Cryptimeleon: A Library for Fast Prototyping of Privacy-Preserving Cryptographic Schemes

Jan Bobolz[1] Fabian Eidens[1] Raphael Heitjohann[1] Jeremy Fell[2]

[1] Paderborn University {jbobolz, feidens, rheitjoh}@mail.uni-paderborn.de
[2] Simon Fraser University jeremy_fell@sfu.ca

July 16, 2021

Abstract

We present a cryptographic Java library called Cryptimeleon designed for prototyping and benchmarking privacy-preserving cryptographic schemes. The library is geared towards researchers wanting to implement their schemes (1) as a sanity check for their constructions, and (2) for benchmark numbers in their papers. To ease the implementation process, Cryptimeleon "speaks the language" of paper writers. It offers a similar degree of abstraction as is commonly used in research papers. For example, bilinear groups can be used as the familiar black-box and Schnorr-style proofs can be described on the level of Camenisch-Stadler notation. It employs several optimizations (such as multi-exponentation) transparently, allowing the developer to phrase computations as written in the paper instead of having to conform to an artificial API for better performance.

Cryptimeleon implements (among others) finite fields, elliptic curve groups and pairings, hashing, Schnorr-style zero-knowledge proofs, accumulators, digital signatures, secret sharing, group signatures, attribute-based encryption, and other modern cryptographic constructions.

In this paper, we present the library, its capabilities, and explain important design decisions.

1 Introduction

Researchers in the field of privacy-preserving schemes *should* implement their schemes much more often: (1) Implementations make researchers' ideas more accessible to practical communities, through benchmark numbers — answering questions like: "is this scheme potentially fast enough for productive use?". Benchmarking becomes especially important in case of a privacy-preserving construction combining many building blocks, where judging performance at a glance is not viable. Additionally, if prototypes are available it gives practitioners the ability to quickly prototype their own product for evaluation in a real environment. If the prototype proves viable it can be used by a programming expert as a basis for a production-ready implementation, additionally considering secure key-storage and standardized formats. (2) Implementations improve the scheme's research paper's editorial quality in the sense that the compiler, test cases, or programmer will bring any typo or implementation-specific problem to light. Thus, prototyping also improves the quality of papers. The implementation process also forces researchers to think about aspects that are often glossed over in papers, but sometimes turn out to be troublesome in practice.

We believe researchers are generally interested in implementing their privacy-preserving schemes that were recently or are yet to be presented in a paper. However, such schemes are often very complex, involving and combining many buildings blocks such as signatures and zero-knowledge protocols. Coming with this is a special set of requirements on a supporting prototyping library. (1) Researchers' time is valuable and limited. Hence such a library has to be easy to use and provide a high-level API supporting a direct paper to code translation. (2) The resulting code should be readable and as close as possible to how schemes are defined in research papers. (3) The library should "speak the language" of paper writers. Meaning that it provides a similar degree of abstraction as is commonly used in research papers. (4) An important requirement is also that the library is feature complete such that researchers can start prototyping their scheme while relying on existing implementations. (5) At the same time such a library must have competitive performance such that benchmarks are meaningful.

We present Cryptimeleon[1] (pronounced *cryp-tee-meleon*, similar to *chameleon*) a library written in Java that is designed for prototyping and benchmarking privacy-preserving cryptographic schemes. The library is geared towards researchers wanting to implement their schemes. To ease the implementation process, Cryptimeleon meets the special set of requirements that comes with research-level prototyping.

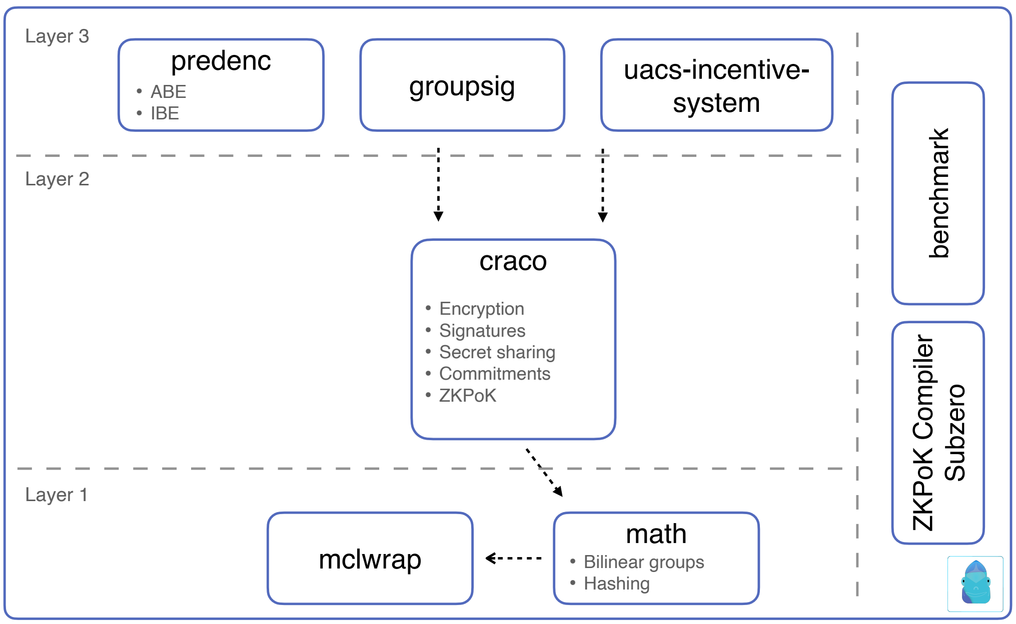

Figure 1: Overview of Cryptimeleon's parts

Cryptimeleon is not one monolithic library, rather it is split into several parts. An overview of Cryptimeleon's parts and implemented schemes is presented in Figure 1. The mathematical underpinning is covered by the Math library. The implemented schemes are collected in a part called Craco and on top of that we provide implementations of predicate encryption, group signatures, and an incentive system based on updatable anonymous credentials. A detailed list of the concrete schemes is given in Appendix A.

Cryptimeleon "speaks the language" of paper writers. It offers a similar degree of abstraction as is commonly used in research papers. For example, pairing groups can be used as the familiar bilinear group black box, and computations like $g^{x^{-1}}$ can be implemented as `g.pow(x.inv())`, where inversion of x is automatically understood to be modulo the group order. As another example, privacy-preserving cryptography often employs Schnorr-style zero-knowledge proofs of knowledge, which are almost universally written down in Camenisch-Stadler notation. Cryptimeleon regards specification of zero-knowledge proofs similarly: it allows researchers to simply feed their Camenisch-Stadler notation into our zero-knowledge compiler *Subzero*. The compiler then generates Cryptimeleon code implementing the protocol (cf. Section 5). The Java framework for

[1] `https://cryptimeleon.org`

Schnorr-style protocols is also structured similarly to how researchers think about them — for example, protocols can be dynamically instantiated with prior exchange of (blinded) values and Schnorr statements are easily composable (cf. Section 4).

Cryptimeleon also implements many of the expected features of Camenisch-Stadler notation protocols: set membership and range proofs, pairing support, nested AND/OR proofs, the Fiat-Shamir heuristic, and others. Other examples of features expected by researchers and implemented by Cryptimeleon are easy hashing into (bilinear) groups or $\mathbb{Z}_p$, and pseudorandom and hash functions with arbitrary input and output lengths. This allows researchers to focus on implementing their specific scheme without having to manually implement some of the building blocks they may take for granted in research papers.

The design of Cryptimeleon is motivated by the following thought process. To benchmark schemes from a paper, you basically have two options. The first option is to implement your scheme from scratch (or based maybe on some existing elliptic curve library), ideally in a low-abstraction language like C. In this scenario, you can manually optimize every single detail specific to your scheme's use case, but this comes with its own challenges. You have to be knowledgeable about possible optimizations and then manually rewrite your scheme into a highly optimized version. While this potentially results in great performance, we believe that this approach, which is also time-consuming, is viable for only a limited number of schemes (e.g. standardized schemes and production-ready schemes) and developers.

For all other developers, researchers, and schemes, there is the second option to rely on a library like Cryptimeleon that alleviates the implementation challenges. In contrast to the first option, with Cryptimeleon (and Java) it is not possible to manually optimize the implementation to get every last drop of performance. This does not mean that implementations with Cryptimeleon are inherently inefficient. We do implement a lot of *automatic* optimizations behind the scenes and our performance numbers are competitive with the simultaneous benefit of fast, convenient, and easy development. Our design decision is to prefer simple, close-to-paper APIs over extremely optimizable APIs. We have put work into optimizations where it benefits the users of Cryptimeleon the most and it can be done without making the code less readable.

For example, to compute two Pedersen commitments $C_1 = g^{x_1} \cdot h^{r_1}, C_2 = g^{x_2} \cdot h^{r_2}$, the code in Cryptimeleon is quite natural and highlights the direct paper to code translation.

```
GroupElement C1 = g.pow(x1).op(h.pow(r1)).compute();
GroupElement C2 = g.pow(x2).op(h.pow(r2)).compute();
// use C1 and C2 ...
```

Behind the scenes, several optimizations are transparently employed: C_1 and C_2 are computed in parallel (signified by the call to `compute()`) and they are each computed as a multi-exponentiation (instead of naively by computing g^{x_i}, then h^{r_i}, and then multiplying the results).

