asynchronous attack on strongSwans's implementation of BLISS-B.

4 AN IMPROVED SIDE-CHANNEL KEY-RECOVERY TECHNIQUE

In this section, we present our new and improved side-channel attack on BLISS, that also works for BLISS-B. Our method consists of four major steps, each step reveals additional information on the secret signing key $\mathbf{s}$.

The first step is equivalent to previous works. That is, the attacker performs a side-channel attack, e.g., a cache attack or power analysis, on the Gaussian-sampler component to recover some of the drawn samples y_i of $\mathbf{y}$. With this information we can construct a (possibly erroneous) system of linear equations over the integers, using knowledge on $z_i - y_i = (-1)^b(\mathbf{s} \cdot \mathbf{c}')$. (Section 4.1)

Due to the previously mentioned sign-uncertainty in BLISS-B (the recovered terms $\mathbf{s} \cdot \mathbf{c}'$ instead of $\mathbf{s} \cdot \mathbf{c}$), the solution cannot be found with simple linear algebra in $\mathbb{Z}$. Instead, in Step 2 we solve this system over the bits, i.e., in GF(2). For error correction, we employ an LPN algorithm that is based on a decoding approach and can incorporate differing error probabilities. (Section 4.2)

This does not give us the full key, but instead $\mathbf{s}^* = \mathbf{s} \mod 2$. For some parameter sets however, there are some coefficients ± 2 (i.e., BLISS-0, BLISS-III and BLISS-IV have $\delta_2 > 0$). In Step 3, we retrieve their positions. We use the current knowledge on the secret key $\mathbf{s}^*$ to derive $\langle \mathbf{s}^*, \mathbf{c}_i \rangle$, and compare this with $z_i - y_i = \langle \mathbf{s}, \mathbf{c}'_i \rangle$ (obtained

from the side channel). Based on that, we give two different methods in Section 4.3 to determine the positions of the ±2 coefficients and derive $|\mathbf{s}| \in \{0,1,2\}^n$.

In the fourth step, we finally recover the full signing key. We use $|\mathbf{s}|$ to reduce the size of the public key. We then perform a lattice reduction and search for $\mathbf{s}_2$ as a short vector in the lattice spanned by this reduced key. Linear algebra then allows recovery of the full private key $(\mathbf{s}_1, \mathbf{s}_2)$. (Section 4.4)

We now give a more detailed description of these steps.

4.1 Step 1: Gathering Samples

Akin to previous attacks (cf. Section 3), we need to observe the generation of multiple signatures and use a side-channel to infer some of the elements of the corresponding noise vector $\mathbf{y} = \mathbf{y}_1$. In previous works, the exploited side channels were based on cache attacks (in [19]) or on power analysis (in [35]). If shuffling is used, then these samples first need to be reassigned to their index in the signature (cf. Section 3.3).

Side-channel analysis has to deal with noise and other uncertainties. Due to these effects a recovered sample y_i might not be correct. In our scenario, the probability ϵ of such an error is known (or can be estimated to a certain extent) and can possibly be different for each sample. We will later use these probabilities to optimize our attack.

For each recovered (and reassigned) sample y_i, we can write an equation $z_i = y_i + (-1)^b \langle \mathbf{s}, \mathbf{c}'_i \rangle$, which holds with probability 1-ϵ. As the signs of coefficients of $\mathbf{c}'_i$ are unknown, we can simply ignore the multiplication with $(-1)^b$ and instead implicitly include this factor into $\mathbf{c}'_i$. Unlike Groot Bruinderink et al., we do not require that $\langle \mathbf{s}_1, \mathbf{c}_i \rangle = 0$ and thus can use all recovered samples. We compute the difference $t_i = z_i - y_i$ and rearrange all gathered $\mathbf{c}'_i$ into a matrix $\mathbf{L}'$ to get $\mathbf{s}\mathbf{L}' = \mathbf{t}$.

This system is defined over $\mathbb{Z}$. However, due to the unknown signs in the $\mathbf{c}'$ it cannot be directly solved using straight-forward linear algebra, even in the case that all recovered samples are correct. Instead, a different technique is required.

4.2 Step 2: Finding $\mathbf{s}_1$ mod 2

In the second attack step, we solve the above system by using the following observation. Line 6 of Algorithm 2, i.e., $\mathbf{z}_1 = \mathbf{y}_1 + \mathbf{s}_1 \cdot \mathbf{c}'$, is defined over $\mathbb{Z}$. That is, there is no reduction mod q involved[2]. Such an equivalence relation in $\mathbb{Z}$ obviously also holds mod 2, i.e., in GF(2), whereas the reverse is not true.

In GF(2), we have that $-1 \equiv 1 \bmod 2$. This resolves the uncertainty in $\mathbf{L}'$ and we can, at least when assuming no errors in the recovered samples, solve the system $\mathbf{s}^*\mathbf{L}' = \mathbf{t}^*$ in GF(2). Here $\mathbf{s}^*$ and $\mathbf{t}^*$ denote $\mathbf{s}$ mod 2 and $\mathbf{t}$ mod 2, respectively. In the BLISS-I parameter set (Table 1), we have that $\delta_2 = 0$. Thus, $\mathbf{s}^*$ reveals the position of all $\lceil \delta_1 n \rceil = 154$ nonzero, i.e., (±1), coefficients. However, a simple enumeration of all 2^{154} possibilities for $\mathbf{s}$ is still not feasible. Before we discuss a method to recover the signs of $\mathbf{s}$ and thus the full key, we show how errors in $\mathbf{t}^*$ can be corrected.

[2] In fact, due to the parameter choices and the tailcut required by a real Gaussian sampler, $|\mathbf{y}_1 + \mathbf{s}_1 \cdot \mathbf{c}'|$ can never exceed q.

Error Correction mod 2. As stated in Section 4.1, a recovered Gaussian sample y_i might not be correct. Hence, the right-hand-side of the system $\mathbf{s}^*\mathbf{L}' = \mathbf{t}^*$ is possibly erroneous. For instance, in the cache attack on CDT sampling algorithm of Groot Bruinderink et al., errors cannot be avoided. For the attack on the shuffling countermeasure, the error probability can be made arbitrarily low by only keeping samples that have a matching probability ≈ 1. However, the need for such an aggressive filtering increases the number of required signatures. Hence, the capability of error correction is crucial.

