GALACTICS: Gaussian Sampling for Lattice-Based Constant-Time Implementation of Cryptographic Signatures, Revisited

GILLES BARTHE, MPI-SP and IMDEA Software Institute
SONIA BELAÏD, CryptoExperts
THOMAS ESPITAU, Sorbonne Université, Laboratoire d'informatique de Paris 6
PIERRE-ALAIN FOUQUE, Université de Rennes
MÉLISSA ROSSI, Thales, ENS Paris, CNRS, PSL University, INRIA
MEHDI TIBOUCHI, NTT Corporation

In this paper, we propose a constant-time implementation of the BLISS lattice-based signature scheme. BLISS is possibly the most efficient lattice-based signature scheme proposed so far, with a level of performance on par with widely used pre-quantum primitives like ECDSA. It is only one of the few postquantum signatures to have seen real-world deployment, as part of the strongSwan VPN software suite.

The outstanding performance of the BLISS signature scheme stems in large part from its reliance on discrete Gaussian distributions, which allow for better parameters and security reductions. However, that advantage has also proved to be its Achilles' heel, as discrete Gaussians pose serious challenges in terms of secure implementations. Implementations of BLISS so far have included secret-dependent branches and memory accesses, both as part of the discrete Gaussian sampling and of the essential rejection sampling step in signature generation. These defects have led to multiple devastating timing attacks, and were a key reason why BLISS was not submitted to the NIST postquantum standardization effort. In fact, almost all of the actual candidates chose to stay away from Gaussians despite their efficiency advantage, due to the serious concerns surrounding implementation security.

Moreover, naive countermeasures will often not cut it: we show that a reasonable-looking countermeasure suggested in previous work to protect the BLISS rejection sampling can again be defeated using novel timing attacks, in which the timing information is fed to phase retrieval machine learning algorithm in order to achieve a full key recovery.

Fortunately, we also present careful implementation techniques that allow us to describe an implementation of BLISS with complete timing attack protection, achieving the same level of efficiency as the original unprotected code, without resorting on floating point arithmetic or platform-specific optimizations like AVX intrinsics. These techniques, including a new approach to the polynomial approximation of transcendental function, can also be applied to the masking of the BLISS signature scheme, and will hopefully make more efficient and secure implementations of lattice-based cryptography possible going forward.

Additional Key Words and Phrases: Timing Attack; Phase Retrieval algorithms; Constant-time Implementation; Lattice-based Cryptography; Masking Countermeasure

INTRODUCTION

The looming threat of general-purpose quantum computers against legacy public-key cryptographic schemes makes it a pressing problem to prepare the concrete transition to postquantum cryptgraphy. Lattice-based cryptography, in particular, offers an attractive alternative to currently deployed schemes based e.g. on RSA and elliptic curves, thanks to strong theoretical security guarantees, a large array of achievable primitives, and a level of efficiency that can rival pre-quantum constructions.

Despite their attractive theoretical properties, however, lattice-based constructions present novel challenges in terms of implementation security, particularly with respect to side-channel attacks. Taking signatures as an example, possibly the most efficient construction proposed so far is the BLISS signature scheme of Ducas et al. [16], which features excellent performance and has seen

real-world deployment via the VPN software suite strongSwan. Later implementations of BLISS show good hardware performance as well [34]. However, existing implementations of BLISS suffer from significant leakage through timing side-channels, which have led to several devastating attacks against the scheme [6, 9, 22, 33, 40]. The main feature of BLISS exploited in these attacks in the use of discrete Gaussian distributions, either as part of the Gaussian sampling used to generate the random nonces in BLISS signatures, or as part of the crucial rejection sampling step that forms the core of the Fiat–Shamir with aborts framework that supports BLISS's security.

Generally speaking, Gaussian distributions are ubiquitous in theoretical works on lattice-based cryptography, thanks to their convenient behavior with respect to proofs of security and parameter choices. However, their role in practical implementations is less clear, largely because of the concerns surrounding implementation attacks. For example, BLISS was not submitted to the NIST postquantum standardization effort due to those concerns, and second round candidate Dilithium [18], which can be seen as a direct successor of BLISS, replaces Gaussian distributions by uniform ones, at the cost of larger parameters and a less efficient implementation, specifically citing implementation issues as their justification.

In this paper we study the security of the BLISS implementation against cache-based timing and power side-channel attacks. Specifically, we develop efficient implementations of BLISS that are secure against these attacks. Although our results target BLISS in particular, our techniques can be applied to the very large class of constructions based on discrete Gaussian distributions (at least those that use Gaussians with fixed standard deviation), which form the bulk of works on lattice-based cryptography. Protecting implementations for these constructions are challenging because state-of-the-art techniques for constant-time and masked implementations mainly consider deterministic programs (and thus in particular for programs with deterministic control-flow). However, schemes that involve Gaussian sampling. In particular, signature schemes within the Fiat–Shamir with aborts framework use rejection sampling, also called the acceptance-rejection method. To sample from a distribution X, with density f, one uses samples from the distribution Y with density g as follows:

(1) Get a sample y from distribution Y and a sample u from the uniform distribution on $(0, 1)$,
(2) If $u < f(y)/Mg(y)$, accept y as a sample drawn from X, and reject otherwise.

This algorithm requires M iterations on average to obtain a sample and in particular does not have deterministic control flow. A further difficulty with BLISS is that floating-point operations are generally not constant-time, and yet the computation of the function $f(y)/g(y)$ involves transcendental functions. It is thus an additional difficulty to implement it purely in terms of integer arithmetic.

Our Contributions. First of all, we present a new timing attack against a countermeasure previously suggested in [22] which avoids some earlier attacks [6, 22]. Previous attacks target the Bernoulli sampling algorithm, while we look at the implementation of the hyperbolic cosine function. We show that the computation of this transcendental function leaks secret information and that by measuring the number of times this algorithm restarts, we can recover the secret key. The available information is similar to the one used in the phase retrieval problem, given $|\langle \mathbf{a}, \mathbf{s} \rangle|$ for known and random samples $\mathbf{a}$, recover $\mathbf{s}$. The general phase retrieval problem [10] is that $\langle \mathbf{a}, \mathbf{s} \rangle$ can be a complex value and we only get the amplitude and not the phase of this value. In the particular case of this problem, where the scalar product is real, we devise 2 new efficient algorithms. The first attack only uses the samples that the scalar product is null, which is not too restrictive here since we have many such samples and performs a Principal Component Analysis algorithm. The second attack takes into account all the information by using maximum likelihood estimator for combining the correlation between many samples and perform a gradient descent. The difficulties in our algorithms come from the truncation at the end of BLISS, to make the signature compact,

that introduces a lot of noise in our samples. Finally, both attacks rely on a clever use of lattice reduction algorithm to recover all the secret information even though some errors are still present at the end of the descent.

Then, we propose a constant-time implementation of BLISS, mainly relying on an alternate implementation of the rejection sampling step, carried out by computing a sufficiently precise polynomial approximation of the rejection probability using pure integer arithmetic. We manage to do so using a novel technique for polynomial approximation, relying on lattice reduction for the Euclidean inner product derived from the Sobolev norm. This approach has several advantageous properties compared to methods based on minimax computations, as implemented e.g. in the Sollya software package [11], especially in terms of its control on the shape of polynomial approximants we can obtain. Our constant-time implementation, written in portable C using pure division-free integer arithmetic, achieves the same level of efficiency as the original, variable-time implementation of BLISS, and outperforms Dilithium by a large margin. We also provide experimental validation using the `dudect` tool of Reparaz et al. that the implementation is indeed constant-time.

Using similar techniques, together with a proof strategy analogous to [3], we also show how to construct a masked implementation of BLISS secure against high-order side-channel attacks.

Related Work. Several works have exhibited side-channel attacks against BLISS [6, 9, 22, 33]. These attacks epitomize the difficulties to implement lattice-based schemes securely, particularly when Gaussians are involved. However, there are also positive results showing that it is possible to make this signature scheme secure against such attacks. For instance, Barthe *et al.* [3] propose a secure implementation against side-channel attack for the GLP signature [23]. This implementation is made secure using the classical masking countermeasure used to prevent SPA and DPA analysis. The security proof uses the strong non-interference property introduced in [2], which allows to reason compositionally, and a relaxation of masking called masking under public outputs. However, the masked implementation and security proof of GLP relies critically on the fact that samplings are drawn from uniform distributions as Dilithium.

There exist a number of works devoted to constant-time techniques for sampling according to discrete Gaussian distributions [20, 28, 32] or related distributions, such as rounded Gaussians [26]. There are different methods according to the size of the standard deviation, as well as whether it is constant or varies: encryption scheme typically require small standard deviations, while signatures use larger ones, which are fixed for Fiat–Shamir schemes and vary for hash-and-sign constructions. To deal with large standard deviations, it is customary to use a small standard deviation "base sampler" and build upon it to achieve the desired standard deviation: this is the approach presented in [32]. These works can also be distinguished according as whether they require floating point arithmetic. In particular, the rounded Gaussians of [26] offer numerous attractive properties, but they have some statistical limitations in terms of distinguishing advantage, and they rely on floating point implementations.

Approximating the exponential function with polynomials has been recently proposed in implementations of Gaussian sampling algorithms. Here, we apply this technique for the transcendental functions used in the rejection sampling part of the signing algorithm. Prest was the first to propose such ideas in [35] and he showed that with 53-bit of precision for floating numbers we can have 256-bit security when the number of signatures is limited to 2^{64}, as is stated in the requirements of the NIST standardization process. NIST second round candidate Falcon [36] is based on a Gaussian sampler that uses Padé approximants to evaluate the exponential function of the rejection probability. More recently, Zhao et al. have proposed a polynomial approximation without floating point division [42], since that operation is known to not be constant-time. They use floating point multiplications instead to compute the exponential, but this instruction does not always

have constant-time execution guarantees either, unfortunately. In this paper, we approximate the exponential and the hyperbolic cosine functions over an interval using integer polynomials to avoid floating operations. Moreover, we aim to approximate using polynomials with small coefficients so that we can use small-sized integers and obtain a straightforward implementation of Horner's algorithm.

Organisation of the paper. In Section 1, we recall the BLISS signature scheme and the implementation of the rejection sampling algorithm. Then, we describe our timing attacks on the hyperbolic cosine implementation in Section 2. In Section 3, we present our constant-time implementation of the BLISS signature scheme. Section 4 presents the technique we used to approximate transcendental functions with integral polynomials. Our masked implementation is introduced in the Appendix A.

1 DESCRIPTION OF THE BLISS SCHEME

Notations. 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 $\|\mathbf{v}\|_\infty = \max_i |v_i|$. The notation $\lfloor x \rceil_d$ denotes the centered d highest-order significant bits of x: $x = \lfloor x \rceil_d \cdot 2^d + x'$ with $x' \in [-2^{d-1}, 2^{d-1})$.

Overall description of BLISS. The BLISS signature scheme [16] is possibly the most efficient lattice-based signature scheme so far. It has been implemented in both software [16] and hardware [34]. BLISS can be seen as a ring-based optimization of the earlier lattice-based scheme of Lyubashevsky [31], sharing the same "Fiat–Shamir with aborts" structure [30].

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. See Figure 1 for a description of the key generation. $\kappa, C, \delta_1, \delta_2, q, p$ are parameters detailed in BLISS specifications and N_κ is the function depicted in Equation 3.2.4. To sign a message μ, one first generates commitment values $\mathbf{y}_1, \mathbf{y}_2 \in \mathcal{R}$ with normally distributed coefficients (distribution denoted $\mathscr{D}$), and then computes a hash $\mathbf{c}$ of the message μ together with $\mathbf{u} = -\mathbf{a}_q\mathbf{y}_1 + \mathbf{y}_2 \bmod q$ using a cryptographic hash function modeled as a random oracle taking values in the set of elements of $\mathcal{R}$ with exactly κ coefficients equal to 1 and the others to 0. 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 signature procedure, described in Figure 2, 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 signature size, the signature element $\mathbf{z}_2$ is actually transmitted in compressed form $\mathbf{z}_2^\dagger$, and accordingly the hash input includes only a compressed version of $\mathbf{u}$.

The verification procedure is given in Figure 3 for completeness since it does not manipulate sensitive data. B_1 and B_2 are parameters detailed in BLISS specifications.

The original BLISS paper describes a family of concrete parameters for the signature scheme, recalled in Table 1. The BLISS–I and BLISS–II parameter sets are optimized for speed and compactness respectively, and both target 128 bits of security. BLISS–III and BLISS–IV are claimed to offer 160 and 192 bits of security respectively. Finally, a BLISS–0 variant is also given as a toy exemple offering a relatively low security level.

```
1: function KeyGen
2:     Generate two polynomials f and g uniformly at random with exactly nδ₁ entries in {±1} and
   nδ₂ entries in {±2}
3:     S = (s₁, s₂) = (f, 2·g + 1)^t
4:     rejection sampling: restart to step 2 if N_κ(S) ≥ C²·5·(⌈δ₁·n⌉ + 4·⌈δ₂·n⌉)·κ
5:     a_q = (2·g + 1)/f  mod q (restart if f is not invertible.)
6:     return (A, S) where A = (2·a_q, q − 2)  mod 2q
7: end function
```

Fig. 1. Description of the BLISS key generation algorithm.

```
1: function Sign(μ, pk = a₁, sk = S)
2:     y₁ ← 𝒟ⁿ, y₂ ← 𝒟ⁿ
3:     u = ζ·a₁·y₁ + y₂ mod 2q                          ▷ ζ(q − 2) = 1 mod 2q
4:     c ← H(⌊u⌉_d mod p, μ)
5:     choose a random bit b
6:     z₁ ← y₁ + (−1)^b s₁c
7:     z₂ ← y₂ + (−1)^b s₂c
8:     rejection sampling: restart to step 2 except with probability
   1/(M exp(−‖Sc‖²/(2σ²)) cosh(⟨z, Sc⟩/σ²))
9:     z₂† ← (⌊u⌉_d − ⌊u − z₂⌉_d) mod p
10:    return (z₁, z₂†, c)
11: end function
```

Fig. 2. Description of the BLISS signature algorithm.

```
1: function Verify(μ, pk = a₁, (z₁, z₂†, c))
2:     if ‖(z₁‖2^d·z₂†)‖₂ > B₂ then reject
3:     if ‖(z₁‖2^d·z₂†)‖_∞ > B_∞ then reject
4:     t ← H(⌊ζ·a₁·z₁ + ζ·q·c⌉_d + z₂† mod p, μ)
5:     if t ≠ c then reject
6:     return accept
7: end function
```

Fig. 3. Description of the BLISS verification algorithm.

2 A TIMING ATTACK ON BLISS

We use statistical learning techniques to recover the second part $\mathbf{s}_2$ of the secret key by using either PCA or either Phase Retrieval algorithm. The main difficulties come from the final compression that adds a lot of noise to the samples. For some BLISS parameters, the noise is too high for the first

Table 1. Concrete parameters for BLISS.

BLISS−	0	I	II	III	IV
n	256	512	512	512	512
q	7681	12289	12289	12289	12289
δ_1, δ_2	.55 , .15	.3 , 0	.3 , 0	.42 , .03	.45, .06
d	5	10	10	9	8
κ	12	23	23	30	39
α	0.5	1.0	0.5	0.7	0.55

```
1: function SampleBernCosh(x)
2:     x ← |x|
3:     Sample a ← SampleBernExp(x)
4:     Sample b ← B_{1/2}
5:     Sample c ← SampleBernExp(x)
6:     if ¬a ∧ (b ∨ c) then restart
7:     return a
8: end function
```

Fig. 4. 1/cosh Bernoulli sampling with countermeasure.

attack to succeed in a complete key recovery. The second attack first uses a maximum likelihood principle to recover an estimate of the of absolute value of the scalar product given the timing information. Then, a phase retrieval algorithm is run. However, since the noise is high and the problem is non-convex, the initialization phase of the gradient descent is crucial. To this end, we develop a new and refined initialization process improving [29]. Finally, we use a lattice reduction to remove a few errors on $\mathbf{s}_2$.

2.1 Leakage of the cosh sampler

Suppose that, as suggested by the countermeasures of [22], the exponential sampler SampleBernExp is constant time. From the specification of SampleBernCosh and following [16], a natural implementation of this function would be given as the pseudocode of Figure 4. However, there still exists a timing leakage from this implementation of the hyperbolic cosine sampler.

Indeed, by definition of the function SampleBernCosh, the probability of outputting a is equal to the probability of the expression $\neg a \wedge (b \vee c)$ to be false, which is

$$\begin{aligned} p(\mathrm{S}, \mathbf{c}, \mathbf{z}) &= 1 - \Pr(\neg a)\Pr(b \vee c) \\ &= 1 - (1 - \Pr(a))(1 - \Pr(\neg b \wedge \neg c)) \\ &= 1 - \left(1 - e^{-\frac{|\langle \mathbf{z}, \mathrm{S}\mathbf{c}\rangle|}{2\sigma^2}}\right)\left(1 - \frac{1 - e^{-\frac{|\langle \mathbf{z}, \mathrm{S}\mathbf{c}\rangle|}{2\sigma^2}}}{2}\right) = \frac{1 + e^{-\frac{|\langle \mathbf{z}, \mathrm{S}\mathbf{c}\rangle|}{\sigma^2}}}{2}. \end{aligned}$$

Hence by measuring the differences in computation time, one can derive traces that shape $(\mathbf{z}, \mathbf{c}, t)$, where $t \in \mathbb{N}$ is the number of restarts performed before outputting the value a. In the following of this section, we describe two ways to exploit this leakage, leading to a full key recovery.

```
1: Collect m traces (z_i, c_i, t_i)
2: for i = 0 to m do
3:     if t_i = 0 then W ← c_i z_i^* · c_i z_i^{*T} end if
4: end for
5: S ←$ N(0,1)^n; S ← S/||s_0||
6: for i = 0 to K then
7:     S ← W^{-1} S; S ← S/||s_0||
8: end for
9: return round(S/||S|| N)
```

Fig. 5. First timing attack on Bernoulli sampler.