To get a similar level of optimization, other frameworks require developers to jump through many more hoops, making the code less readable. To illustrate, the following code is semantically equivalent in a hypothetical framework without our automatic optimizations.

```
GroupElement[] bases = new GroupElement[] {g,h};
ZnElement[] exp1 = new ZnElement[] {x1, r1};
ZnElement[] exp2 = new ZnElement[] {x2, r2};

Future<GroupElement> C1future =
        executor.submit(() -> group.multiexp(bases, exp1));
Future<GroupElement> C2future =
        executor.submit(() -> group.multiexp(bases, exp2));
try {
        GroupElement C1 = C1future.get();
        GroupElement C2 = C2future.get();
        // use C1 and C2 ...
```

```
} catch (ExecutionException ex) { return; }
```

Another small illustration of the "readability first" design paradigm can be seen with our zero-knowledge framework. Assume a paper requires you to prove knowledge of a secret value r such that $e(g^2 \cdot h^r, y) = C$ in zero-knowledge. This equation can directly be translated to Cryptimeleon code as `new LinearStatement(e.apply(g.pow(2).op(h.pow(r)),y).isEqualTo(C))`. The expression is then automatically rewritten as $e(h^r, y) = C \cdot e(g^2, y)^{-1}$ internally (because the Schnorr-like proof system requires the format "linear expression = constant"). In a hypothetical maximally optimized implementation (or even a less readability-centric library), the programmer would manually (and statically) rewrite the equation to conform to the required format. The downside, however, is that this makes the code less readable (much harder to map the paper's equations onto the optimized implementation). Hence, we opted not to go in this direction — we offer well established and generally applicable optimizations and performance gains though parallelism, multi-exponentiation, and lazy evaluation of group operations, cf. Section 2. These optimizations are transparent to the developer and code reader.

Another benefit of Cryptimeleon's design is that benchmarking schemes is easy and outputs metrics that can be used in papers. It offers a simple approach for hardware-independent performance evaluations via automatic counting of group operations, cf. Section 3. One can also switch to very efficient pairing libraries for hardware-dependent evaluations. To support this Cryptimeleon provides wrappers for `mcl` [Shi] (C++ and Assembly, bilinear group BN254) and `ECCelerate` [Sec] (Java, BN256 and BN464)[2]. Regardless of the pairing library, the optimizations in the rest of our Math library and benefits of Cryptimeleon's API are still in place.

1.1 Related Work

For the work on Cryptimeleon we classify papers or rather their results in a three layer hierarchy. This hierarchy is used in this work to classify related libraries and the parts of Cryptimeleon. In short, layer 1 encompasses foundational research, e.g. choice of security parameter, groups, elliptic curves, pairings, and lattices. To layer 2 we ascribe building blocks such as secret sharing, message-authentication codes, signature schemes, encryption schemes, and (non-)interactive proof systems. Therefore, higher level protocols that rely on layer 2 are assigned to layer 3, e.g. anonymous credentials, predicate encryption, and group signatures.

For on overview of the layers in Cryptimeleon see Figure 1, where layer 1 is the Math library. Additionally, we show mclwrap to highlight the flexibility of Math to rely on other pairing libraries. With Math our layer 1 provides the mathematical foundation through groups, rings, fields, and bilinear groups equipped with pairings, where the transparent optimization of the layer is the most important API feature for the layers above.

Layer 2 is covered by Craco since it provides everything concerned with cryptographic schemes including interfaces and classes for public parameters, keys, ciphertexts, signatures, and message blocks. Researchers can directly implement their scheme with the support of the provided interfaces and class structure of Craco. For layer 3, Craco includes implementations of important building blocks for higher level cryptographic constructions, e.g. accumulators, commitments, signatures, encryption schemes, key encapsulation mechanisms, secret sharing schemes, and zero-knowledge proof of knowledge protocols. A detailed list of the concrete schemes is given in Appendix A.

Cryptimeleon's layer 3 provides three examples, namely predicate encryption (predenc), group signatures (groupsig), and an incentive system based on updatable anonymous credentials (uacs-incentive-system) that rely on the lower levels, see Figure 1. We use uacs-incentive-system in Section 3 to show that Cryptimeleon is ready for prototyping and benchmarking of privacy-preserving schemes. The predenc part features implementations of identity-based and attribute-based encryption that build upon the secret sharing schemes, e.g. monotone-span programs, of Craco. The last layer 3 part is groupsig which is derived from the interfaces of libgroupsig [DAR15]. The goal of

[2] Cryptimeleon also provides Java-implemented elliptic curve groups out of the box.

libgroupsig is to provide group signature interfaces that can be used as a standard API. This is, based on our knowledge, the first adaptation of libgrougsig besides the original C implementation.

In the following we give an overview of related libraries, classify them according to the introduced three layers, and comparing the libraries with Cryptimeleon.

We start with a look at traditional cryptography libraries that do not focus on research-level prototyping. There are many libraries [AGM+,BLS12,Leg,ZBPB17] that provide implementations of standardized cryptographic schemes such that you never have to implement one of them yourself or even have to think about secure parameters such as correct padding in RSA. You can just build on top of vetted implementations with secure parameters, side-channel security, and standardized key formats. For very efficient implementations of select cryptographic schemes, one can rely on one of the NaCl [BLS12] variants like libsodium [libb] and HACL* [PBP+20, ZBPB17]. Because they focus on standardized established schemes instead of more modern research-level schemes, they offer no direct support for prototyping of modern privacy-preserving schemes.

1.1.1 mcl

The library mcl [Shi] focuses on a highly performant implementation of the Ate pairing over common BN and BLS12-381 curves written in Assembly and C++. Through its architectural versatility it supports all major systems including M1 macOS, Android, and WebAssembly. Because of these features it is the primary external pairing library used in Cryptimeleon via a wrapper. This wrapper also is an example of how to implement one for a layer 1 library of your choice. Note that many of the libraries listed here provide bindings for Java which simplifies the implementation of a wrapper.

Layer 1: Type 3 bilinear pairings using Barreto-Naehrig (BN254, BN381, BN462) and Barreto-Lynn-Scott curves (BLS12-381)

1.1.2 ECCelerate

Developed at TU Graz, specifically the Institute for Applied Information Processing and Communication (IAIK), ECCelerate [Sec] is a commercial product with the exception that is free for education and research. It mostly implements standardized schemes from ANSI X9.62-2005, ANSI X9.63, IEEE P1363a, FIPS 186-4, SEC1 v2.0, SEC2 v2.0, RFC 5639 and ANSSI. What makes it usable for Java developers that rely on the official Java Cryptography Extension (JCE) is that ECCelerate is JCE compatible by using the IAIK JCE provider[3]. Since ECCelerate covers standardized cryptography, it is mostly situated at Layer 1 and 2 with schemes that Cryptimeleon does not deal with. For our goals, only the provided asymmetric bilinear pairings are of interest. Therefore, we provide in Cryptimeleon a wrapper for the ECCelerate pairing.

Layer 1: Type 3 bilinear pairings using Barreto-Naehrig curves; Curve25519 and Curve448 for EdDSA

Layer 2: EdDSA, ECDSA, ECDH, ECMQV (key agreement), ECIES

1.1.3 Bouncy Castle

The Bouncy Castle library [Leg] is written in Java and mainly provides implementations of standardized schemes for the official Java Cryptography Architecture (JCA). In detail it provides implementations, among others, for S/MIME, TLS, X.509 certificates, and PKCS#12. From the privacy-preserving cryptography view, the focus of Bouncy Castle is therefore on layer 1 and 2 with the goal to provide Java developers easy access to common and standardized cryptographic operations.

[3] `https://jce.iaik.tugraz.at/products/core-crypto-toolkits/jca-jce/`

1.1.4 RELIC

The library RELIC [AGM+] is mainly written in C and aims at a performant implementation of common cryptography schemes. Therefore, it includes very efficient architecture-dependent code in the form of multiple implementations for each scheme to tailor for specific CPU and memory features. Given the implementation of many standardized schemes, researchers tend to use RELIC to extend existing systems. Thereby enabling benchmarks in real world execution environments, e.g. testing forward-secure 0-RTT key exchange in the QUIC protocol [DDG+20]. RELIC is available under the Apache-2.0 License or LGPL-2.1 and provides the following implementations:

Layer 1 : Elliptic curves (NIST curves and pairing-friendly curves), integer arithmetic, prime and binary field arithmetic

Layer 2 : RSA, ECDSA, ECMQV, ECSS (Schnorr), ECIES, BLS [BLS04], BBS [BBS04], PS signatures [PS16], Paillier [Pai99] and Benaloh [Ben94] homomorphic encryption schemes

1.1.5 NaCl

Introduced in [BLS12] NaCl (pronounced *salt*), with its simple and high-level API, performant and secure implementations in C, is the basis of many other libraries like libsodium [libb]. NaCl defines a core API consisting of six functions concentrating on public-key authenticated encryption and signatures. The library follows the mantra that programmers should never be asked to define the correct key size and padding for RSA signatures, rather they should just call a function that signs a given message and the rest is done internally. Underneath this, NaCl features very efficient (memory and computation) constant-time implementations of well established cryptographic schemes ranging from AES to state-of-the-art schemes. The efficiency also comes from an automatic selection of an implementation specific for your CPU. Besides being very rigorous with its mostly verified implementations, NaCl also defines the aforementioned core of the API and therefore a naming convention adapted by other libraries such as HACL* [ZBPB17], libsodium [libb], libhydrogen [liba], and TweetNaCl [BvJ+15]. It follows an excerpt of the provided primitives in NaCl.