We can rewrite the above equations in GF(2) as $\mathbf{s}^*\mathbf{L}' = \mathbf{t}^* + \mathbf{e}$. Here, $\mathbf{t}^*$ is errorless and the error is instead modeled as vector $\mathbf{e}$. Solving this system is exactly the LPN problem described in Section 2.4, thus we employ an LPN solving algorithm to recover $\mathbf{s}^*$. The most time-efficient algorithms to solve LPN are based on the work of Blum, Kalai, and Wasserman [10]. A caveat of this and improved versions [20, 24] are the large memory and LPN-sample requirement. For instance, with $n = 512$ and an error probability ϵ of just 0.01, the often quoted LF1 algorithm by Levieil and Fouque [24] requires 2^{52} bytes of memory. Thus, for BLISS and the already somewhat high dimension of $n = 512$ this class of algorithms is not ideal for the problem at hand.

Also, note that in the definition of the LPN problem (Definition 2.1) the error probability ϵ is constant for all samples. This, however, does not reflect the reality of our side-channel attack. There, each recovered Gaussian sample can be assigned a potentially different error probability ϵ_i. By making use of this additional knowledge, the solving process can potentially be sped up.

An LPN-solving algorithm that can utilize such differing probabilities and that does not require an extensive amount of memory was presented by Pessl and Mangard [36]. First, they perform a filtering, i.e., only keep samples those samples with the lowest error probability. All other samples are discarded. Then, they use a decoding approach, i.e., solving LPN by decoding a random linear code, on the remaining samples. They tweaked Stern's decoding algorithm [44] such that it can utilize differing error probabilities. They use their method in context of a side-channel attack on polynomial multiplication in GF(2).

We use their algorithm for our attack. It is easy to see that due to the initial filtering of highly reliable equations, there exists a possible trade-off between gathered samples and computational runtime. That is, with more equations one can expect a lower error probability of the few best samples, which decreases the runtime of decoding. We will explore this trade-off in Section 5.

Determining error probabilities. Thus far, we did not discuss how the error probabilities of the samples are computed. They mainly depend on the used side-channel attack. Groot Bruinderink et al. [19] attack two different samplers using a cache attack. In their (idealized) attack on a Bernoulli sampler, they can recover samples perfectly. Hence, no error correction is required. The attack on a CDT sampler, however, cannot exclude errors. There, the error probability depends on the used cache weakness[3].

In the case of the attack on the shuffling countermeasure by Pessl [35], we reuse the assumption that all recovered samples are correct. Hence, errors are introduced by incorrect reassignments of

[3] The error probabilities are specified in Appendix B of the full version of [19].

samples to their respective index in the signature. The probability of a correct match $\Pr(z_i \sim y)$ is computed during the attack and was used to filter for highly probable matches. While it is possible to simply use these probabilities, there is optimization potential that can decrease the number of required samples.

In our attack, we solve the system $\mathbf{s}^*\mathbf{L}' = \mathbf{t}^*$ over GF(2). Thus, the we only require knowledge of $\Pr(z_i - y \equiv 0 \mod 2)$. Assume for now that the adversary is in possession of the full, but shuffled, $\mathbf{y}$. This $\mathbf{y}$ contains two large elements of values, for example values 1498 and 1502, and there is one signature coefficient that is large, for example one element $z = 1500$. Only one of the two elements of y belong to this z, and both elements have a probability of 50 %. However, the probability that the difference of $z - y$ is even is $\Pr(z_i - y \equiv 0 \mod 2) = 1$. In general, if given full (or parts of) shuffled $\mathbf{y}$, all these probabilities can be easily computed as:

$$\Pr(z_i - y \equiv 0 \bmod 2) = \sum_{y_j \in \mathbf{y}: z_i - y_j \equiv 0 \bmod 2} \Pr(z_i \sim y_j)$$

Thus, the error probability for computation over GF(2) at least as small as over $\mathbb{Z}$, and in most cases significantly smaller. Thus, less signatures are required to gather enough samples with low-enough error probability.

4.3 Step 3: Recovering the Position of Twos

After the above second attack step, we know $\mathbf{s}^* \equiv \mathbf{s} \bmod 2$. If we have $d_2 = \delta_2 n > 0$ (i.e., in BLISS-0, BLISS-III or BLISS-IV), we denote $\bar{\mathbf{s}} \in \{0,1\}^n$ the vector with $\bar{\mathbf{s}}_i = 1$ whenever $s_i = \pm 2$, i.e. this vector is non-zero at each coefficient where vector $\mathbf{s}$ has coefficient ± 2.

In the third attack step, we use one of two methods to recover $\bar{\mathbf{s}}$, one based on integer programming and the other based on a maximum likelihood test. Both make use of the fact that the weight κ of the challenge vector $\mathbf{c}$ (and hence also $\mathbf{c}'$) is relatively small. Thus, in any inner product $\langle \mathbf{s}, \mathbf{c}'_i \rangle$, only a small number of coefficients in $\mathbf{s}$ are relevant. From knowledge of $\mathbf{s}^*$, we can immediately derive how many of the selected coefficients are ± 1. We define this quantity as $\eta_1 = \langle \mathbf{s}^*, |\mathbf{c}_i| \rangle$. The other $\kappa - \eta_1$ are then either 0 or ± 2. We define the (unknown) number of twos as $\eta_2 = \langle \bar{\mathbf{s}}, |\mathbf{c}_i| \rangle$, this number is bound by $0 \le \eta_2 \le \min(d_2, \kappa - \eta_1)$

Both methods then compare the output of the side-channel analysis, i.e., $|z_i - y_i| = |\langle \mathbf{s}, \mathbf{c}'_i \rangle|$, to η_1 and use this to derive information on η_2. We will now discuss both methods.

Integer Programming Method. Our first method recovers $\bar{\mathbf{s}}$ by formulating it into an Integer Program. First, suppose we perfectly retrieved y_{ji} from a side-channel. If

$$|z_i - y_i| = |\langle \mathbf{s}, \mathbf{c}'_i \rangle| > \eta_1 + 1,$$

we know that $\eta_2 > 0$, i.e. there has to be a at least one ± 2 involved making up for the difference in the above inequality. We save all $|\mathbf{c}_i|$ for which the above is true in a list $\mathbf{M}$. Then, we need to find a solution $\mathbf{r}$ for the following constraints:

$$\mathbf{Mr} \ge \mathbf{1}.$$

We also add another constraint stating that a solution must satisfy $||\mathbf{r}||_1 = \delta_2 n$, so that we end up with the correct number coefficients in the solution.

