# The LaZer Library: Lattice-Based Zero Knowledge and Succinct Proofs for Quantum-Safe Privacy

Vadim Lyubashevsky
IBM Research Europe
Zurich, Switzerland
vad@zurich.ibm.com

Gregor Seiler
IBM Research Europe
Zurich, Switzerland
gseiler@posteo.net

Patrick Steuer
IBM Research Europe
Zurich, Switzerland
ick@zurich.ibm.com

## Abstract

The hardness of lattice problems offers one of the most promising security foundations for quantum-safe cryptography. Basic schemes for public key encryption and digital signatures are already close to standardization at NIST and several other standardization bodies, and the research frontier has moved on to building primitives with more advanced privacy features. At the core of many such primitives are zero-knowledge proofs. In recent years, zero-knowledge proofs for (and using) lattice relations have seen a dramatic jump in efficiency and they currently provide arguably the shortest, and most computationally efficient, quantum-safe proofs for many scenarios. The main difficulty in using these proofs by non-experts (and experts!) is that they have a lot of moving parts and a lot of internal parameters depend on the particular instance that one is trying to prove.

Our main contribution is a library for zero-knowledge and succinct proofs which consists of efficient and flexible C code underneath a simple-to-use Python interface. Users without any background in lattice-based proofs should be able to specify the lattice relations and the norm bounds that they would like to prove and the library will automatically create a proof system, complete with the intrinsic parameters, using either the succinct proofs of LaBRADOR (Beullens and Seiler, Crypto 2023) or the linear-size, though smaller for certain application, proofs of Lyubashevsky et al. (Crypto 2022). The Python interface also allows for common operations used in lattice-based cryptography which will enable users to write and prototype their full protocols within the syntactically simple Python environment.

We showcase some of the library's usefulness by giving protocol implementations for blind signatures, anonymous credentials, the zero-knowledge proof needed in the recent Swoosh protocol (Gajland et al., Usenix 2024), proving knowledge of Kyber keys, and an aggregate signature scheme. Most of these are the most efficient, from a size, speed, and memory perspective, known quantum-safe instantiations.

Publication rights licensed to ACM. ACM acknowledges that this contribution was authored or co-authored by an employee, contractor or affiliate of a national government. As such, the Government retains a nonexclusive, royalty-free right to publish or reproduce this article, or to allow others to do so, for Government purposes only. Request permissions from owner/author(s).

*CCS '24, October 14–18, 2024, Salt Lake City, UT, USA*

ACM ISBN 979-8-4007-0636-3/24/10
https://doi.org/10.1145/3658644.3690330

## CCS Concepts

• **Security and privacy → Cryptography**; **Privacy-preserving protocols**.

## Keywords

Lattice Cryptography, Zero-Knowledge, Succinct Proofs, Implementation, Privacy, Quantum-Safe

**ACM Reference Format:**
Vadim Lyubashevsky, Gregor Seiler, and Patrick Steuer. 2024. The LaZer Library: Lattice-Based Zero Knowledge and Succinct Proofs for Quantum-Safe Privacy. In *Proceedings of the 2024 ACM SIGSAC Conference on Computer and Communications Security (CCS '24), October 14–18, 2024, Salt Lake City, UT, USA*. ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3658644.3690330

## 1 Introduction

With NIST having recently released standards for the next generation of public key cryptography [8, 12, 16, 31**? ? ?** ], the information security community is currently in the middle of a (mandated) transition to quantum-safe encryption and digital signature schemes. While the focus of the current transition is on encryption and digital signatures, it's fairly clear that in the near future one will have to transition other types of algorithms as well. With more advanced privacy-based cryptography gaining traction in applications, the cryptography research community has been working on creating more efficient *advanced* cryptographic schemes which should still remain secure against quantum attacks.

A research area that has seen a lot of recent activity in terms of moving from theory to practice (or at least to prototypes) is privacy-enhancing cryptography which includes protocols such as blind signatures, anonymous credentials, private digital currency, ring signatures, electronic voting, etc. Another related area which has gained importance due to the rise of distributed ledgers and blockchains is succinct proofs. A simple example of the latter is aggregate signatures, where the prover can combine a lot of signatures into a short proof that all the messages have been signed by the respective parties. Most of the above-mentioned protocols have zero-knowledge proofs, or succinct proofs, as the key component, and so a lot of research has focused on making this important building block faster and more compact.

Since three out of the four NIST standards have algebraic lattices as their underlying hardness assumption, it is natural to wonder whether lattices can be the most optimal quantum-safe foundation for zero-knowledge proofs as well. Algebraic lattices are extremely fast, allowing for faster encryption and signature schemes than even their classical counterparts based on discrete log and factoring. Since speed is often a limiting factor when it comes to succinct proofs, it makes the appeal of lattices interesting even apart from their presumed quantum-safety.

Research in lattice-based zero-knowledge proofs has, however, lagged behind other quantum-safe (and non-quantum-safe) approaches. For example, in 2018, Ligero [3] and Aurora [7] were already highly-promising succinct proof systems for arbitrary circuits,[1] whereas lattice-based proofs were still in their infancy, being extremely inefficient both theoretically and concretely. Proofs for the most basic cryptographically-interesting lattice relations required several megabytes [25, 26].

In the last few years, however, research on improving the state of the art progressed rapidly. Improvements of linear-sized proof systems [28] has resulted in advanced cryptographic schemes like blind signatures [11], anonymous credentials [4, 11], verifiable random functions[17, 18], ring signatures [27], etc. having outputs ranging between a dozen and a few dozen kilobytes. And the more recent LaBRADOR [9] succinct proof system promises proof sizes of around 60KB for arbitrarily-large witnesses, which is shorter than all other quantum-safe constructions.

The one area in which lattice-based proof systems have yet to catch up is usability in applications. While there are available software libraries that allow users to create proofs for various relations using other quantum-safe foundations (e.g. [14, 15, 20]), such a tool is non-existent for lattice-based proofs. There are papers in the literature implementing some version of the proofs, but they are all created from scratch and are very specific to the primitive being constructed in the paper. As of yet, we are not aware of any tool which helps protocol designers use lattice-based proofs as a black box.

## 1.1 The Challenge with Incorporating ZK Proofs

The main challenge with instantiating lattice-based zero knowledge proofs is that they consist of several steps and each of the steps may require different parametrization based on the concrete statement that is being proved. It is therefore up to the protocol designer to create the parameters that will optimize the run-time of the whole protocol. For example, the most basic proof system that one could generate for a lattice-based protocol is, for a public matrix $A$ and vector $t$, a proof of knowledge of a vector $\vec{s}$ such that

$$A\vec{s} = \vec{t} \bmod p \text{ such that } \|s\| \leq \beta \tag{1}$$

over a polynomial ring $R_p = \mathbb{Z}_p[X]/(X^d + 1)$.

Just being able to efficiently prove this equation, and some slight variations of it,[2] is enough to construct various advanced cryptographic primitives like blind signatures, anonymous credentials, and aggregated signatures. But even if we restrict ourselves to the most basic equation in (1), and only ask for linear-size (as opposed to succinct) ZK proofs, the solutions are already rather non-trivial. For example, the steps in [28] for proving (1) are as follows:

(1) Commit to $\vec{s}$ using a lattice-based commitment scheme over $R_q$ for $q \geq p$.
(2) Convert the equation $A\vec{s} = \vec{t} \bmod p$ into an equation $A\vec{s} + p\vec{v} = \vec{t}$ over $R_q$, with an additional secret polynomial vector $\vec{v}$ with small coefficients such that proving it implies proving (1) (i.e. one needs to make sure there is no overflow modulo $q$ and this equation is in essence being proved over $\mathbb{Z}$).
(3) For a random small challenge integer matrix $T$, commit to a masking vector $y$ and output $\vec{u} = T[\vec{s}; \vec{v}] + \vec{y}$. If $\vec{u}$ has small coefficients, then so do $\vec{s}, \vec{u}$ and there is no overflow modulo $q$.
(4) Prove the above set of linear equations by first creating linear combinations of them so that one only needs to prove a few equations (via the Schwartz-Zippel lemma) over $\mathbb{Z}$, and then convert these equations to ones over $\mathbb{Z}_q[X]/(X^d+1)$ masked by some auxiliary vectors to not leak any extra information.
(5) Combine the above linear equations (again applying the Schwartz-Zippel lemma) for proving $\vec{u} = T[\vec{s}; \vec{v}] + \vec{y}$ with the equation $A\vec{s} + p\vec{v} = \vec{t}$ together with the quadratic equation $\|\vec{s}\|^2 \bmod q \leq \beta^2$ into one quadratic equation.
(6) Apply a ZK proof of knowledge to prove the one quadratic equation.

The modulus $q$ that needs to be used in the above zero-knowledge proof needs to be either large-enough to accommodate the arithmetic being done modulo $p$ in the actual equation that we would like to prove, or be a multiple of $p$. Furthermore, in many scenarios, the equation in (1) is actually of the form $A_1 s_1 + \ldots + A_k s_k$ where $\|s_i\| \leq \beta_i$ for different $\beta_i$. Thus one may want to commit to the vectors $s_i$ in different ways – in particular, one may choose to put some $s_i$ into the "Ajtai" part of the ABDLOP commitment scheme from [28] and others into the "BDLOP" part. Furthermore, for the ones in the Ajtai part, one needs to decide what kind of rejection sampling strategy to use – the usual Gaussian one, a bimodal one for "one-time" commitments [27], or perhaps in the future not use rejection sampling at all by slightly increasing the standard deviation [24].[3]

The good news is that if one has freedom to work modulo any $q$, then picking the optimal parameters for obtaining the smallest proof sizes can be done via a script that tries many possibilities for the internal working of the zero-knowledge proof. Because our implementation of the linear-size proofs does not restrict us to any particular modulus $q$, we can incorporate such a script into LaZer.[4] Thus a protocol designer wishing to incorporate zero-knowledge proofs proving relations of the form (1), can simply write down the public parameters $A$ and $t$ and the bounds on the various parts of the secret vector, and LaZer will set the parameters that the ZK proof will need to use and output them to a header file which will be read at compile-time. The designer can then simply invoke the prover and verifier procedure of the ZK proof without needing to understand anything about the internals of these proofs.