2.2 Spectral attack with samples with $t = 0$

Remark that if a trace satisfies $t = 0$, then it is likely for the geometric distribution parameter $p(\mathrm{S}, \mathbf{c}, \mathbf{z})$ to be large, since for $t = 0$, the likelihood function is exactly p. Therefore, for such a sample, $\langle \mathbf{z}, \mathrm{S}\mathbf{c} \rangle$ should be close to zero, i.e., S should be *close to* orthogonal to the vector $\mathbf{z}\mathbf{c}^*$, where $\mathbf{c}^*$ is the adjoint of $\mathbf{c}$: $\langle \mathbf{z}, \mathrm{S}\mathbf{c} \rangle = \langle \mathbf{z}\mathbf{c}^*, \mathrm{S} \rangle$.

If the vector S was actually orthogonal to each of these $\mathbf{z}\mathbf{c}^*$ then it would be enough to collect sufficiently of them so that they generate an hyperplane $\mathcal{H}$ of the ambient space $\mathbb{R}^n$ and return the unique (up to sign) vector of $\mathcal{H}^\perp$ of norm compatible with the specification of BLISS (secret vectors in BLISS all have the same known norm by construction). This would practically translate in constructing the empirical covariance matrix $W = \sum_i \mathbf{w}_i \mathbf{w}_i^T$ ($\mathbf{w}_i = \mathbf{z}_i \mathbf{c}_i^*$) for a series of trace $(\mathbf{z}_i, \mathbf{c}_i, 0)$ and get a basis of its kernel. Remark now that since the secret is not actually orthogonal to these vectors, the obtained matrix is not singular. To overcome this difficulty we thus do not seek a vector in the kernel but instead in the eigenspace associated with the smallest eigenvalue of W. This technique can be seen as a continuous relaxation of the kernel computation in the ideal case. It translates directly into pseudocode in Figure 5, where the computation of the eigenvector is performed iteratively and $N = \lceil \delta_1 n \rceil + 4\lceil \delta_2 n \rceil$ is the norm of the secret key. Remark that this technique does not recover exactly the secret but an *approximate* solution over the reals. To recover the secret we need to find the closest integral vector to the output candidate, which is simply done by rounding each coefficient to the nearest integral elements. In addition, remark that by the contruction of the public key from the secret one, recovering solely $\mathbf{s}_2$ is sufficient to reconstruct the full secret key. Hence the rounding can be carried to $2\mathbb{Z}$ on the second part of the eigenvector to conclude, as $\mathbf{s}_2$ has its coefficients equal to 0, ± 2 or ± 4 by construction.

2.3 A timing attack by phase retrieval

Exploiting the leakage described in Section 2.1 boils down to retrieve S up to sign from a family of values of the shape $(\mathbf{z}_i, \mathbf{c}_i, t_i)$ where t_i is sampled under a *geometric* distribution of parameter $p(\mathrm{S}, \mathbf{c}_i, \mathbf{z}_i)$. A natural approach would then consist in starting by estimating the values of $p(\mathrm{S}, \mathbf{c}_i, \mathbf{z}_i)$ for each trace $(\mathbf{c}_i, \mathbf{z}_i, t_i)$, yielding a (noisy) estimate of the absolute value of the inner product $|\langle \mathbf{z}_i, \mathrm{S}\mathbf{c}_i \rangle| = |\langle \mathbf{z}_i \mathbf{c}_i^*, \mathrm{S} \rangle|$. In a second time we then fall back on retrieving S from samples of the form $(|\langle \mathbf{w}_i, \mathrm{S} \rangle|, \mathbf{w}_i)$. This is an instance of so-called (noisy) *phase retrieval problem.*

2.3.1 First phase: estimation of the phases. In order to get a (noisy) evaluation of the phases, we devise an estimator of maximum likelihood. Set $\mathcal{L}_i(\omega)$ to be the logarithm of the probability $\Pr[|\langle \mathrm{S}, \mathbf{w}_i \rangle| = x | t = \omega]$. We then set the estimator y_i to be the arguments of the maximum of $\mathcal{L}_i(t_i)$ for each trace. Such a computation is classically done using Bayes' theorem and seeking for critical values from the derivates of $\mathcal{L}_i(\omega)$.

```
1: A ← [w_1 | ··· | w_m]
2: s_0 ←$ N(0,1)^n
3: for i = 0 to K then
4:     s_0 ← A^T diag(y_1, ..., y_m) A s_0
5:     s_0 ← (A^T A)^{-1} s_0
6: s_0 ← s_0 / ||s_0||
7: end for
8: s_0 ← (s_0 / ||s_0||) N
9: return rounding(s_0)
```

Fig. 6. Spectral initializer algorithm.

2.3.2 Second phase: solving the phase retrieval instance. Phase retrieval aims at solving quadratic equations of the shape

$$|\langle \mathrm{S}, \mathbf{w}_i \rangle|^2 = y_i \quad i = 1, \ldots, m,$$

where S is the decision variable, the $\mathbf{w}_i$ are known sampling vectors and the $y_i \in \mathbb{R}$ are the phase measurements. The noisy version of this problem consists in retrieving the variable S from noisy quadratic equations:

$$|\langle \mathrm{S}, \mathbf{w}_i \rangle|^2 + e_i = y_i \quad i = 1, \ldots, m,$$

for e_i independents (usually gaussian) random variables. This problem has been widely studied in the fields of statistical learning and the most common approach to tackle it consists of a two-step strategy:

2.3.3 Initialization via spectral method. First, find a candidate vector $\mathbf{s}_0$ that is sufficiently close to the actual solution to make the second step converges towards the actual solution. The usual way to initialize the candidate vector can be seen as a generalization of the principal component analysis (PCA): the initial guess is given via a spectral method; in short, $\mathbf{s}_0$ is the leading eigenvector of the positive definite symmetric matrix $\sum_i y_i \mathbf{w}_i \mathbf{w}_i^T$. The intuition behind this method is to remark that the secret vector will have a greater inner product with the test vectors $\mathbf{w}_i$ which have a small angle with it. Hence we want to extract the direction of the w_i for which the inner product is the largest, that is, favorizing the components inducing high y_i's. This corresponds to extract the largest eigenvalue of the Gram-matrix of the w_i, normalized by a diagonal matrix of y_i. It is nothing more than a principal componant analysis on the test vectors $\mathbf{w}_i$. In practice, we use a slightly different version of the (iterative version of the) spectral initializer, outlined in Figure 6, which provides slightly better practical results than the classical method of [10]. $\mathcal{N}(0,1)$ is the centered normal reduced distribution, K is a constant, set sufficiently large.

2.3.4 The descent phase. Once an initialization vector is found, we iteratively try to make it closer to the actual secret by a series of updates like in a gradient descent scheme. Note that in the problem of phase retrieval the problem is non-convex so that a direct gradient descent would not be directly applicable. As stated in [10], the phase retrieval problem can be stated as a minimization problem:

$$\text{minimize} \quad \frac{1}{2m} \sum_{r=1}^{m} \ell(y_r, |\langle \mathbf{w}_r, \mathbf{x} \rangle|^2), \mathbf{z} \in \mathbb{R}^n, \tag{1}$$

where ℓ is a distance function over the reals (such as the Euclidean distance $\ell_2(a,b) = (a-b)^2$). The corresponding descent, called Wirtinger flow, is then simply stated in Figure 7 where $t \mapsto \mu_t$ is a step function, which has to be experimentally tailored to optimize the convergence. The value $\epsilon > 0$ is a small constant that determines the desired precision of the solution.

```
1: t ← 0
2: do
3:     s_{t+1} ← s_t − μ_t/(m‖s_0‖^2) Σ_{r=1}^{m} (|⟨w_r, s_t⟩|^2 − y_r)(w_r w_r^t) s_t
4:     t ← t + 1
5: while ‖s_t − s_{t+1}‖ > ε
6: return S
```

1: $t \leftarrow 0$
2: **do**
3: $\quad \mathbf{s}_{t+1} \leftarrow \mathbf{s}_t - \frac{\mu_t}{m\|\mathbf{s}_0\|^2} \sum_{r=1}^{m} (|\langle \mathbf{w}_r, \mathbf{s}_t \rangle|^2 - y_r)(\mathbf{w}_r \mathbf{w}_r^t)\mathbf{s}_t$
4: $\quad t \leftarrow t + 1$
5: **while** $\|\mathbf{s}_t - \mathbf{s}_{t+1}\| > \epsilon$
6: **return** S

Fig. 7. Wirtinger flow descent algorithm.

1: Collect m traces $(\mathbf{z_i}, \mathbf{c_i}, t_i)$
2: **for** $i = 0$ **to** m **do**
3: $\quad y_i \leftarrow \left(\mathrm{argmax}_x \mathcal{L}_i(t_i)\right)^2$
4: **end for**
5: $\mathbf{s}_0 \leftarrow$ Spectral initialization (Figure 6)
6: S $\leftarrow$ Descent($\mathbf{s}_0$) (Figure 7)
7: **return** S

Fig. 8. Timing attack on the Bernoulli sampler.

It is well known that minimizing non-convex objectives, which may have very many stationary points is in general NP-hard. Nonetheless if the initialization $\mathbf{s}_0$ is sufficiently accurate, then the sequence $\mathbf{s_i}$ will converge toward a solution to the problem given by Equation (1).

As in the first attack, the descent algorithm does not directly give an integral solution to the retrieval problem, so that we eventually need to round the coefficients before outputting the solution.

The full outline of the attack is given in Figure 8.

2.4 Reducing the number of samples by error localization and dimension reduction

By the inherent noisy nature of the problem, if not enough samples are used to mount the attack, the recovery might fail on a certain amount of bits. In such a case one cannot figure *a priori* where these errors are and would be forced to enumerate the possible errors, using, for instance, the hybrid MiTM technique of Howgrave-Graham [25]. Since the dimension ($n = 512$) is large, such an approach becomes quickly untractable as the number of errors is greater than 8.

However, as the final step of both of the attacks consists of a coefficient-wise rounding, we can study the distance of each coefficient to $2\mathbb{Z}$. Heuristically since the descent is supposed ultimately to converge to the secret, the retrieved coefficients should be close to $2\mathbb{Z}$. Hence if some of them are far from this lattice, we can consider them as problematic coefficients and likely to be prone to induce an error after rounding. Suppose that we discriminate these problematic coefficients in a finite set T and that each coefficient outside T is correctly retrieved by rounding. Then we can find the correct value of the coefficients in T by lattice reduction in dimension slightly larger than $|T|$ by the exploitation of *dimension reduction techniques* described in [21].

If this dimension is sufficiently small (less than 100 for typical computers), this approach allows to still perform a full key recovery in cases where the sole descent algorithm would have led to some errors.

2.5 Practicality of the attacks and discussion

We provide the attack scripts in [4] and summarize in Table 2 the number of samples required to perform a full key recovery with both of the attacks. The first column corresponds to the first attack

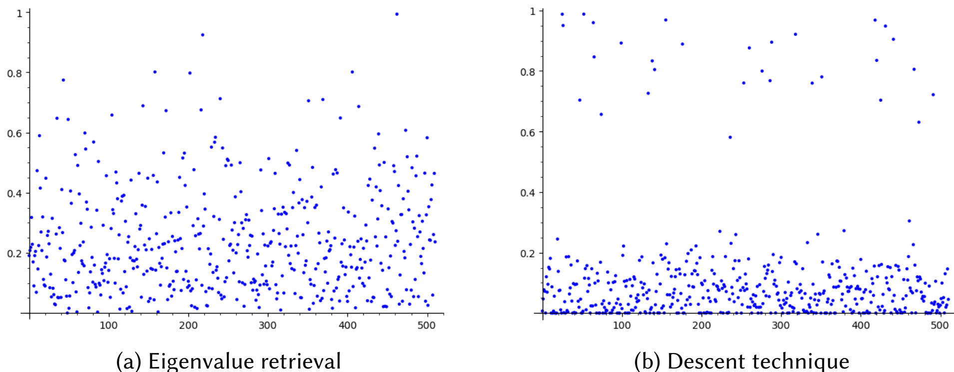
Table 2. Experimental number of samples required to perform a full key recovery. The average CPU time for a full key recovery is 40h on a Intel Xeon E5-2697v3 workstation.

		PCA_0+MiTM	Spectral+Descent
w/o compress	BLISS-I	180k	65k
	BLISS-II	250k	130k
	BLISS-III	209k	100k
	BLISS-VI	308k	120k
w/ compress	BLISS-I	4200k	700k
	BLISS-II	27500k	2000k
	BLISS-III	2100k	350k
	BLISS-VI	unfeasible	200k

described in Section 2.2 with the MiTM technique of [25] to correct the errors. The second column corresponds the Wirtinger flow technique coupled with the lattice reduction and the localization of Section 2.4. Since the descent attack is an improvement build on a spectral method, it is natural to see that this algorithm indeed requires far fewer samples to mount the attack than the first method presented in Section 2.2. It should also be noticed that this attack discards every samples for which $t > 0$, implying that a certain amount of the information provided by the samples is not used. For instance when attacking BLISS-II with compression, almost 30 millions of samples are necessary to retrieve the secret, but among those, only 18 millions of them are actually conserved to mount the attack. The number of required samples may seems high compared to the dimension of the problem, but it can be noticed that the size of the errors obtained by obtaining the estimation of the phases by maximum likelihood is of the same magnitude as the actual phase we are trying to retrieve. Hence, canceling the noise actually costs a significant amount of samples, as evoked above.

As far as the correction of errors is concerned, with the two techniques introduced in Section 2.4 (i.e. the MiTM and the localization), the two attacks have different behaviors. Indeed, the MiTM exhaustive search appeared to be more tailored to the first attack whereas the localization worked far better for the descent attack. A more detailed discussion on the causes of this phenomena is provided in Section 2.6 below. The results presented in Table 2 are obtained by making the maximum use of these correction techniques. Hence, the running time of a full key recovery is contributed almost exclusively by this final phase: practically the parameters given allows the descent to yield a lattice problem in dimension at most 110. On a Intel Xeon E5-2697v3 workstation this phase takes less than an hour to complete. Using the BKZ reduction with blocksize 25 takes then around 38h to complete the recovery.

A striking observation is that in both of the attacks the compression on $\mathbf{z}_2$ used in actual BLISS signatures, makes the recovery significantly harder: indeed, there is an order of magnitude between the number of samples needed to make a full key recovery. Indeed the bit dropping yields noisier estimates for the recovery problem. Finally, note that BLISS-II is the hardest variant to attack with this method. This is due to the fact that this parameter set provides the highest rate of compression.



(a) Eigenvalue retrieval (b) Descent technique

Fig. 9. Comparison of the repartition of the distance to the lattice $2\mathbb{Z}$.

2.6 Convergence behavior

In 9 we present the result of an experiment picturing the distance of each coefficient of the candidate secret from the lattice $2\mathbb{Z}$ before the final rounding, for both of the proposed attacks.

A striking observation is that the descent attack pushes way more the distances towards either 0 or 1 and as such makes it easy to localize the coefficients that are prone to be problematic. Indeed setting a threshold at 0.5 clearly discriminates the "good" coefficients form the potentially problematic ones. On the contrary, the situation is way more blurry in the other attack, where the distances are much more close to 0.5. As such being able to distinguish the "good" coefficients from the "bad" ones is much more difficult in order not to create false positives.

As a consequence, it is experimentally less costly to rely on MitM technique to resolve the errors in this latter case as setting a threshold too low would imply reducing lattices of dimension too large.

3 IMPLEMENTING BLISS IN CONSTANT TIME

In order to protect against timing attacks such as the one of Section 2 and most types of microarchitectural side-channel attacks (including [9, 22, 33]), it would be desirable to design an implementation of BLISS that runs in *constant time*.

As noted in the introduction, doing so seems to present fundamental difficulties related to the fact that the BLISS signing algorithm, in keeping with the Fiat–Shamir with aborts framework, includes a probabilistic *rejection sampling* step that makes the running time intrisically vary from one execution to the next. Moreover, the rejection probability computed at each step depends on the secret key and the generated signature, so it may seem that secret-dependent branching is unavoidable when implementing the scheme.

Fortunately, the problem is in fact crucial, because the distribution of the number of repetitions in the signing algorithm is actually independent of all secrets. As a result, it is possible to aim for an implementation that is *constant time with public outputs*, where the public outputs leak to the adversary the number of repetitions. Since that number can be perfectly simulated independently of the secret key, this is just as good as a truly constant time implementation.

In fact, although it is not really discussed in those terms, the same issue arises in existing "constant time" implementations of Fiat–Shamir with aborts signature schemes such as the NIST second round candidate Dilithium [18]. The main obstacle in implementing BLISS in constant time lies

elsewhere, in what forms the key difference between those two lattice-based schemes: BLISS's reliance on discrete Gaussian distributions, whereas Dilithium only uses uniform distributions, with the explicit goal of avoiding side-channel vulnerabilities in the implementation.

The use of Gaussian distributions leads to two main implementation challenges: the constant time implementation of Gaussian sampling, and that of the rejection sampling, corresponding to Step 2 and Step 8 of Figure 2. In addition, some care must be taken regarding the implementation of the ring-valued hash function from Step 4, as well as the sign flips in Steps 6 and 7. We describe our implementation choices below and provide further technical details at the end of this section.

We note that most of these implementation techniques would apply equally well to other Fiat–Shamir signatures schemes using Gaussian distributions, and in particular to the optimized variant BLISS–B [15]. Regarding BLISS–B, the only subtle point is the computation of $\mathbf{Sc}$, which now involves sign flips and can no longer be carried out using an NTT; it is still easy if $\mathbf{c}$ is considered non-sensitive (which is reasonable but requires additional assumptions), but becomes significantly more expensive otherwise. We also note that our approach supports arbitrary Gaussian standard deviations, which could in principle allow for more efficient parameter settings; to make comparisons more meaningful, we did not attempt to select new parameters, but this could be interesting further work.

3.1 Overview of our constant-time implementation

The main design goal of our implementation is to obtain a *fast, constant-time* implementation of BLISS (focusing on the BLISS–I parameter set, which offers the best trade-off between security and efficiency) while maintaining a high degree of portability. With the latter goal in mind, we choose to rely entirely on *integer arithmetic* (limited to additions, multiplications and shifts on 32-bit and 64-bit operands). Indeed, division instructions and floating point operations rarely offer constant-time execution guarantees,[1] and they can present serious security challenges related to weak determinism [17].