Finding the solution $\bar{\mathbf{s}}$ can be seen as a minimal set cover problem. Here, the indices of $\mathbf{M}_i$ form sets and $\mathbf{r}$ a cover. We find the smallest solution for this problem using an Integer Program solver, namely GLPK [40]. Note that by adding more constraints, i.e., more rows in $\mathbf{M}$, the probability that the solver finds the correct solution increases.

The above method cannot be used if the errors in the recovered samples y exceeds ± 2. Such errors could break the Integer program due to conflicting constraints. However, it is possible to deal with ± 1 errors, as the difference between $|z_i - y_i|$ and η_1 needs to be at least 2. Samples with an error of ± 1 can be detected an discarded, simply due to knowledge of the correct parity. Note that in the work of Groot Bruinderink et al. [19], an (idealized) adversary targeting the CDT sampling algorithm only makes errors of ± 1. Hence, this method can be used for this scenario.

Statistical Approach. We now give a second approach that can recover the position of twos in the $\mathbf{s}_1$. It differs from the first as we use a statistical approach rather than integer programming. Thus, it can withstand errors more easily.

We use the following observation. The probability that a certain $z_i - y_i$ is observed clearly depends on η_1 and η_2. If $\eta_2 = 0$, then the probability density function is essentially a binomial distribution picking from $\{\pm 1\}$ instead of the usual $\{0, 1\}$. If $\eta_2 \neq 0$ but $\eta_1 = 0$, the same goes with $\{\pm 2\}$. We compute the joint distribution for all possible combinations of η_1 and η_2

We then perform a standard hypothesis testing. That is, for every recovered sample we compute $\Pr(Z - Y = z_i - y_i | \mathrm{H}_1 = \eta_1, \mathrm{H}_2 = \eta_2)$ for the correct η_1 and for all $0 \le \eta_2 \le \min(d_2, \kappa - \eta_1)$. Note that all distributions, and thus also the joint one, are symmetric. Thus, the actual sign of $z_i - y_i$ is not relevant. Then, we apply Bayes' Theorem to get every $\Pr(\mathrm{H}_2 = \eta_2 | Z - Y = z_i - y_i)$, compute the expected value of H_2, and divide this number by $\min(d_2, \kappa - \eta_1)$. This gives us the probability that any one of the $\min(d_2, \kappa - \eta_1)$ unknown but involved key coefficients is 2.

Finally, we perform a log-likelihood testing. For each unknown coefficient s_k in $\mathbf{s}_1$, we compute the mean of the logarithm of above probability, over all recovered samples where $\mathbf{c}_i$ is 1 at index k. We then set the d_2 coefficients with the highest score to 2.

4.4 Step 4: Recovering $\mathbf{s}_1$ with the Public Key

After the above 3 steps we have recovered $|\mathbf{s}|$. In the fourth and final step, we recover the signs of all its nonzero coefficients and thereby the full signing key $\mathbf{s}$.

We do so by combining all knowledge on $|\mathbf{s}| = |\mathbf{s}_1|$ with the public key. Key generation (Algorithm 1) computes a public key $\mathbf{A} = \{2\mathbf{a}_q, q - 2\}$, with $\mathbf{a}_q = \mathbf{s}_2/\mathbf{s}_1 = (2\mathbf{g} + 1)/\mathbf{f}$ in the ring $\mathcal{R}_q$. In case of the BLISS-I and BLISS-II parameter sets (Table 1), both $\mathbf{f}, \mathbf{g}$ have $\lceil \delta_1 n \rceil = 154$ entries in $\{\pm 1\}$, while all other elements are zero. Thus, both these vectors are *small*.

When writing $\mathbf{s}_1 \cdot \mathbf{a}_q = \mathbf{s}_2$, it is easy to see that $\mathbf{s}_2 = 2\mathbf{g}+1$ is a short vector in the q-ary lattice generated by $\mathbf{a}_q$ (or more correctly, the columns of $\mathbf{A}_q$). Obviously, the parameters of BLISS were chosen in a way such that a straight-forward lattice-basis reduction approach is not feasible. However, knowledge of $|\mathbf{s}|$ allows a reduction of the problem size and thus the ability to recover the key.

With matrix-vector notation, i.e., $\mathbf{s}_1\mathbf{A}_q = \mathbf{s}_2$, it becomes evident that all rows of $\mathbf{A}_q$ at indices where the coefficients of $|\mathbf{s}|$ (and thus $\mathbf{s}_1$) are zero can be simply ignored. Thus, we discard these rows and generate a matrix $\mathbf{A}_q^\star$ with size $(n \times \lceil \delta_1 n \rceil)$, i.e., (512×154) for parameter sets BLISS-I and BLISS-II). Hence, the rank of the lattice, i.e., the number of basis vectors, is decreased.

We further transform the key-recovery problem as follows. First, we do not search for $\mathbf{s}_2$ directly, but instead search for the even shorter $\mathbf{g}$ used in the key-generation process. We have that $\mathbf{f} \cdot \mathbf{a}_q = 2\mathbf{g}+1$, thus $\mathbf{f} \cdot \mathbf{a_q} \cdot 2^{-1} = \mathbf{g}+2^{-1}$ and we simply multiply all elements of $\mathbf{A}_q^\star$ with $2^{-1} \mod q$. We discard the computation of the first coefficient, which contains the added $2^{-1} \mod q$, and thus reduce the dimension of the lattice to $n-1$.

And second, we reduce the lattice dimension further to some d with $\delta_1 n < d < n-1$ by discarding the upper $n-1-d$ coefficients. Hence, we do not search for the full $\mathbf{g}$ but for the d-dimensional sub-vector $\mathbf{g}^\star$. If, on the one hand, this dimension d is too low, then $\mathbf{g}^\star$ is not the shortest vector in the q-ary lattice spanned by the now $(n \times \lceil \delta_1 n \rceil)$ matrix $\mathbf{A}_q^\star$. If, on the other hand, d is chosen too large, then a lattice-reduction algorithm might not be able to find the short $\mathbf{g}^\star$. For our experiments with parameter sets BLISS-I and BLISS-II, we set $d = 250$.