Layer 1: Curve25519, Poly1305 MAC, AES-GCM

Layer 2: Authenticated encryption: Curve25519 elliptic-curve-Diffie–Hellman function, Salsa20 stream cipher, and Poly1305 MAC [4], Ed25519 signature scheme

Other NaCl API compatible libraries improve it in performance, compatibility and implementation of more schemes. For example libsodium [libb] deals with downsides of NaCl. Unlike NaCl the build process and resulting library can be installed system-wide and is portable in the sense that it also runs on machines different from the compiling machine. In addition there are bindings for all major programming languages[5] making it even more useable, e.g. one can sign a message with libsodium in Python and verifying it in Rust.

1.1.6 HACL*: A Verified Modern Cryptographic Library

As a member of Project Everest [Mic], the library HACL* [ZBPB17] focuses on providing a compact and verified library written in the F* programming language that supports the full NaCl API. Further development presented in [PBP+20] concentrates heavily on optimizing the implementation for multiple architectures using single-instruction multiple data parallelism. The use case of HACL* is Signal and especially TLS together with other Everest projects, e.g. ValeCrypt (primitives in assembly) and EverCrypt (automatically selects from HACL* and ValeCrypt the best implementation depending on the execution environment). Since HACL*, available under

[4] For details on authenticated encryption see `https://cr.yp.to/highspeed/naclcrypto-20090310.pdf`

[5] `https://doc.libsodium.org/bindings_for_other_languages`

Apache-2.0 license, covers layer 1 with standardized implementations, it is a candidate for future support in Cryptimeleon such that you can build layer 3 systems while relying on the performant and verified code base of HACL*. Note, that the extensive performance evaluation of NaCl compatible libraries in [ZBPB17] shows that the verification guarantees of HACL* are no detriment to the performance.

Layer 1: Among others[6]; ChaCha20 and Salsa20 stream ciphers, AES-GCM, SHA-3

Layer 2: Same as NaCl: Salsa20 based authenticated encryption, Ed25519 signature scheme

There are also libraries that pursue the goal of research-level prototyping. Two notable libraries are Kyber [Decb] and Charm [AGM+13].

1.1.7 Kyber

The library Kyber [Decb], written in Go, is developed in the Cothority project [Deca, STV+16] that provides tools for decentralized cryptographic schemes. Kyber's goal is to provide a high-level API for developers that want to use modern cryptography (ZKPoK, secret sharing, pairing-based signature schemes) in real-world applications. Therefore their goal is similar to ours. Their ZKPoK framework supports proving knowledge of discrete logarithms with AND and OR statements, where Cryptimeleon additionally provides range and set membership proofs.

Layer 1: Finite field arithmetic, Edwards curve 25519, NIST P-256 elliptic curve, Barreto-Naehrig (BN256)

Layer 2: Among others; EdDSA, Schnorr signatures, Polynomial commitment, Shamir secret sharing, ZKPoK, elliptic curve integrated encryption scheme (ECIES), Boneh-Lynn-Shacham (BLS) [BLS04]

1.1.8 Charm

The library Charm [AGM+13] is a framework for prototyping of academic cryptographic schemes and provides a starting point for benchmarks against other implemented schemes. Charm is written in Python and tackles challenges that just occur during implementation of a scheme. Hence, it provides solutions for serialization of cryptographic objects, error handling and basic checks that are not present in research papers, but are necessary for any application.

The focus of Charm is on API usability, scheme variety and composition. Schemes in Charm can be combined in predefined ways via so called adapters, e.g. an adapter for hybrid encryption. In Cryptimeleon this is directly achieved by appropriate types and class hierarchies due to Java. Therefore, Java type safety helps in limiting developers choices where security demands it. Comparing the attribute-based encryption capabilities, in Cryptimeleon the creation and handling of predicates in the form of threshold and boolean polices is more sophisticated. In Charm the process of defining policies as a text string that is then interpreted is error-prone. In Cryptimeleon, polices and attributes are typed and have a fixed set of operators. In both libraries policies get internally transformed to monotone span programs to be used by the schemes.

Available under the LGPL-3.0 license Charm features the following primitives:

Layer 1: Relies on Python bindings for OpenSSL, PBC library, RELIC, and MIRACL.

Layer 2: encryption schemes (including identity-based and attribute-based schemes), e.g. Waters11 [Wat11, Wat08], digital signatures, e.g. BLS [BLS04] and PS signatures [PS16], commitment schemes, and zero-knowledge proofs[7]

[6] For full list of supported algorithms refer to `https://hacl-star.github.io/Supported`

[7] For a listing of schemes see `https://github.com/JHUISI/charm/wiki/Cryptographic-schemes-and-protocols`

1.2 Related Work on ZKPoK (compilers)

Besides Cryptimeleon, there are a number of libraries that help implementing Schnorr-style protocols.

1.2.1 zksk

Most recently, Lueks et al. presented the *Zero-Knowledge Swiss Knife* Python library [LKF+19]. Similar to our library, it is design for fast prototyping of Schnorr-style protocols. Like Cryptimeleon, they offer extendable "primitives" (similar in functionality to our "fragments", see Section 4), Camenisch-Stadler notation, linear equations, range proofs, OR-composition, interactive and non-interactive execution, etc. over bilinear groups. Overall, *zksk* can be seen as a Python alternative to our protocol implementation offering, with (as of 2021) minor differences in capabilities.

1.2.2 dalek zkp and Merlin

Written in rust, *dalek zkp*[8] implements Schnorr-style proofs (only) for basic (homomorphism preimage) statements. While it is less expressive than Cryptimeleon, it is intended to be eventually used in production code (whereas Cryptimeleon is purely meant for academic prototyping). It is compatible with *Merlin*[9], which enables secure protocol compilation (and composition) to a non-interactive proof.

1.2.3 CACE ZK Toolbox / YAZKC / ZKCrypt

Within the *CACE* project a zero-knowledge compiler [ABB+10] was implemented. It allows users to specify a statement in a Camenisch-Stadler-like domain-specific language and supports many of the standard extensions that Cryptimeleon also supports. In contrast to Cryptimeleon, its generated protocols can be verified [ABB+12]. Unfortunately, the compiler does not seem to be publicly available (anymore).

1.2.4 ZKPDL / Cashlib

Similar to CACE, *ZKPDL* [MEK+10] compiles protocols specified in a tailor-made domain-specific language to a working protocol. While expressive, they do not support OR composition of protocols or reusable subprotocols.

1.2.5 ZQL

Focusing less on protocol prototyping, *ZQL* [FKDL13] compiles a (general-purpose) computation specified in their domain-specific language into a Sigma protocol.

1.2.6 Additional pointers

Further libraries with *some* support to implement Schnorr-style protocols include emmy [SBM+19], Charm [AGM+13], SCAPI [EFLL12], and DEDIS Kyber[10].

2 The Math Library's Computational Model

When it comes to computation of expressions of group elements, there are several optimizations that are often employed: Multi-exponentiation, parallelism, and precomputation. We will detail

[8] `https://github.com/dalek-cryptography/zkp`
[9] https://merlin.cool
[10] `https://github.com/dedis/kyber`

these and how they are realized in the Cryptimeleon Math library in the following. We note that Cryptimeleon does not support constant-time group operations.

2.1 Structure of Group Implementations

On the implementation level, a group is split into a "frontend" and a "backend". The backend only defines how to compute a group's operations (e.g., using our custom Java implementation or by delegating to an efficient C++ implementation [Shi]). The frontend is what a user of our library usually interacts with. It handles generic optimizations, delegating the actual group operations to the backend. This split into frontend and backend means that when implementing new groups for our library, there is no need to reimplement generic optimization. In the remainder of this section, we generally describe the `LazyGroup` frontend for groups, in which group operations are done lazily, which enables many optimizations to be done automatically behind the scenes.

2.2 Multi-exponentiation

The idea behind multi-exponentiation is that an expression such as $g^a \cdot h^b$ can be computed more efficiently than naively computing g^a and h^b and then multiplying them.