The ingredients needed to implement the signing algorithm are as follows: we need *Gaussian sampling* for Step 2 of Figure 2; *ring multiplication* for Steps 3, 6 and 7; *ring-valued hashing* for Step 4; and *rejection sampling* for Step 8. Other operations like constant-time sign flips, ring additions and signature compression are straightforward. We now give a description of our implemention choices for each of these steps. Note that in terms of efficiency, the critical elements are the Gaussian sampling and the ring multiplication, with the ring-valued hashing also taking up a significant amount of time. The other operations take negligible time in comparison.

3.1.1 Gaussian sampling. The Gaussian sampling step is key to obtaining a fast implementation of BLISS, as it represents half or more of the computation time of signature generation: for each signature, one needs to generate 1024 samples of the discrete Gaussian distribution $\mathscr{D}_\sigma$ (possibly several times over, in case a rejection occurs), and the standard deviation is relatively large ($\sigma = 205$ for BLISS–I). This step has also been specifically targeted by cache timing attacks such as [9].

Several approaches can be considered for implementing it in constant time, but they have wildly different running times. All approaches first generate samples from the non-negative Gaussian $\mathscr{D}_\sigma^+$, and then use a random sign flip (in constant time) to recover the entire distribution.

The most naive way would be to rely on cumulative distribution table (CDT) sampling: precompute a table of the cumulative distribution function of $\mathscr{D}_\sigma^+$ covering the inverval at the points

[1] Regarding floating point arithmetic, it is often variable time even in the presence of an FPU, and even for simpler operations like multiplications. For example, the `fmul` multiplication instruction can have variable latency on several x86 architectures, including the Intel Pentium III!

of which the distribution has a non-negligible probability[2] $\gtrsim 2^{-128}$; then, to produce a sample, generate a random value in $[0, 1]$ with 128 bits of precision, and return the index of the first entry in the table greater than that value. In variable time, this can be done relatively efficiently with a binary search, but this leaks the resulting sample through memory access patterns. As a result, a constant time implementation has essentially no choice but to read the entire table each time and carry out each and every comparison. Although a basic CDT implementation would store the cumulative probabilities with 128 bits of precision, it is in fact possible to only store lower precision approximations, as discussed in [34**?**] (see also [35] for an alternate approach using "conditional distribution functions"). Nevertheless, since the table should contain $\sigma\sqrt{2\lambda \log 2} \approx 2730$ entries for BLISS–I, we are looking at 22 kB's worth of memory access for every generated sample. The resulting implementation is obviously highly inefficient. Other table-based approaches like the Knuth-Yao algorithm similarly suffer from constant time constraints.

A more efficient approach, originally introduced by Pöppelmann et al. [34] and later improved and generalized by Micciancio and Walter [32], assumes that we can generate a base Gaussian distribution $\mathscr{D}^+_{\sigma_0}$ with not too small standard deviation σ_0, and allows to then combine samples from that base distribution to achieve larger standard deviations. For the parameters of BLISS–I, one can check that the optimal choice is to let $\sigma_0^2 = \frac{\sigma^2}{(9^2+7^2)(3^2+2^2)}$. One can then generate a sample x statistically close to $\mathscr{D}^+_\sigma$ from 4 samples $x_{0,0}, x_{0,1}, x_{1,0}, x_{1,1}$ from $\mathscr{D}^+_{\sigma_0}$, as $x = 9x_0 + 7x_1$, where $x_i = 3x_{i,0} + 2x_{i,1}$. Since $\sigma_0 \approx 4.99$ is much smaller than σ, using a CDT approach for the base sampler is more reasonable: the CDT table now stores 63 entries. Generating a sample requires reading through the table 4 times, for a total of 2 kB of memory access and 128 bits of randomness per sample. It turns out, however, that the performance of the resulting implementation in our setting is still somewhat underwhelming.

The authors of the qTesla[3] second round NIST submission [1] proposed an ingenious approach to improve constant-time CDT-based discrete Gaussian sampling. In practice, one needs to generate many samples from the discrete Gaussian distribution in each signature (one for each coefficient of the $\mathbf{y}_i$ polynomials). The idea is then to batch all of the searches through the CDT table corresponding to those samples. This can be done in constant time by applying an oblivious sorting algorithm (e.g. network sorting) to the concatenation of the CDT with the list of uniform random samples. This can be used in conjunction with the convolution technique of [32, 34] in order to reduce the total size of the table to be sorted (which is the sum of the CDT size and of the desired number of samples). Preliminary attempts to use this approach in the case of BLISS did not result in compelling performance numbers, but there is likely room for improvement in terms of the oblivious sorting algorithm involved as well as the way is algorithm is combined with various optimization tricks: detailed investigation of this question is left as interesting further work.

Finally, yet another strategy is to generate a discrete Gaussian of very small standard deviation, use it to construct a distribution that looks somewhat like $\mathscr{D}^+_\sigma$ but is not statistically close, and use rejection sampling to correct the discrepancy. This is actually the approach taken in the original BLISS paper [16]. Concretely, what that paper essentially does is sample some x from the distribution $\mathscr{D}^+_{\sigma_2}$ where $\sigma_2 = \sigma/k$, and some y uniform in $\{0, \dots, k-1\}$. Then, $z = kx + y$ looks "somewhat like" a sample from $\mathscr{D}^+_\sigma$, and one can check that rejecting z except with probability $\exp\left(-y(y + 2kx)/(2\sigma^2)\right)$ yields a value that actually follows $\mathscr{D}^+_\sigma$. As observed in the BLISS paper, this rejection sampling step is exactly of the same form as the one used for the overall signing algorithm. The constant time implementation of that step is described in Section 3.1.4 below, and we can simply reuse that work to obtain our Gaussian sampling. The only ingredient to add is

[2]Even taking Rényi divergence arguments into account, values taken with probability $\geq 2^{-117}$ should included.
[3]While qTesla uses uniform randomness during signature generation, it does use discrete Gaussians for key generation.

a base sampler for the distribution $\mathscr{D}^+_{\sigma_2}$, since the one in the original BLISS paper does not lend itself to a convenient constant time implementation. Fortunately, choosing $k = 256$, the standard deviation $\sigma_2 \approx 0.80$ is really small, and hence a CDT approach only requires 10 table entries. In practice, this yields a Gaussian sampling of very reasonable efficiency, whose cost is dominated by the cost of the rejection sampling step, and of the generation of the uniform randomness. This is the approach we choose for our implementation. Its security directly follows from that of the rejection sampling (see Section 4 for technical details).

3.1.2 Ring multiplications. As usual in ideal lattice-based schemes, ring multiplications such as the one in Step 3 of Figure 2 are carried out using the number-theoretic transform (NTT). Since the NTT does not use any secret-dependent conditional branches or memory accesses, constant-time implementation does not pose any particular difficulty. In our case, we directly adapt the NTT from the reference implementation of Dilithium, which uses the bit-reversed order for coefficients in the NTT domain, lazy modular reductions, and the Montgomery representation for values modulo q. Only a few simple changes are needed compared to Dilithium, in order to account for the different modulus $q = 12289$ and the higher degree $n = 512$ (instead of $q = 8380417$ and $n = 256$ respectively). At the cost of more frequent modular reductions, we could do the entire computation on 16-bit integers (which could yield to faster automatic vectorization), but for simplicity, we keep the 32-bit arithmetic from the Dilithium NTT.

The implementation choice for the ring multiplications in Steps 6 and 7 of Figure 2 is less obvious. Indeed, those steps involve the multiplication of the secret key elements, which are small, by the hash value $\mathbf{c}$, which has 23 coefficients equal to 1, and the others equal to 0. Moreover, we can show that, under a non-standard but reasonable LWE-like assumption, BLISS remains secure even when $\mathbf{u}$ and hence $\mathbf{c}$ are made public (including for rejected instances). It would therefore not jeopardize security to implement the multiplications $\mathbf{s}_1\mathbf{c}$ and $\mathbf{s}_2\mathbf{c}$ as repeated additions of shifted versions of $\mathbf{s}_1$ and $\mathbf{s}_2$, where the memory access patterns in the shifts reveal the coefficients of $\mathbf{c}$ (but nothing about the secret key vectors themselves). Interestingly, however, it turns out that, at least on our target platform, implementing the multiplications that way is not faster than using the NTT, probably because the NTT has a much better cache locality. As a result, all ring multiplications in our implementation simply use the NTT.

3.1.3 Ring-valued hashing. Step 4 of the signing algorithm in Figure 2 computes the "challenge" ring element $\mathbf{c} = H(\lfloor \mathbf{u} \rceil_d \bmod q, \mu)$ from the "commitment" $\mathbf{u}$ and the input message μ. That ring element should be a polynomial uniformly sampled among those with $\kappa = 23$ coefficients equal to 1, and all other coefficients equal to 0. To construct such a polynomial, we first pass the inputs of H to an extendable output function (XOF), in our case SHAKE128, and then use the resulting random stream to sample the list $(i_1, \dots, i_\kappa)$ of indices in $\mathbf{c}$ equal to 1.

Concretely speaking, we again follow Dilithium's approach, which proceeds as follows. We pick i_1 uniformly in $\{0, \dots, n-\kappa\}$. Then i_2 is chosen uniformly in $\{0, \dots, n-\kappa+1\}$, and if it happens to collide with i_1, it is set to $n-\kappa+1$ instead. Continuing, i_k is chosen uniformly in $\{0, \dots, n-\kappa-1+k\}$, and replaced by $n-\kappa-1+k$ if it coincides with one of the previous values. It is easy to check that $\{i_1, \dots, i_\kappa\}$ is then a uniformly distributed κ-element subset of $\{0, \dots, n-1\}$ as required.

However, Dilithium's implementation of this strategy is not in fact constant-time, as it works by updating an n-element array and modifying the elements at indices i_k and $n-\kappa-1+k$ for each k. As a result, the algorithm leaks the entirety of $\mathbf{c}$ through memory accesses. This is not a critical problem, since as we have mentioned, the values $\mathbf{u}$ and hence $\mathbf{c}$ are not really sensitive in BLISS (security is still achieved for the variant in which those values are revealed, albeit under a less standard hardness assumption: this is exactly analogous to how the security of the $\mathbf{r}$-GLP scheme from [3] is proved under the non-standard $\mathbf{r}$-DCK assumption).

Nevertheless, in order to avoid relying on additional assumptions compared to the original BLISS paper, we opt for a completely constant time implementation of the same approach instead. Our idea is to add i_k to the list of previously obtained indices using a constant-time insertion sort, and do a constant-time swap between i_k and $n - \kappa - 1 + k$ in case a collision occurs. In principle, that approach has quadratic complexity in κ, but since κ is so small, the overhead is negligible: we find that our constant-time approach is only a few thousand cycles slower than the variable time algorithm (about 1–2% of the entire running time of signature generation).

3.1.4 Rejection sampling. Finally, the last step we need to implement in constant time is the rejection sampling. In other words, at the end of the signature generation algorithm, we need to sample bits $b_{\exp}$ and $b_{1/\cosh}$ that take the value 1 with probability

$$p_{\exp} = \exp\left(-\frac{K - \|\mathbf{Sc}\|^2}{2\sigma^2}\right) \quad \text{and} \quad p_{1/\cosh} = 1/\cosh\left(\frac{\langle \mathbf{z}, \mathbf{Sc}\rangle}{\sigma^2}\right)$$

respectively (where K is a known constant).

To do so, the approach taken in the original BLISS paper relies on iterated Bernoulli trials with known constant probabilities for $b_{\exp}$, and recursively calls this exponential sampling algorithm to sample $b_{1/\cosh}$. Again, the variable time nature of these algorithms has led to multiple attacks.

As mentioned in [22], it is relatively easy to modify the function SampleBernExp from Figure 4 to run in constant time: simply carry out every iteration every time, and accumulate the results of the Bernoulli trials using constant time logic expressions. However, the performance penalty of doing so is significant, due to the lack of early aborts. This is not a serious problem for the rejection sampling step itself, since it is only carried out a handful of times per signature. However, since this exponential rejection sampling function is also called as part of Gaussian sampling (as we recall from Section 3.1.1 above), any slow down will strongly affect the running time of the entire signature generation. Moreover, while the $b_{\exp}$ part can be made constant time, doing so is much harder for $b_{1/\cosh}$, as we have discussed in Section 2.

An alternate approach is to simply evaluate the values $p_{\exp}$ and $p_{1/\cosh}$ with sufficient precision, and compare them to uniform random values in $[0, 1]$. The challenge is to do so in constant time, using only integer arithmetic. In particular, we cannot rely on floating point implementations of transcendental functions like exp and cosh.

The approach we take is to replace the exp function by a sufficiently close polynomial approximation, and similarly for cosh. Then, $p_{\exp}$ can be evaluated in fixed point to sufficient precision using an application of Horner's algorithm, entirely with integer arithmetic; and $1/p_{1/\cosh}$ can be evaluated using the same code by expressing cosh in terms of exp. There are several steps involved in carrying out that strategy:

(1) determine the precision we need to ensure security. To do so, we use a methodology introduced by Prest [35] based on the Rényi divergence. It shows that 45 bits of relative precision suffice for security, provided that the number of generated signatures is at most 2^{64} (as specified in the NIST competition).
(2) compute a polynomial approximation on the required interval of the function $f\colon x \mapsto \exp\left(x/(2\sigma^2)\right)$, achieving the relative precision we need. To do so, we introduce a novel technique based on lattice reduction for the Sobolev Euclidean norm on polynomials. This technique lets us precisely control the shape of the polynomial we get, in order to ensure that Horner's algorithm can be applied without any overflow using 64-bit integer arithmetic. Compared to earlier techniques such as the L^∞ approximations of Brisebarre and Chevillard [8], it also has the advantage of eliminating heuristics (since a bound on the Sobolev norm directly yields a bound on the functional $\|\cdot\|_\infty$ norm), and of avoiding the computation

Table 3. Performance results and comparison (kcycles).

	LQ	Median	UQ	Const. time?
Dilithium (`ref`)	286	515	1526	✓
Dilithium (`avx2`)	142	332	428	✓
qTesla-I (`ref`)	243	418	781	✗
Original BLISS	188	194	313	✗
Our implementation	218	220	223	✓

of minimax polynomials (since the closest polynomial in the Sobolev norm can be obtained using a simple Euclidean projection).

(3) extend the range of that polynomial approximation in order to support the larger interval required for the cosh computation (as well as for the rejection step in Gaussian sampling). This is done by computing a constant c such that $f(c)$ is very close to 2, so that $f(k \cdot c + x) = f(c)^k \cdot f(x)$ can be easily obtained from $f(x)$ using small multiplications and shifts.

(4) deduce an algorithm for the cosh part of the rejection sampling. The nontrivial point here is that we end up evaluating a good approximation of $p' = 1/p_{1/\cosh}$. Testing if $u < p_{1/\cosh}$, for some $u \in [0, 1]$, reduces to testing if $u \cdot p' < 1$. The multiplication involves numbers with over 45 bits of precision, however, so the result does not fit within 64 bits, and thus requires some degree of bit fiddling. Intermediate conditional branches also need to be written in constant time.

Full technical details regarding these various steps are provided in Section 4 below.

The idea of using polynomial approximations to evaluate $p_{\exp}$ already appears in earlier work: as part of the FACCT Gaussian sampler described in [42]. In particular, our own Gaussian sampler can be seen as a variant of FACCT. There are multiple differences between our works, however: in particular, FACCT relies on floating point arithmetic, which we specifically seek to avoid,[4] and uses off-the-shelf software to obtain a double precision floating point polynomial approximation of the function f. Moreover, since FACCT focuses on Gaussian sampling, that paper does not directly address the cosh issue.

3.2 Security and performance

Using the techniques described above, we wrote a constant-time implementation of BLISS in portable C (specifically for the BLISS–I parameters), that can be found in [4]. We now provide some data regarding its performance, and provide a short formal treatment of its security.

We point out that our code only implements *signature generation* in constant time. Obviously, signature verification does not manipulate any secret, and hence does not need to be made constant time; however, one may wish to ensure that *key generation* is constant time as well. We have not attempted to do so, since key generation is carried out much less often and usually in much more controlled conditions than actual signing. However, it is not difficult to modify our implementation to make key generation constant time as well. The building blocks involved are briefly discussed at the end of this section.

[4]We think the argument from [42] to the effect that floating point multiplications are constant time is overly optimistic. As mentioned earlier, this is not true on some older x86 platforms, to say nothing of more exotic, more lightweight or FPU-less architectures. Besides, as highlighted in [17], floating points arithmetic may lead to a vulnerbability called weak determinism which can sometimes lead to complete breaks.

3.2.1 Performance measurement and comparison. Our implementation is written for the SUPERCOP toolkit for measuring cryptographic software performance [5]. Accordingly, it follows the SUPERCOP API, and uses the corresponding utility functions for operations like randomness generation (for which SUPERCOP automatically selects the most efficient machine-specific candidate, in our case ChaCha20). We therefore use SUPERCOP's latest version as of this writing[5] to evaluate the performance its performance on our testbench platform, and compare its speed with the closest competitors Dilithium [18] and qTesla [1] on the same machine.

We also provide a comparison to Ducas and Lepoint's original, variable-time implementation of BLISS on the same platform [19]. Unfortunately, that implementation does not follow the SUPERCOP API, so the comparison is not entirely apples to apples: on the one hand hashing and randomness generation are carried out with OpenSSL's implementation of SHA2 (instead of SHAKE128 and ChaCha20 respectively); on the other hand, all the serialization routines required by SUPERCOP are omitted. On balance, this should not strongly bias the comparison in either direction.

Our testbench platform is an Intel Xeon Platinum 8160-based server (Skylake-SP architecture) with Ubuntu 18.04 and gcc 7.3.0 with the default SUPERCOP compiler options (`-march=native -mtune=native -O3 -fomit-frame-pointer -fwrapv`), with hyperthreading disabled and scaling governor set to `performance`. The choice of machine may seem overkill, but it was the newest CPU we had access to, and hence made it possible to compare our portable C implementation with the hand-vectorized AVX2 implementation of Dilithium available in SUPERCOP.

Performance results are presented in Table 3: they indicate the lower quartile, median and upper quartile cycle counts measured by SUPERCOP (or in the case of BLISS, measured by the RDTSC instruction) for the signature of a 59-byte message, which is the standard performance figure presented on the eBATS website. The Dilithium performance numbers are for the fastest parameter set available in SUPERCOP, namely the `dilithium2` implementation, corresponding to "medium" security parameters in [18] (no implementation is provided for the "weak" parameters); we give timings both for the portable C (`ref`) and AVX2 platform specific (`avx2`) implementations. For qTesla, we also use the fastest available implementation (`qtesla1`, only in portable C[6]), which corresponds to essentially the same lattice security level as BLISS–I.