Finally, we feed the basis of the q-ary lattice generated by the columns of $\mathbf{A}_q^\star$ into a basis-reduction, i.e, the BKZ algorithm. The returned shortest-vector is the sought-after $\mathbf{g}^\star$. We then simply solve $\mathbf{f}^\star\mathbf{A}_q^\star = \mathbf{g}^\star$ for $\mathbf{f}^\star \in \mathbb{Z}^{\lceil \delta_1 n \rceil}$. This $\mathbf{f}^\star$ will only consist of elements in ± 1, which are the signs of the nonzero coefficients of the full $\mathbf{f}$. By simply putting the elements of $\mathbf{f}^\star$ into the nonzero coefficients of $\mathbf{s}_1'$, we can fully recover the first part of the signing key $\mathbf{f} = \mathbf{s}_1$. Finally, the second part of the key is $\mathbf{s}_2 = \mathbf{a}_q \cdot \mathbf{s}_1$. Thus, the full signing key is now recovered.

5 EVALUATION OF KEY RECOVERY

In this section, we give an evaluation of our new key-recovery technique. That is, we apply our algorithm on attacks presented in earlier work on original BLISS and compare its performance. Recall, however, that all previous work was unable to perform key-recovery for BLISS-B.

In order to allow a fair comparison, we reuse the modeled and idealized adversaries of earlier work. Concretely, we look at the idealized cache-adversary targeting the CDT sampling algorithm of Groot Bruinderink et al. [19] and the modeled adversaries for the attack on shuffling by Pessl [35]. Thus, for the evaluation our Step 1 is identical to theirs.

We analyze the performance of the following steps in our key recovery. We analyze the key recovery mod 2, i.e., the LPN solving approach (Step 2). Then, we evaluate the success rate of both two-recovery approaches (Step 3). And finally, we state figures for the full-key recovery using a lattice reduction (Step 4).

5.1 Step 2: Key-Recovery mod 2

For evaluation of the second attack step, i.e., mod-2 key recovery, we only consider the BLISS-I parameter set.

Our used LPN approach utilizes differing error probabilities of samples. Its first step is to filter samples, i.e., keep only those with lowest error probability. Evidently, this means that the success probability increases with the number of gathered LPN samples. Thus, we tested the performance for a broad set of observed signatures. For each test, we ran decoding on all 16 hyperthreads of a Xeon E5-2630v3 CPU running at 2.4GHz. If this does not find a solution after at most 10 minutes, then we abort and mark the experiment as failed.

Shuffling. Figure 1 shows the success rate for the attack on shuffling. It was already shown that shuffling once cannot increase security [35]. Thus, we focus solely on the shuffling-twice approach, i.e., $\mathbf{y} = k \cdot \text{Shuffle}(\mathbf{y}') + \text{Shuffle}(\mathbf{y}'')$. For the analysis, we reused previously proposed attacker models [35]. The idealized attacker A1 is given the full $\mathbf{y}', \mathbf{y}''$ but in a random order. Attacker A2 models a profiled side-channel adversary and can classify samples that have some minimum value. Attacker A3 is non-profiled and only detects samples that are uniquely determined with the control flow of the sampling algorithm.

We give results for BLISS-AI (Figure 1a) and BLISS-BI (Figure 1b) separately, as the introduction of the GreedySC algorithm leads to different results and slightly better performance for BLISS-B[4]. For A1, one needs approximately 70 000 signatures in order to achieve a success rate larger than 0.9. A2 needs 90 000 signatures, A3 200 000. Compared to [35], the number of required signatures is cut by a factor of around 3. Note that their numbers require the recoverability of the secret bit b, which we do not need. If this is not given, then their numbers increase by a factor of 6.6. Then, our attack only needs about 1/20th of their signatures.

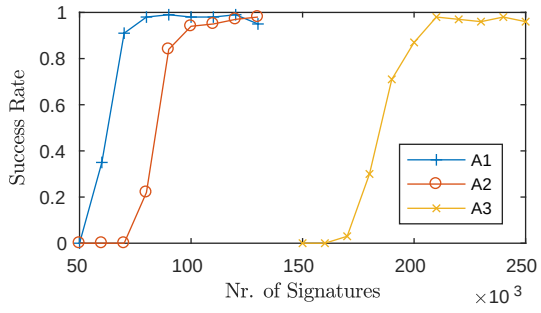
Cache attack on CDT sampling. Figure 2 shows the results of the idealized cache-attack on a CDT sampler by Groot Bruinderink et al. [19]. We did not perceive any significant differences between BLISS-A and BLISS-B here, so we performed experiments for both versions and give the average. We reach a success rate of about 0.9 when using 325 signatures. This is roughly 28 % less than the 450 signatures required in previous work. These savings can be explained as follows. We can now use all recovered samples, and not only those where $z = y$. However, this is somewhat offset by the fact that our LPN-based approach is not as error-tolerant as their lattice-based method which is not applicable in our setting.

5.2 Step 3: Recovery of Twos

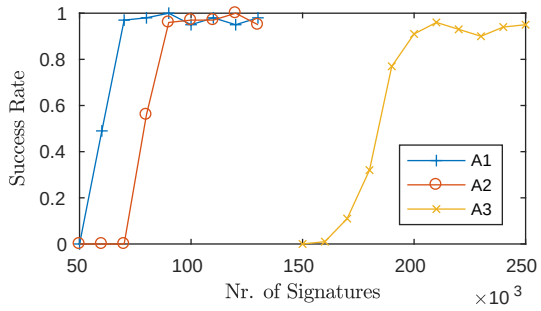
For evaluation of the third attack step, we analyzed the success rate of both twos-recovery procedures (Section 4.3) with the idealized CDT adversary. We consider all parameter sets with $\delta_2 > 0$, i.e., BLISS-0, BLISS-III, and BLISS-IV.

We show the success rate as a function of the number of recovered samples in Figure 3. Please note that this is not equal to the number of required signatures (see [19]). As seen in Figure 3a, the linear-programming approach requires 30 000 samples for BLISS-0 and 400 000 samples for BLISS-III, respectively. Here we did not evaluate the performance with BLISS-IV due to even higher sample requirements. The second approach, which is based on statistical methods, requires more samples for BLISS-0 (45 000) but performs better for BLISS-III (35 000) and BLISS-IV (130 000).