For illustration, take $a = 6 = (110)_2$ and $b = 3 = (011)_2$ and let $g_i = g^i$ and $h_i = h^i$ for $i \in \{0, 1\}$. We can *interleave* the usual Square and Multiply computation of $g^a = (g_1^2 \cdot g_1)^2 \cdot g_0$ (note the $(110)_2$ pattern in this expression) and $h^b = (h_0^2 \cdot h_1)^2 \cdot h_1$ to compute $g^a \cdot h^b$. In the interleaved version, the two bases essentially share the squaring step: $g^a \cdot h^b = ((g_1 h_0)^2 \cdot (g_1 h_1))^2 \cdot g_0 h_1$. This approach results in fewer group operations than the naive computation. While this example illustrates the idea, the library uses slightly more advanced multi-exponentiation methods (see [Möl01] for a good overview).

Because the `LazyGroup` is lazy, the expression `g.pow(a)` does not compute g^a immediately, but instead returns a placeholder object. This placeholer object behaves semantically equivalent to g^a, but internally the computation of its concrete value is deferred until needed (which is mostly upon serialization or comparison with another group element, which may, as in the following example, never actually happen). The advantage is that, in the expression `C = g.pow(a).op(h.pow(b))`, the `g.pow(a)` subexpression does not force the computation of g^a (same for h^b), so these intermediate values are never actually computed. Instead, `C` is again a placeholder object. The concrete value $g^a \cdot h^b$ of `C` is computed *using an efficient multi-exponentiation method* when `C`'s value is accessed. As a result, programmers benefit from the performance improvements of multi-exponentiation without any additional effort.

2.3 Precomputation

Precomputation can be used to speed up (multi-)exponentiation for a specific group element. For illustration, take $a = 14 = (1110)_2$ and $g_i = g^i$ for $i \in \{0, 1\}$. The Square and Multiply algorithm would compute g^a as $(((g_1^2) \cdot g_1)^2 \cdot g_1)^2 \cdot g_0$ (note the 1110 pattern in the expression). With precomputation of $(g_{00}, g_{01}, g_{10}, g_{11}) := (g^0, g^1, g^2, g^3)$ in use, we can more efficiently compute g^a as $((g_{11})^2)^2 \cdot g_{10}$ (i.e. handle two bits per multiply step instead of only one).

To run precomputation for a group element g, one calls `g.precomputePow()`. This triggers a similar precomputation of small powers of the group element as explained above to speed up future exponentiations or multi-exponentiations involving `g`.[11] During the setup phase of a scheme, this should be used on group elements involved in (several) future exponentiations, like the bases of a Pedersen commitment scheme.

[11] The library actually implements the more efficient wNAF (multi-)exponentiation technique instead of of simple Square and Multiply (cf. [Möl01])

2.4 Parallelism

As is commonly known, modern processors can process multiple threads at the same time. The `LazyGroup` enables a very easy model to exploit parallelism.

Suppose we have code to compute two Pedersen commitments `C1`, `C2` and send them to another party.

```
1 GroupElement C1 = g.pow(x1).op(h.pow(r1));
2 GroupElement C2 = g.pow(x2).op(h.pow(r2));
3 //...
4 send(C1.getRepresentation());
5 send(C2.getRepresentation());
```

Here, because group elements are lazy, the actual value of C1 is (internally) computed in line 4 and afterwards the value of C2 is computed in line 5.

To enable parallelism in this scenario, one can use the `compute()` method on group elements. This method is non-blocking and returns immediately, but starts computation of the result of a group element on a background thread. So we change lines 1 and 2, adding a `compute()` call as follows.

```
1 GroupElement C1 = g.pow(x1).op(h.pow(r1)).compute();
2 GroupElement C2 = g.pow(x2).op(h.pow(r2)).compute();
3 //...
4 send(C1.getRepresentation());
5 send(C2.getRepresentation());
```

Here, computation of C1 begins on a background thread in line 1 and computation of C2 begins on a background thread in line 2. So in this code snippet, computation of C1 and C2 happens concurrently. Line 4 then blocks until the value of C1 has finished computing and then line 5 blocks until the value of C2 has finished computing.

Calls to `compute()` are never *necessary* (i.e. the code produces the same result with or without them) but they enable concurrent computation of results. The need to *manually* call `compute()` (instead of, say, having the library automatically call `compute()` implicitly after every operation) arises from the need to mark some results as relevant to distinguish them from intermediate results. For example, `C1 = g.pow(x1).compute().op(h.pow(r1).compute()).compute()` is semantically valid, but triggers more computation than necessary. This is because this forces computation of unwanted intermediate results (`g.pow(x1)` and `h.pow(r1)`) even when `C1` can be more efficiently computed using multi-exponentation.

3 Benchmarking

An important purpose of implementing a prototype is gathering performance information. This information then allows for comparisons with other schemes and to evaluate practicality. Achieving acceptable runtime and/or memory usage is essential for demonstrating the potential for practical usage.

There are many different kinds of performance metrics. These include runtime in the form of CPU cycles or CPU time, memory usage, and network usage. Furthermore, one may want to collect hardware-independent information such as the number of group operations or pairings. Both of these types of metrics have their use cases. Counting group operations and pairings has the advantage of being hardware-independent. To the layperson and potential user, applied metrics such as CPU time or memory usage can be more meaningful as they demonstrate practicality better than the more abstract group operations metrics. Cryptimeleon supports collecting all of these metrics in a coherent way. This way the user can choose the metrics that are most suitable to their use case.

3.1 Collecting hardware-independent metrics

The collection of hardware-independent metrics such as group operations is implemented by the Cryptimeleon Math library. The main points of interest here are the `DebugBilinearGroup` and `DebugGroup` classes. The former allows for counting pairings, and the latter allows for counting group operations, group squarings (relevant for elliptic curves), group inversions, exponentiations, and multi-exponentiations. It is also able to track the number of times group elements have been serialized.

The counting is done in two modes: The "NoExp" mode and the "Total" mode. Group operations metrics from the "NoExp" mode disregard operations done inside (multi-)exponentiations while the "Total" mode does account for operations inside (multi-)exponentiations. "NoExp" measurements are therefore independent of the actual (multi-)exponentiation algorithm while "Total" measurements are more expressive in regards to the actual runtime (since estimating group operation runtime is easier than that of a (multi-)exponentiation).

As an example we consider the computation of $g^a \cdot h^b$ over a group size of 128 bit. The "NoExp" mode counts this as a single multi-exponentiation with two terms. No group operations are counted since they are all part of the multi-exponentiation. The "Total" mode does not consider the multi-exponentiation as its own unit. Instead, it counts the group operations, inversions, and squarings that are part of evaluating the multi-exponentiation (using a wNAF-type algorithm). Combining these metrics gives us therefore a more complete picture of the computational costs.

To collect operation metrics one just has to replace the `BilinearGroup` or `Group` used by `DebugBilinearGroup` or `DebugGroup`, respectively. Then execute the code you want to collect metrics for, and display the results. The measurements are stored within `DebugBilinearGroup` and/or `DebugGroup` and can be retrieved via getter methods. These getter methods are split up by metric and by mode.

We now take a look at how one can use the debug groups to obtain operation metrics for the verification algorithm of the signature scheme from [PS18]. Given the public key $\mathsf{pk} = (\tilde{g}, \tilde{X}, \tilde{Y}_1, \ldots, \tilde{Y}_{r+1})$, the message vector $\mathbf{m} = (m_1, \ldots, m_r)$, and the signature $\sigma = (m', \sigma_1, \sigma_2)$, the verification algorithm checks whether $\sigma_1 \neq 1$ and $e(\sigma_1, \tilde{X} \cdot \prod_{j=1}^{r} \tilde{Y}_j^{m_j} \cdot \tilde{Y}_{r+1}^{m'}) = e(\sigma_2, \tilde{g})$. We show the code to count the number of operations during verification in Listing 1, and show the result in Table 1 (we omit the results for $\mathbb{G}_1$ and $\mathbb{G}_\mathrm{T}$).

List. 1: Operation measurements for verification algorithm of [PS18]

```
// Want a type 3 bilinear group; debug group is flexible
var bilGroup = new DebugBilinearGroup(TYPE_3);
// Enable counting by instantiating scheme using debug group
PSPublicParameters pp = new PSPublicParameters(bilGroup);
// [Set up plainText, sig, and vk]

// Reset all counters before executing the verification
bilGroup.resetCounters();
// Verification
scheme.verify(plainText, sig, vk);
// Print results
System.out.println(bilGroup.formatCounterData());
```

3.2 Collecting applied metrics

Applied metrics, such as runtime or memory usage, can be collected using any existing Java benchmark framework. An example of such a framework is the Java Microbenchmarking Harness (JMH). It allows for very accurate measurements and integrates with many existing profilers. Due to these existing options, we have decided against implementing any such capabilities ourselves.

Table 1: Results of running the benchmark in Listing 1. The multi-exponentiation results mean that a single multi-exponentiation with two terms has been done.