As we can see in the table, we achieve a performance level similar to the original, variable-time BLISS implementation, while preventing the serious timing attack vulnerabilities exposed in multiple papers so far.

In addition, our implementation is multiple times faster than than qTesla-I and the portable C implementation of Dilithium, and even outperforms the AVX2 implementation of Dilithium by a significant margin, while providing stronger constant-time guarantees (since the Dilithium ring-valued hash function presents a mild timing leakage that causes the security in the constant-time model to rely on non-standard assumptions). Admittedly, the Dilithium parameters were derived using a more conservative methodology for assessing the cost of lattice attacks, and hence probably achieve a significantly higher level of security against them. Nevertheless, according to Wunderer's recent reevaluation [41] of what is likely the strongest attack against BLISS (namely the Howgrave-Graham hybrid attack), it is reasonable to think that BLISS–I does reach its stated security level of around 128 bits.

Note that the "Const. time?" column in Table 3 indicates whether the implementation satisfies constant-time *security guarantees* (i.e. the absence of secret dependent branches and memory

[5]https://bench.cr.yp.to/supercop/supercop-20190110.tar.xz

[6]The "heuristic" qTesla-I parameters were recently removed from the qTesla submission documents and the remaining "provable" parameters are significantly less efficient. Since our goal is to compare to *fast* comparable schemes, however, qTesla-I appears to be the most suitable parameter choice.

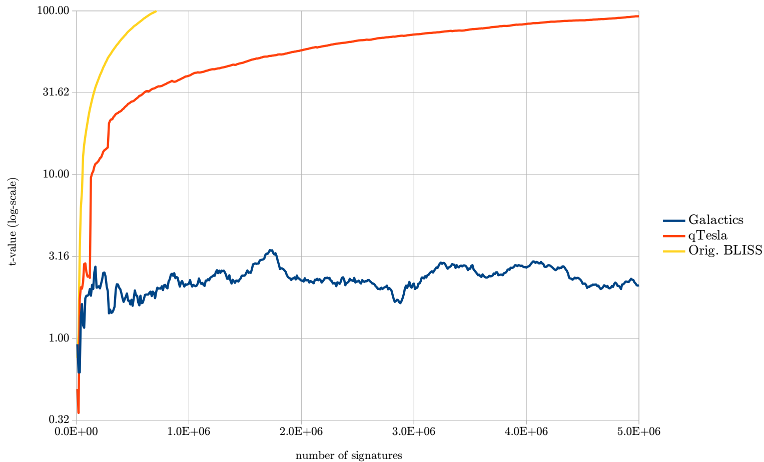


Fig. 10. Leakage assessment with dudect [37].

accesses). This is of course achievable without having strictly constant *running time*, since secret-independent branches and loops are permitted and heavily relied on in Fiat–Shamir with aborts-type schemes.

3.2.2 Experimental validation. In order to further validate the constant-time nature of our implementation, we carried out experiments with the `dudect` tool of Reparaz et al. [37]. The basic idea of the experiment is to generate two key pairs for the signature scheme, sign many messages randomly with either of the two signing keys, and check if a statistical difference can be observed between timings among the two keys.

Clearly, such an experimental approach cannot be used to conclusively establish that an algorithm is constant-time: for example, it will not detect if a very small fraction of weak keys with different timing profiles exist; in the case of Fiat–Shamir signatures, it is also incomplete in the sense that the signing key is not the only sensitive part of the algorithm—the *randomness* is also sensitive. And in principle, experimental validation should also not be necessary if the security analysis provides sufficient evidence that the algorithm is constant-time.

We did find experiments with `dudect` to be quite useful in practice, as it *did* find timing leakage in an earlier version of our implementation, mostly due to the fact that `gcc` would compile apparently branch-free code containing comparison instructions into actual conditional branches in the assembly. After fixing those issues (by replacing comparisons with bit fiddling), we obtained our final implementation, for which no leakage is detected in `dudect`: we ran it on 30 different sets of two random key pairs, and in all cases, the t values in Welch's t-test remained below 3 or 4 even with tens of millions of signatures.

In contrast, `dudect` easily detects the timing leakage in the original BLISS implementation: t values quickly exceed 10, and shoot up above 100 after a few hundred thousand signatures. Similarly, significant timing leakage is detected in the qTesla-I implementation available in SUPERCOP, even though it is advertised [1] as constant-time: in our experiments, t values exceed 10 after around 100,000 signatures, and eventually increase to above 100; whether this leakage can be exploited to attack the scheme is unclear, but it does rule out the implementation being constant-time.

Welch t-test values measured by `dudect` for our implementation as well as the original BLISS and qTesla are shown in Figure 10.

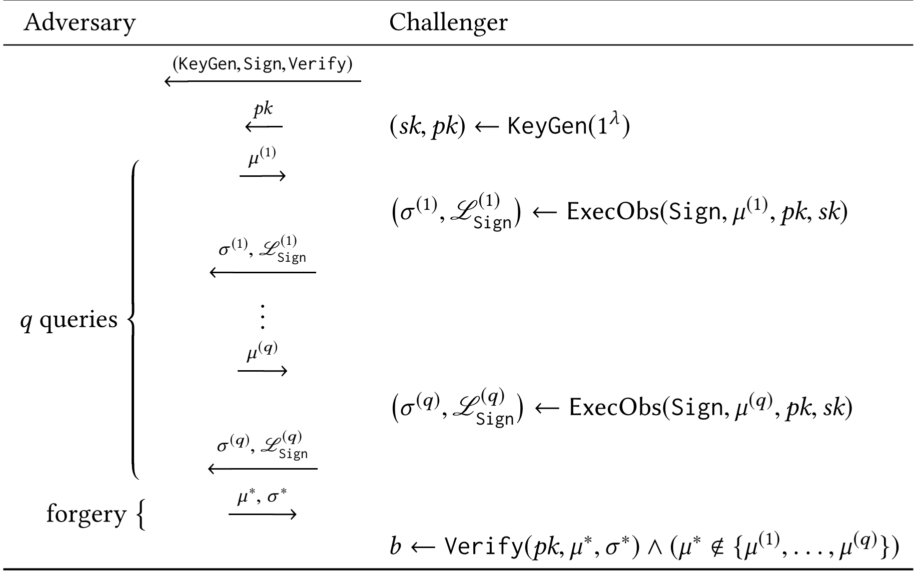


Fig. 11. The CT-EUF-CMA security game.

3.2.3 Security argument. Let us now try and formalize the constant time security guarantees that we claim are provided by our implementation. To do so, we introduce the notion of *existential unforgeablity under chosen message attack in the constant-time model* (CT-EUF-CMA), which combines standard EUF-CMA security property with the security in the constant-time model. It can be seen as a constant-time model counterpart of the "EUF-CMA in the d-probing model" notion introduced in [3] in the context of masking security.

Definition 3.1. An implementation (KeyGen, Sign, Verify) of a signature scheme is EUF-CMA-secure in the constant-time model, or CT-EUF-CMA secure for short, if any PPT adversary has a negligible winning probability in the experiment from Figure 11. In that security experiment, ExecObs is a universal RAM machine that takes as input an algorithm and its arguments, executes the program, and outputs the result of the computation together with the timing leakage $\mathscr{L}$, consisting of the sequence of *visited program points* and *memory accesses*.

In the context of that definition, the constant-time properties discussed in Section 3 above can be summed up as follows.

Proposition 3.2. *For any execution of our implementation of the signature generation* $(\sigma, \mathscr{L}_{\mathtt{Sign}}) \leftarrow$ *ExecObs*$(\mathtt{Sign}, \mu, pk, sk)$*, the leakage* $\mathscr{L}_{\mathtt{Sign}}$ *can be perfectly publicly simulated from the number of executions the main loop (Steps 2–8 of Figure 2).*

Proof. Indeed, we have made sure that each step of algorithm, except for the execution or not of the rejection sampling, is devoid of secret-dependent branches or memory accesses. As a result, the sequence of visited program points and memory accesses from each step is perfectly publicly simulatable, and the overall leakage $\mathscr{L}_{Sign}$ is obtained from repeating those simulations a number of times equal to the number of executions the main loop. □

From that result, together with the security of the rejection sampling, we can deduce that our implementation achieves CT-EUF-CMA security.

Theorem 3.3. *The CT-EUF-CMA security of our implementation in the random oracle model tightly reduces to the standard EUF-CMA security of BLISS.*

Proof. There are three hybrids to this security argument.

First, by Proposition 3.2, we can replace the CT-EUF-CMA security game by a game in which the signing oracle simply returns pairs (σ, ℓ) where the value ℓ is the number of execution the main signing loop when generating the signature σ.

Then, in a second hybrid, we replace the approximate discrete Gaussian distributions for $\mathbf{y}_1, \mathbf{y}_2$ and the approximate values of the rejection probabilities computed by our implementation by the exact values. The Rényi divergence estimates of Section 4.1 below prove that the advantage of an adversary in distinguishing this hybrid from the previous one is negligible.

Finally, since the ring-valued hash function H in Figure 2 (or at least the XOF it uses internally) is modeled as a random oracle, one can easily show that the value ℓ exactly follows a geometric distribution of parameter $1/M$, where M is the constant appearing in Step 8 of Figure 2. This is already noted in the original BLISS paper [16, Lemma 2.1], and follows from a general result of Lyubashevsky [31, Lemma 4.7]. Since the constant M is public, there is zero difference in advantage with another game in which the value ℓ is removed from oracle replies. But that game is exactly the standard EUF-CMA security experiment for BLISS. □

3.2.4 Making key generation constant-time. As we have noted, our implementation of the key generation algorithm of Figure 1 is not actually constant time. However, there are no major obstacle in making it constant time if desired. In this paragraph, we briefly describe how this can be done.

There are mainly three steps in key generation that are not trivially constant time: the sampling of the sparse polynomials $\mathbf{f}$ and $\mathbf{g}$ in Step 1; the computation of the value $N_\kappa(\mathbf{S})$ in Step 4; and the ring division in Step 5. Note on the other hand that the rejection sampling in and of itself is not problematic, because any secret generated at that point is discarded if rejection happens; therefore, the number of rejections leaks no secret information per se.

To implement the sampling of $\mathbf{f}$ and $\mathbf{g}$ in constant time, one possible approach is to use the same algorithm as the one we described in Section 3.1.3 above for ring-valued hashing. Since the number of coefficients is larger in this case, the approach is not highly efficient, but it is not a serious issue for key generation.

Regarding the ring division, we implement it by computing the NTT of f, and inverting the NTT coefficients modulo q in Montgomery representation. The only change that needs to be done to make it constant time is to use a constant time version of the modular inversion, as described e.g. by Bos [7].

Finally, the more subtle problem is to obtain a constant time implementation of the computation of:

$$N_\kappa(\mathbf{S}) = \max_{\substack{I \subset \{1,\ldots,n\} \\ \#I=\kappa}} \left(\sum_{i\in I} \max_{\substack{J \subset \{1,\ldots,n\} \\ \#J=\kappa}} \left(\sum_{j\in J} T_{i,j} \right) \right) \text{ where } \mathbf{T} = \mathbf{S}^T \cdot \mathbf{S},$$

As explained in [16, Sec. 4.1], one possible approach to carry out that computation is to construct $\mathbf{T} \in \mathbb{Z}^{n\times n} = \mathbf{S}^T \cdot \mathbf{S}$, sort its columns, sum the κ largest values of each line, sort the resulting vector and sum its κ largest components. All these operations are naturally constant-time, except for the sorting steps. To make sure that these sorts implemented in constant time as well, we suggest to rely on a data-oblivious sorting algorithm, such as a sorting network. The resulting constant-time implementation is presented in Figure 12.

```
1: function N_κ(S)
2:     T ← S^t · S
3:     T′ ← (NetworkSort(T_1)|...|NetworkSort(T_n))          ▷ where T_i are the columns of T
4:     (v_k)_{0≤k≤n} ← (∑_{j=0}^{j=κ} T′[k, j])_{0≤k≤n}
5:     v' ← NetworkSort(v)
6:     return N_κ(S) = ∑_{j=0}^{j=κ} v'[j]
7: end function
```

Fig. 12. Computation of N_κ.

1: Compute

$$x_1 \in I_1 := \left[-\frac{\sigma^2}{\alpha^2}, 0\right] \text{ and } x_2 \in I_2 := \left[-\frac{2B_2\sigma}{\alpha}, \frac{2B_2\sigma}{\alpha}\right]$$

such that

$$x_1 = \|\mathbf{Sc}\|^2 - \frac{\sigma^2}{\alpha^2} \text{ and } x_2 = 2\langle \mathbf{z}, \mathbf{Sc}\rangle$$

2: Generate a pair (u_1, u_2) of fixed-precision numbers uniformly at random in $[0, 1]^2$

3: Let $a = 1$ if $u_1 \leq \exp(\frac{x_1}{2\sigma^2})$, and $a = 0$ otherwise

4: Let $b = 1$ if $\cosh(\frac{x_2}{2\sigma^2}) \cdot u_2 \leq 1$, and $b = 0$ otherwise

5: Return $a \wedge b$

Fig. 13. The BLISS rejection sampling.

4 REJECTION SAMPLING WITH POLYNOMIAL APPROXIMATIONS

In the BLISS signing algorithm of Figure 2, candidate signatures $(\mathbf{z}, \mathbf{c})$ are rejected with probability $1 - 1/\big(M \exp(-\frac{\|\mathbf{Sc}\|^2}{2\sigma^2}) \cdot \cosh(\frac{\langle \mathbf{z}, \mathbf{Sc}\rangle}{\sigma^2})\big)$. As this probability depends on the secret S, we aim at giving a constant time implementation of this rejection step. This construction relies on a polynomial approximation to compute the transcendental terms $\exp(-\frac{\|\mathbf{Sc}\|^2}{2\sigma^2})$ and $\cosh(\frac{\langle \mathbf{z}, \mathbf{Sc}\rangle}{\sigma^2})$ in constant time, as explained in Section 3 above.

More precisely, in view of the rejection sampling algorithm described in Figure 13, our goal is to first determine the *number of bits of precision* on the various values involved we need to ensure security, and to then construct polynomial approximations of exp and cosh that make it possible to evaluate the transcendental expression in Steps 3 and 4 to that level of precision, in *constant time*, using only *integer arithmetic*. As discussed previously, this construction is carried out using a novel approach based on a Sobolev norm, which is Euclidean and hence allows us to use lattice reduction techniques to obtain approximations of our chosen shape easily.

In the following, we first recall the recent results based on *Rényi divergence* that we use to evaluate the quality of our approximations. Afterwards, we aim at deriving a polynomial that approximates the exponential, and then the hyperbolic cosine.

4.1 Rényi Divergence

In [35], Prest introduces an inequality that evaluates the security gap between two cryptographic schemes that query an ideal distribution $\mathcal{D}$ and an approximate distribution $\mathcal{D}'$ using Rényi divergence.

Definition 4.1 (Rényi Divergence). Let $\mathcal{P}, \mathcal{Q}$ be two distributions such that $Supp(\mathcal{P}) \subseteq Supp(\mathcal{Q})$. For $a \in (1, +\infty)$, we define the Rényi divergence of order a by

$$R_a(\mathcal{P}, \mathcal{Q}) = \left(\sum_{x \in Supp(\mathcal{P})} \frac{\mathcal{P}(x)^a}{\mathcal{Q}(x)^{a-1}} \right)^{\frac{1}{a-1}}.$$

In addition, we define the Rényi divergence of order $+\infty$ by

$$R_\infty(\mathcal{P}, \mathcal{Q}) = \max_{x \in Supp(\mathcal{P})} \frac{\mathcal{P}(x)}{\mathcal{Q}(x)}.$$

According to [35], by taking $a = 2 \cdot \lambda$ where λ is the security parameter of the cryptosystem using $\mathcal{D}$; the following inequation ensures that the use of the approximate distribution $\mathcal{D}'$ provides at least $\lambda - 1$ bits of security.

$$R_{2\cdot\lambda}(\mathcal{D}||\mathcal{D}') \leq 1 + \frac{1}{4q_{\mathcal{D}}}. \tag{2}$$

The integer $q_{\mathcal{D}}$ denotes the maximum number of queries to the distributions.

Number of queries. NIST suggested $q_s = 2^{64}$ maximum signature queries for post-quantum standardization. In the BLISS signing algorithm, the Bernoulli distribution with exponential parameter is called once per attempt at generating a Gaussian sample, which is repeated a small number of times (≤ 2 on average) due to rejection in Gaussian sampling, $2n$ times to generate all the coefficients of $\mathbf{y}_1, \mathbf{y}_2$, and M times overall where M is the repetition rate of the signature scheme. Therefore, the expected number of calls $q_{\mathcal{D}}$ to the Bernoulli distribution as part of Gaussian sampling when generation q_s signatures is bounded as $q_{\mathcal{D}} \leq 2M \cdot 2n \cdot q_s \leq 2^{78}$ for BLISS–I. Note on the other hand that the final rejection sampling is only called $M < 2$ times per signatures on average, so the polynomial approximations used in the rejection sampling step can assume $q_{\mathcal{D}} \approx q_s = 2^{64}$.

Lemma 4.2 (Condition of the relative error ([35])). *Assume that $Supp(\mathcal{D}') = Supp(\mathcal{D})$ and that the cryptosystem using $\mathcal{D}$ provides $\lambda + 1 \leq 256$ bits of security. For $q_{\mathcal{D}} = 2^{78}$ (resp. $q_{\mathcal{D}} = 2^{64}$), the replacement of $\mathcal{D}$ by a distribution $\mathcal{D}'$ satisfying*

$$\left| \frac{\mathcal{D} - \mathcal{D}'}{\mathcal{D}} \right| \leq 2^{-45} \quad (\textit{resp.} \leq 2^{-37}) \tag{3}$$

ensures at least λ bits of security.

The proof that directly follows [35] is in Appendix B.1. We denote by K the exponent in Equation 3. This parameter represents the quality of the approximation using the relative precision. Let us introduce the notion of polynomial approximation of a distribution. This is a particular case where $\mathcal{D}'$ is a polynomial.