[4] GreedySC aims at minimizing the norm of $\mathbf{s} \cdot \mathbf{c}'$, thus the difference $z - y$ is, on average, smaller. The attack on shuffling benefits from this, as it tests this difference.



(a) BLISS-AI



(b) BLISS-BI

Figure 1: Success rate of LPN decoding for the attack on shuffling

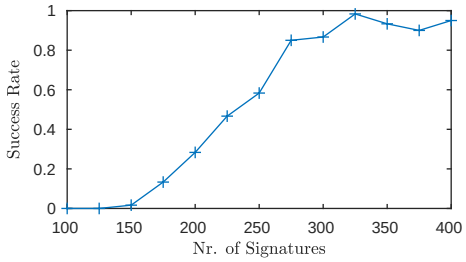
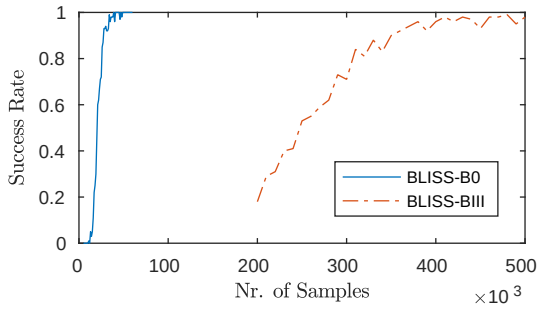
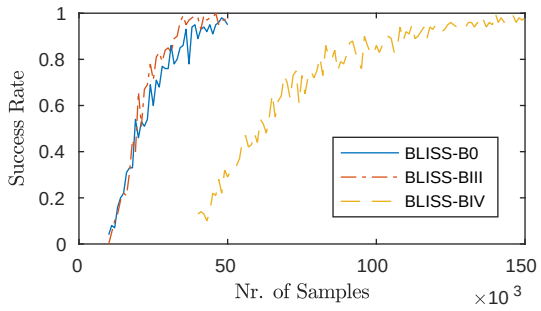


Figure 2: Success rate of LPN decoding for an idealized attack on CDT sampling



(a) Linear Programming



(b) Statistical

Figure 3: Success rate for Twos recovery

5.3 Step 4: Key-Recovery using Lattice Reduction

In the last step, i.e., recovery of the full signing key $\mathbf{s}$ from $|\mathbf{s}|$ (Section 4.4), we use the BKZ lattice-reduction algorithm. Concretely, we use the implementation provided by Shoup's Number Theory Library NTL [43]. We set the BKZ block size to 25 and abort the reduction algorithm as soon as a fitting, i.e., short enough, candidate for the d-dimensional vector $\mathbf{g}^\star$ is found. Such a candidate vector must have a Hamming weight of at most $\lceil \delta_1 n \rceil$ and must consist solely of elements in $\{\pm 1\}$.

We evaluated the correctness and performance of this method by running over 250 key-recovery experiments for both BLISS-AI and BLISS-BI. In each experiment, we generated a new key, performed a key recovery mod 2 (assuming a perfect and errorless side-channel), and finally performed a lattice reduction. All our experiments were successful, hence we can assume that once $\mathbf{s}^* = \mathbf{s}_1 \bmod 2$ is known, the full signing key can always be recovered. The average runtime of lattice reduction (with early abort) was roughly 4-5 minutes on an Intel Xeon E5-2660 v3 running at 2.6GHz.

Other parameter sets. For parameter sets BLISS-0, BLISS-III, and BLISS-IV, we were not able to perform full key-recovery using the above method. In case of BLISS-I and BLISS-II, the Hamming weight of $\mathbf{s}_1$ and hence the rank of the reduced q-ary lattice is $\delta_1 n = 154$. For BLISS-III and BLISS-IV, this quantity increases to 232 and 262, respectively. Due to the resulting increased rank of the lattice, we were not able to recover the key using BKZ.

6 ATTACKING STRONGSWANS BLISS-B

In this section, we perform a cache attack on the BLISS-B implementation of the strongSwan IPsec-based VPN suite [45]. Concretely, we use the parameter set BLISS-I. We describe the setup and the execution of the cache attack in Section 6.1. Our adversary is not synchronized with the victim, thus we perform synchronization based on the signature output (Section 6.2). This corresponds to the first step of our key-recovery method. Finally, we apply the other three steps and describe the outcome.

6.1 Asynchronous Cache Attack

We carry out the experiment on a server featuring an 8 core Intel Xeon E5-2618L v3 2.3GHz processor and 8GB of memory, running a CentOS 6.8 Linux, with gcc 4.4.7. We use strongSwan version 5.5.2, which is the current version at the time of writing. We build strongSwan from the sources with BLISS enabled and with C compile options `-g -falign-functions=64`. To validate the side-channel results against the ground-truth, we collect a trace of key operations executed as part of the signature generation. The trace only has a negligible effect on the timing behaviour of the code and is not used for key extraction.

For the side channel atack, we use the `FR-trace` tool of the Mastik toolkit version 0.02 [47]. `FR-trace` is a command line utility that allows mounting the Flush+Reload attack with amplification. We set `FR-trace` to perform the Flush+Reload attack every 30000 cycles. We describe the locations we monitor below. We set an amplification attack against the function `pos_binary`, which is used as part of Line 1 of Algorithm 7. This slows the average running time of the function from 500 to 233000 cycles, creating a temporal separation between calls to Algorithm 7. However, this slowdown is not uniform and 26% of the calls take less than 30000 cycles, i.e. below the temporal resolution of our attack.

strongSwans implementation of BLISS uses the Bernoulli-sampling approach described in Section 2.3. Thus, we reuse the exploit of Groot Bruinderink et al. [19] and detect if the input to Algorithm 6 was 0. Our cache adversary is asynchronous. Thus, to detect the zero input we have to keep track of several events. First, we detect calls to the Gaussian sampler (Algorithm 7). Second, strongSwan interleaves the sampling of the two noise vectors $\mathbf{y}_1$ and $\mathbf{y}_2$, i.e., it calls the sampler twice in each of the 512 iterations of a loop. As we only target the generation of $\mathbf{y}_1$, we detect the end of each iteration and only use the first call to the Gaussian sampler in each iteration. Third, we track the entry to Algorithm 6 and only use the last entry per sampled value. Other calls to this function correspond to rejections (Lines 5 and 6 of Algorithm 7) and thus cannot be used. Finally, if we detect that Line 3 of Algorithm 6 was not executed, we know that $x = 0$. In this case, the sampled value y is a multiple of $K = 254$.