Metric	Result
Number of total group operations in $\mathbb{G}_2$	62
Number of total group inversions in $\mathbb{G}_2$	27
Number of total group squarings in $\mathbb{G}_2$	128
Number of terms in each multi-exponentiation in $\mathbb{G}_2$	2

Table 2: Results of running the JMH benchmark in Listing 2. "numMessages" denotes the length of the message vector.

Benchmark	numMessages	Mode	Cnt	Score	Error	Units
measureVerify	1	ss	50	5.845	$\pm$ 0.661	ms/op
measureVerify	10	ss	50	9.259	$\pm$ 1.097	ms/op

In Listing 2, we show JMH runtime measurement code for the verification algorithm of the signature scheme from [PS18]. The results are depicted in Table 2.

List. 2: Runtime measurements for verification algorithm of [PS18]

```
@State(Scope.Thread)
public class PS18VerifyBenchmark {
  // Test with one message and ten
  @Param({"1", "10"})
  int numMessages;

  // [Insert remaining fields here]

  @Setup(Level.Iteration)
  public void setup() {
    // Use the efficient mcl BN254 wrapper
    var pp = new PSPublicParameters(new MclBilinearGroup());
    // [Set up plainText, sig, and vk]
  }

  // The benchmark method. Includes settings for JMH
  @Benchmark
  @BenchmarkMode(Mode.SingleShotTime)
  @Warmup(iterations = 3, batchSize = 1)
  @Measurement(iterations = 10, batchSize = 1)
  @OutputTimeUnit(TimeUnit.MILLISECONDS)
  public Boolean measureVerify() {
    return scheme.verify(plainText, sig, vk);
  }
}
```

Using the right metric to illustrate the performance of a construction can be helpful for driving further adoption. Cryptimeleon is flexible and allows for collecting a wide variety of important metrics.

Table 3: Average performance of our implementation of [BBDE19a] over 100 runs in milliseconds or number of operations. Emphasized: typical execution platform for each algorithm.

Measurement	Issue	Join	Credit	Earn	Deduct	Spend
Google Pixel (Phone, Snapdragon 821)	37	**38**	43	**37**	139	**88**
Macbook Pro (Laptop, i9-9980HK)	**3**	3	**4**	3	**15**	16
$\mathbb{G}_1$ operations (multiply or square)	6432	4419	1811	1570	8662	8276
$\mathbb{G}_2$ operations (multiply or square)	0	1587	855	804	7911	3159
$\mathbb{G}_\mathrm{T}$ operations (multiply or square)	0	0	599	1	9779	2211
Pairings	0	2	6	4	30	12

3.3 Example: Benchmarking a Privacy-Preserving Incentive System

As a test run for the library, we have reimplemented[12] the privacy-preserving incentive system based on updatable anonymous credentials [BBDE19a]. This construction uses Pointcheval Sanders (blind) signatures and Schnorr-style zero-knowledge protocols. Such systems are exactly what Cryptimeleon is geared towards, so implementation of the system was straightforward.

See the original paper's full version [BBDE19b, Appendix E] for more information on the construction. Because of various performance optimizations in Cryptimeleon, we were able to improve upon the original paper's numbers by a factor of about 2 to 4.

Table 3 shows the running times on a laptop and an Android phone, and (device-independent) group operation counts. The device benchmarks were created using mcl's BN254 as the bilinear group and a maximum point count of 256^8 (for the range proof). The operation counts may fluctuate over multiple runs because, for example, random choices of exponents may lead to more or fewer operations during exponentiation.

4 Sigma Protocol Framework

Zero-knowledge proofs of knowledge are a powerful tool for modern privacy-preserving protocols. They allow a prover to prove knowledge of values without revealing these values to the verifier. For this reason they are a natural fit for privacy-preserving protocols, where the hidden values may be something like a user's identity, attributes, messages, or keys.

There are many different zero-knowledge proof systems, but proof systems based on Schnorr's protocol are often the most convenient for modern privacy-preserving protocols. This is because Schnorr protocols (and their generalizations) are very efficient for algebraic statements (for example knowledge of discrete logarithms over (bilinear) groups). Furthermore, they are Sigma protocols, which enables many generic extensions (e.g., they can be made non-interactive using the Fiat-Shamir heuristic [FS87]).

Theoretically, we can characterize Schnorr protocols as being able to prove preimages of a group homomorphism ψ [Mau09]. This means that given some y, the verifier can check that the prover knows some x with $\psi(x) = y$. For example, for the original Schnorr proof of knowledge of a discrete logarithm, the homomorphism is $\psi(x) = g^x$.

However, much more complex protocols can be built. We call protocols that build upon Schnorr's protocol "Schnorr-style protocols". Schnorr-style protocols have been used, for example, for

- Proving knowledge of a signature (the main ingredient for *anonymous credentials*)
- Proving statements about the contents of an ElGamal ciphertext or a Pedersen commitment
- Proving well-formedness of values (e.g., that something has been raised to the right exponent)

[12] `https://github.com/cryptimeleon/uacs-incentive-system`

- Range proofs
- ... any combination of the above

Usually, these protocols are denoted in the style of the Camenisch-Stadler notation [CS97]. For example, a typical research paper designing privacy-preserving protocols may contain an expression such as the following.

$$\mathrm{ZK}\{(m_1, m_2, r) : \; C_1 = h_1^{m_1} \cdot h_2^{m_2} \cdot g^r \wedge 20 \leq m_1 + m_2 \leq 100\} \tag{1}$$

It denotes a zero-knowledge proof of knowledge protocol in which the prover proves knowledge of an opening $m_1, m_2, r \in \mathbb{Z}_p$ to the public Pedersen commitment C such that $m_1 + m_2 \bmod p$ is a number between 20 and 100.

The concrete protocol corresponding to Equation (1) is quite complicated to write down manually. Cryptimeleon's Craco library offers a protocol framework to conveniently implement Schnorr-style protocols. In our framework, Equation (1) can be implemented by extending the `DelegateProtocol` class and overriding the `provideSubprotocolSpec` method as in Listing 3.

List. 3: Implementation of Equation (1) using `DelegateProtocol`

```java
@Override
protected SubprotocolSpec provideSubprotocolSpec(
        CommonInput input, SubprotocolSpecBuilder builder) {
  SchnorrZnVariable m1 = builder.addZnVariable("m1", zn);
  SchnorrZnVariable m2 = builder.addZnVariable("m2", zn);
  SchnorrZnVariable r = builder.addZnVariable("r", zn);

  builder.addSubprotocol("commitmentOpen",
    new LinearStatementFragment(
      h1.pow(m1).op(h2.pow(m2)).op(g.pow(r))
        .isEqualTo(((MyCommonInput) input).commitmentC)
    )
  );

  builder.addSubprotocol("rangeProof", //m1+m2 \in [20,100]
    new TwoSidedRangeProof(m1.add(m2), 20, 100, pp)
  );

  return builder.build();
}
```

Composing the actual protocol is then done behind the scenes. In particular, the range proof is (conceptually) split into two range proofs of the form $x \in [0, b^\ell)$, which in turn run set membership subprotocols for the base b digits of x [CCs08].

This complexity is hidden from scheme implementors. Our goal is to enable developers to write code that is not much more complex than the corresponding protocol specification in research papers.

4.1 Schnorr Fragments

As in the example above, protocols usually contain more than a single statement. For simplicity, consider the following proof for equality of discrete logarithms:

$$\mathrm{ZK}\{(x) : \; h_1 = g_1^x \wedge h_2 = g_2^x\} \tag{2}$$

This proof is clearly composed of two parts: one for the statement $h_1 = g_1^x$ and one for $h_2 = g_2^x$. One could naively consider coming up with a protocol for the first and another protocol for the second statement, and then running them in parallel with the same challenge (which is the generic way of AND-composing Sigma protocols). However, if we go with this generic composition, we end up with a protocol implementing

$$\mathrm{ZK}\{(x_1, x_2):\ h_1 = g_1^{x_1} \wedge h_2 = g_2^{x_2}\}$$

i.e. for each of the protocols run in parallel, the prover may choose an independent witness. As known in folklore, we can *appropriately* instantiate Equation (2) with the protocol in Figure 2. In contrast to the generic composition of two Schnorr proofs, here the two parts share the same

Prover $\mathcal{P}(x)$		Verifier $\mathcal{V}()$
$s_x \leftarrow \mathbb{Z}_p$		
$\boxed{a_1 = g_1^{s_x}}, \boxed{a_2 = g_2^{s_x}}$	$\xrightarrow{a_1, a_2}$	
	$\xleftarrow{c}$	$c \leftarrow \mathbb{Z}_p$
$r_x = x \cdot c + s_x$	$\xrightarrow{r_x}$	
		accept if
		$\boxed{g_1^{r_x} = h_1^c \cdot a_1}$ and $\boxed{g_2^{r_x} = h_2^c \cdot a_2}$

Figure 2: A protocol for Equation (2). Contains Schnorr fragments for $h_1 = g_1^x$ (in red, dashed) and $h_2 = g_2^x$ (in blue, dotted)

variable x, i.e. the same randomness s_x in the announcement and the same response r_x. This kind of composition, which allows for sharing variables between the statements/subprotocols, is clearly superior to its generic counterpart, which does not allow variable sharing.