Definition 4.3. We denote by P_f^I a polynomial that satisfies

$$\forall x \in I, \quad \left| \frac{P_f^I(x) - f(x)}{f(x)} \right| = \left| \frac{P_f^I(x)}{f(x)} - 1 \right| < 2^{-K}. \tag{4}$$

Such a polynomial is referred to as an approximation that coincides with f up to K bits of relative precision on I.

4.2 Polynomial approximation of the exponential

We aim to exhibit a polynomial that approximates function $f = \exp\left(\frac{\cdot}{2\sigma^2}\right)$ on I_1 (defined in Figure 13) that we denote $P_{\exp}^{I_1}$. The latter must minimize two parameters, namely

- η, the number of bits of its coefficients,
- γ, its degree,

in order to achieve Equation (4) as tightly as possible. The procedure is as follows:

(1) In a first attempt, we exhibit a candidate polynomial for exp whose coefficients are in $\mathbb{R}$. This step gives us the minimum degree γ that is needed.
(2) In a second attempt, the coefficient of the candidate polynomial are rounded to fulfill the requirement on the number of bits η.

(1). We start by looking for a polynomial $P_{\mathbb{R}}$ in $\mathbb{R}[x]$ that approximates exp. We define the infinite norm as $\|f\|_\infty = \sup_I |f(x)|$ where I is the interval I_1. Such a bound is convenient for Equation (4) as it manipulates function sup. However, the main drawback of the infinite norm is that it is not Euclidean. One possible efficient method to polynomially approximate with the infinite norm is introduced in [8]. In the following, we present a different method which trades efficiency with accuracy: the approximation consists in the interpolation of a continuous interval instead of a discrete set of samples. This procedure is more adapted to our setting since we want an approximation for all x in an interval I_1. To approximate on I_1, we can get use of Sobolev H^2 inner product. This Euclidean metric was introduced in [39] and allows an inequality with the infinite norm.

Definition 4.4 (Sobolev H^2 inner product). For u and v two differentiable functions defined on an interval I, Sobolev H^2 inner product is defined by

$$\langle u, v\rangle = \frac{1}{|I|}\int_I uv + |I|\int_I u'v'.$$

The corresponding norm $|.|_S$ is

$$|u|_S^2 = \frac{1}{|I|}\int_I u^2 + |I|\int_I u'^2.$$

And we have the following result, whose proof is provided as supplementary material in Section B.2.

Lemma 4.5. *The Sobolev norm $|.|_S$ satisfies*

$$\|u\|_\infty \leq \sqrt{2}\cdot |u|_S\,.$$

Based on this norm, we compute a polynomial $P_{\mathbb{R}}$ minimizing

$$\left| P_{\mathbb{R}}(\cdot)/\exp\left(\frac{\cdot}{2\sigma^2}\right) - 1\right|_S.$$

For several possible degrees d, we then compute the orthogonal projection of the function $x \mapsto 1$ on the space

$$E_{1,d} = \left\{x \mapsto P(x)\cdot\exp\left(\frac{-x}{2\sigma^2}\right) \mid P \in \mathbb{R}_{<d}[x]\right\}$$

with respect to Sobolev H^2 inner product. Let π denotes the projection of $x \mapsto 1$ on $E_{1,d}$. With Lemma 4.5, $\|\pi - 1\|_\infty \leq \sqrt{2}\cdot|\pi - 1|_S$. With the application of the log function, a slightly underestimated quality of the approximation can be obtained as

$$\kappa(d) = -\log_2\left(\sqrt{2}\cdot\left|\pi - 1\right|_S\right).$$

Therefore, to achieve a precision K, it is sufficient to select the degree γ as being the minimum degree d such that

$$\kappa(\gamma) > K \text{ i.e. } \|\Pi - 1\|_\infty < 2^{-K} \text{ and set } P_{\mathbb{R}} = \Pi \cdot \exp\left(\frac{\cdot}{2\sigma^2}\right).$$

(2). In order to obtain integer coefficients, we then minimize the precision loss on the approximation and operate a rounding of $P_{\mathbb{R}}$ using lattice reduction. Concretely, to get an approximation of $\exp\left(\frac{\cdot}{2\sigma^2}\right)$ with a polynomial in $\mathbb{Z}[x]$, the float coefficients must be rounded into integers of size η (introduced earlier in this Section 4.2). In a nutshell, the idea is to round $P_{\mathbb{R}}$ with its closest element in a Euclidean lattice that represents the elements in $\mathbb{Z}_{2\eta}[x] \cdot \exp\left(\frac{-x}{2\sigma^2}\right)$. In this objective, let us create an Euclidean lattice with the following basis

$$\mathcal{B}_1 = \left(2^{-\eta} \cdot x^i \cdot \exp\left(\frac{-x}{2\sigma^2}\right)\right)_{i \in [\gamma]}.$$

Our notion of closeness still refers to the Sobolev norm, which is an unusual norm for Euclidean lattices. The lattice reduction must be adapted to use Sobolev norm (using Gram matrix corresponding to Sobolev inner product). Then, this lattice can be LLL-reduced[7] with respect to the Sobolev H^2 norm. And a Babai rounding of the polynomial Π with respect to the same Sobolev H^2 norm gives a rounded element denoted $\Pi_{\mathbb{Z}}$. The quality of the rounding can be evaluated as

$$\kappa^{\text{round}}(\gamma, \eta) = -\log_2\left(\sqrt{2} \cdot \left|\Pi_{\mathbb{Z}} - \Pi\right|_S\right).$$

Finally, η must be chosen and the degree γ can be modified s.t.

$$2^{-\kappa(\gamma)} + 2^{-\kappa^{\text{round}}(\gamma,\eta)} < 2^{-K}.$$

Hence, the following polynomial appears as an approximation

$$P_{\exp}^{I_1} = \Pi_{\mathbb{Z}} \cdot \exp\left(\frac{\cdot}{2\sigma^2}\right) \in \mathbb{Z}_{2^\eta,\gamma}[x].$$

whose quality can be checked from Equation (4):

$$\begin{aligned}\forall x \in I_1, \quad \frac{\left|P_{\exp}^{I_1}(x) - \exp\left(\frac{x}{2\sigma^2}\right)\right|}{\left|\exp\left(\frac{x}{2\sigma^2}\right)\right|} &\leq \left\|\frac{P_{\exp}^{I_1} - \exp\left(\frac{\cdot}{2\sigma^2}\right)}{\exp\left(\frac{\cdot}{2\sigma^2}\right)}\right\|_\infty = \|\Pi_{\mathbb{Z}} - 1\|_\infty \\ &\leq \|\Pi_{\mathbb{Z}} - \Pi\|_\infty + \|\Pi - 1\|_\infty \\ &\leq \sqrt{2} \cdot |\Pi_{\mathbb{Z}} - \Pi|_S + \sqrt{2} \cdot |\Pi - 1|_S \\ &= 2^{-\kappa(\gamma)} + 2^{-\kappa^{\text{round}}(\gamma,\eta)} \\ &\leq 2^{-K}.\end{aligned}$$

4.3 Polynomial approximation of the hyperbolic cosine

The above method to approximate $\exp\left(\frac{\cdot}{2\sigma^2}\right)$ on I_1 can be applied to approximate $\cosh\left(\frac{\cdot}{2\sigma^2}\right)$ on I_2. However, the interval I_2 is larger, namely $I_2 = \left[-\frac{2B_2\sigma}{\alpha}, \frac{2B_2\sigma}{\alpha}\right] \approx [-5534960, 5534960]$ for BLISS-I. Due to the parity of the hyperbolic cosine, the study on I_2 can be reduced to $I_2 = \left[0, \frac{2B_2\sigma}{\alpha}\right] \approx [0, 5534960]$ for BLISS-I. A direct application gives around 48 coefficients for the integer polynomial (see Appendix B.4 for the polynomial $P_{\cosh}^{I_2}$ obtained with a direct approximation). This approximation is used for the masking countermeasure (we refer to Section 5 in the supplementary material for

[7]Using BKZ for more precise reduction was not relevant for the sizes manipulated. Besides, unlike for LLL, there is no function for BKZ that allows to give the Gram matrix as input.

details). However, in constant time, shifting the interval with multiplications is not costly, so we present an optimization in the sequel. For $x \in I_2$, let $c = 2\sigma^2 \ln(2)$, we define t as the remainder of the following Euclidean division

$$t(x) = x - \left\lceil \frac{x}{c} \right\rceil c.$$

By definition, $t(x)$ belongs in $I_3 := [-c, 0]$. Thus, we apply all the following shifts.

$$\exp\left(\frac{x}{2\sigma^2}\right) = \exp\left(\frac{\lfloor \frac{x}{c} \rfloor c}{2\sigma^2}\right) \cdot \exp\left(\frac{t(x)}{2\sigma^2}\right) = 2^{\lfloor \frac{x}{c} \rfloor} \exp\left(\frac{t(x)}{2\sigma^2}\right).$$

We thus define $P_{\cosh}^{I_2}$ as

$$P_{\cosh}^{I_2}(x) = \frac{2^{\lfloor \frac{x}{c} \rfloor} \cdot P_{\exp}^{I_3}(t(x)) + 2^{-\lfloor \frac{x}{c} \rfloor} \cdot P_{\exp}^{I_3}(-t(x))}{2}$$

where $P_{\exp}^{I_3}$ is the approximation of the $\exp\left(\frac{\cdot}{2\sigma^2}\right)$ on I_3 as obtained.

Remark 1. *Since the difference between I_1 and I_3 is small, we only compute $P_{\exp}^{I_3}$ for both. Indeed, since $I_1 \subset I_3$, $P_{\exp}^{I_3}$ satisfies both approximations.*

Lemma 4.6. *If $P_{\exp}^{I_3}$ is a precision K approximation then, $P_{\cosh}^{I_2}$ is a precision (K-1) approximation.*

Computations are provided in Section B.3. Note that the factor $2^{\lfloor \frac{x}{c} \rfloor}$ can be computed exactly in constant time in a fast way because $\lfloor \frac{x}{c} \rfloor \leq 86$. Namely, it consists in at most 86 shifts.

4.4 Bit precision of the inputs and intermediate values

Lemma 4.2 imposes $K \geq 45$ for the initial Gaussian sampling, and ≥ 37 for the final rejection sampling in BLISS. In order to account for the slight loss of precision as part of the fixed point evaluation of the polynomials using Horner's rule, we need to take slightly more precise values: we verify that $K = 48$ for Gaussian sampling (resp. $K = 40$ for rejection sampling) suffice with more than one bit of margin (one could use general error bounds for Horner such as [24, Eq. (5.3)], but since our evaluations occur on small integer intervals, it is easy to check their precision by exhausting the intervals).

For the constant time implementation, we also face a precision issue for applying Lemma 4.6. For an implementation with 64 bits integers, the value c can be stored with at most $64 - \log(x_2) = 16$ bits of precision. Thus, $\exp\left(\frac{c}{2\sigma^2}\right)$ is not exactly equal to 2. To preserve Lemma 4.6's $K - 1$ bit security, we simply use a first order approximation of the exponential at c:

$$\left| \exp\left(\frac{c}{2\sigma^2}\right) - 2 \cdot \left(1 + 17933 \cdot 2^{-43}\right) \right| \leq 2^{-46}.$$

4.5 Implementation of the approximation

We provide a SageMath code [4] which takes a function and an interval I as input and generates a polynomial approximation of the function on I according to the previously described procedure. This program was used to generate $P_{\exp}^{I_3}$, the approximation of exp on $I_3 = [-\ln(2), 0]$ and $P_{\cosh}^{I_2}$, the direct approximation of cosh on $I_2 = [0, \frac{B_2}{\sigma\alpha}]$. Given the refinement provided for cosh approximation, the direct approximation is not used for the constant time implementation. Although, it is suited for the masking of the Rejection Sampling of Section 5.3.2.

Remark 2. *We actually added some granularity on η and turned it into a vector that indicates the number of bits for each coefficient of the polynomial. This makes it possible to select more precision*

on the high degree coefficients and less on the lower degree ones. In this setting, η corresponds to the maximum size of the coefficients.

For BLISS-I parameters, we get the parameters

$$\begin{cases} \exp \text{ on } I_3, & (\gamma, \eta) = (9, 35) \text{ with } K = 40 \\ \exp \text{ on } I_3, & (\gamma, \eta) = (11, 45) \text{ with } K = 48 \\ \cosh \text{ on } I_2, & (\gamma, \eta) = (96, 110) \text{ with } K = 48. \end{cases}$$

Note that $P^{I_2}_{\cosh}$ has only 48 coefficients due to its parity. The description of the polynomials are given in Appendix B.4.

5 HIGH ORDER MASKING OF BLISS

In this section, BLISS is turned into a functionally equivalent scheme which is both constant-time (from the previous sections) and secure against more powerful side-channel attacks which exploit the leakage of several executions. This can be done after a preliminary step in which BLISS is slightly tweaked into a new scheme referred to as u-BLISS which outputs u even in case of failure. Then, only the key derivation scheme and the signature scheme must be protected, since the verification step does not manipulate sensitive data.

5.1 Side-channel attacks and masking

Theoretical leakage models have been introduced in order to properly reason on the security of implementations exposed to side-channel attacks.

The *probing model* or *ISW model* from its inventors [27] is undoubtedly the most deployed. In a nutshell, a cryptographic implementation is d-probing secure iff any set of at most d intermediate variables is independent from the secrets. This model is practically sound from the reduction established in [14] and also convenient to prove the security of an implementation as it manipulates finite sets of exact values.

The *masking* countermeasure, which performs computations on secret-shared data, appears as a natural countermeasure in this landscape. Basically, each input secret x is split into $d + 1$ variables $(x_i)_{0 \le i \le d}$ referred to as shares. d of them are generated uniformly at random whereas the last one is computed such that their additive combination reveals the secret value x. d is called *masking order* and represents the security level of an implementation.

While the conceptual idea behind the masking countermeasure is pretty simple, implementing it to achieve d-probing security has been shown to be a complex and error-prone task. Although it is straightforward on linear operations on which masking is equivalent to applying the original operation on each share of the sensitive data, the procedure is much more complicated on non-linear functions. In the latter, the mix of shares to compute the result makes it mandatory to introduce random variables and the bigger the program is, the more dependencies to be considered. This is why Barthe et al. formally defined in [2] two security properties, namely *non-interference* and *strong non-interference*, which (1) ease the security proofs for small gadgets (as algorithms operating on shared data), and (2) allows to securely combine secure gadgets by inserting refreshing gadgets (which refresh sharings using fresh randomness) at carefully chosen locations[8]. In a nutshell, a gadget is d-non-interfering (d-`NI`) iff any set of at most d observations can be perfectly simulated from at most d shares of each input. A gadget is d-strong non-interfering (d-`SNI`) iff any set of at most d observations whose d_{int} observations on the internal data and d_{out} observations on the outputs can be perfectly simulated from at most d_{int} shares of each input. It is easy to check that

[8]Notice that non-interference was already used in practice [13, 38] to prove probing security of implementations.

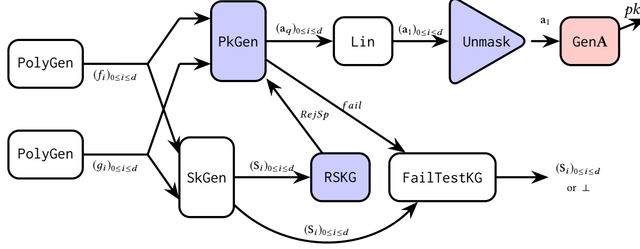


Fig. 14. Masked BLISS key generation. The white (resp. blue, red) gadgets are proved d-NI (resp. d-NIo with public outputs, unmasked).

d-SNI implies d-NI which implies d-probing security. An additional notion was introduced in [3] to reason on the security of lattices-based schemes in which some intermediate variables may be revealed to the adversary. Intuitively, a gadget with public outputs X is d-non-interfering with public outputs (d-NIo) iff every set of at most d intermediate variables can be perfectly simulated with the public outputs and at most d shares of each input.

Let u-BLISS be the variant of BLISS that outputs u even in case of failure. The following sections justify the security of $\mathbf{u}$-BLISS and prove the probing security of the key derivation and the signature procedures from the security properties fulfilled by their gadgets among d-NI, d-NIo, and d-SNI.

5.2 Overall structure

To achieve d-probing security, we need to carefully mask both the key derivation scheme and the signature scheme. For the sake of clarity, we focus on a single iteration of the latter. In other words, from now on, the signature algorithm considered is the same as in Figure 2 except that if the rejection sampling asks for a restart, the algorithm output $\perp$. The masking can be generalized by calling the masked signature algorithm when it fails.

For efficiency purposes, our masking countermeasure splits each sensitive data into $d + 1$ shares, namely $\mathbf{y}_1$, $\mathbf{y}_2$, $\mathbf{s}_1$, $\mathbf{s}_2$, $\mathbf{z}_1$, $\mathbf{z}_2$, and the intermediate variables that stricly depend on them. The public variables $(\mathbf{a}_1, \mathbf{a}_2)$ (i.e., the public key), μ (i.e., the message), $RejSp$ (i.e., the bit corresponding to the success of the rejection sampling), $fail$ (i.e., the bit corresponding to the success of the public key generation), $(\mathbf{z}_1, \mathbf{z}_2, \mathbf{c})$ (i.e., the signature) are left unmasked. Furthermore, because anyone can recombine $\lfloor \mathbf{u} \rceil_d \bmod p$, even if u is an intermediate value, it is considered as a public output, as well as bits $RejSp$ and $fail$.

Decompositions into sub-gadgets are provided in Figure 14 for the key derivation scheme and in Figure 15 for the signature scheme.

Some of these sub-gadgets are either trivial to mask or an efficient masked version is already provided in [3]. Efficient masked versions are designed and proved for the other ones in section 5.3. To further achieve global probing security for the signature, two calls of a d-SNI refreshing gadget are inserted at the outputs of Sign's calls for the signature. Table 4 recalls the security property achieved by the masked version of each one of the sub-gadgets used in the key derivation scheme, the signature scheme, or both. In all these reported cases, masking is efficiently performed either through a Boolean sharing or an arithmetic sharing coming with a dedicated modulus, depending on the sub-gadget and the manipulated data. To go from one sharing to another while preserving the d-probing security, we need to apply a conversion algorithm. An efficient algorithm between Boolean and arithmetic sharing for a non-prime modulus is already defined in [3]. Two tweaks to convert between two arithmetic sharings with different non-prime moduli are discussed afterwards.