For BLISS-I, the above events, which we will dub *zero events* from now on, happen on average twice per signature. In order to minimize the error rate, we apply aggressive filtering. Also, we found that possibly due to prefetching, access to Line 3 of Algorithm 6 is often detected although $x = 0$. As a result, we detect zero events on average 0.74 times per signature. 92% of these detections were correct, the other 8% were false positives in which the access to Line 3 was missed by the cache attack.

6.2 Resynchronization

Even though zero events can be detected by an adversary, due to the asynchronous nature of the attack it is not obvious which of the 512 samples corresponds to this detection. In other words, we can detect (with high probability) that there exists a sample $y \in \{0, \pm K, \pm 2K, \ldots\}$, but we do not know *which* sample.

We recover the index i of a detected zero event as follows. First, we locate the first and the last call to the Gaussian sampler in the cache trace. We then estimate the positions of the other 510 calls by placing them evenly in between. Note that Algorithm 7 does not run in constant time, hence this can only give a rough approximation. However, we found that run-time differences average out and that the estimated positions are relatively close to the real calls. In fact, this method gives better results than counting the calls to Algorithm 7 in the trace, as some calls are missed and counting errors accumulate. We also found that the error, i.e., the difference from the estimated index of an event to its real index in the signature, roughly follows a Gaussian distribution with standard deviation 3.5. We then compute the time span between the detected event and the estimated calls to the sampler, match it against the above Gaussian distribution, and then apply Bayes theorem to derive the probability that the detected call to the Gaussian sampler corresponds to each index $0 \ldots 511$ in the signature.

This alone, however, does not allow a sufficient resynchronization. We use the signature output $\mathbf{z}$ in order to further narrow down the index i. For each coefficient in $\mathbf{z}$, we compute the distance d to the closest multiple of parameter K used in Algorithm 7. Then we look up the prior-probability that the sample y corresponding to any signature coefficient z was a multiple of K, this is simply the probability that a coefficient of $\mathbf{s}_1 \cdot \mathbf{c}'$ is equal to d. We estimated this distribution using a histogram approach, it is shown in Figure 4 (for BLISS-I). As $K = 254$ and the coefficient-wise probability distribution of $\mathbf{s}_1 \cdot \mathbf{c}'$ is narrow, many elements of the unknown $\mathbf{y}$ have a zero or very small probability of being a multiple of K. Note that this approach is somewhat similar to the attack on the shuffling countermeasure described in Section 3.3.

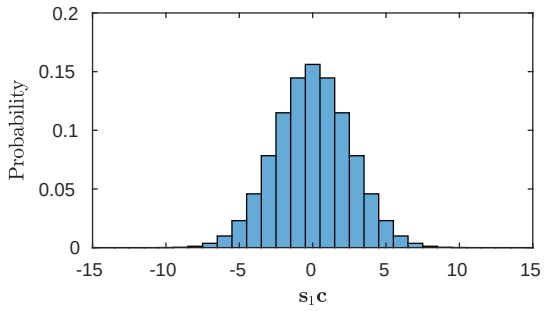


Figure 4: Coefficient-wise probability distribution of $\mathbf{s}_1 \cdot \mathbf{c}'$

Finally, we combine the prior-probabilities derived from the signature output $\mathbf{z}$ with the matching of the trace, which we do by applying Bayes theorem once more. We then use only these zero events that can be reassigned to a single signature index with high probability, i.e., > 0.975, and where the prior-probability $\Pr(\langle \mathbf{s}_1, \mathbf{c}'_i \rangle = d)$ is also high, i.e., $d < 3$.

Roughly 1/3 of detected zero events fulfill both criteria. Out of these, 95% are correct, i.e., correspond to a real zero event and were reassigned to the correct index. Recall that our key-recovery approach only requires the value of $z_i - y_i \bmod 2$. Thus, 97.5% of all recovered samples are correct in GF(2).

6.3 LPN and Results

For LPN-decoding (Section 4.2) we set the code length to 1024. With the above detection rates, we require on average 6 000 signatures in

order to collect this number of samples. Note that for the selection of used samples we again made use of probabilities. For instance, we use only zero events that can be reassigned to a single sample with high probability. Unlike in the case of attacking the shuffling countermeasure or the attack on the CDT sampling algorithm, however, we were not able to accurately determine any differing error probabilities within the selected 1024 samples. Thus, we used a used a non-modified algorithm for decoding the random linear code. Our used decoding algorithm is based on the descriptions in [8].

We performed 100 decoding experiments using the error distribution obtained from the previous step. In the 1024 used samples we encountered between 26 and 36 errors. We ran decoding using 64 threads on two Xeon E5-2699 v4 running at 2.2 GHz. Similar to Section 5.1, we abort decoding after 10 minutes and then consider it to have failed. 98 experiments were successful, 82 of them finished within the first minute.

As we used the parameter set BLISS-I and thus have $\mathbf{s} \in \{0, \pm 1\}$, the third attack step is not required. The fourth attack step, lattice reduction, then finally returns the secret signing key. The runtime of this step was already stated in Section 5.3.

7 CONCLUSION

In this work we present the first side-channel attack against the BLISS-B variant of the BLISS signature scheme. Apart from being able to attack this improved version, the theoretical attack is also more efficient than prior attacks on the BLISS family, requiring only 325 observations by an ideal attacker. We complement the theoretical attack with the first asynchronous cache-based attack against lattice-based cryptography. When using the BLISS-I parameter set, our combined attack is able to recover the BLISS-B secret key after observing 6 000 signatures.

We now give a brief discussion on possible future work, countermeasures against our attack, and ways to avoid Gaussian noise altogether.

Future work. The discrepancy between the low number of observations required for the theoretical attack and the much higher number required for the actual attack is due to the high level of noise we see in the cache channel. Reducing the noise would result in a stronger attack. A promising direction is to combine our attack with the work of Moghimi et al. [27], which investigates attack on the Intel Software Guard Extension (SGX), to see if it is possible to reduce the noise in the SGX setting.