The LaBRADOR succinct proofs are, in our opinion, even more complicated to implement and use than the linear size proofs. The scheme consists of many rounds, with each round needing its own parameter setting. Furthermore, the scheme from [9] only supports specific relations related to R1CS proofs, whereas we would like to use is to support proofs of multiple relations as in (1). The main difference is that we need to prove an $\ell_2$-bound on the witness of

[1] Pairing-based schemes were already almost fully-mature by then
[2] For example, prove different bounds on various parts of the witness $s$.
[3] Since this paper is relatively new, we have not yet incorporated it into our work.
[4] The succinct proof allows a choice of several moduli.

each equation, rather than just proving one bound for the entire proof system.[5] This requires a wrapper around the LaBRADOR protocol and is described in Section 4.

**The main goal of our library is so that the protocol designer need not understand anything written in this section!**

## 1.2 Our Contribution

As alluded to above, ease-of-use of incorporating lattice-based proof systems is the primary objective of the LaZer library. In addition to this main goal, we also want the library to be efficient and amenable to future optimizations. The library consists of four layers (see Figure 1). At the bottom layer are the algebraic operations that are needed for ZK proofs and lattice cryptography in general. In order to have the necessary flexibility, we need to support operations over arbitrary moduli, and therefore we cannot specifically optimize for a particular polynomial ring. Nevertheless, one can come fairly close to optimal by optimizing for operations over small moduli and then using the Chinese remainder theorem to perform operations over a modulus that's their product. Working with such small moduli is particularly efficient when one further optimizes using AVX2 (or AVX512) instructions. Some experimental evidence in [13] showed that working over moduli that do not natively support the Number Theoretic Transform (which would lead to optimal implementations) are actually quite efficient using the Chinese Remainder Theorem approach. We give more details about this layer in Section 3.

The second layer of the library is the ZK / succinct proof layer. The ZK protocol of [28], which we implemented, is most compact when implemented over polynomial rings of small degree, such as $\mathbb{Z}_q[X]/(X^{64}+1)$. We furthermore expect that these zero-knowledge proofs will be the most expensive part of many protocols. We therefore optimized the expensive operations (such as multiplication) specifically over rings of this degree, which allowed to have a computationally-efficient implementation of the ZK proof from [28]. This ZK proof can then be used to prove relations from (1) over any ring with degree $\geq 64$.[6] More details are in Section 3.

The second layer of the library also includes the LaBRADOR succinct proof system. The original purpose of LaBRADOR was to be able to provide succinct proofs for R1CS, but in applications to lattice protocols, we believe that the most useful application will be proving (multiple) statements of the form (1). In order to be able to prove such statements, we needed to implement a wrapper algorithm on top, which is in the third layer of LaZer. Because the proof size of LaBRADOR is succinct (asymptotically $O(\log \log n)$ in the witness length), we did not need to optimize its parameters to allow arbitrary moduli. In particular, we allow the user to pick from one of several moduli to work with, and then adapt the statement to this modulus. More details about this layer are in Section 4.

The third layer of the library consists of useful common tools that can be built on top of the layer below (such as proofs of (1)) or tools that are independent of proof systems, such as the trapdoor sampling algorithm such that when given a trapdoor for a (pseudo)-random matrix $A$, one is able to produce vectors $\vec{s}$ with small norms that satisfy $A\vec{s} = \vec{t} \bmod p$ for arbitrary $p$. A very compact such sampler is exactly the one used for the recently-chosen NIST standard signature FALCON [31]. We therefore added this trapdoor sampling functionality to our library. This pre-image sampler is used in our sample constructions of blind signatures, anonymous credentials, and aggregate signatures.

The last layer is the main layer that we expect most users will interact with. It is the Python layer which, using CFFI [32], wraps the algebraic C library and the zero knowledge functionality of the lower layers. The result allows the user to set up and call all the proof functions (whether from [28] or [9]) from Python. In addition to the functions related to proving and verifying statements, the user also has access to functions that allow for creating, manipulating, and operating on vectors and matrices over polynomial rings. We believe that the protocol designer should be able to write his entire protocol in our Python layer and perhaps only add the performance-critical tools to the layer below written in C. Because Python is an interpreted language, we tried to make this layer as "thin" as possible so as to mitigate the performance hit. For example, writing a somewhat involved anonymous credential protocol in the Python layer only resulted in about a 30% overhead.

## 1.3 Protocol Examples and Comparison to Other Works.

As far as we are aware, there have not been any general libraries that support either efficient lattice-based linear or succinct proof systems. There are of course existing libraries for fast polynomial arithmetic (e.g. [10, 30, 33]), and we do make use of the HEXL library [10] as a subroutine for polynomial multiplications and for the linear-size proof system. We cannot exclusively rely on HEXL for polynomial multiplication because it only works for special rings, whereas we would like our linear-size proofs to work for *any* ring of the form $\mathbb{Z}_q[X]/(X^d+1)$ where $d$ is a power of 2. For fast polynomial multiplication over such a ring, we use the CRT to work over a ring $\mathbb{Z}_{p_1\cdots p_k}[X]/(X^d+1)$ such that the product $p_1\cdots p_k > dq^2$ is large enough so that no carries occur when operations with coefficients at most $q$ are performed in this ring. The primes $p_i$ are picked such that one can perform efficient NTT multiplications over the rings $\mathbb{Z}_{p_i}[X]/(X^d+1)$. The HEXL library is used to do multiplications over these latter rings. For succinct proofs, we only work over very specific primes and created specifically-tailored operations over these primes optimized for the AVX-512 instruction set.

Together with the LaZer library, we provide fully-documented examples of protocols written in Python. We hope that in addition to being interesting in of themselves, they will also serve as guides for how one would use the library to construct new protocols. The example protocols include a proof of knowledge of a Kyber KEM secret key and a proof of knowledge of the secret key for the Swoosh NIKE [21]. These two protocols demonstrate the extremes of the moduli sizes that one may encounter in practice. While Kyber

[5]For R1CS systems, it's enough to prove that all the coefficient vectors are 0/1, which can be done by proving one quadratic equation $\langle \vec{w}, \vec{1} - \vec{w} \rangle = 0$ over the integers.
[6]Currently, the algebraic operations over rings $\mathbb{Z}_q[X]/(X^d+1)$ with $d > 64$ use automorphisms to map down to vectors of polynomials over the ring $\mathbb{Z}_q[X]/(X^{64}+1)$ and then perform operations over this ring. This, as mentioned, is not the optimal way to perform such operations, and will be sped up in the future. Nevertheless, because the runtime is dominated by the ZK proof, this currently does not make too much practical difference.

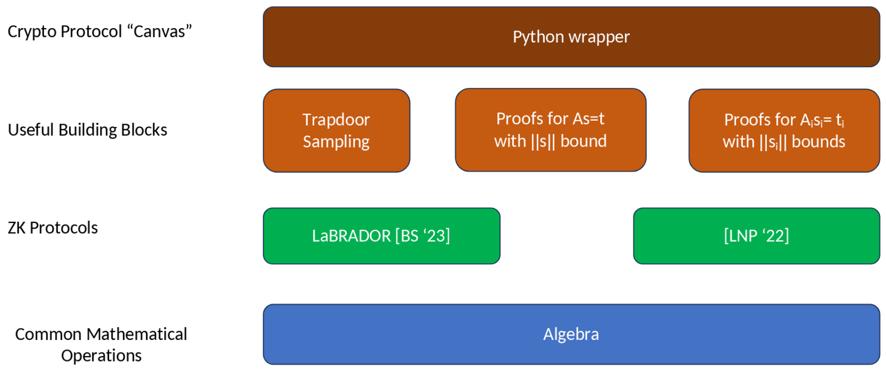

**Figure 1: The Layers of the LaZer library. All but the top Python layer are written in C. The LaBRADOR part requires the AVX-512 instruction set.**

| Scheme | [4] | [11] (with LaZer) |
|---|---|---|
| Assumption | Mod-LWE / Mod-SIS | Mod-LWE / Mod-ISIS$_f$ |
| Organization PK | 47.5KB | 1 KB |
| Blinded Sig | 6.8 KB | 1.3 KB |
| Credential | 79.6 KB | 29 KB |
| Issuance Time | 0.383 sec | 0.117 sec |
| Showing Time | 0.504 sec | 0.198 sec |

**Table 1: Anonymous Credential Scheme from [4] and one from [11]. The latter is implemented using the LaZer library.**

| | [23] | FALCON + LaZer | FALCON + LaZer |
|---|---|---|---|
| Signatures | 1024 | 1024 | 100, 000 |
| Prover Time | 25.7 sec | 0.6 sec | 65 sec |
| Prover RAM | 9.5 GB | 200 MB | 25.5 GB |
| Sig. Size | 165 KB | 73.5 KB | 72.3 KB |

**Table 2: Performance of a STARK-based aggregate signature from [23] for 1024 signatures and the implementation using the FALCON signature and LaZer for 1024 and 100,000 signatures.**

has a short 12-bit modulus, the Swoosh moduli are over 200 bits. We further use our linear-size proof system in two full-fledged protocols – a blind signature scheme and an anonymous credential scheme from [11]. We also illustrate the workings of the succinct proof system built on LaBRADOR to give an example of a generic instantiation of an aggregate signature scheme that combines a hash-and-sign signature scheme (in our case we use FALCON [31]) and a succinct proof system. We describe these schemes in more detail in Section 6, and below give a brief comparison of our protocol instantiation of an anonymous credential scheme and a lattice-based aggregate signature scheme to similar quantum-safe protocols in the literature.