This motivates our notion of a *Schnorr fragment*. Schnorr fragments are designed to be composable in a way that enables shared variables. Furthermore, they are powerful enough to encapsulate complex subprotocols such as range proofs on shared variables. Figure 2 shows how the protocol for Equation (2) is divided into two fragments (framed red and blue, respectively) and how the two fragments share the variable x (by sharing s_x and r_x).

The notion of Schnorr fragments actually closes a general gap in the formalization of Schnorr-style protocols. For example, range proofs are always formalized as "the prover knows *how to open a commitment to* a number in the interval $[A, B]$". This is because the statement "the prover knows a number in the interval $[A, B]$" is trivial on its own — obviously, the prover knows, for example, the number A. However, range proofs based on Schnorr-style protocols almost always work on arbitrary variables in a larger Schnorr proof (e.g., a *signed* number in the interval $[A, B]$). With Schnorr fragments, there is now a convenient way to express this compatibility formally and to enable implementations such as Listing 3.

4.2 A Formal View on Schnorr Fragments

In this section, we formally define Schnorr fragments. Intuitively, a Schnorr fragment is a partial Schnorr-style protocol, where everything regarding external variables is contributed from outside of the fragment. This allows sharing the same variable between multiple fragments. More specifically, the parts that are contributed from outside are the witness value w_{ext}, the announcement randomness rnd_{ext}, and the response r_{ext}. A Schnorr fragment may also internally generate additional announcement randomness as and an additional response r.

Definition 1 *Let $p \in \mathbb{N}$ be a prime. Let $\mathcal{W}$ be a finite (additive) group whose elements have either order 1 or order p (e.g., $(\mathbb{Z}_p^n, +)$). Let $\phi : \mathcal{W} \to \{0, 1\}$ be a predicate. A* Schnorr fragment *for witness space $\mathcal{W}$ and predicate ϕ is an efficient three-message protocol*

$$\begin{array}{lcr}
\textit{Prover } \mathcal{P}(w_{\text{ext}}, rnd_{\text{ext}}) & & \textit{Verifier } \mathcal{V}() \\
 & & \\
as \leftarrow \mathsf{genAnncmntSecret}(w_{\text{ext}}) & & \\
a \leftarrow \mathsf{genAnncmnt}(w_{\text{ext}}, as, rnd_{\text{ext}}) & & \\
 & \xrightarrow{\quad a \quad} & \\
 & \xleftarrow{\quad c \quad} & c \leftarrow \mathbb{Z}_p \\
r \leftarrow \mathsf{genResponse}(w_{\text{ext}}, as, c) & \xrightarrow{\quad r \quad} & \\
 & & \textit{accept if} \\
 & & \mathsf{chkTrnscrpt}(a, c, r, r_{\text{ext}}) = 1 \\
 & & \textit{where } r_{\text{ext}} = c \cdot w_{\text{ext}} + rnd_{\text{ext}}
\end{array}$$

together with an algorithm $(a, c, r) \leftarrow \mathsf{generateSimulatedTranscript}(c, r_{\text{ext}})$.

A Schnorr fragment is correct*, meaning that for all* $w_{\text{ext}}, rnd_{\text{ext}} \in \mathcal{W}$ *with* $\phi(w_{\text{ext}}) = 1$*, all possible transcripts* (a, c, r) *generated by* $\mathcal{P}(w_{\text{ext}}, rnd_{\text{ext}}) \leftrightarrow \mathcal{V}()$ *are accepting, i.e.* $\mathsf{chkTrnscrpt}(a, c, r, r_{\text{ext}}) = 1$ *for* $r_{\text{ext}} = c \cdot w_{\text{ext}} + rnd_{\text{ext}}$*.*

Note that in order for the verifier to check whether a transcript is accepting using $\mathsf{chkTrnscrpt}$, it needs to know $r_{\text{ext}} = c \cdot w_{\text{ext}} + rnd_{\text{ext}}$. We imagine this value to be sent in the response of some wrapping protocol around the fragment (cf. Section 4.3).

Furthermore, note that in a Schnorr fragment, there is no common input for the prover and verifier. `SchnorrFragment` objects are meant to be essentially created with hardcoded common input.

`LinearStatementFragment` is the easiest and perhaps most important example for a Schnorr fragment, which encapsulates the homomorphism preimage capabilities of Schnorr protocols. As such, its transcripts are essentially partial Schnorr transcripts (with the parts concerning (shared) variables cut out).

Example 1 *Let* $\psi : \mathbb{Z}_p^n \to \mathbb{G}$ *be a homomorphism (e.g.,* $\psi(x, a) = g^x \cdot h^a$*) and let* $C \in \mathbb{G}$ *be a constant. The* `LinearStatementFragment` $\Pi_{\psi,C}$ *for* ψ *and* C *works as follows:*

- $\mathsf{genAnncmntSecret}_{\psi,C}(w_{\text{ext}})$ *outputs an empty secret* $\emptyset$*.*
- $\mathsf{genAnncmnt}_{\psi,C}(w_{\text{ext}}, as, rnd_{\text{ext}})$ *outputs the announcement* $a = \psi(rnd_{\text{ext}})$*.*
- $\mathsf{genResponse}_{\psi,C}(w_{\text{ext}}, as, c)$ *outputs an empty response* $r = \emptyset$*.*
- $\mathsf{chkTrnscrpt}_{\psi,C}(a, c, r, r_{\text{ext}})$ *checks that* $\psi(r_{\text{ext}}) = C^c \cdot a$
- $\mathsf{generateSimulatedTranscript}_{\psi,C}(c, r_{\text{ext}})$ *sets* $a = \psi(r_{\text{ext}}) \cdot C^{-c}$*,* $r = \emptyset$ *and outputs the transcript* (a, c, r)*.*

$\Pi_{\psi,C}$ *is a Schnorr fragment for predicate* $\phi(w_{\text{ext}}) = 1 \Leftrightarrow \psi(w_{\text{ext}}) = C$*.*

One special case of this example shows how to implement the fragments for Figure 2. The fragment for $h_i = g_i^x$ is an instantiation of Example 1 with $\psi(x) = g_i^x$. The external variable is $w_{\text{ext}} = x$, its external randomness is $rnd_{\text{ext}} = s_x \leftarrow \mathbb{Z}_p$ and its external response is $r_{\text{ext}} = r_x = x \cdot c + s_x$.

The ability for a fragment to choose its own announcement secret and send something in the response is used for more complex fragments. For example, it allows a fragment to prove knowledge of its own (internal) Schnorr variables. This can be seen in Section 4.5.

We proceed with defining security properties. First, a Schnorr fragment shall be honest-verifier zero-knowledge. This notion is analogous to its Sigma protocol counterpart, but taking external variables into account for simulation.

Definition 2 (Honest-verifier zero-knowledge) *A Schnorr fragment for witness space $\mathcal{W}$ and predicate ϕ is* honest-verifier zero-knowledge *if for all $w_{\text{ext}} \in \mathcal{W}$ with $\phi(w_{\text{ext}}) = 1$ and all (a, c, r), the following two probabilities are the same:*

- $\Pr[(a, c, r) \leftarrow \mathcal{P}(w_{\text{ext}}, rnd_{\text{ext}}) \leftrightarrow \mathcal{V}() \mid \mathcal{V}$ *chooses* $c]$ *where* $rnd_{\text{ext}} \leftarrow \mathcal{W}$
- $\Pr[(a, c, r) \leftarrow \mathsf{generateSimulatedTranscript}(c, r_{\text{ext}})]$ *where* $r_{\text{ext}} \leftarrow \mathcal{W}$

Furthermore, a Schnorr fragment shall ensure properties of proven values. Essentially, this definition says that if the standard Schnorr extractor is applied to (efficiently created) transcripts in order to compute a witness w_{ext}, then w_{ext} must conform to the proven predicate ϕ.

Definition 3 ((Computational) special soundness) *A Schnorr fragment for witness space $\mathcal{W}$ and predicate ϕ has* (computational) special soundness *if it is computationally infeasible*[13] *to find $(a, c, r), (a, c', r')$ and $r_{\text{ext}}, r'_{\text{ext}} \in \mathcal{W}$ such that $c \neq c'$, $\mathsf{chkTrnscrpt}(a, c, r, r_{\text{ext}}) = 1$, and $\mathsf{chkTrnscrpt}(a, c', r', r'_{\text{ext}}) = 1$, but $\phi(w_{\text{ext}}) = 0$ for $w_{\text{ext}} = ((c - c')^{-1} \bmod p) \cdot (r_{\text{ext}} - r'_{\text{ext}})$.*

We say that a Schnorr fragment is *secure* if it is honest-verifier zero-knowledge and has computational special soundness.

4.3 From Schnorr Fragment to Sigma Protocol

The idea is to first compose a Schnorr fragment for desired statements ϕ_x (Section 4.4) from several fragments that share variables w_{ext}, then convert the resulting fragment to a Sigma protocol as follows.