Possible countermeasures. To protect against the side-channel attack described in this paper, it is vital that Algorithm 6 is implemented in constant-time and without secret-dependent branching. More specifically, the handling of rejections and table look-ups should not depend on the input. As shown in Algorithm 8, this can be done by performing all ℓ steps in the loop and always sample an A_i. The return value v is then updated according to the values of A_i and x_i in constant time. We use C-style bitwise-logic operands to describe this update.

Alternatives to Gaussian noise. As our and previous work clearly shows, high-precision Gaussian samplers are a prime target for attacking lattice-based schemes. And while the above countermeasure can fix the exploited leak in this specific implementation, different attack techniques and side-channels can still allow key recovery. Implementing thoroughly secured samplers seems to be a difficult task. In fact, due to their complex structure, implementing them both correctly and efficiently is challenging and error prone already, even without considering implementation security.

Some cryptographers seem to have noted this, as there already exist lattice-based schemes that avoid Gaussians for these reasons. For instance, the NewHope key exchange [2] uses the centered binomial distribution (which is trivial to sample from) as a low-precision approximation to Gaussians. The more recently proposed signature schemes ring-Tesla [1] and Dilithium [15] also avoid discrete Gaussians and use uniform noise instead. Both cite implementation concerns as a motivation for this design choice.

Algorithm 8 Sampling a bit from $\mathcal{B}(\exp(-x/(2\sigma^2)))$ for $x \in [0, 2^\ell)$, constant-time version

Input: $x \in [0, 2^\ell)$ an integer in binary form $x = x_{\ell-1} \dots x_0$. Precomputed table E with $E[i] = \exp(-2^i/(2\sigma^2))$ for $0 \le i < \ell$
Output: A bit b from $\mathcal{B}(\exp(-x/(2\sigma^2)))$

```
v = 1
for i = ℓ − 1 downto 0 do
    sample A_i from B(E[i])
    v = v & (A_i | ~x_i)
return v
```

Acknowledgements
Yuval Yarom performed part of this work as a visiting scholar at the University of Pennsylvania.

This work was supported by the Austrian Research Promotion Agency (FFG) under the COMET K-Project DeSSnet (grant number 862235); by an Endeavour Research Fellowship from the Australian Department of Education and Training; and by the Commission of the European Communities through the Horizon 2020 program under project number 645622 (PQCRYPTO).

REFERENCES

[1] Sedat Akleylek, Nina Bindel, Johannes A. Buchmann, Juliane Krämer, and Giorgia Azzurra Marson. 2016. An Efficient Lattice-Based Signature Scheme with Provably Secure Instantiation. In *AFRICACRYPT 2016*. 44–60.
[2] Erdem Alkim, Léo Ducas, Thomas Pöppelmann, and Peter Schwabe. 2016. Post-quantum Key Exchange - A New Hope. In *25th USENIX Security Symposium*. 327–343.
[3] Thomas Allan, Billy Bob Brumley, Katrina E. Falkner, Joop van de Pol, and Yuval Yarom. 2016. Amplifying side channels through performance degradation. In *ACSAC 2016*. 422–435.
[4] Sonia Belaïd, Jean-Sébastien Coron, Pierre-Alain Fouque, Benoît Gérard, Jean-Gabriel Kammerer, and Emmanuel Prouff. 2015. Improved Side-Channel Analysis of Finite-Field Multiplication. In *CHES 2015*. 395–415.
[5] Sonia Belaïd, Pierre-Alain Fouque, and Benoît Gérard. 2014. Side-Channel Analysis of Multiplications in GF(2^{128}) - Application to AES-GCM. In *ASIACRYPT 2014*. 306–325.
[6] Naomi Benger, Joop van de Pol, Nigel P. Smart, and Yuval Yarom. 2014. "Ooh Aah..., Just a Little Bit": A Small Amount of Side Channel can Go a Long Way. In *CHES 2014*. 75–92.
[7] Daniel J. Bernstein. 2005. Cache-timing attacks on AES. (2005). Preprint available at http://cr.yp.to/antiforgery/cachetiming-20050414.pdf.
[8] Daniel J. Bernstein, Tanja Lange, and Christiane Peters. 2008. Attacking and Defending the McEliece Cryptosystem. In *PQCrypto 2008*. 31–46.
[9] Nina Bindel, Johannes A. Buchmann, and Juliane Krämer. 2016. Lattice-Based Signature Schemes and Their Sensitivity to Fault Attacks. In *FDTC 2016*. 63–77.
[10] Avrim Blum, Adam Kalai, and Hal Wasserman. 2003. Noise-tolerant learning, the parity problem, and the statistical query model. *J. ACM* 50, 4 (2003), 506–519.

[11] Matt Braithwaite. 2016. Experimenting with Post-Quantum Cryptography. (July 2016). https://security.googleblog.com/2016/07/experimenting-with-post-quantum.html.

[12] Yuanmi Chen and Phong Q. Nguyen. 2011. BKZ 2.0: Better Lattice Security Estimates. In *ASIACRYPT 2011*. 1–20.

[13] Léo Ducas. 2014. Accelerating Bliss: the geometry of ternary polynomials. IACR Cryptology ePrint Archive, Report 2014/874. (2014).

[14] Léo Ducas, Alain Durmus, Tancrède Lepoint, and Vadim Lyubashevsky. 2013. Lattice Signatures and Bimodal Gaussians. In *CRYPTO 2013*. 40–56.

[15] Léo Ducas, Tancrède Lepoint, Vadim Lyubashevsky, Peter Schwabe, Gregor Seiler, and Damien Stehlé. 2017. CRYSTALS - Dilithium: Digital Signatures from Module Lattices. Cryptology ePrint Archive, Report 2017/633. (2017).

[16] Thomas Espitau, Pierre-Alain Fouque, Benoît Gérard, and Mehdi Tibouchi. 2016. Loop abort Faults on Lattice-Based Fiat-Shamir & Hash'n Sign signatures. IACR Cryptology ePrint Archive, Report 2016/449. (2016).

[17] Amos Fiat and Adi Shamir. 1986. How to Prove Yourself: Practical Solutions to Identification and Signature Problems. In *CRYPTO 1986*. 186–194.