In [4], the authors construct and implement an anonymous credential scheme based on the hardness of standard lattice problems using the zero-knowledge proof from [28]. As an example of an application of LaZer, we implement (using the Python layer) the anonymous credential scheme from [11]. The run-times and output sizes of both protocols are given in Table 1. It is not really possible to make an apples-to-apples comparison between the running time of the schemes – the schemes are somewhat different, the computers on which they are not exactly the same, the scheme in [4] was in C using AVX-2 instructions, while ours is a combination of C with AVX-512 combined with Python, etc. The main takeaway from Table 1 is that one can create a simple implementation in Python of a somewhat-involved lattice-based protocol using the LaZer library, and the performance is comparable to that of implementations that were specifically protocol-tailored. We hope that the efficiency and relative ease of use of our Python layer therefore encourages more researchers to write their schemes using LaZer.

In Table 2, we give a comparison between our generic aggregate signature scheme implementation which combines the FALCON hash-and-sign signature scheme and a succinct proof system to that of a *one-time* aggregate signature scheme created from Winternitz signatures and a STARK-based succinct proof. [7] The run-time comparison is again not apples-to-apples. The signature scheme in [23] is a one-time signature scheme, whereas FALCON is a general signature scheme. Furthermore, the implementation of [23] used multi-threading on an 8-core processor, whereas our implementation is single-thread, but did use the AVX-512 instruction set. Nevertheless, it appears that the LaZer implementation has at least an order of magnitude speed advantage and perhaps, even more importantly, has a much lower RAM requirement and shorter signature size than the STARK-based implementation.

The lattice-based multi-signature scheme Chipmunk [19] can be seen as a conversion of a lattice-based one-time signature into a *synchronous* multi-signature where the restriction is that all time is broken down into epochs, each signer can sign only once per

[7] [1] also discusses a lattice-based aggregate signature using FALCON and a lattice-based succinct proof, and proposes a slightly different version of LaBRADOR. But there is no implementation, and so a comparison cannot be made.

epoch (and the message is the same for all signers), and one can only aggregate signatures from the same epoch. For 1024 signatures, the aggregate signature size was 118 KB, and the Rust run-time (which used 24 threads) was essentially the same as our C-based (single-threaded, but utilizing AVX-512 instructions) scheme. Our instantiation of FALCON + succinct proof is more general in that there is no same-message restriction and no additional synchronicity requirement, and yet it still has shorter signatures and a similar prover run-time. The advantage of Chipmunk is that the verification time is much smaller, whereas ours is only about 40% faster than that of the prover. As far as we are aware, the aggregate signature construction in this paper has the smallest signature length and prover running time of all quantum-safe constructions, and we hope that this example encourages more generic constructions that use (lattice-based) succinct proofs as the core building block.

### 1.4 Future Work

The LaZer library in its current form already allows for fairly efficient constructions of many privacy-based primitives, but there are still numerous improvements that can be made to improve the performance and expand the functionality of the library. There are several improvements that one can make to the implementation of the zero-knowledge proof system from [28]. There has been interesting recent work on cheaply removing the need for rejection sampling when masking the randomness of a commitment scheme [24]. This technique could be used to speed up the scheme and slightly reduce the output length of the linear-size ZK proofs. There are also some internal mechanics of the proof that can be improved. One of the more expensive parts of the proof is the approximate range proof that uses a modular version of the Johnson-Lindenstrauss lemma [22] and the protocol description in [28] calls for two applications of it. Reducing it to one application should result in a noticeable performance improvement.

Lattice cryptography is known to be highly parallelizable, and practical uses of the LaZer library should eventually take advantage of this feature. It would therefore be useful to add the ability to support multi-threaded processors to all layers of the library. Exploring the implementation of the library on GPU's can also result in a significant performance improvement.

Currently, the succinct proof system only supports succinctness, but not yet zero-knowledge. While this is already useful for several applications, adding zero-knowledge (which was already described in [9]) would allow the applicability of the proof system to be expanded to domains like voting systems and privacy-based digital currency systems where privacy is important.

The linear-size proof system currently supports native operations over the ring $\mathbb{Z}_p[X]/(X^d + 1)$ for $d = 64$ and all the internal ZK components (such as the commitment scheme) is over this ring. One can increase efficiency by using larger rings for the commitment scheme, as this requires fewer polynomials and thus a lot fewer multiplications. This, together with natively supporting multiplication for rings $\mathbb{Z}_p[X]/(X^d + 1)$ where $d \neq 64$ should result in a noticeable performance improvement for our linear-size proof system.

There are also many "intermediate" layer useful building blocks (see Figure 1) that one could construct on top of the proof system layer. For example, one could build ring signatures by implementing the one-out-of many proof system from [27] or, when aggregating the equations in (1), one could aggregate not just the signatures, but also the public keys. This requires some extra wrapper layers around LaBRADOR, but would then result in a rather efficient lattice-based threshold signature scheme without requiring any interaction during signing.

## 2 Obtaining and building LaZer and its documentation

The LaZer library can be build on Linux AMD64 systems with the following software installed: gcc or clang, make, python 3.10 or newer (including the cffi module) and the gmp and mpfr development packages. The LaZer code generator requires an installation of the SageMath computer algebra system version 10.2 or newer.

The LaBRADOR proof system is currently implemented in separate libraries (one per supported prime). While the LaZer library benefits from AVX512 instruction set extensions, LaBRADOR requires them (plus a couple of features only found in newer compiler versions). That is why it is possible to build the rest of LaZer separately from LaBRADOR such that LaZer can also be build on systems lacking those prerequisites.

A sphinx installation[8] (including the sphinxcontrib-bibtex package) is required to build the documentation.

Clone the repository from GitHub, change to the lazer directory and clone the submodules

```
git clone https://github.com/lazer-crypto/lazer
cd lazer
git submodule init
git submodule update
```

In the following, all directory changes are relative to the lazer directory. Now either build just the LaZer library ..

```
make
```

.. or both the LaZer and the LaBRADOR library

```
make all
```

Change to the python directory and build the python modules

```
cd python
make
```

The python directory contains a couple of examples, each in its own subdirectory. The demo subdirectory has an implementation of some example instance of 1. The anon_cred subdirectory has an anonymous credentials implementation. The blindsig subdirectory has a blind signature implementation. The swoosh subdirectory has an implementation of proving well-formedness of a swoosh public key. The kyber1024 subdirectory has an implementation of proving knowledge of a Kyber1024 secret key. Let <name> be one of these subdirectories, build and run the corresponding example

[8]https://www.sphinx-doc.org/en/master/usage/installation.html

```
cd <name>
make
python3 <name>.py
```

Optionally, change to the docs directory and build the html documentation

```
cd docs
make html
```

Use any browser (e.g. firefox) to view the documentation

```
firefox build/html/index.html
```

## 3 The C Linear Proof Layer

We now describe in detail how to prove lattice linear relations with norms using the linear-sized proof system. For simplicity, we present only the case where there is a single euclidean norm bound on the witness. Note that LaZer allows to prove multiple norm bounds on the elements of some arbitrary partition of the witness or to prove that some elements have binary coefficients only. These features are needed to implement the protocols we are interested in. However, even the simplest case highlights some of the challenges encountered when setting parameters for these proof systems. The last section provides insights into LaZer implementation details.

### 3.1 Linear-sized proofs for lattice relations

In this section, we show how to prove knowledge of a secret short vector $(\mathbf{s}, \mathbf{e}) \in \mathcal{R}_{p,d'}^m \times \mathcal{R}_{p,d'}^n$ such that

$$\mathbf{t} = \mathbf{A}\mathbf{s} + \mathbf{e} \tag{2}$$

over $\mathcal{R}_{p,d'}$ and

$$\left\| \begin{pmatrix} \mathbf{s} \\ \mathbf{e} \end{pmatrix} \right\| \leq B \tag{3}$$

for public $\mathbf{A} \in \mathcal{R}_{p,d'}^{n\times m}$, $\mathbf{t} \in \mathcal{R}_{p,d'}^n$ and $B$.

In the linear-sized proof system, we can prove knowledge of above secret as follows: First commit to $\mathbf{s}$ using the ABDLOP commitment [28].

If $B$ is a Euclidean norm bound, do an exact Euclidean norm proof for the following linear functions of $\mathbf{s}$ over $\mathcal{R}_{p,d'}$:

$$\left\| \begin{bmatrix} \mathbf{s} \\ \mathbf{e} \end{bmatrix} \right\|_2 = \left\| \begin{bmatrix} \mathbf{1}_m \\ -\mathbf{A} \end{bmatrix} \mathbf{s} + \begin{bmatrix} \mathbf{0} \\ \mathbf{t} \end{bmatrix} \right\|_2 \leq B$$

This implies a proof of 2 and 3.

Applying these strategies directly works only if the proof system modulus $q$ can be set to $p$ and the degree $d'$ is 64, that is, it requires the statement's parameters to be proof system friendly in the sense that they exactly correspond to the rings over which the zero-knowledge proof operates. In most cases, this is unfortunately not the case, and usually one would like to prove lattice relations over rings which were set up with optimality of the underlying construction in mind, whereas the zero-knowledge proof system works over rings which optimize the zero-knowledge proof.

For example, many lattice-based schemes use moduli $p = 2l + 1 \mod 4l$ with $l$ close or equal to the ring degree $d'$ such that $X^{d'} + 1$ splits into many ($l$) irreducible factors of low degree ($\frac{d'}{l}$), while the linear-sized proof system [28] uses moduli $q = 5 \mod 8$ (i.e., $l = 2$) to guarantee the invertability of its challenges. In the following, we show how to prove statements modulo any $p$.