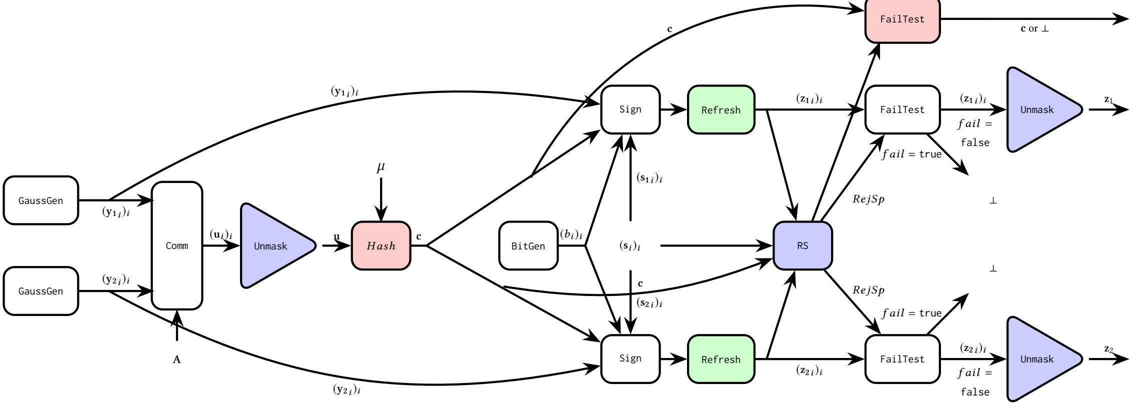


Fig. 15. Masked BLISS signature. The green (resp. white, blue, red) gadgets are proved d-SNI (resp. d-NI, d-NIo with public outputs, unmasked).

Table 4. Gadgets and their security properties.

Key Derivation		
Gadget	Property	Reference
PolyGen	d-NI	sec. 5.3
PkGen	d-NIo	sec. 5.3
SkGen	d-NI	sec. 5.3
RSKG	d-NIo	sec. 5.3
FailTestKG	d-NI	sec. 5.3
Lin	d-NI	sec. 5.3
Unmask	d-NIo	[3]
Gen**A**	none	

Signature		
Gadget	Property	Reference
GaussGen	d-NI	sec. 5.3
Comm	d-NI	[3]
Unmask	d-NIo	[3]
Hash	none	
BitGen	d-NI	sec. 5.3
Sign	d-NI	[3]
Refresh	d-SNI	[27]
RS	d-NIo	sec. 5.3
FailTest	d-NI	sec. 5.3

The security of masked key derivation and signature schemes as displayed in Figures 14 and 15 with gadgets' security properties in Table 4 is captured in Theorems 5.1 and 5.2. Proofs are given in the supplementary material (sections A.1 and A.2, and 5.3 for the individual gadgets).

THEOREM 5.1. *The masked* **u**-*BLISS key generation algorithm is d-NIo secure with public outputs pk, RejSp, and $fail$.*

The proof is given in Appendix A.

THEOREM 5.2. *The masked* **u**-*BLISS sign algorithm is d-NIo secure with public outputs u, RejSp, and $fail$.*

The proof is given in Appendix A.

Theorems 5.1 and 5.2 allow to reduce the EUF-CMA security of the BLISS signature scheme masked at order d in the d-probing model and in the random oracle model to the EUF-CMA security of the u-BLISS variant of the scheme in the random oracle model.

Based on the work of [3] we can prove that the security of u-BLISS reduces to the EUF-CMA security of the orginal BLISS scheme by introducing a mild computational assumption which is

close to the classical LWE problem. This problem informally states that distinguishing the output distribution of u when a rejection occurs from the uniform distribution on $\mathcal{R}_{2q}$ is hard.

It can seem artifical and ad-hoc to introduce such a new problem. However we can avoid it by hashing not $\mathbf{u}$ but $f(\mathbf{u})$ for some *statistically hiding commitment* f (which can itself be constructed under standard lattice assumptions). The downside of that approach is that it has a non negligible overhead in terms of key size, signature size, and to a lesser extent signature generation time.

5.3 Masked Gadgets

In this section, we give the masked versions of the sub-gadgets listed in Table 4 that are involved in the computation of the key derivation and/or the signature. They come with a sketch of proof of the property they achieve from Table 4. All the proofs use the security properties of smaller gadgets together with the compositional properties of [2]. We additionally discuss new methods to go from one arithmetic sharing to another with a different modulus, which can be used as independent contributions in other masking schemes.

5.3.1 Gadgets for Key Generation Scheme. As illustrated on Figure 14, the masked key generation algorithm can be divided into eight different sub-gadgets. We briefly describe the sub-gadgets for which masked version is trivially achieved and we provide deeper explanations for the more complex ones.

PolyGen. The first masked sub-gadget to be called is PolyGen for the uniformly random generation of two polynomials $\mathbf{f}$ and $\mathbf{g}$ with exactly d_1 entries in $\{\pm 1\}$ and d_2 entries in $\{\pm 2\}$ (Step 2 in Figure 1). Basically, for each polynomial, we first attribute the d_1 first coefficients to 1 and the next d_2 coefficients to 2. Then, a d-NI linear refresh gadget from [38] is applied on the sharing of $d+1$ elements made by the newly generated polynomial and d zeros. Then each shared coefficient of the polynomial is securely multiplied (using d-SNI function SecMult from [38]) with an arithmetic sharing of 1 or -1 generated with function BitGen. The last step consists in a random permutation of these coefficients, as in the constant-time version.

Lemma 5.3. PolyGen *is d-NI secure.*

Proof. The algorithm does not take any sensitive inputs. We thus show that any set of $\delta \leq d$ observations can be perfectly simulated with at most δ shares of each coefficient of the output polynomial $\mathbf{f}$ (resp. $\mathbf{g}$). Since there is no cycle, from the composition results of [2], it is enough to prove that each sub-gadget is d-NI to achieve global security. The first generation of coefficients only manipulates constants. Then, Linear-Refresh is d-NI from [2]. SecMult and BitGen are also proven to be d-NI, respectively in [2] and further in Lemma 5.10. Finally, the random permutation does not mix coefficients and only switch sharings, it is thus also d-NI. □

RSKG. Once the secret key is generated, a rejection sampling step is performed (Steps 4 and 5 in Figure 1). Its constant time version is given in Section 3.2.4. In the masked version, the first step (Step 2 in Figure 12) consists in matrices multiplications where matrices are defined by $\mathbf{s_1}$ and $\mathbf{s_2}$. For intermediate multiplications involving $\mathbf{s_1}$ and $\mathbf{s_2}$ (or their transposes) as operands, function d-SNI SecMult can be applied. When both operands involve the same part of the secret key, a secure refreshing function is called prior to the multiplication using FullRefresh (refreshing gadget introduced in [27] and proven to be d-SNI in [2]). Function NetworkSort basically compares and performs computations on coefficients of matrix $\mathbf{T}$. Each comparison can be performed using a d-SNI comparison algorithm as given in [3], and the computations can make use of the d-SNI SecMult function. For each row, the κ first matrix coefficients are added together via their arithmetic

sharings. Finally, a secure comparison can be performed with a final call to `Unmask` to safely output the Boolean value *RejSp*.

Lemma 5.4. `RSKG` *is* d-`NIo` *secure with public output RejSp.*

Proof. Each step of `RSKG` is computed with a d-`NI` or d-`SNI` function. Some cycles occur for functions taking as operands two inputs issued from the same secret element. Nevertheless, they have no impact since for each such cycle in Step 2, a d-`SNI` refreshing algorithm is performed to break dependencies and the additions in Steps 4 and 6 also manipulates data that are previously refreshed by d-`SNI` multiplications. Finally, function `Unmask` to output a single Boolean value makes function `RSKG` d-`NIo` secure with public output *RejSp*. □

`PkGen`. Our masked version of `PkGen` is a bit more complicated and we thus give its graphical description in Figure 16. Note that `SecArithBoolModp` (`SABModP` on the figure) was introduced in [3] and `NTT` is the classical Number Theoretic Transform and applies independently on each share. `SecIsNull` tests whether a shared value is equal to zero without revealing information on its sharing. Basically, all the complementary sharings for each bit (by complementing only the first share) are multiplied with function `SecMult` and function `Unmask` is then applied on the result.

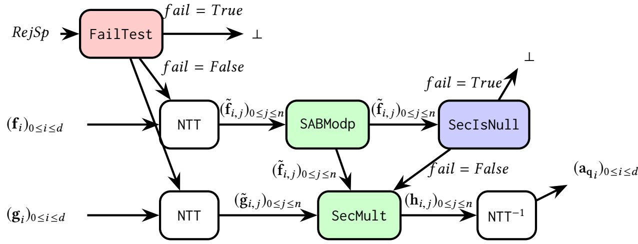


Fig. 16. Public key generation (PkGen). The green (resp. white, blue, red) gadgets are proved d-SNI (resp. d-NI, d-NIo with public outputs, unmasked).

Lemma 5.5. `PkGen` *is* d-`NIo` *secure with public output* $fail$.

Proof. `PkGen` involves three external functions, namely `NTT`, `SecArithBoolModp`, and `SecIsNull`. `NTT` applies independently on each share and `SecArithBoolModp` was proven to be d-`SNI` secure in [3]. We show that `SecIsNull` is d-`NIo` secure. Successive d-`SNI` multiplications (i.e., `SecMult`) are first performed sequentially to multiply the complementary of every input bit. Each of these multiplications applies on the sharing resulting from the previous multiplication and on the sharing of a new bit. There is no cycle in this procedure. Then, function d-`NIo` `Unmask` is applied on the result, making the whole scheme `SecIsNull` d-`NIo` with pubic output $fail$.

Let us now get back to `PkGen` algorithm. First, if *RejSp* is `false`, `NTT` is applied on each share (or linear function of share) independently for f and g inputs. Then, the d-`SNI` function `SecArithBoolModp` is applied on the updated variable f, followed by the d-`NIo` `SecIsNull` function. From this point, the current variable $\tilde{f}$ is processed with $\tilde{g}$ with sharewise product. So far, no intermediate variable depends on two shares of the same input. Finally, the inversion NTT^{-1} is applied on the product result sharewisely to get the output. Since there is no cycle in the successive calls of these secure functions, the global algorithm `PkGen` is d-`NIo` secure with the public output $fail$. □

Other sub-gadgets. `SkGen` (Step 3 in Figure 1) only modifies polynomial $\mathbf{g}$ with a linear transformation which can thus be applied separately on each share for the masked version. It returns a

sharing of the secret key $(\mathbf{s_1}, \mathbf{s_2})$. Function `FailTestKG` takes as inputs Boolean results from `PkGen` and `RSKG` and a sharing of the secret key. It propagates the sharing if and only if both Boolean values are true, and returns $\perp$ otherwise. Its only change from the constant-time version is the propagation of a sharing instead of the original secret key. `Lin` simply doubles variable a_q (cf. Step 7 of Figure 1). In the masked version, it applies this constant-time linear transformation on each share of its input independently. Function `Unmask` simply refreshes its input with `FullRefresh` (refreshing gadget introduced in [27] and proven to be d-`SNI` in [2]) and unmasks the resulting sharing through a basic addition. Eventually, `Gen`**A** only manipulates non-sensitive data and is left unmodified for a masked implementation.

LEMMA 5.6. `SkGen`, `FailTestKG` *and* `Lin` *are* d-`NI` *secure, and* `Unmask` *is* d-`NIo` *secure with public output* $\mathbf{a_1}$.

PROOF. `SkGen`, `FailTestKG` and `Lin` are trivially d-`NI` secure as they apply linear transformation on each share independently. `Unmask` is d-`NIo` secure from [3] with public output $\mathbf{a_1}$. □

5.3.2 *Gadgets for Signature Scheme.*

Sub-gadgets used in the masked signature scheme are displayed on Figure 15. A brief description is given for sub-gadgets that are trivial to mask whereas deeper explanations are given for the other ones.

`GaussGen`. As in the constant-time version, the masked Gaussian generation relies on a table of w Gaussian values p_j. Basically the idea is to generate a uniform value r and to return the index j such that $p_j \leq r < p_{j+1}$. In the masked version, r is a sensitive value which is generated as a $(d+1)$ Boolean sharing $(r_i)_{0 \leq i \leq d}$. Then at each step j, a secure comparison is performed between the sharing $(r_i)_{0 \leq i \leq d}$ and the current value p_j. The result of the comparison is a $(d+1)$ Boolean sharing $(b_{j,i})_{0 \leq i \leq d}$ which represents a value 0 when $r < p_j$ or 1 otherwise. This value is safely multiplied with the complementary of $(b_{j-1,i})_{0 \leq i \leq d}$ to ensure that the shared value represents 1 only when $p_j \leq r < p_{j+1}$. At that point, for $1 \leq j \leq w$, each sharing $(b_{j,i})_{0 \leq i \leq d}$ is multiplied (with `SecMult`) with the sharing $(j, 0, \ldots, 0)$ on $\log(w) + 1$ bits. The w resulting Boolean sharings are all added together share by share. The result is a secure $(d+1)$ Boolean sharing of index j such that $p_j \leq r < p_{j+1}$. A Boolean to arithmetic conversion is applied to output an arithmetic sharing.

LEMMA 5.7. `GaussGen` *is* d-`NI` *secure.*

PROOF. Each step of the process above is performed with secure operations, namely `SecMult` for logical and or multiplications or linear transformations share by share. The comparison and the conversion are provided in [3] and proven to be d-`SNI` secure. The global composition contains cycles due to the dependency of the last additions operands with the common input r. Nevertheless, the dependency is broken with the use of the d-`SNI` `SecMult` function for the multiplication of each sharing $(b_{j,i})_{0 \leq i \leq d}$ with $(j, 0, \ldots, 0)$. □

`Sign`. `Sign` corresponds to Steps 6 and 7 of Figure 2. In the masked version, $\tilde{b}_i$ is an arithmetic sharing of $(-1)^b$, and $\mathbf{s_i}$ and $\mathbf{y_i}$ are arithmetic sharings of s_i and y_i. A first call to `SecMult` is performed between $\tilde{b}_i$ and $\mathbf{s_i}$ to safely compute a sharing of $(-1)^b s_i$. Then a linear combination is applied to generate independently each share of $\mathbf{z}$ from a share of the multiplication result, a share of $\mathbf{y_i}$, and the commitment $\mathbf{c}$.

LEMMA 5.8. `Sign` *is* d-`NI` *secure.*

PROOF. The first multiplication step is perfectly handled with algorithm `SecMult` from [38] which was proven to be d-`SNI` secure in [2]. The second step is linear and manipulates two independent

inputs (an input of `Sign` and the output of a d-SNI gadget) share by share. It is thus d-NI. The absence of cycles makes the masked version of `Sign` d-NI secure from the compositional properties established in [2]. □

RS. The steps which compose the constant-time version of RS are given in Section 4. They can easily be transformed to ensure d-probing security. Step 1 computes two elements x_1 and x_2 from sensitive values $\mathbf{s}$ and $\mathbf{z}$. Multiplications must be processed with function `SecMult` in the masked version. As for Step 2, two sets of $d+1$ Boolean shares are generated at random in $\{0,1\}$ to represent the secret bits u_1 and u_2. Steps 3 and 4 require the computation of $\exp(x_1)$ and $\cosh(x_2)$ with x_1 and x_2 sensitive values shared in Step 1. Thanks to the polynomial approximation of these two functions, as described in Section 4, the evaluation of exp and cosh for these two sharings is only a combination of linear squarings and `SecMult` operations. As for their comparison with functions of u_1 and u_2, the computed arithmetic sharings are first converted into Boolean sharings as suggested in [3] (sharing of u_1 can be first converted into an arithmetic masking to be subtracted to $\exp(x_1)$ and allows a comparison with public values). Then, a secure comparison is performed between Boolean sharings and outputs two masked bits of a and b. Finally, the last multiplication in Step 5 is computed with `SecMult`, before a call to Unmask outputs *RejSp*.

Lemma 5.9. RS *is* d-NIo *secure with the public output RejSp.*

Proof. All the steps in RS are either d-NI secure (random generations of Boolean sharings), d-NIo secure (Unmask) or d-SNI secure (masking conversions, comparisons, polynomial evaluation, and multiplication). Thus, it is enough that each cycle is properly handled. Basically, the algorithm produces one cycle since the logical and of both Boolean values a and b takes as input variables that both depend on the secret key. Nevertheless, both multiplication inputs are refreshed with d-SNI gadgets which is enough to break the dependency. □

Other Gadgets. The commitment function Comm takes as inputs two arithmetic sharings and the public key. The linear transformation applied in the unmasked version (Step 3, Figure 2) is here applied on each share of the secret inputs. As proven in [3], function Unmask is d-NIo secure with public output a signature part or $\mathbf{u}$. Hash applies on a public output, so it is left unchanged in the masked version. `BitGen` first generates $d+1$ bits uniformly at random to build a Boolean sharing of a value in $\{0,1\}$. The resulting sharing is then converted into an arithmetic sharing using the secure conversion method provided in [3]. In the masked version of the signature, `FailTest` simply returns $\perp$ if *RejSp* is true and the input sharing $(\mathbf{z}_i)_{0\leq i\leq d}$ or c otherwise.

Lemma 5.10. Comm, `BitGen`*, and* `FailTest` *are* d-NI *secure and* Unmask *is* d-NIo *secure with the public output a signature part or* $\mathbf{u}$.