Observation 1 *If for all x, Π_x is a secure Schnorr fragment with predicate $\phi_x : \mathcal{W} \to \{0, 1\}$, then the following is a (computationally sound) Sigma protocol for relation $\{(x, w) \mid \phi_x(w) = 1\}$.*

Prover $\mathcal{P}(x, w)$		*Verifier* $\mathcal{V}(x)$
$w_{\text{ext}} := w$ $rnd_{\text{ext}} \leftarrow \mathcal{W}$ $as \leftarrow \mathsf{genAnncmntSecret}_x(w_{\text{ext}})$ $a \leftarrow \mathsf{genAnncmnt}_x(w_{\text{ext}}, as, rnd_{\text{ext}})$		
	$\xrightarrow{\quad a \quad}$	
	$\xleftarrow{\quad c \quad}$	$c \leftarrow \mathbb{Z}_p$
$r \leftarrow \mathsf{genResponse}_x(w_{\text{ext}}, as, c)$ $r_{\text{ext}} := c \cdot w_{\text{ext}} + rnd_{\text{ext}}$		
	$\xrightarrow{\quad r, r_{\text{ext}} \quad}$	
		accept if $\mathsf{chkTrnscrpt}_x(a, c, r, r_{\text{ext}}) = 1$

Completeness follows immediately from the completeness of the fragment. The simulator for honest-verifier zero-knowledgeness $\mathcal{S}(x, c)$ first chooses random $r_{\text{ext}} \leftarrow \mathcal{W}$ and outputs $(a, c, r) \leftarrow \mathsf{generateSimulatedTranscript}_x(c, r_{\text{ext}})$, which outputs the transcripts with the expected distribution according to Definition 2. Computational soundness ("computational" in the sense that it is hard to find transcripts for which a witness cannot be extracted) follows immediately from the fragment's corresponding property (Definition 3).

[13] For this definition to make sense asymptotically, we imagine that the prime number p is picked appropriately to scale with a security parameter and that prover and verifier have access to honestly generated public parameters. For the sake of simplicity, we leave out these details.

4.4 Composing Schnorr Fragments

The following observation establishes the desired property that Schnorr fragments can be composed over common witnesses.

Observation 2 *Let Π_i be a fragment for $\phi_i : \mathcal{W} \to \{0,1\}$ for all $i \in \{1,\dots,n\}$. Then the following is a fragment for $\phi(w) = \bigwedge_{i=1}^{n} \phi_i(w)$.*

Prover $\mathcal{P}(w_{\text{ext}}, rnd_{\text{ext}})$		Verifier $\mathcal{V}()$
$as_i \leftarrow \mathsf{genAnncmntSecret}_i(w_{\text{ext}})$		
$as = (as_1,\dots,as_n)$		
$a_i \leftarrow \mathsf{genAnncmnt}_i(w_{\text{ext}}, as_i, rnd_{\text{ext}})$		
$a = (a_1,\dots,a_n)$	$\xrightarrow{\quad a \quad}$	
	$\xleftarrow{\quad c \quad}$	$c \leftarrow \mathbb{Z}_p$
$r_i \leftarrow \mathsf{genResponse}_i(w_{\text{ext}}, as_i, c)$		
$r = (r_1,\dots,r_n)$	$\xrightarrow{\quad r \quad}$	
		accept if for all i
		$\mathsf{chkTrnscrpt}_i(a_i, c, r_i, r_{\text{ext}}) = 1$
		where $r_{\text{ext}} = c \cdot w_{\text{ext}} + rnd_{\text{ext}}$

$$\mathsf{generateSimulatedTranscript}(c, r_{\text{ext}}) = (\mathsf{generateSimulatedTranscript}_i(c, r_{\text{ext}}))_{i=1}^{n}$$

4.5 Implementing a Set Membership Schnorr Fragment

As a more advanced example for a Schnorr fragment, consider the famous set membership proof [CCs08] for the statement "$m \in S$" (for hidden m and fixed public S). It works as follows:

- A trusted party computes (weakly secure) Boneh-Boyen [BB04] signatures $\sigma_m = g_1^{1/(sk+m)}$ on messages $m \in S$ and publishes $pk = g_2^{sk}$ and all σ_m.
- The prover with witness m chooses $\alpha \leftarrow \mathbb{Z}_p$, randomizes σ_m as $\sigma' = \sigma_m^{\alpha}$, and sends σ' to the verifier.
- Prover and verifier engage in the proof $\mathrm{ZK}\{(m,\alpha) :\ e(\sigma', pk \cdot g_2^m) = e(g_1, g_2)^{\alpha}\}$ where the prover essentially shows he knows how to derandomize σ' to a valid signature on (hidden) m.

This set membership proof can be implemented as a Schnorr fragment. For this, we first observe that the "inner" proof can be easily handled by the homomorphism-preimage fragment from Example 1, which we will run as a subprotocol $\Pi_{\sigma'}$.

Example 2 *Let S be a (small) set. Assume $pp = (pk, (\sigma_m)_{m \in S})$ has been generated by a trusted third party. For all $\sigma' \in \mathbb{G}$, let $\Pi_{\sigma'}$ be a fragment for $\phi_{\sigma'} : \mathbb{Z}_p^2 \to \{0,1\}$ with $\phi_{\sigma'}(m,\alpha) = 1 \Leftrightarrow e(\sigma', pk \cdot g_2^m) = e(g_1, g_2)^{\alpha}$. Then the following is a Schnorr fragment for $\phi : \mathbb{Z}_p \to \{0,1\}$ with $\phi(m) = 1 \Leftrightarrow m \in S$.*

Prover $\mathcal{P}(w_{\text{ext}}, rnd_{\text{ext}})$ — *Verifier* $\mathcal{V}()$

$m := w_{\text{ext}}$
Set up internal α *and randomness:*
$\alpha \leftarrow \mathbb{Z}_p$
$s_\alpha \leftarrow \mathbb{Z}_p$
Set up randomized σ'*:*
$\sigma' = \sigma_m^{r_\sigma}$
Set up subprotocol external variables:
$w'_{\text{ext}} = (m, \alpha),\ rnd'_{\text{ext}} = (rnd_{\text{ext}}, s_\alpha)$
Send σ' *and subprotocol announcement:*
$as_{\sigma'} \leftarrow \mathsf{genAnncmntSecret}_{\sigma'}(w'_{\text{ext}})$
$as = (s_\alpha, as_{\sigma'})$
$a_{\sigma'} \leftarrow \mathsf{genAnncmnt}_{\sigma'}(w'_{\text{ext}}, as_{\sigma'}, rnd'_{\text{ext}})$
$a = (\sigma', a_{\sigma'})$ $\xrightarrow{\quad a \quad}$
$\xleftarrow{\quad c \quad}$ $c \leftarrow \mathbb{Z}_p$
Make α *extractable:*
$r_\alpha = c \cdot \alpha + s_\alpha$
Send subprotocol response (empty for $\Pi_{\sigma'}$*):*
$r_{\sigma'} \leftarrow \mathsf{genResponse}_{\sigma'}(w'_{\text{ext}}, as_{\sigma'}, c)$
$r = (r_\alpha, r_{\sigma'})$ $\xrightarrow{\quad r \quad}$

accept if
$\mathsf{chkTrnscrpt}_{\sigma'}(a_{\sigma'}, c, r_{\sigma'}, r'_{\text{ext}} = (r_{\text{ext}}, r_\alpha)) = 1$
and $\sigma' \neq 1$
where $r_{\text{ext}} = c \cdot w_{\text{ext}} + rnd_{\text{ext}}$

This example also shows how fragments can be arranged hierarchically, as in this example, the set membership fragment delegates the $\phi_{\sigma'}$ check to an inner fragment. α is considered an internal variable for the set membership fragment, but external for the inner fragment. $m = w_{\text{ext}}$ is considered an external variable for both fragments.

4.6 Other Features and Implementation Considerations

Apart from the modeling of Schnorr-style proofs using fragments, the library supports several other useful constructs regarding Sigma protocols.

Range proofs The library contains a range proof fragment for arbitrary ranges [CCs08].

Protocol optimization We implement several optimizations. For example, accepting Schnorr protocol transcripts can be compressed (i.e. instead of storing (a, c, r), we store only (c, r) and compute the unique a that makes the transcript accepting). This is useful for Fiat-Shamir signatures of knowledge. Furthermore, computation of protocol messages and verification happens concurrently.

Sigma protocol transformations A Schnorr protocol is a Sigma protocol. Our library enables AND and OR composition of Sigma protocols [CDS94], Damgård's transformation to a proper zero-knowledge protocol (in the common reference string model) [Dam00], and the Fiat-Shamir heuristic [FS87] for a non-interactive proof.

5 Zero-Knowledge Compiler

Cryptimeleon offers an even easier method to instantiate Schnorr-style protocols. Instead of coding the protocols using the framework explained in Section 4, one can utilize our compiler *Subzero* to generate the code automatically. For example, $\mathrm{ZK}\{(m_1, m_2, r) : \; C_1 = h_1^{m_1} \cdot h_2^{m_2} \cdot g^r \wedge 20 \leq m_1 + m_2 \leq 100\}$ from Section 4 can be implemented in Subzero with Listing 4, resulting in code equivalent to Listing 3.