If the statement modulus $p$ is not a suitable proof system modulus, we consider a proof of knowledge for $(\mathbf{s}, \mathbf{e}, \mathbf{v}) \in \mathcal{R}_{d'}^m \times \mathcal{R}_{d'}^n \times \mathcal{R}_{d'}^n$ such that 3 holds and

$$\mathbf{t} = \mathbf{A}\mathbf{s} + \mathbf{e} + p\mathbf{v} \tag{4}$$

holds over $\mathcal{R}_{d'}$.

Clearly, such a proof implies a proof for the original statement. Then again, this proof is implied by a proof of knowledge of $(\mathbf{s}, \mathbf{e}, \mathbf{v})$ such that 3 holds and 4 holds over $\mathcal{R}_{q,d'}$ if $q$ is large enough such that there is no wrap-around modulo $q$. For the latter to hold, we need to add an approximate range proof (ARP) for $\mathbf{v}$.[9]

A lower bound for the required proof system modulus can be determined as follows. For $P = \frac{p-1}{2}$, let $S \leq P$ and $E \leq P$ be bounds such that $\|s\|_\infty \leq S$ and $\|e\|_\infty \leq E$.[10] We can bound $p\mathbf{v}$ by $\|p\mathbf{v}\|_\infty = \|\mathbf{t} - \mathbf{A}\mathbf{s} - \mathbf{e}\|_\infty \leq \|\mathbf{t}\|_\infty + \|\mathbf{A}\mathbf{s}\|_\infty + \|\mathbf{e}\|_\infty \leq P + mPd'S + E$. So we know that

$$\|\mathbf{v}\|_\infty \leq \frac{1}{p}(P + mPd'S + E) \tag{5}$$

but can only prove $\|\mathbf{v}\|_\infty \leq \frac{\psi}{p}(P + mPd'S + E)$, where $\psi$ is the slack of the ARP. Then we have $\|\mathbf{t} + \mathbf{A}\mathbf{s} + \mathbf{e} + p\mathbf{v}\|_\infty \leq \|\mathbf{t}\|_\infty + \|\mathbf{A}\mathbf{s}\|_\infty + \|\mathbf{e}\|_\infty + \|p\mathbf{v}\|_\infty \leq (\psi + 1)(P + mPd'S + E)$. Set $q$ such that $\frac{q-1}{2} \geq (\psi + 1)(P + mPd'S + E)$.

We can then commit to $(\mathbf{s}, \mathbf{e}, \mathbf{v})$, prove the linear (in $(\mathbf{s}, \mathbf{e}, \mathbf{v})$) equation 4 over $\mathcal{R}_{q,d'}$, do a norm proof for 3 and do the approximate range proof for 5.

Doing the ARP adds 4 "full-sized" polynomials to the proof size, for our proof system ring degree of 64, and 256 "short" polynomials. Since $\mathbf{s}$ and $\mathbf{e}$ are short they are put in the "Ajtai" part of the ABDLOP commitment.[11].

From the above computation, $\|\mathbf{v}\|_\infty \leq V$ for $V = \frac{1}{p}(P + mPd'S + E)$. Putting $\mathbf{v}$ in the BDLOP part of the ABDLOP commitment increases the commitment size by $n$ "full-sized" polynomials, i.e., by $nd'\lceil \log(q) \rceil$ bits.

If $\|(\mathbf{s}, \mathbf{e})\|_2 \leq B$, then putting $\mathbf{v}$ into the Ajtai part increases this bound: $\|(\mathbf{s}, \mathbf{e}, \mathbf{v})\|_2 \leq \sqrt{B^2 + nV^2}$. While this adds only $n$ "short" polynomials (the additional masked openings) to the proof (and makes those slightly larger), the main disadvantage is that this increases the bound on the extracted MSIS solution. This may require an increase in the MSIS problem dimension by $x$ which increases the commitment size by $x$ compressed "full-sized" polynomials and the corresponding hints. Moreover, it may require to increase the modulus $q$, which negatively affects the proof size.

[9]An approximate range proof [6, 22] is a "cheap" proof that proves that the coefficients of the commitment are in a certain loose range. They are mostly used in zero-knowledge proofs to prove that no wraparound occurs.
[10]We can always bound $\|s\|_\infty$ and $\|e\|_\infty$ by $B$: In case B is an $l_2$-norm bound, that is just the corresponding naive infinity norm bound.
[11]The ABDLOP commitment scheme in [28] contains slots for committing to elements with small norms (i.e. the "Ajtai" part, named so because it is related to the original commitment scheme implicit in [2]) and slots for committing to arbitrary-sized polynomials (named the BDLOP part because it is related to the BDLOP commitment scheme [5])

Therefore, the best strategy is to avoid committing to $\mathbf{v}$ altogether, and make the proof of the linear equation 4 over $\mathcal{R}_{q,d'}$ implicit in the ARP. Notice that $\mathbf{v} = \frac{\mathbf{t}-\mathbf{A}\mathbf{s}-\mathbf{e}}{p}$ over $\mathcal{R}_{d'}$ and $\mathbf{v}$ can also be computed over $\mathcal{R}_{q,d'}$ if $q$ is chosen coprime to $p$. Instead of 5, we do an ARP for

$$\left\| \begin{bmatrix} -p^{-1}\mathbf{A} & -p^{-1}\mathbf{1}_n \end{bmatrix} \begin{bmatrix} \mathbf{s} \\ \mathbf{e} \end{bmatrix} + \begin{bmatrix} p^{-1}\mathbf{t} \end{bmatrix} \right\|_\infty \leq \frac{1}{p}(P + mPd'S + E) \quad (6)$$

This implies a proof for 5 and for 4 over $\mathcal{R}_{q,d'}$ which (for appropriate $q$) implies that 4 also holds over $\mathcal{R}_{d'}$. Together with the norm proof for 3, this implies a proof for the original statement.

We now consider the case, where the degree $d'$ of the statement's ring may be greater than the degree $d = 64$ of the rings the proof system naturally works with. In this case, a ring isomorphism can be applied to convert the statement into an equivalent statement over the smaller ring.

Both strategies (lifting the modulus and reducing the ring degree) are easily combined such that we can prove arbitrary statements of above form.

However, in the more general case we want to prove multiple norm bounds on different parts of the witness or prove that other parts of the witness have binary coefficients only. In this case, parameter setting becomes more involved. The next section describes how the LaZer code generator can be used to set parameters from a simple input specification such that

- The underlying MLWE and MSIS problems are hard.
- Proof size is optimized heuristically considering trade-offs between putting witness parts in the Ajtai or BDLOP part of the commitment.
- The right proof system modulus is determined.
- The input statement is converted into an equivalent statement over the proof system ring.

## 3.2 Code generation

When we need to prove different statements of the form (1) as a subroutine of some protocol, note that norm bounds on parts of the witness are usually fixed across instances. From an implementation point of view, this means that we may fix them at compile-time, together with the public parameters: the polynomial ring, the module dimension and the partition of the witness.

In fact, specifying the above inputs (polynomial ring, module rank, partition of the witness and corresponding norm bounds) is all one needs to do to prove (1) using LaZer.

Such a specification is then used as an input to a LaZer's code generator, which heuristically optimizes proof size and runtime and outputs the resulting parameters encapsulated in a constant C structure, that can be referenced by the prover and the verifier. More specifically, the specification is just a python script with variables corresponding to the required inputs.

For example the specification for proving 1 over $R_p = \mathbb{Z}_{3329}[X]/(X^{256}+1)$ for $\beta = 1.2\sqrt{2048}$ and $A \in R_p^{4\times 8}$ (that is, proving knowledge of a Kyber1024 secret key) would look as follows:

```python
from math import sqrt
vname = "params"

deg   = 256
mod   = 3329
dim   = (4,8)

wpart = [ [list(range(8))] ]
wl2   = [ 1.2*sqrt(2048) ]
wbin  = [ 0              ]
```

The string `vname` is some variable name for the generated C structure (and thus must be a valid C identifier). The partition of the witness is encoded as a list `wpart`. The $n$-th element of the partition is itself encoded as as list of indices of the witness vector. In the example, the partition has just a single element i.e., the whole witness vector (indices 0 to 7). The euclidean norm bound corresponding to the $n$-th part of the witness is the $n$-th element of the list `wl2`. The $n$-th element of the list `wbin` is a boolean which, if set to 1, bounds the $n$-th part of the witness to have binary coefficients only. The latter feature is not needed in our example.

## 3.3 Using generated code from an application

Below code implements generic prover and verifier functions for 1 in LaZer. By "generic" we mean that, assuming the generated code (a constant structure named "params" in our example) is in `params.h`, the contents of `params.h` could be replaced by any other output from the code generator, corresponding to a possibly totally different instance.

```c
#include "lazer.h"
#include "params.h"

void prover(uint8_t *proof, polyvec_t s, polymat_t A,
     polyvec_t t, const uint8_t pp[32])
{
  lin_prover_state_t prover;

  lin_prover_init (prover, pp, params);

  lin_prover_set_statement (prover, A, t);
  lin_prover_set_witness (prover, s);
  lin_prover_prove (prover, proof, NULL, NULL);

  lin_prover_clear (prover);
}

int verifier(const uint8_t *proof, polymat_t A,
     polyvec_t t, const uint8_t pp[32])
{
  lin_verifier_state_t verifier;

  lin_verifier_init (verifier, pp, params);

  lin_verifier_set_statement (verifier, A, t);
  int accept = lin_verifier_verify (verifier, proof,
      NULL);

  lin_verifier_clear (verifier);
  return accept;
}
```

The `NULL` parameter in the verify function and first `NULL` parameter in the prove function can instead be used to avoid out-of-bounds reads or writes of the proof buffer. We ignore this here for simplicity and assume the size of the proof buffer is an upper bound of the proof length. The second `NULL` parameter in the prove function can instead be used to pass internal coins to the prover (instead of tossing them interally).