Proof. Functions Comm and `FailTest` manipulates shares separately and thus are trivially d-NI secure. The first step of `BitGen` separately generates uniform random bits. They are then processed in a d-SNI secure conversion function as proven in [3]. `BitGen` is thus (at least) d-NI secure. Unmask is d-NIo secure from [3] with public output $\mathbf{u}$ or the signature. □

5.3.3 *Masking Conversion.* The state of the art provides efficient techniques to convert Boolean masking into arithmetic masking with power-of-two modulus and the reverse for higher-order implementations [12]. A recent paper additionally extends these tools to convert from Boolean masking to arithmetic masking with any modulus [3]. To efficiently mask the polynomial approximation in our constant-time implementation of BLISS, we need an unusual conversion between arithmetic masking with a modulus q and arithmetic masking with a modulus $q' \gg q$. Our approximations being of high degrees, we need to update our modulus accordingly. One easy way to do it is to

convert the first arithmetic masking with modulus q into a Boolean masking and then to convert it back to the second arithmetic masking with modulus q'. This requires two full conversions. Another possible method is to adapt one of the conversion algorithms given in [12] and extended in [3] for any modulus to an arithmetic to arithmetic masking. The only step to modify is the operation SecAdd which takes two Boolean sharings of x and y in inputs and outputs a Boolean sharing of z such that $z = x + y$ with the arithmetic modulus. In our case, the Boolean sharings are replaced by arithmetic sharings with a modulus q' and the arithmetic addition to perform is to be done with a modulus $q \ll q'$. Namely, we have two arithmetic sharings $(x_i)_{0 \le i \le d}$ and $(y_i)_{0 \le i \le d}$ modulo q' of values x and y and we want to obtain an arithmetic $(z_i)_{0 \le i \le d}$ modulo q' of a value z such that $z = x + y \mod q$. Basically, we can perform an arithmetic addition modulo q of the lowest part (i.e., the less significant bits) of x and y's sharings to avoid the carry management. Then, only the highest part (i.e., the most significant bits) of the sharings are to be converted into Boolean shares. The addition is then performed as in the paper [3] and a final Boolean to arithmetic conversion ends the operation. Note that in this case, we also need to have an arithmetic to Boolean and a Boolean to arithmetic conversions. However, these two conversions are dependent on the number of bits to convert. And by saving the less significant bits of the sharings, both conversions are cheaper. Concretely, as x and y are values between 0 and $q - 1$, we can save $\log_2(q) - \log_2(2(d + 1))$ bits in the lowest part, leaving $\log_2(q') - \log_2(q) + \log_2(2(d + 1))$ to convert.

6 ACKNOWLEDGEMENTS

We thank the anonymous reviewers for their very relevant comments and their analysis of our code. We acknowledge the support of the French Programme d'Investissement d'Avenir under national project RISQ P14158. This work is also partially supported by the European Union's H2020 Programme under grant agreement number ERC-669891 and the PROMETHEUS project (grant 780701), and by the French FUI-AAP25 VERISICC project. This research has been partially funded by ANRT under the programs CIFRE N 2016/1583. This work was partially supported by Office of Naval Research under projects N00014-12-1-0914, N00014-15-1-2750 and N00014-19-1-2292.

REFERENCES

[1] Erdem Alkim, Paulo S. L. M. Barreto, Nina Bindel, Patrick Longa, and Jefferson E. Ricardini. 2019. The lattice-based digital signature scheme qTESLA. Cryptology ePrint Archive, Report 2019/085. https://eprint.iacr.org/2019/085.

[2] Gilles Barthe, Sonia Belaïd, François Dupressoir, Pierre-Alain Fouque, Benjamin Grégoire, Pierre-Yves Strub, and Rébecca Zucchini. 2016. Strong Non-Interference and Type-Directed Higher-Order Masking. In *ACM CCS 2016*, Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi (Eds.). ACM Press, 116–129. https://doi.org/10.1145/2976749.2978427

[3] Gilles Barthe, Sonia Belaïd, Thomas Espitau, Pierre-Alain Fouque, Benjamin Grégoire, Mélissa Rossi, and Mehdi Tibouchi. 2018. Masking the GLP Lattice-Based Signature Scheme at Any Order. In *EUROCRYPT 2018, Part II (LNCS)*, Jesper Buus Nielsen and Vincent Rijmen (Eds.), Vol. 10821. Springer, Heidelberg, 354–384. https://doi.org/10.1007/978-3-319-78375-8_12

[4] Gilles Barthes, Sonia Belaïd, Thomas Espitau, Mélissa Rossi, and Mehdi Tibouchi. 2019. GALACTICS implementations. https://github.com/espitau/GALACTICS

[5] Daniel J. Bernstein and VAMPIRE Lab others. 2016. System for Unified Performance Evaluation Related to Cryptographic Operations and Primitives. https://bench.cr.yp.to/supercop.html.

[6] Jonathan Bootle, Claire Delaplace, Thomas Espitau, Pierre-Alain Fouque, and Mehdi Tibouchi. 2018. LWE Without Modular Reduction and Improved Side-Channel Attacks Against BLISS. In *ASIACRYPT 2018, Part I (LNCS)*, Thomas Peyrin and Steven Galbraith (Eds.), Vol. 11272. Springer, Heidelberg, 494–524. https://doi.org/10.1007/978-3-030-03326-2_17

[7] Joppe W. Bos. 2014. Constant time modular inversion. *Journal of Cryptographic Engineering* 4, 4 (Nov. 2014), 275–281. https://doi.org/10.1007/s13389-014-0084-8

[8] Nicolas Brisebarre and Sylvain Chevillard. 2018. Efficient polynomial L^∞-approximations. In *18th IEEE Symposium on Computer Arithmetic (ARITH 18)*. IEEE, 169–176.

[9] Leon Groot Bruinderink, Andreas Hülsing, Tanja Lange, and Yuval Yarom. 2016. Flush, Gauss, and Reload - A Cache Attack on the BLISS Lattice-Based Signature Scheme. In *CHES 2016 (LNCS)*, Benedikt Gierlichs and Axel Y. Poschmann (Eds.), Vol. 9813. Springer, Heidelberg, 323–345. https://doi.org/10.1007/978-3-662-53140-2_16

[10] E. J. Candés, X. Li, and M. Soltanolkotabi. 2015. Phase Retrieval via Wirtinger Flow: Theory and Algorithms. *IEEE Transactions on Information Theory* 61, 4 (2015), 1985–2007.

[11] S. Chevillard, M. Joldeş, and C. Lauter. 2010. Sollya: An Environment for the Development of Numerical Codes. In *Mathematical Software - ICMS 2010 (Lecture Notes in Computer Science)*, K. Fukuda, J. van der Hoeven, M. Joswig, and N. Takayama (Eds.), Vol. 6327. Springer, Heidelberg, Germany, 28–31.

[12] Jean-Sébastien Coron, Johann Großschädl, and Praveen Kumar Vadnala. 2014. Secure Conversion between Boolean and Arithmetic Masking of Any Order. In *CHES 2014 (LNCS)*, Lejla Batina and Matthew Robshaw (Eds.), Vol. 8731. Springer, Heidelberg, 188–205. https://doi.org/10.1007/978-3-662-44709-3_11

[13] Jean-Sébastien Coron, Emmanuel Prouff, Matthieu Rivain, and Thomas Roche. 2014. Higher-Order Side Channel Security and Mask Refreshing. In *FSE 2013 (LNCS)*, Shiho Moriai (Ed.), Vol. 8424. Springer, Heidelberg, 410–424. https://doi.org/10.1007/978-3-662-43933-3_21

[14] Alexandre Duc, Stefan Dziembowski, and Sebastian Faust. 2014. Unifying Leakage Models: From Probing Attacks to Noisy Leakage. In *EUROCRYPT 2014 (LNCS)*, Phong Q. Nguyen and Elisabeth Oswald (Eds.), Vol. 8441. Springer, Heidelberg, 423–440. https://doi.org/10.1007/978-3-642-55220-5_24

[15] Léo Ducas. 2014. Accelerating Bliss: the geometry of ternary polynomials. Cryptology ePrint Archive, Report 2014/874. http://eprint.iacr.org/2014/874.

[16] Léo Ducas, Alain Durmus, Tancrède Lepoint, and Vadim Lyubashevsky. 2013. Lattice Signatures and Bimodal Gaussians. In *CRYPTO 2013, Part I (LNCS)*, Ran Canetti and Juan A. Garay (Eds.), Vol. 8042. Springer, Heidelberg, 40–56. https://doi.org/10.1007/978-3-642-40041-4_3

[17] Léo Ducas, Steven Galbraith, Thomas Prest, and Yang Yu. 2019. Integral matrix Gram root and lattice Gaussian sampling without floats. Cryptology ePrint Archive, Report 2019/320. https://eprint.iacr.org/2019/320.

[18] Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, Peter Schwabe, Gregor Seiler, and Damien Stehlé. 2018. CRYSTALS-Dilithium: A Lattice-Based Digital Signature Scheme. *IACR TCHES* 2018, 1 (2018), 238–268. https://doi.org/10.13154/tches.v2018.i1.238-268 https://tches.iacr.org/index.php/TCHES/article/view/839.

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

[20] Léo Ducas and Phong Q. Nguyen. 2012. Faster Gaussian Lattice Sampling Using Lazy Floating-Point Arithmetic. In *ASIACRYPT 2012 (LNCS)*, Xiaoyun Wang and Kazue Sako (Eds.), Vol. 7658. Springer, Heidelberg, 415–432. https://doi.org/10.1007/978-3-642-34961-4_26

[21] Thomas Espitau, Pierre-Alain Fouque, Benoît Gérard, and Mehdi Tibouchi. 2018. Loop-Abort Faults on Lattice-Based Signature Schemes and Key Exchange Protocols. *IEEE Trans. Computers* 67, 11 (2018), 1535–1549.

[22] Thomas Espitau, Pierre-Alain Fouque, Benoît Gérard, and Mehdi Tibouchi. 2017. Side-Channel Attacks on BLISS Lattice-Based Signatures: Exploiting Branch Tracing against strongSwan and Electromagnetic Emanations in Microcontrollers. In *ACM CCS 2017*, Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu (Eds.). ACM Press, 1857–1874. https://doi.org/10.1145/3133956.3134028

[23] Tim Güneysu, Vadim Lyubashevsky, and Thomas Pöppelmann. 2012. Practical Lattice-Based Cryptography: A Signature Scheme for Embedded Systems. In *CHES 2012 (LNCS)*, Emmanuel Prouff and Patrick Schaumont (Eds.), Vol. 7428. Springer, Heidelberg, 530–547. https://doi.org/10.1007/978-3-642-33027-8_31

[24] Nicholas J. Higham. 2002. *Accuracy and Stability of Numerical Algorithms* (second ed.). SIAM.

[25] Nick Howgrave-Graham. 2007. A Hybrid Lattice-Reduction and Meet-in-the-Middle Attack Against NTRU. In *CRYPTO 2007 (LNCS)*, Alfred Menezes (Ed.), Vol. 4622. Springer, Heidelberg, 150–169. https://doi.org/10.1007/978-3-540-74143-5_9

[26] Andreas Hülsing, Tanja Lange, and Kit Smeets. 2018. Rounded Gaussians - Fast and Secure Constant-Time Sampling for Lattice-Based Crypto. In *PKC 2018, Part II (LNCS)*, Michel Abdalla and Ricardo Dahab (Eds.), Vol. 10770. Springer, Heidelberg, 728–757. https://doi.org/10.1007/978-3-319-76581-5_25

[27] Yuval Ishai, Amit Sahai, and David Wagner. 2003. Private Circuits: Securing Hardware against Probing Attacks. In *CRYPTO 2003 (LNCS)*, Dan Boneh (Ed.), Vol. 2729. Springer, Heidelberg, 463–481. https://doi.org/10.1007/978-3-540-45146-4_27

[28] Angshuman Karmakar, Sujoy Sinha Roy, Oscar Reparaz, Frederik Vercauteren, and Ingrid Verbauwhede. 2018. Constant-Time Discrete Gaussian Sampling. *IEEE Trans. Computers* 67, 11 (2018), 1561–1571.

[29] Wangyu Luo, Wael Alghamdi, and Yue M. Lu. 2019. Optimal Spectral Initialization for Signal Recovery With Applications to Phase Retrieval. *IEEE Trans. Signal Processing* 67, 9 (2019), 2347–2356.

[30] Vadim Lyubashevsky. 2009. Fiat-Shamir with Aborts: Applications to Lattice and Factoring-Based Signatures. In *ASIACRYPT 2009 (LNCS)*, Mitsuru Matsui (Ed.), Vol. 5912. Springer, Heidelberg, 598–616. https://doi.org/10.1007/978-

3-642-10366-7_35
[31] Vadim Lyubashevsky. 2012. Lattice Signatures without Trapdoors. In *EUROCRYPT 2012 (LNCS)*, David Pointcheval and Thomas Johansson (Eds.), Vol. 7237. Springer, Heidelberg, 738–755. https://doi.org/10.1007/978-3-642-29011-4_43
[32] Daniele Micciancio and Michael Walter. 2017. Gaussian Sampling over the Integers: Efficient, Generic, Constant-Time. In *CRYPTO 2017, Part II (LNCS)*, Jonathan Katz and Hovav Shacham (Eds.), Vol. 10402. Springer, Heidelberg, 455–485. https://doi.org/10.1007/978-3-319-63715-0_16
[33] Peter Pessl, Leon Groot Bruinderink, and Yuval Yarom. 2017. To BLISS-B or not to be: Attacking strongSwan's Implementation of Post-Quantum Signatures. In *ACM CCS 2017*, Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu (Eds.). ACM Press, 1843–1855. https://doi.org/10.1145/3133956.3134023
[34] Thomas Pöppelmann, Léo Ducas, and Tim Güneysu. 2014. Enhanced Lattice-Based Signatures on Reconfigurable Hardware. In *CHES 2014 (LNCS)*, Lejla Batina and Matthew Robshaw (Eds.), Vol. 8731. Springer, Heidelberg, 353–370. https://doi.org/10.1007/978-3-662-44709-3_20
[35] Thomas Prest. 2017. Sharper Bounds in Lattice-Based Cryptography Using the Rényi Divergence. In *ASIACRYPT 2017, Part I (LNCS)*, Tsuyoshi Takagi and Thomas Peyrin (Eds.), Vol. 10624. Springer, Heidelberg, 347–374. https://doi.org/10.1007/978-3-319-70694-8_13
[36] Thomas Prest, Pierre-Alain Fouque, Jeffrey Hoffstein, Paul Kirchner, Vadim Lyubashevsky, Thomas Pornin, Thomas Ricosset, Gregor Seiler, William Whyte, and Zhenfei Zhang. 2017. FALCON: Fast-Fourier Lattice-based Compact Signatures over NTRU. https://falcon-sign.info.
[37] Oscar Reparaz, Josep Balasch, and Ingrid Verbauwhede. 2017. Dude, is my code constant time?. In *DATE*, David Atienza and Giorgio Di Natale (Eds.). IEEE, 1697–1702.
[38] Matthieu Rivain and Emmanuel Prouff. 2010. Provably Secure Higher-Order Masking of AES. In *CHES 2010 (LNCS)*, Stefan Mangard and François-Xavier Standaert (Eds.), Vol. 6225. Springer, Heidelberg, 413–427. https://doi.org/10.1007/978-3-642-15031-9_28
[39] S.L Sobolev. 1963. On a theorem of functional analysis. *Transl. Amer. Math. Soc.* 34 (1963), 39–68.
[40] Mehdi Tibouchi and Alexandre Wallet. 2019. One bit is all it takes: a devastating timing attack on BLISS's non-constant time sign flips. Cryptology ePrint Archive, Report 2019/898. https://eprint.iacr.org/2019/898.
[41] Thomas Wunderer. 2019. A detailed analysis of the hybrid lattice-reduction and meet-in-the-middle attack. *J. Mathematical Cryptology* 13, 1 (2019), 1–26.
[42] Raymond K. Zhao, Ron Steinfeld, and Amin Sakzad. 2018. FACCT: FAst, Compact, and Constant-Time Discrete Gaussian Sampler over Integers. Cryptology ePrint Archive, Report 2018/1234. https://eprint.iacr.org/2018/1234.

Appendices

A MASKING

A.1 Proof of Theorem 5.1

Proof. From Table 4, all the sub-gadgets involved in the computation of the key derivation are either d-NI secure, d-NIo secure, or they do not manipulate sensitive data. In all cases, this means that no probing attack can be performed on only one of these gadgets. We prove here the d-probing security with outputs of their composition. In the d-probing model, we assume that an attacker has access to $\delta \leq d$ exact variables in the whole execution of the key derivation. Then, we want to prove that all these δ observations can be perfectly simulated with at most δ shares of each secret and the public variables. We consider the following distribution of the attacker's δ observations: δ_1 (resp. δ_2) on the instance of PolyGen which produces f (resp. g), δ_3 on SkGen, δ_4 on RSKG, δ_5 on PkGen, δ_6 on FailTest, δ_7 on Lin, δ_8 on Unmask, and δ_9 on Gen**A**, such that $\sum_{i=1}^{9} \delta_i = \delta$.