[18] Qian Ge, Yuval Yarom, David Cock, and Gernot Heiser. 2016. A Survey of Microarchitectural Timing Attacks and Countermeasures on Contemporary Hardware. *Journal of Cryptographic Engineering* (2016).

[19] Leon Groot Bruinderink, Andreas Hülsing, Tanja Lange, and Yuval Yarom. 2016. Flush, Gauss, and Reload - A Cache Attack on the BLISS Lattice-Based Signature Scheme. In *CHES 2016*. 323–345. Full version available at: http://ia.cr/2016/300.

[20] Qian Guo, Thomas Johansson, and Carl Löndahl. 2014. Solving LPN Using Covering Codes. In *ASIACRYPT 2014*. 1–20.

[21] Gorka Irazoqui, Mehmet Sinan Inci, Thomas Eisenbarth, and Berk Sunar. 2014. Wait a minute! A fast, Cross-VM attack on AES. In *RAID 2014*. 299–319.

[22] Adam Langley. 2016. CECPQ1 results. (November 2016). https://www.imperialviolet.org/2016/11/28/cecpq1.html.

[23] A. K. Lenstra, H. W. Lenstra, and L. Lovász. 1982. Factoring polynomials with rational coefficients. *Math. Ann.* 261, 4 (1982), 515–534.

[24] Éric Levieil and Pierre-Alain Fouque. 2006. An Improved LPN Algorithm. In *SCN 2006*. 348–359.

[25] Richard Lindner and Chris Peikert. 2011. Better Key Sizes (and Attacks) for LWE-Based Encryption. In *CT-RSA 2011*. 319–339.

[26] Matteo Mariantoni. 2014. Building a superconducting quantum computer - Invited Talk in PQCrypto 2014. (October 2014). https://www.youtube.com/watch?v=wWHAs--HA1c.

[27] Ahmad Moghimi, Gorka Irazoqui, and Thomas Eisenbarth. 2017. CacheZoom: How SGX Amplifies The Power of Cache Attacks. *CoRR* abs/1703.06986 (2017).

[28] NIST. 2016. Post-Quantum crypto standardization. (December 2016). http://csrc.nist.gov/groups/ST/post-quantum-crypto/call-for-proposals-2016.html.

[29] Tobias Oder, Thomas Pöppelmann, and Tim Güneysu. 2014. Beyond ECDSA and RSA: Lattice-based Digital Signatures on Constrained Devices. In *DAC 2014*. 110:1–110:6.

[30] Committee on National Security Systems. 2015. Use of Public Standards for the Secure Sharing of Information Among National Security Systems. CNSS Advisory Memorandum Information Assurance 02-15. (July 2015).

[31] Yossef Oren, Vasileios P. Kemerlis, Simha Sethumadhavan, and Angelos D. Keromytis. 2015. The Spy in the Sandbox: Practical Cache Attacks in JavaScript and their Implications. In *23rd CCS*. 1406–1418.

[32] Dag Arne Osvik, Adi Shamir, and Eran Tromer. 2006. Cache Attacks and Countermeasures: The Case of AES. In *CT-RSA 2006*. 1–20.

[33] Cesar Pereida García and Billy Bob Brumley. 2016. Constant-Time Callees with Variable-Time Callers. IACR Cryptology ePrint Archive, Report 2016/1195. (2016).

[34] Cesar Pereida García, Billy Bob Brumley, and Yuval Yarom. 2016. "Make Sure DSA Signing Exponentiations Really are Constant-Time". In *CCS 2016*. 1639–1650.

[35] Peter Pessl. 2016. Analyzing the Shuffling Side-Channel Countermeasure for Lattice-Based Signatures. In *INDOCRYPT 2016*. 153–170.

[36] Peter Pessl and Stefan Mangard. 2016. Enhancing Side-Channel Analysis of Binary-Field Multiplication with Bit Reliability. In *CT-RSA 2016*. 255–270.

[37] Krzysztof Pietrzak. 2012. Cryptography from Learning Parity with Noise. In *SOFSEM 2012*. 99–114.

[38] Joop van de Pol, Nigel P. Smart, and Yuval Yarom. 2015. Just a Little Bit More. In *CT-RSA 2015*. 3–21.

[39] Thomas Pöppelmann, Léo Ducas, and Tim Güneysu. 2014. Enhanced Lattice-Based Signatures on Reconfigurable Hardware. In *CHES 2014*. 353–370.

[40] GNU Project. n.d.. GLPK (GNU Linear Programming Kit). (n.d.). https://www.gnu.org/software/glpk/.

[41] Oded Regev. 2005. On lattices, learning with errors, random linear codes, and cryptography. In *STOC 2005*. 84–93.

[42] Markku-Juhani O. Saarinen. 2017. Arithmetic coding and blinding countermeasures for lattice signatures. *Journal of Cryptographic Engineering* (2017).

[43] Victor Shoup. n.d.. NTL: A Library for doing Number Theory. (n.d.). http://www.shoup.net/ntl/.

[44] Jacques Stern. 1988. A method for finding codewords of small weight. In *Coding Theory and Applications 1988*. 106–113.

[45] strongSwan. 2015. strongSwan 5.2.2 Released. https://www.strongswan.org/blog/2015/01/05/strongswan-5.2.2-released.html. (2015).

[46] Yukiyasu Tsunoo, Etsuko Tsujihara, Kazuhiko Minematsu, and Hiroshi Hiyauchi. 2002. Cryptanalysis of Block Ciphers Implemented on Computers with Cache. In *International Symposium on Information Theory and Its Applications*. 803–806.

[47] Yuval Yarom. 2016. Mastik: A Micro-Architectural Side-Channel Toolkit. http://cs.adelaide.edu.au/~yval/Mastik/Mastik.pdf. (Sept. 2016).

[48] Yuval Yarom and Naomi Benger. 2014. Recovering OpenSSL ECDSA Nonces Using the Flush+Reload Cache Side-channel Attack. IACR Cryptology ePrint Archive, Report 2014/140. (2014).

[49] Yuval Yarom and Katrina Falkner. 2014. FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack. In *23rd USENIX Security Symposium*. 719–732.

[50] Yinqian Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. 2014. Cross-Tenant Side-Channel Attacks in PaaS Clouds. In *22nd CCS*. 990–1003.