## 3.4 Implementation details

The code generator is approximately 600 lines of sage code. The LWE estimator[12] is called as a subroutine. From the input specification, it sets the proof system modulus and computes an equivalent statement over the proof system ring. It starts by putting the whole witness in the Ajtai part of the commitment, setting all other parameters and computing the resulting proof size. Then it continues to move the biggest witness part to the BDLOP part of the commitment and comparing the resulting proof size. This process is iterated as long as the proof size decreases. While this process only is guaranteed to find a local optimum, the best possible proof sizes were obtained in our real-world examples.

The C library is more than 10000 lines of code. The HEXL library's NTT implementation and the FALCON reference implementation's trapdoor sampling algorithm are used as subroutines. Since the NTT is specialized to fully-splitting rings, we implemented polynomial multiplications in the proof system ring via the explicit Chinese-remainder theorem (ECRT). The base arithmetic layer also implements samplers (uniform, gaussian, binomial), an encoder for those distributions and rejection sampling algorithms.

The combined toolchain of the code generator and the library aims at fixing the maximum amount of quantities possible at compile time.

## 4 The C LaBRADOR Proof Layer

Our starting point is an implementation of the LaBRADOR proof system from an overlapping anonymous submission aimed towards proving polynomial evaluations that arise in the arithmetization of arbitrary circuits. For this work and its focus on proving lattice schemes we have implemented a new frontend for LaBRADOR that efficiently supports proving relations that are most suitable for this setting. The new frontend builds on top of the carefully AVX512-optimized library for polynomial arithmetic and Johnson-Lindenstrauss projection from the LaBRADOR implementation.

Recall that LaBRADOR originally supports the so-called principle relation where the witnesses consist of $r$ vectors $\boldsymbol{s}_i \in \mathcal{R}_q^n$ that fulfill dot-product constraints of the form

$$\sum_{i,j} a_{ij}\langle \boldsymbol{s}_i, \boldsymbol{s}_j\rangle + \sum_i \langle \boldsymbol{\varphi}_i, \boldsymbol{s}_i\rangle + b = 0$$

that optionally can only be required to hold in the constant coefficient, together with one global norm constraint $\sum_i \|\boldsymbol{s}_i\|^2 \leq \beta^2$. In the code the LaBRADOR implementation is actually written for another relation that is optimized for internal purposes and allows for more efficient recursion. Then there is a frontend, called the Chihuahua frontend, that reduces the principle relation to the internal LaBRADOR relation.

[12]https://bitbucket.org/malb/lwe-estimator/src/master/

The problem with the principle relation for our purposes is twofold. First, consider our example application of signature aggregation. Here the witness consists of many constant-length vectors that represent the individual signatures and one needs to prove tight $\ell_2$-norm bounds on each individual vector, which means that the global norm bound in the principle relation does not suffice. Second, proving the principle relation is not efficient when the multiplicity $r$ is large, which is the case in an aggregate signature. The reasons is that Chihuahua computes $O(r^2)$ garbage polynomials that have a computational cost of $O(n)$ each, and they become part of the witness for the following LaBRADOR proof layer. Our new frontend is called Dachshund and supports a relation which differs from the principal relation in that statements have individual norm bounds for every vector, $\|\boldsymbol{s}_i\|^2 \leq \beta_i^2$. Moreover, Dachshund is efficient even for very large multiplicities $r$.

*Proving individual norm bounds.* For proving the individual norm bounds, Dachshund expands the witness and produces dot-product constraints in the resulting principle relation that prove them together with only one remaining global norm bound. It essentially uses standard techniques for this. Concretely, note that the constant coefficient of the polynomial $\langle \boldsymbol{s}_i, \sigma_{-1}(\boldsymbol{s}_i)\rangle$ is equal to the squared $\ell_2$-norm $\|\boldsymbol{s}\|^2$. So if the global norm bound on all the $\boldsymbol{s}_i$ implies that this can not wrap-around modulo $q$, it suffices to prove that $h_i = \beta_i^2 - \|\boldsymbol{s}_i\|^2 \bmod {}^{\pm}q$ is positive which in turn can be proven by expanding it in binary in order to prove that it is below $q/2$. So let $h_i$ be the polynomial whose coefficients are given by the binary expansion of $h_i$. Then we get the dot-product constraint

$$\mathsf{cnst}(\langle \boldsymbol{s}_i, \boldsymbol{s}_i'\rangle - \beta_i^2 - h_i g) = 0,$$

where $\sigma_{-1}(g) = 1 + 2X + \cdots + 2^{\lceil \log q\rceil - 2}X^{\lceil \log q\rceil - 2}$ is a gadget polynomial and $\boldsymbol{s}_i' = \sigma_{-1}(\boldsymbol{s}_i)$. The $\boldsymbol{s}_i'$ and $h_i$ become part of the witness. To prove that $\boldsymbol{s}_i'$ is correctly conjugated constraints of the form $\mathsf{cnst}(\langle \sigma_{-1}(\boldsymbol{\alpha}_i), \boldsymbol{s}_i\rangle - \langle \boldsymbol{\alpha}_i, \boldsymbol{s}_i'\rangle) = 0$ for uniformly random challenge vectors $\boldsymbol{\alpha}_i \in \mathcal{R}_q^n$ are used, and the fact that the $h_i$ are binary follows from $\mathsf{cnst}(h_i\sigma_{-1}(h_i)) = 0$.

*Rebalancing into fewer vectors of higher rank.* While LaBRADOR allows that vectors are split further into even more vectors it is not possible to join them and one needs a more elaborate way of rebalancing the vectors. The reason is that the quadratic garbage polynomials $\langle \boldsymbol{s}_i, \boldsymbol{s}_j\rangle$ transform nicely under splitting in that also the garbage polynomials split and the original garbage polynomials can be recovered by adding them together. On the other hand joining vectors also joins garbage polynomials and it is not possible anymore to recover the original ones. To circumvent this problem we perform committing to the witness in two phases. First we commit to the unconjugated vectors $\boldsymbol{s}_i$. Then we get challenge polynomials $c_i \in C$, multiply the conjugated $\boldsymbol{s}_i'$ by the corresponding $c_i$, and commit to the vectors $c_i\boldsymbol{s}_i'$. Now, joining all the $\boldsymbol{s}_i$ and all the $c_i\boldsymbol{s}_i'$ into long vectors $\boldsymbol{s} = \boldsymbol{s}_1 \parallel \cdots \parallel \boldsymbol{s}_r$ and $\boldsymbol{s}' = c_1\boldsymbol{s}_1' \parallel \cdots \parallel c_r\boldsymbol{s}_r'$, respectively, results in the garbage polynomial

$$\langle \boldsymbol{s}, \boldsymbol{s}'\rangle = \sum_i c_i\langle \boldsymbol{s}_i, \boldsymbol{s}_i'\rangle.$$

Here the garbage polynomials that we need for the quadratic norm-check constraints are all separated by the challenge polynomials $c_i$.

There is one issue remaining. Namely, in order to lift the quadratic norm-check constraints to hold in all of $\mathcal{R}_q$ one would want to first collapse all of them to a single constraint over $\mathbb{Z}_q$ by linear combining them with integer challenges. Then this single $\mathbb{Z}_q$-constraint could be lifted by letting the prover send the polynomial with zero constant coefficient that makes the constraint hold over $\mathbb{Z}_q$. In our case we now already have a linear combination of the $\mathbb{Z}_q$-constraints with polynomial challenges $c_i$. This destroys their $\mathbb{Z}_q$ structure and the constant coefficient does not vanish anymore. It means we need to lift all of the constraints individually inside the linear combination. But the prover can not send all of the lifting polynomials as this would dramatically increase the proof size. Hence, we let the prover compute the lifting polynomials $b_i$ with $\mathrm{cnst}(b_i) = 0$ such that

$$\langle s_i, s_i' \rangle - \beta_i^2 - h_i g + b_i = 0 \text{ in } \mathcal{R}_q,$$

and commit to them together with the $\boldsymbol{s}_i$ in the first commitment phase. Since the lifting polynomials are not short they need to be decomposed with respect to some small base before committing. Now, the final quadratic constraint for the norm-checks is

$$\langle s, s' \rangle - \sum_i c_i(\beta_i^2 + g h_i - b_i) = 0.$$

The linear constraints for proving the conjugated vectors become $\mathrm{cnst}(\langle \sigma_{-1}(c_i \boldsymbol{\alpha}_i), \boldsymbol{s}_i \rangle - \langle \boldsymbol{\alpha}_i, \boldsymbol{s}_i' \rangle) = 0$. They can be collapsed and lifted together, in conjunction with the lifting polynomials $b_i$ so that it is shown that their constant coefficients are zero.

## 5 The Python Layer

The Python layer wraps the C layer of the library and allows the user to easily and efficiently work with algebraic rings of the form $\mathbb{Z}_q[X]/(X^d + 1)$. Most importantly, it also allows the user to create and prove statements using either the [28] or [9] proof systems. There is full documentation, together with detailed examples, provided with the code, and in this section we just provide a general overview. All elements (e.g. polynomials, polynomial, vectors, polynomial matrices) are defined over some polynomial ring $\mathbb{Z}_q[X]/(X^d + 1)$. To create such a ring with, e.g. $d = 512, q = 12289$, one would simply write `R=polyring_t(512,12289)`. Then we can instantiate polynomials (`poly_t`), polynomial vectors (`polyvec_t`), and polynomial matrices (`polymat_t`) over the ring `R` by passing a python list consisting either of integer coefficients to create a polynomial or a list of `poly_t` or `polyvec_t` types to create vectors and matrices. One can also create these types randomly by generating the coefficients from some bounded uniform, binomial, or discrete Gaussian distribution.