As first demonstrated in [2], we build the proof from right to left. Gen**A** only manipulates non-sensitive data, so any of the δ_9 observations are non-sensitive as well and can be revealed. Unmask is d-NIo secure with public output the public key part a_1 and does not return any sensitive variable. Then all its observations can be perfectly simulated with at most δ_8 shares of $(\mathbf{a_1})_{0 \leq i \leq d}$ and the knowledge of the output a_1. Lin is also d-NI secure with δ_7 internal observations and at most δ_8 output observations. As $\delta_7 + \delta_8 \leq \delta$, all further observations can be perfectly simulated with at most $\delta_7 + \delta_8$ shares of a_q. FailTest is d-NI secure with δ_6 observations. Thus, the latter can be perfectly simulated with at most δ_6 shares of s_i and the knowledge of its non-sensitive Boolean inputs. PkGen is d-NIo secure with δ_5 observations on its internal data and at most $\delta_7 + \delta_8$ observations on its outputs. $\delta_5 + \delta_7 + \delta_8 \leq \delta$ so all further observations can be perfectly simulated with at most $\delta_5 + \delta_7 + \delta_8$ shares of f_i and $\delta_5 + \delta_7 + \delta_8$ shares of g_i and the knowledge of the non-sensitive value *RejSp* and of its output. RSKG is d-NIo secure with public output the Boolean value *RejSp*. It comes with $\delta_4 \leq \delta$ observations on its internal data and its only output is non-sensitive. All its observations can be perfectly simulated with at most δ_4 shares of s_i. SkGen is d-NI secure with δ_3 observations on its internal data and at most $\delta_4 + \delta_6$ observations on its outputs. As $\delta_3 + \delta_4 + \delta_6 \leq \delta$, all these observations can be perfectly simulated with at most $\delta_3 + \delta_4 + \delta_6$ shares of f_i and $\delta_3 + \delta_4 + \delta_6$ shares of g_i. At the end, PolyGen is d-NI secure with no inputs. We thus need to check, for each of its two instances, that the sum of all its observations does not exceed δ. The instance involving f_i gathers δ_1 internal observations and $\delta_5 + \delta_7 + \delta_8 + \delta_3 + \delta_4 + \delta_6$ output observations. The instance involving g_i gathers δ_2 internal observations and $\delta_5 + \delta_7 + \delta_8 + \delta_3 + \delta_4 + \delta_6$ output observations. The number of observations remains less than δ, which concludes the proof. □

A.2 Proof of Theorem 5.2

Proof. From Table 4, all the sub-gadgets involved in the computation of the signature are either d-NI secure, d-NIo secure, d-SNI secure, or they do not manipulate sensitive data. In all cases, this means that no probing attack can be performed on only one of these gadgets. We prove here the d-probing security with outputs of their composition. In the d-probing model, we assume that an attacker has access to $\delta \leq d$ exact variables in the whole execution of the key derivation. Then, we want to prove that all these δ observations can be perfectly simulated with at most δ shares of each secret among y_1, y_2, s_1, and s_2, and the public variables. We consider the following distribution of the attacker's δ observations: δ_1 (resp. δ_2) on the instance of GaussGen which produces y_1 (resp. y_2), δ_3 on Comm, δ_4 on the instance of Unmask following Comm, δ_5 on *Hash*, δ_6 on BitGen, δ_7 (resp. δ_8) on the instance of Sign involving y_1 (resp. y_2), δ_9 (resp. δ_{10}) on the instance of Refresh which outputs

z_1 (resp. z_2), δ_{11} on RS, δ_{12} (resp. δ_{13} and δ_{14}) on the instance of `FailTest` involving c (resp. z_1 and z_2), and δ_{15} (resp. δ_{16}) on the instance of `Unmask` involving z_1 (resp. z_2), such that $\sum_{i=1}^{16} \delta_i = \delta$.

We build the proof from right to left. `Unmask` is d-NIo secure with public output both part of the signature z_1 and z_2. As a consequence, all the observations from its call involving z_1 (resp z_2) can be perfectly simulated with at most $\delta_{15} \leq \delta$ shares of z_1 (resp. at most $\delta_{16} \leq \delta$ shares of z_2) and the knowledge of the signature part z_1 (resp z_2). The algorithm referred to as `FailTest` is also d-NI secure. Thus, all the observations from its call involving z_1 (resp z_2) can be perfectly simulated with at most $\delta_{13} + \delta_{15} \leq \delta$ shares of z_1 (resp. at most $\delta_{14} + \delta_{16} \leq \delta$ shares of z_2) and the bit *RejSp* which is public information indicating whether or not the rejection sampling failed. The third instance of `FailTest` involving c does not manipulate any sensitive data and can be safely left unmasked. Then, RS is d-NIo secure with public output *RejSp* and does not return any sensitive element. All the observations performed in this gadget and its output can be perfectly simulated with at most δ_{11} shares of each input among s_1, s_2, z_1, z_2, the knowledge of c, which is here public, and the knowledge of *RejSp*. Continuing from right to left, we consider both instances of `Refresh`. From its d-SNI security property and since the output and local observations are still less than δ, all observations from its call can be perfectly simulated with at most $\delta_9 \leq \delta$ (resp. $\delta_{10} \leq \delta$) input shares. Both instances of `Sign` outputs variables that are immediately refreshed. `Sign` is additionally d-NI secure and has δ_9 (resp. δ_{10}) output observations and δ_7 (resp. δ_8) internal ones. In both cases the addition of the internal and output observations remains below δ and the d-NI property makes it possible to simulate all further observations with $\delta_7 + \delta_9 \leq \delta$ shares of $y_{1,i}$, $s_{1,i}$, b_i, and the knowledge of c (resp. $\delta_8 + \delta_{10} \leq \delta$ shares of $y_{2,i}$, $s_{2,i}$, b_i, and the knowledge of c). `BitGen` is d-NI secure and since it has no inputs, the fact that the $\delta_7 + \delta_9 + \delta_8 + \delta_{10} \leq \delta$ output observations and δ_6 internal observations are less than δ is enough to guarantee the global security from its location. `Hash` only manipulates public data. `Unmask` is d-NIo secure and does not return any sensitive variable. Then all the observations performed from this gadget can be perfectly simulated with at most δ_4 shares of u_i. `Comm` is d-NI secure. δ_3 observations are performed on its intermediate variables, and at most δ_4 observations are performed on its outputs. As $\delta_3 + \delta_4 \leq \delta$, all further observations can be perfectly simulated with at most $\delta_3 + \delta_4$ shares of y_1, $\delta_3 + \delta_4$ shares of y_2 and the knowledge of the public value A. The last step of the proof is to verify that all the internal and output observations on each instance of `GaussGen` are less than δ. Internal observations are respectively δ_1 and δ_2 while output observations are bounded by $\delta_3 + \delta_4 + \delta_7 + \delta_9$ and $\delta_3 + \delta_4 + \delta_8 + \delta_{10}$ which are both less than δ. The d-NIo property of `GaussGen` concludes the proof. □

B PROOFS

B.1 Proof of Corollary 4.2

Proof. Let us suppose that $\left|\frac{\mathcal{D}-\mathcal{D}'}{\mathcal{D}}\right| \leq 2^{-45}$ is verified. Then, $1 - 2^{-45} \leq \frac{\mathcal{D}'}{\mathcal{D}} \leq 1 + 2^{-45}$. Since $Supp(\mathcal{D}') = Supp(\mathcal{D})$, we apply the relative error Lemma (Lemma 3 of [35]), and get

$$
\begin{aligned}
R_{2\cdot\lambda}(\mathcal{D}||\mathcal{D}') &\leq 1 + \tfrac{2\cdot\lambda\cdot 2^{-2\cdot 45}}{2} \\
&\leq 1 + 256 \cdot 2^{-2\cdot 45} \\
&\leq 1 + 2^{-82} \\
&= 1 + \tfrac{1}{4q_{\mathcal{D}}}.
\end{aligned}
$$

This corresponds to Equation 2 and completes the proof.

□

B.2 Proof of Lemma 4.5

Proof. Let $x_0 \in I$ be such that $|u(x)| \geq |u(x_0)|$ for all $x \in I$. We then write

$$u(x) = u(x_0) + \int_{x_0}^{x} u'$$

Hence,

$$|u(x)| \leq |u(x_0)| + \int_{x_0}^{x} |u'| \leq \frac{1}{|I|} \int_I |u| + \int_I |u'|$$

Using Cauchy-Schwarz, we have $\left(\int_I |u| \right)^2 \leq |I| \int_I |u|^2$ and $\left(\int_I |u'| \right)^2 \leq |I| \int_I |u'|^2$. Then,

$$|u(x)| \leq \frac{1}{|I|} \sqrt{|I| \int_I |u|^2} + \sqrt{|I| \int_I |u'|^2}$$

Using the equality $x + y \leq \sqrt{2}\sqrt{x^2 + y^2}$ for $x, y \geq 0$, we have

$$|u(x)| \leq \sqrt{2} \sqrt{\frac{1}{|I|^2} \cdot |I| \int_I |u|^2 + |I| \int_I |u'|^2}$$

Then,

$$||u||_\infty \leq \sqrt{2} \cdot |u|_S$$

which concludes the proof. □

B.3 Proof of Lemma 4.6

Proof. By hypothesis,

$$\forall t \in I_3 \quad \frac{|P_{\exp}^{I_3}(t) - \exp\left(\frac{t}{2\sigma^2}\right)|}{|\exp\left(\frac{t}{2\sigma^2}\right)|} \leq \left\| \frac{P_{\exp}^{I_3} - \exp\left(\frac{\cdot}{2\sigma^2}\right)}{\exp\left(\frac{\cdot}{2\sigma^2}\right)} \right\|_\infty \leq 2^{-K}$$

Then, let us compute the relative error for $P^{I_2}_{\cosh}$. For $x \in I_2$,

$$
\begin{aligned}
&\frac{|P^{I_2}_{\cosh}(x) - \cosh\left(\frac{x}{2\sigma^2}\right)|}{|\cosh\left(\frac{x}{2\sigma^2}\right)|} \\
&= \left|\frac{P^{I_2}_{\cosh}(x)}{\cosh\left(\frac{x}{2\sigma^2}\right)} - 1\right| \\
&= \left|\frac{2^{\lfloor \frac{x}{c} \rfloor} \cdot P^{I_3}_{\exp}(t(x)) + 2^{\lfloor \frac{-x}{c} \rfloor} \cdot P^{I_3}_{\exp}(-t(x))}{\exp\left(\frac{x}{2\sigma^2}\right) + \exp\left(\frac{-x}{2\sigma^2}\right)} - 1\right| \\
&= \left|\frac{2^{\lfloor \frac{x}{c} \rfloor} \cdot P^{I_3}_{\exp}(t(x)) + 2^{\lfloor \frac{-x}{c} \rfloor} \cdot P^{I_3}_{\exp}(-t(x)) - \exp\left(\frac{x}{2\sigma^2}\right) - \exp\left(\frac{-x}{2\sigma^2}\right)}{\exp\left(\frac{x}{2\sigma^2}\right) + \exp\left(\frac{-x}{2\sigma^2}\right)}\right| \\
&\leq \left|\frac{2^{\lfloor \frac{x}{c} \rfloor} \cdot P^{I_3}_{\exp}(t(x)) - \exp\left(\frac{x}{2\sigma^2}\right)}{(\exp\left(\frac{x}{2\sigma^2}\right) + \exp\left(\frac{-x}{2\sigma^2}\right))}\right| + \left|\frac{2^{\lfloor \frac{-x}{c} \rfloor} \cdot P^{I_3}_{\exp}(-t(x)) - \exp\left(\frac{-x}{2\sigma^2}\right)}{\exp\left(\frac{x}{2\sigma^2}\right) + \exp\left(\frac{-x}{2\sigma^2}\right))}\right| \\
&\leq \frac{\left|2^{\lfloor \frac{x}{c} \rfloor} \cdot P^{I_3}_{\exp}(t(x)) - \exp\left(\frac{x}{2\sigma^2}\right)\right|}{\left|\exp\left(\frac{x}{2\sigma^2}\right) + \underbrace{\exp\left(\frac{-x}{2\sigma^2}\right)}_{>0}\right|} + \frac{\left|2^{\lfloor \frac{-x}{c} \rfloor} \cdot P^{I_3}_{\exp}(-t(x)) - \exp\left(\frac{-x}{2\sigma^2}\right)\right|}{\left|(\underbrace{\exp\left(\frac{x}{2\sigma^2}\right)}_{>0} + \exp\left(\frac{-x}{2\sigma^2}\right))\right|} \\
&= \left|\frac{2^{\lfloor \frac{x}{c} \rfloor} \cdot P^{I_3}_{\exp}(t(x)) - \exp\left(\frac{x}{2\sigma^2}\right)}{\exp\left(\frac{x}{2\sigma^2}\right)}\right| \\
&+ \left|\frac{2^{\lfloor \frac{-x}{c} \rfloor} \cdot P^{I_3}_{\exp}(-t(x)) - \exp\left(\frac{-x}{2\sigma^2}\right)}{\exp\left(\frac{-x}{2\sigma^2}\right)}\right| \\
&= \left|\frac{P^{I_3}_{\exp}(t(x))}{\exp\left(\frac{t(x)}{2\sigma^2}\right)} - 1\right| + \left|\frac{P^{I_3}_{\exp}(-t(x))}{\exp\left(\frac{-t(x)}{2\sigma^2}\right)} - 1\right| \\
&\leq \left\|\frac{P^{I_3}_{\exp} - \exp\left(\frac{\cdot}{2\sigma^2}\right)}{\exp\left(\frac{\cdot}{2\sigma^2}\right)}\right\|_\infty + \left\|\frac{P^{I_3}_{\exp} - \exp\left(\frac{\cdot}{2\sigma^2}\right)}{\exp\left(\frac{\cdot}{2\sigma^2}\right)}\right\|_\infty \\
&\leq 2^{-K} + 2^{-K} \\
&\leq 2^{-K+1}
\end{aligned}
$$

□

B.4 Values of Section 4 polynomials

Here are the values of the polynomials that approximate $\exp\left(\frac{\cdot}{2\sigma^2}\right)$ on I_3 for $K = 48$, $K = 40$ and $\cosh\left(\frac{\cdot}{2\sigma^2}\right)$ on I_2

$$
\begin{aligned}
(K=48) \quad P_{\exp}^{I_3}(x) \quad &= 1 \\
&+24941514431733 \cdot 2^{-61} \cdot x^1 \\
&+17680552620868 \cdot 2^{-78} \cdot x^2 \\
&+33422396152215 \cdot 2^{-97} \cdot x^3 \\
&+23692484119014 \cdot 2^{-115} \cdot x^4 \\
&+26872223790743 \cdot 2^{-134} \cdot x^5 \\
&+25398935908394 \cdot 2^{-153} \cdot x^6 \\
&+20576945247259 \cdot 2^{-172} \cdot x^7 \\
&+29170523303177 \cdot 2^{-192} \cdot x^8 \\
&+18337363552744 \cdot 2^{-211} \cdot x^9 \\
&+20154220626818 \cdot 2^{-231} \cdot x^{10} \\
&+31849608726558 \cdot 2^{-252} \cdot x^{11}
\end{aligned}
$$

$$
\begin{aligned}
(K=40) \quad P_{\exp}^{I_3}(x) \quad &= 1 \\
&+24356947687 \cdot 2^{-51} \cdot x^1 \\
&+17266164657 \cdot 2^{-68} \cdot x^2 \\
&+2039941109 \cdot 2^{-83} \cdot x^3 \\
&+5784292223 \cdot 2^{-103} \cdot x^4 \\
&+26241795903 \cdot 2^{-124} \cdot x^5 \\
&+24793374123 \cdot 2^{-143} \cdot x^6 \\
&+19985200707 \cdot 2^{-162} \cdot x^7 \\
&+27042670991 \cdot 2^{-182} \cdot x^8 \\
&+24859304099 \cdot 2^{-202} \cdot x^9
\end{aligned}
$$

$$
\begin{aligned}
P_{\cosh}^{I_2}(x) \quad &= 1 \\
&+579356348280174377 \cdot 2^{-93} \cdot x^{2} \\
&+776355318672508413 \cdot 2^{-130} \cdot x^{4} \\
&+915092737617992344722936430591 \cdot 2^{-208} \cdot x^{6} \\
&+477972884153784129 \cdot 2^{-206} \cdot x^{8} \\
&+683198250418101963 \cdot 2^{-246} \cdot x^{10} \\
&+732080158605593691080137637887 \cdot 2^{-326} \cdot x^{12} \\
&+941246584139471315 \cdot 2^{-327} \cdot x^{14} \\
&+1009039488665294857 \cdot 2^{-368} \cdot x^{16} \\
&+848404038966405199 \cdot 2^{-409} \cdot x^{18} \\
&+574427366621078729 \cdot 2^{-450} \cdot x^{20} \\
&+639792480494866645 \cdot 2^{-492} \cdot x^{22} \\
&+596411567633024933 \cdot 2^{-534} \cdot x^{24} \\
&+1038265388175552617133165772799 \cdot 2^{-617} \cdot x^{26} \\
&+642737561059887937 \cdot 2^{-619} \cdot x^{28} \\
&+835957399885546368823128489983 \cdot 2^{-702} \cdot x^{30} \\
&+788833826198999147 \cdot 2^{-705} \cdot x^{32} \\
&+180800091273276529 \cdot 2^{-746} \cdot x^{34} \\
&+296114798686759457 \cdot 2^{-790} \cdot x^{36} \\
&+940728566765727412999436632063 \cdot 2^{-875} \cdot x^{38} \\
&+150149332642775351 \cdot 2^{-876} \cdot x^{40} \\
&+572450757456113259692666388479 \cdot 2^{-962} \cdot x^{42} \\
&+748157035942843819 \cdot 2^{-965} \cdot x^{44} \\
&-2318931663498473299 \cdot 2^{-1010} \cdot x^{46} \\
&+11469920785630361661 \cdot 2^{-1054} \cdot x^{48} \\
&-720704975489956588178017804615 67 \cdot 2^{-1139} \cdot x^{50} \\
&+45346083844645413249 \cdot 2^{-1141} \cdot x^{52} \\
&-85738915116177191317 9 \cdot 2^{-1188} \cdot x^{54} \\
&+11238985938428339667 7 \cdot 2^{-1228} \cdot x^{56} \\
&-41435507672351356869 7 \cdot 2^{-1273} \cdot x^{58} \\
&+33706033762918354021 5 \cdot 2^{-1316} \cdot x^{60} \\
&-2658306635674898909421952386990 07 \cdot 2^{-1399} \cdot x^{62} \\
&+3363190191763256831121750071705 59 \cdot 2^{-1443} \cdot x^{64} \\
&-17047523299494673982 5 \cdot 2^{-1446} \cdot x^{66} \\
&+16718397753871067093 3 \cdot 2^{-1490} \cdot x^{68} \\
&-71970509923746855587 \cdot 2^{-1533} \cdot x^{70} \\
&+21696700177675978741 7 \cdot 2^{-1579} \cdot x^{72} \\
&-35646442335471927163 \cdot 2^{-1621} \cdot x^{74} \\
&+81298181032640091129 \cdot 2^{-1667} \cdot x^{76} \\
&-15983070798130234326 1 \cdot 2^{-1713} \cdot x^{78} \\
&+1476776720725659053681024801177 59 \cdot 2^{-1798} \cdot x^{80} \\
&-23863574853001513117 \cdot 2^{-1801} \cdot x^{82} \\
&+11312857910076672293 1 \cdot 2^{-1849} \cdot x^{84} \\
&-10967496476957148225 5 \cdot 2^{-1895} \cdot x^{86} \\
&+21170189159645535701 \cdot 2^{-1939} \cdot x^{88} \\
&-3128582308920305019 \cdot 2^{-1983} \cdot x^{90} \\
&+85067057155831645461 \cdot 2^{-2035} \cdot x^{92} \\
&-11559169938839276831 \cdot 2^{-2080} \cdot x^{94} \\
&+96586818007198473957 \cdot 2^{-2132} \cdot x^{96}
\end{aligned}
$$