The library tries to make polynomial arithmetic simple and intuitive. The operators `+`,`-`,`*` have been overloaded and so one can write things of the form

```
vec_a=mat_B*vec_c+5*pol_d*vec_e-(vec_f*vec_g)*vec_h
```

to correspond to

$$\vec{a} = B * \vec{c} + 5 * d * \vec{e} - \langle \vec{f}, \vec{g} \rangle * \vec{h}.$$

The library automatically performs the needed NTT conversions and then stores the variable in the last form (either NTT or coefficient). In the vast majority of scenarios, this is optimal – i.e. it is fairly rare that one needs a variable in both NTT and coefficient representation.

The initialization function for the classes `poly_t,polyvec_t,polymat_t` use CFFI to initialize a pointer to the respective equivalent C class and the Python functions implementing arithmetic similarly call their respective C equivalents. The C functions that are used in our Python wrapper are all declared in the `lazer_cffi_build.py` file. It's important to note that the user never needs to interact with pointers or anything unrelated to Python when interacting with LaZer through the Python interface.

In addition to the basic arithmetic, we also provide the FALCON trapdoor generation and sampling functions for the ring $\mathbb{Z}_q[X]/(X^d + 1)$ where $d = 512, q = 12289$. As mentioned before, this function is very useful for creating blind signatures, anonymous credentials, and multi/threshold-signatures. One can create a FALCON public key/secret key pair `skenc,pkenc,pkpol=falcon_keygen()`, where `skenc` is a secret key (which corresponds to a short basis stored in some form, which is not important), `pkenc` is the public key stored in some form (we can ignore this[13]), and `pkpol`, which is a FALCON public key. And one can do trapdoor sampling as

```
s1, s2 = falcon_preimage_sample(skenc,t)
```

to create polynomials `s1,s2` such that `pkenc*s2 + s1 = t`.

The main purpose of the LaZer library is to be able to easily work with proof systems. If, for example, we would like to prove knowledge of short `s` satisfying `A*s + t = 0`, after initializing the proof system (see Section 6) by creating a `prover`, we would simply write

```
prover.set_statement(A, t)
prover.set_witness(s)
proof = prover.prove()
```

where `proof` is the zero-knowledge proof in which $\|\vec{s}\|$ satisfies some norm bounds that we specified in a separate file that was first fed to LaZer so that LaZer could create C header files with the optimal parameters (this C header file is always imported at the beginning of the protocol).

The verifier would verify this proof system via (after initialization)

```
verifier.set_statement(A, t)
result = verifier.verify(proof)
```

When creating a succinct proof using LaBRADOR, we will generally be proving many relations of the form (1). We will need to initialize the prover by letting it know how many equations there are, what is the degree of each polynomial, and what are the bounds on the witness norm. Afterwards, we can just enter each polynomial relation one at a time. For example, if we would like to prove knowledge of `s1,s2` such that `pkenc*s2 + s1 = t`, we would write

```
ID=int_to_poly(1,R)
left_statement=[pkenc,ID]
witness=[s2,s1]
```

[13]It is used temporarily in the current version of the library because we do not yet implement general optimal multiplication for any ring except $\mathbb{Z}_q[X]/(X^d + 1)$ with $d = 64$.

```
ProofSystem.fresh_statement(left_statement,witness,t)
```

Then once all the statements are entered, one creates a proof using the `ProofSystem.pack_prove()` routine.

It may happen that we would like to use the same witness polynomial/polynomial vector in several statements. In this case, instead of adding a witness polynomial to the witness list, one can instead add the witness number corresponding to an already witness that one would like to use. So for example, if, instead of `s1`, we wanted to use a polynomial that was the fifth witness, we would set `witness=[s2,4]` (since the numbering starts from 0). Furthermore, the various individual statements can be over rings $\mathbb{Z}_q[X]/(X^d+1)$ of different degrees (i.e. $X^d+1$ for any $d \geq 64$ and a power of 2), though if two statements share a witness, they should be over the same ring.

## 6 Protocol Samples

In this section we give a small taste for how one would use the Python layer of the LaZer library to create protocols that utilize linear-size ZK proofs and succinct proofs. The full protocols can be found in the `python` directory and their descriptions are fully documented in the Python Module / Examples section of the accompanying documentation.

### 6.1 Kyber Proof

One of the example applications for the linear-size ZK proof that we provide is a proof of knowledge of a Kyber-1024 [12] secret key. All the files for this are found in the `python/kyber1024` directory.

Define $R_p = \mathbb{Z}_p[X]/(X^d+1)$ for $p = 3329$ and $d = 256$. The secret key of the Kyber-1024 encryption scheme consists of polynomial vectors $\vec{s}, \vec{e} \in R_p^4$ each of whose coefficients is independently generated from the binomial distribution as $\eta_1 + \eta_2 - \eta_3 - \eta_4$ where $\eta_i \leftarrow \{0,1\}$. The public key consists of a uniformly-random matrix $B \in R_p^{4\times 4}$ and a vector $\vec{t} = B\vec{s} + \vec{e}$. The important feature of Kyber's secret key $(\vec{s}, \vec{e})$, which controls the security and decryption failure probabilities, is that its $\ell_2$-norm is approximately $\sqrt{2048}$. We will show how to use the LaZer library to give a ZK proof that the norm of $(\vec{s}, \vec{e})$ is at most $1.2 \cdot \sqrt{2048}$.

We first need to create the file that defines the parameters of the proof system.

```
vname = "param"             # variable name

deg   = 256                 # ring Rp degree d
mod   = 3329                # ring Rp modulus p
m,n   = 4,8
dim   = (m,n)               # dimensions of A

wpart = [ list(range(n)) ] # partition of w
wl2   = [ 1.2*sqrt(deg*n) ] # l2-norm bounds
wbin  = [ 0               ]  # binary coeffs?
```

The equation that the ZK proof system proves is of the form $[B \mid I] \cdot \begin{bmatrix} \vec{s} \\ \vec{e} \end{bmatrix} + \vec{t} = 0$. In the code above, we define the degree and the modulus of the ring $\mathbb{Z}_p[X]/(X^d+1)$ and then the dimensions of the matrix $A = [B \mid I]$. The list of lists wpart defines how the witness is split. In our case, we want to prove a norm bound on the entire witness $(\vec{s}, \vec{e})$, which consists of 8 polynomials, and so wpart consists of just one list [0,1,2,3,4,5,6,7]. The list wl2 specifies the norm bound that we would like to prove on the corresponding witness in wpart. In this case, we would like to prove that the norm is less than $1.2 \cdot \sqrt{2048}$. If instead of the $\ell_2$-norm, we wanted to prove that the witness were binary, we would instead enter 0 in the wl2 list, and a 1 in the wbin one. In this case, we want to prove the $\ell_2$-norm, so we set wbin to 0.

If the above file is named `kyber1024_params.py`, then we run the sage script (in the `scripts` directory) to create the `kyber1024_params.h` header file that figures out the optimal parameters for the ZK proof. We do this via

```
sage lin-codegen.sage kyber1024_params.py >
        kyber1024_params.h
```

Once we have the header file, we can compile the generated C code into a python module

```
python3 params_cffi_build.py kyber1024_params.h
```

This will create a python module `_kyber1024_params_cffi` exporting a lib object that contains the parameter set. The lib object can contain more than one parameter set (see next Section). One can then import this object into the python code as

```
from _kyber1024_params_cffi import lib
```

We now create the main file which will create the statement and the ZK proof (and verification) of it. This file can be found in the python directory of our source code, and we just give the important details here. We import the LaZer library and also the parameters `mod,deg,m,n` from the `kyber1024_params.py` file. Then we have the only place that the C functions are accessed from Python, and this part is the same in every example, and can just be copied (adjusting for the file names) into any new construction. We import the object lib containing all the parameters described above via `from _kyber1024_params_cffi import lib` and then set up the prover and verifier with the proof parameters and the public randomness KYBERPP (which is the seed that creates a part of the Kyber public key) and the common randomness P1PP which is used in the proof system (e.g. to create the random commitment matrix).

```
from lazer import *

shake128 = hashlib.shake_128(bytes.fromhex("00"))
KYBERPP = shake128.digest(32) # kyber public randomness
shake128 = hashlib.shake_128(bytes.fromhex("01"))
P1PP = shake128.digest(32)   # proof system public
    randomness

from kyber1024_params import mod, deg, m, n
from _kyber1024_params_cffi import lib
prover = lin_prover_state_t(P1PP, lib.get_params("param"))
verifier = lin_verifier_state_t(P1PP,
    lib.get_params("param"))
```

Afterwards, we create the polynomial ring $R = \mathbb{Z}_p[X]/(X^d+1)$ with $d = deg$ and $p = mod$, create the random matrix $A1 \in R^{m\times m}$ using the and the identity matrix $A2$, and then define $A$ as their concatenation. We then create the random secret vector $\vec{sk} = (\vec{s}, \vec{e})$ with binomial coefficients, and finally define $\vec{pk} = A \cdot \vec{sk}$.

```
R = polyring_t(deg, mod)
A1 = polymat_t.urandom_static(R, m, m, mod, KYBERPP, 0)
A2 = polymat_t.identity(R, m)
A = polymat_t(R, m, n, [A1, A2])
sk = polyvec_t.brandom_static(R, n, 2,
     secrets.token_bytes(32), 0)
pk = A*sk
```

We then set the statement for the prover and create the ZK proof. That we know a vector $\vec{w}$ in $R^8$ with $\ell_2$-norm at most $1.2 \cdot \sqrt{2048}$ satisfying $A \cdot \vec{w} - \vec{pk} = 0$.

```
prover.set_statement(A, -pk)
prover.set_witness(sk)
proof = prover.prove()
```

To verify the proof, we simply do

```
verifier.set_statement(A, -pk)
verifier.verify(proof)
```

We can then run the entire protocol simulating the creation of the proof and its verification via `python3 kyber1024_demo.py` where the latter is the file name we saved the above code to.

## 6.2 Anonymous Credentials

A more complicated example using the linear-size ZK proof system is the anonymous credential scheme from [11]. That scheme includes several rounds and two ZK proofs and also requires combining operations over rings $\mathbb{Z}_p[X]/(X^d + 1)$ for different values of $d$. The full protocol is in the `python/anon_cred` directory and we recommend to also read through the blind signature and anonymous credential section in the code documentation. In this section, we will just give a taste for how these more involved schemes can still be fairly easily instantiated using LaZer.

The first ZK proof that we will want to create is to prove knowledge of $\vec{s}, \vec{m}$ satisfying

$$A\vec{r} + B\vec{m} + \vec{t} = 0 \tag{7}$$

over the ring $R_p = \mathbb{Z}_p[X]/(X^d + 1)$ with $p = 12289$ and $d = 64$. The public matrices $A, B$ are in $R_p^{8\times 16}$ and $R_p^{8\times 8}$ respectively. We want to prove that $\|\vec{r}\| < 109$ and that $\vec{m}$ is binary. We therefore set up the parameter file as follows:

```
vname = "p1_param"

deg   = 64
mod   = 12289
dim   = (8,24)

wpart = [ list(range(0,16)), list(range(16,24)) ]
wl2   = [   109,     0 ]
wbin  = [     0,     1 ]
```

The splitting of the witness wpart is more interesting than in the previous Kyber example. The witness now consists of two parts $\vec{r}$ and $\vec{m}$, and we would like to prove different things about each of them. The list wpart thus consists of two lists – one containing the placing of $\vec{r}$, which is in position 0-15, and the then the placing of $\vec{m}$ in positions 16-23. Then the wl2 and wbin lists specify that we would like to prove that $\|\vec{r}\|$ is at most 109 and that the coefficients of $\vec{m}$ are binary.

The second ZK proof proves knowledge of $\vec{r} \in R_p^{16}, \vec{m} \in R_p^8, \vec{s} \in R_p^{16}$, and $\vec{\tau} \in R_p^8$ that satisfy

$$A\vec{r} + B\vec{m} + C\vec{s} + D\vec{\tau} + \vec{t} = 0 \tag{8}$$

with $\|\vec{r}\| < 109$, $\vec{m}$ and $\vec{\tau}$ being binary, and $\|\vec{s}\| < \sqrt{34034726}$. The parameter file, which has a different name would look as follows

```
vname = "p2_param"
deg   = 64
mod   = 12289
dim   = (8,48)

wpart = [ list(range(0,16)), list(range(16,24)),
     list(range(24,40)), list(range(40,48)) ]
wl2   = [   109,   0, sqrt(34034726), 0 ]
wbin  = [     0,   1,              0, 1 ]
```

Note that the wpart list specifies that the witness should be $(\vec{r}, \vec{m}, \vec{s}, \vec{\tau})$ and the wl2 and wbin lists specify the $\ell_2$ norms for $\vec{r}$ and $\vec{s}$ and that $\vec{m}$ and $\vec{\tau}$ should be binary.

To create the parameters, we use the same method as in the Kyber example, except we now have two parameter files. So we first generate the parameter for each script:

```
sage lin-codegen.sage anon_cred_p1_params.py > params1.h
sage lin-codegen.sage anon_cred_p2_params.py > params2.h
```

and then simply paste the code from both of the new .h files into one file `anon_cred_params.h` Once we have the header file, we can compile the generated C code into a python module

```
python3 params_cffi_build.py anon_cred_params.h
```

This will create a python module `_anon_cred_params_cffi` exporting a lib object that contains the parameter set.

In the anonymous credential scheme, the user first generates $\vec{r}, \vec{m}$ and creates the ZK proof for (7). He then receives the vectors $\vec{s}$ and $\vec{\tau}$ from the credential authority, and then has to create a proof for something resembling (8) to show his credential. He therefore needs to initialize two proof systems.

```
self.p1_prover = lin_prover_state_t(P1PP,
     lib.get_params("p1_param"))
self.p2_prover = lin_prover_state_t(P2PP,
     lib.get_params("p2_param"))
```

Another useful feature of the library worth pointing out here is that we can easily work over rings $\mathbb{Z}_p[X]/(X^d + 1)$ of different dimensions $d$. For example, all the above equations were over the ring with $d = 64$, but we also use the FALCON pre-image sampler, which uses $d = 512$. Define $R$ to be the ring with $d = 64$ and $R'$ to be the one with $d = 512$. If we have a FALCON public polynomial $a$ and a signature $s_2, s_1$ such that $as_2 + s_1 = t$ over the ring $R'$, then we can use the standard algebra mapping $\sigma : R' \rightarrow R^8$ (e.g. see [29, Section 2.8]) to convert this to the equation $A\vec{s}_2 + \vec{s}_1 = \vec{t}$ over $R$. The LaZer library supports such a linear algebra transformation and one can convert between the polynomial $a \in R'$ and a matrix $A \in R^{8\times 8}$ by taking an encoding of the FALCON public key generated via

```
sk, pk, _ = falcon_keygen()
```

and then transform pk into $A$ by running `A = falcon_decode_pk(pk,RING)`, where `RING = polyring_t(64,12289)`. Similarly, one can convert the sampled pre-images $s_1, s_2 \in R'$ into $R^8$ via

```
s1, s2 = falcon_preimage_sample(sk,t,RING)
```

Thus one can use the pre-image sampling of FALCON and fork over any polynomial ring $\mathbb{Z}_p[X]/(X^d + 1)$ with $p = 12289$ and $d$ being a power-of-2 at most 512.

## 6.3 Aggregate Signature

A simple example that uses (a wrapper for) the LaBRADOR succinct proof combines the proof with the FALCON signature scheme to create an aggregate signature scheme. The file for the aggregate signature scheme is `python/agg_sig.py`. The general idea is that we want to prove the knowledge of $s_2^{(i)}, s_1^{(i)}$ with small norms satisfying

$$a^{(i)} * s_2^{(i)} + s_1^{(i)} = t^{(i)} \bmod p \tag{9}$$

over the polynomial ring $R_p = \mathbb{Z}_p[X]/(X^d + 1)$ for $p = 12289$ and $d = 512$, for many $i$. At the present, our succinct proof requires all the equations to be written over one of its supported moduli $q$ (which are 24,32,40,or 48 bit) and so we can rewrite (9) as

$$a^{(i)} * s_2^{(i)} + s_1^{(i)} + p * v^{(i)} = t^{(i)} \bmod q \tag{10}$$

over the ring $R_q = \mathbb{Z}_q[X]/(X^d + 1)$ for some polynomial $v$ and $q \gg p$ (while the degree $d$ is still 512). As long as we prove that the norms of $s_2^{(i)}, s_1^{(i)}, v^{(i)}$ are all small, there will be no wraparound modulo $q$ and the above equation holds true over the integers, and therefore (9) is satisfied.

We begin constructing the proof system by importing the LaZer and LaBRADOR python libraries and specifying the number of signatures that will be aggregated, and the $\ell_2$-squared norms of $s_2^{(i)}, s_1^{(i)}, v^{(i)}$ in (10).

```
from lazer import *
from labrador import *
sig_num=1024
norms=[17017363,17017363,round(1248245003*.75)]
```

We now move to setting up the parameters for the proof system. To initialize the proof system, we need to specify the structure of the entire witness vector $\vec{w}$ (which is the concatenation of all the $s_2^{(i)}, s_1^{(i)}, v^{(i)}$). Every element of $\vec{w}$ needs to be either a polynomial or a polynomial vector. We need to specify the degree of the ring in which each of these polynomials (or polynomial vectors) is in. In the case of the aggregate signature scheme, each equation (10) consists of three polynomials of degree 512. Thus the entire witness consists of `3*sig_num` witnesses of degree 512, ad we set `deg_list=[deg]*(3*sig_num)`

Next we specify the number of polynomials that each witness has. In our case, each witness consists of one polynomial, and so we define the num_pols_list variable as a list of `3*sig_num` ones, as `num_pols_list=[1]*(3*sig_num)`.

We then specify the square norm bounds that we would like to prove for each of the witnesses. Because we plan to store the witnesses as

$$[s_2^{(1)}, s_1^{(1)}, v^{(1)}, s_2^{(2)}, s_1^{(2)}, v^{(2)}, \ldots] \tag{11}$$

the norm bounds are going to be as in the `norms` list defined in the beginning,repeated sig_num times, and so we define this list as `norm_list=norms*sig_num`

We finally define the number of constraints (which is the number of signatures) and initialize the proof statement specifying that we are using the 40-bit modulus.

```
num_constraints=sig_num
PS=proof_statement(deg_list, num_pols_list,norm_list,
    num_constraints, "40")
```

To create and add statements in (10), we create a loop that creates a random polynomial (which would normally correspond to a hash of the message) in $R_p$ and then lift it to $R_q$.

```
f_t=poly_t.urandom_static(FALCON_RING,FALCON_RING.mod,TARGPP,0)
l_t=f_t.lift(BIGMOD_RING)
```

We would then pre-image sample the $s_2^{(i)}, s_1^{(i)}$ and again lift them to the ring $R_q$.

```
l_s1, l_s2 = falcon_preimage_sample(skenc, f_t)
l_s1=l_s1.lift(BIGMOD_RING)
l_s2=l_s2.lift(BIGMOD_RING)
```

Afterwards, we initialize and compute the polynomial $v^{(i)}$ in (10) as

$$v^{(i)} = (t^{(i)} - s_1^{(i)} - a^{(i)} * s_2^{(i)}) * p^{-1} \bmod q$$

```
l_v=poly_t(BIGMOD_RING)
v=(l_t-l_s1-l_pk*l_s2)*inv_fal_mod
```

where the variable `inv_fal_mod` was perviously defined to be the inverse of $p$ over $\mathbb{Z}_q$. After checking that the norms of the pre-image sampling outputs are small enough so as the statement we want to prove is actually satisfied, we add the equation to the statement.

```
stat_left=[l_pk,ID,ID*mod]
wit=[l_s2,l_s1,v]
PS.fresh_statement(stat_left,wit,l_t)
```

The equation specified above is `l_pk*l_s2+ID*l_s1+ID*mod*v=l_t`, which is exactly (10). Once we have collected all the equations into our statement, we can output the statement data structure as `stmnt=PS.output_statement()`, and then create the proof via `proof = PS.pack_prove()`. Then the verifier can verify the statement using the `pack_verify(proof,stmnt,"40")` command.

## Acknowledgments

This work was supported by the EU H2020 ERC Project 101002845 PLAZA.

## References

[1] Marius A. Aardal, Diego F. Aranha, Katharina Boudgoust, Sebastian Kolby, and Akira Takahashi. 2024. Aggregating Falcon Signatures with LaBRADOR. In *CRYPTO (1) (Lecture Notes in Computer Science, Vol. 14920)*. Springer, 71–106.

[2] Miklós Ajtai. 1996. Generating Hard Instances of Lattice Problems (Extended Abstract). In *STOC*. 99–108.

[3] Scott Ames, Carmit Hazay, Yuval Ishai, and Muthuramakrishnan Venkitasubramaniam. 2017. Ligero: Lightweight Sublinear Arguments Without a Trusted Setup. In *ACM Conference on Computer and Communications Security*. ACM, 2087–2104.

[4] Sven Argo, Tim Güneysu, Corentin Jeudy, Georg Land, Adeline Roux-Langlois, and Olivier Sanders. 2024. Practical Post-Quantum Signatures for Privacy. *IACR Cryptol. ePrint Arch.* (2024), 131.

[5] Carsten Baum, Ivan Damgård, Vadim Lyubashevsky, Sabine Oechsner, and Chris Peikert. 2018. More Efficient Commitments from Structured Lattice Assumptions. In *SCN*. 368–385.

[6] Carsten Baum and Vadim Lyubashevsky. 2017. Simple Amortized Proofs of Shortness for Linear Relations over Polynomial Rings. *IACR Cryptology ePrint Archive* 2017 (2017), 759. http://eprint.iacr.org/2017/759

[7] Eli Ben-Sasson, Alessandro Chiesa, Michael Riabzev, Nicholas Spooner, Madars Virza, and Nicholas P. Ward. 2019. Aurora: Transparent Succinct Arguments for R1CS. In *EUROCRYPT (1) (Lecture Notes in Computer Science, Vol. 11476)*. Springer, 103–128.

[8] Daniel J. Bernstein, Andreas Hülsing, Stefan Kölbl, Ruben Niederhagen, Joost Rijneveld, and Peter Schwabe. 2019. The SPHINCS$^+$ Signature Framework. In *CCS*. ACM, 2129–2146.

[9] Ward Beullens and Gregor Seiler. 2023. LaBRADOR: Compact Proofs for R1CS from Module-SIS. In *CRYPTO (5) (Lecture Notes in Computer Science, Vol. 14085)*. Springer, 518–548.

[10] Fabian Boemer, Sejun Kim, Gelila Seifu, Fillipe D. M. de Souza, and Vinodh Gopal. 2021. Intel HEXL: Accelerating Homomorphic Encryption with Intel AVX512-IFMA52. *CoRR* abs/2103.16400 (2021).

[11] Jonathan Bootle, Vadim Lyubashevsky, Ngoc Khanh Nguyen, and Alessandro Sorniotti. 2023. A Framework for Practical Anonymous Credentials from Lattices. In *CRYPTO (2) (Lecture Notes in Computer Science, Vol. 14082)*. Springer, 384–417.

[12] Joppe W. Bos, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, John M. Schanck, Peter Schwabe, Gregor Seiler, and Damien Stehlé. 2018. CRYSTALS - Kyber: A CCA-Secure Module-Lattice-Based KEM. In *2018 IEEE European Symposium on Security and Privacy, EuroS&P*. 353–367.

[13] Chi-Ming Marvin Chung, Vincent Hwang, Matthias J. Kannwischer, Gregor Seiler, Cheng-Jhih Shih, and Bo-Yin Yang. 2021. NTT Multiplication for NTT-unfriendly Rings New Speed Records for Saber and NTRU on Cortex-M4 and AVX2. *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2021, 2 (2021), 159–188.

[14] Plonky3 Library Contributors. 2024. Plonky3. https://github.com/Plonky3/Plonky3.

[15] STARK Library Contributors. 2018. stark. https://github.com/elibensasson/libSTARK.

[16] Leo Ducas, Tancrede Lepoint, Vadim Lyubashevsky, Peter Schwabe, Gregor Seiler, and Damien Stehle. 2017. CRYSTALS – Dilithium: Digital Signatures from Module Lattices. Cryptology ePrint Archive, Report 2017/633. https://eprint.iacr.org/eprint-bin/getfile.pl?entry=2017/633&version=20170627:201152&file=633.pdf.

[17] Muhammed F. Esgin, Veronika Kuchta, Amin Sakzad, Ron Steinfeld, Zhenfei Zhang, Shifeng Sun, and Shumo Chu. 2021. Practical Post-quantum Few-Time Verifiable Random Function with Applications to Algorand. In *Financial Cryptography (2) (Lecture Notes in Computer Science, Vol. 12675)*. Springer, 560–578.

[18] Muhammed F. Esgin, Ron Steinfeld, Dongxi Liu, and Sushmita Ruj. 2023. Efficient Hybrid Exact/Relaxed Lattice Proofs and Applications to Rounding and VRFs. In *CRYPTO (5) (Lecture Notes in Computer Science, Vol. 14085)*. Springer, 484–517.

[19] Nils Fleischhacker, Gottfried Herold, Mark Simkin, and Zhenfei Zhang. 2023. Chipmunk: Better Synchronized Multi-Signatures from Lattices. In *CCS*. ACM, 386–400.

[20] Ariel Gabizon, Zachary J. Williamson, and Oana Ciobotaru. 2019. PLONK: Permutations over Lagrange-bases for Oecumenical Noninteractive arguments of Knowledge. Cryptology ePrint Archive, Paper 2019/953. https://eprint.iacr.org/2019/953

[21] Phillip Gajland, Bor de Kock, Miguel Quaresma, Giulio Malavolta, and Peter Schwabe. 2023. Swoosh: Practical Lattice-Based Non-Interactive Key Exchange. *IACR Cryptol. ePrint Arch.* (2023), 271.

[22] Craig Gentry, Shai Halevi, and Vadim Lyubashevsky. 2022. Practical Non-interactive Publicly Verifiable Secret Sharing with Thousands of Parties. 13275 (2022), 458–487.

[23] Irakliy Khaburzaniya, Konstantinos Chalkias, Kevin Lewi, and Harjasleen Malvai. 2022. Aggregating and Thresholdizing Hash-based Signatures using STARKs. In *AsiaCCS*. ACM, 393–407.

[24] Duhyeong Kim, Dongwon Lee, Jinyeong Seo, and Yongsoo Song. 2023. Toward Practical Lattice-Based Proof of Knowledge from Hint-MLWE. In *CRYPTO (5) (Lecture Notes in Computer Science, Vol. 14085)*. Springer, 549–580.

[25] Benoît Libert, San Ling, Khoa Nguyen, and Huaxiong Wang. 2018. Lattice-Based Zero-Knowledge Arguments for Integer Relations. In *CRYPTO (2) (Lecture Notes in Computer Science, Vol. 10992)*. Springer, 700–732.

[26] San Ling, Khoa Nguyen, Damien Stehlé, and Huaxiong Wang. 2013. Improved Zero-Knowledge Proofs of Knowledge for the ISIS Problem, and Applications. In *PKC*. 107–124.

[27] Vadim Lyubashevsky and Ngoc Khanh Nguyen. 2022. BLOOM: Bimodal Lattice One-out-of-Many Proofs and Applications. 13794 (2022), 95–125.

[28] Vadim Lyubashevsky, Ngoc Khanh Nguyen, and Maxime Plançon. 2022. Lattice-Based Zero-Knowledge Proofs and Applications: Shorter, Simpler, and More General. In *CRYPTO (2) (Lecture Notes in Computer Science, Vol. 13508)*. Springer, 71–101. https://eprint.iacr.org/2022/284 https://eprint.iacr.org/2022/284.

[29] Vadim Lyubashevsky, Ngoc Khanh Nguyen, Maxime Plançon, and Gregor Seiler. 2021. Shorter Lattice-Based Group Signatures via "Almost Free" Encryption and Other Optimizations. In *ASIACRYPT (4)*. Springer, 218–248.

[30] Carlos Aguilar Melchor, Joris Barrier, Serge Guelton, Adrien Guinet, Marc-Olivier Killijian, and Tancrède Lepoint. 2016. NFLlib: NTT-Based Fast Lattice Library. In *CT-RSA (Lecture Notes in Computer Science, Vol. 9610)*. Springer, 341–356.

[31] Thomas Prest, Pierre-Alain Fouque, Jeffrey Hoffstein, Paul Kirchner, Vadim Lyubashevsky, Thomas Pornin, Thomas Ricosset, Gregor Seiler, William Whyte, , and Zhenfei Zhang. 2017. *FALCON*. Technical Report. National Institute of Standards and Technology. https://csrc.nist.gov/projects/post-quantum-cryptography/round-1-submissions..

[32] Armin Rigo and Maciej Fijalkowski. 2012-2018. *CFFI documentation*. https://cffi.readthedocs.io/en/latest/index.html.

[33] Victor Shoup. 1996-2021. *NTL: A Library for doing Number Theory*. https://libntl.org/.