List. 4: Implementation of Equation (1) in Subzero

```
witness: m_1, m_2, r
C_1 = h_1^m_1 * h_2^m_2 * g^r & 20 <= m_1 + m_2 <= 100
```

Subzero is a declarative domain-specific language (DSL) for the specification of zero-knowledge proof of knowledge protocols. It uses a concise grammar based on Camenisch-Stadler notation [CS97] to describe a protocol. It compiles to Java code that uses the Cryptimeleon Math and Craco libraries. The language allows for fast prototyping of protocols, provides a higher-level interface for the Cryptimeleon API, and automates much of the boilerplate code that is common across zero-knowledge protocol specifications. Each valid Subzero protocol compiles to a complete Java project (buildable with Gradle) containing the classes necessary to specify the protocol with the Cryptimeleon API, as well as to run the protocol.

The compiler is made publicly accessible through `https://cryptimeleon.org/subzero`, which includes a code editor for the language. The editor supports many of the standard features expected of a development environment, including syntax highlighting, automatic bracket matching and indentation, and syntax error messages. Additionally, a semantic validator provides error messages as a protocol is typed to ensure semantic correctness, providing quicker feedback than raising the errors during compilation. Because the language grammar is similar to mathematical notation, there is also a natural translation from Subzero code to LaTeX text. The website can generate a formatted LaTeX preview in real-time as code is typed, and the LaTeX text used to create this preview can also be downloaded.

A Subzero program specifies a single zero-knowledge protocol. A basic protocol begins with an optional protocol name[14], followed by a list declaring witness variables, and finally a proof expression. User-defined functions[15] are also supported for creating more complex protocols, as well as public parameter variables. Every variable in a Subzero protocol has both an algebraic type and a proof role. The type is either group element or exponent, and is inferred based on the variable's context within the protocol. The role determines the variable's usage within the protocol: it can be a witness, public parameter, or common input variable. Witness and public parameter variables are declared explicitly, whereas any implicitly declared variable becomes common input. This type/role system allows for more readable code and simplifies writing protocols.

The language supports linear exponent statements, linear group statements, range proofs (both single and double inequalities), and pairings. A protocol can be composed of several subprotocols, joined by the conjunction operator `&`. Proofs of partial knowledge are also supported, with subprotocols joined by the disjunctive operator `|`.

As an example, the following denotes a proof of knowledge of a randomized Pointcheval Sanders signature [PS16] $(\sigma_1', \sigma_2') = (\sigma_1^t, (\sigma_2 \cdot \sigma_1^r)^t)$ on attributes *age* and *position* such that either the person is very young or has the position "student" (which we encode as $position = 17$).

$$\begin{aligned} &\mathrm{ZK}\{(age, pos, r) : \\ &\quad e(\sigma_1', \tilde{X}) \cdot e(\sigma_1', \tilde{Y}_1^{age} \cdot \tilde{Y}_2^{pos}) \cdot e(\sigma_1', \tilde{g})^r = e(\sigma_2', \tilde{g}) \; \text{// valid signature} \\ &\quad \wedge \; (age < 18 \vee pos = 17)\} \; \text{// young or student} \end{aligned} \tag{3}$$

The proof's translation into Subzero code is very straightforward and intuitive.

[14] The protocol name is used in naming sal generated Java classes
[15] Functions operate as pure functions, with no side effects

List. 5: Implementation of Equation (3) in Subzero

```
[Pointcheval Sanders signature]
witness: age, pos, r
e(sigma_1', X~) * e(sigma_1', Y_1~ ^ age * Y_2~ ^ pos)
  * e(sigma_1', g~) ^ r = e(sigma_2', g~) // valid signature
& (age < 18 | pos = 17) // young or student
```

The Java code generated for Listing 5 consists of three Sigma protocols arranged in a proof of partial knowledge (to account for the "OR" statement). To ensure that the values of *age* and *pos* are consistent between the OR subproofs (`age < 18` and `pos = 17`) and the signature proof, the generated code contains a Pedersen commitment proof of consistency.

Variable identifiers support special characters such as underscores, tildes, and quotes, which allows for subscripts and common diacritical marks in the LaTeX preview. Identifiers with the name of Greek letters will also be converted to the equivalent symbol. Thus, the LaTeX preview created by the website for the Listing 5 code is nearly identical to Equation (3).

The Subzero compiler was developed using Xtext[16], a framework for creating DSLs and programming languages. It is written in Java and Xtend[17], which is a programming language that transpiles to Java. Both Xtext and Xtend are maintained by the Eclipse Foundation. The code editor makes use of the Ace editor[18].

6 Serialization of Cryptographic Objects

Serialization is the process of converting a Java object into a format that can be stored or sent over the network. When serializing and deserializing cryptographic objects, there are several pitfalls that developers usually need to be aware of. For example, for elliptic curve points, we have to make sure that ...

- ... when serializing, the representation of the point does not leak unwanted information (e.g., the normalization factor in projective coordinates).
- ... when deserializing, the resulting Java object references the correct curve parameters (instead of a false curve with weak parameters).
- ... when deserializing, the point is in the right subgroup.

In cryptographic papers, these concerns are almost never mentioned, so it is especially crucial to abstract them away for the user of Cryptimeleon, too.

For this reason, most cryptographic objects are deserialized *with the help of some parent object.* In the case of a ciphertext, the parent is an encryption scheme, for a group element, the parent is a group. The idea is that these parent objects serve as trust anchors – their task is to instantiate objects from an untrusted serialized format in a way that is secure. For example, a group would check that a point belongs to the right subgroup while deserializing and make sure that the right `GroupElement` object is instantiated.

To keep complexity of writing serialization and deserialization code down, we implemented an intermediate serialization format `Representation`. Representations constitute a convenient Java-object-based format for hierarchical structures. For example, an elliptic curve point's representation contains the representation of a finite field element. Additionally, typical cases of serialization can be easily implemented by simply annotating object variables with an annotation like `@Represented(restorer="parentObjectName")`.

[16] Xtext: https://www.eclipse.org/Xtext/
[17] Xtend: https://www.eclipse.org/xtend/
[18] Ace: https://ace.c9.io/

Acknowledgments

This work was partially supported by the German Research Foundation (DFG) within the Collaborative Research Centre On-The-Fly Computing (GZ: SFB 901/3) under the project number 160364472.

Special thanks to Gennadij Liske, Peter Günther, Mirko Jürgens, Denis Diemert, Swante Scholz, and our student project groups CrACo and Re(AC)^t for their work on early versions of the library. Thanks to Paul Kramer and Patrick Schürmann for their recent contributions and discussions. Thanks to Johannes Blömer for fostering an environment where Cryptimeleon could be created.

References

[ABB+10] José Bacelar Almeida, Endre Bangerter, Manuel Barbosa, Stephan Krenn, Ahmad-Reza Sadeghi, and Thomas Schneider. A certifying compiler for zero-knowledge proofs of knowledge based on sigma-protocols. In Dimitris Gritzalis, Bart Preneel, and Marianthi Theoharidou, editors, *ESORICS 2010*, volume 6345 of *LNCS*, pages 151–167, Athens, Greece, September 20–22, 2010. Springer, Heidelberg, Germany. doi:10.1007/978-3-642-15497-3_10. 8

[ABB+12] José Bacelar Almeida, Manuel Barbosa, Endre Bangerter, Gilles Barthe, Stephan Krenn, and Santiago Zanella Béguelin. Full proof cryptography: verifiable compilation of efficient zero-knowledge protocols. In Ting Yu, George Danezis, and Virgil D. Gligor, editors, *ACM CCS 2012*, pages 488–500, Raleigh, NC, USA, October 16–18, 2012. ACM Press. doi:10.1145/2382196.2382249. 8

[AGM+] D. F. Aranha, C. P. L. Gouvêa, T. Markmann, R. S. Wahby, and K. Liao. RELIC is an Efficient LIbrary for Cryptography. https://github.com/relic-toolkit/relic. 5, 6

[AGM+13] Joseph A. Akinyele, Christina Garman, Ian Miers, Matthew W. Pagano, Michael Rushanan, Matthew Green, and Aviel D. Rubin. Charm: a framework for rapidly prototyping cryptosystems. *Journal of Cryptographic Engineering*, 3(2):111–128, June 2013. doi:10.1007/s13389-013-0057-3. 7, 8

[ASM06] Man Ho Au, Willy Susilo, and Yi Mu. Constant-size dynamic k-TAA. In Roberto De Prisco and Moti Yung, editors, *SCN 06*, volume 4116 of *LNCS*, pages 111–125, Maiori, Italy, September 6–8, 2006. Springer, Heidelberg, Germany. doi:10.1007/11832072_8. 26

[BB04] Dan Boneh and Xavier Boyen. Short signatures without random oracles. In Christian Cachin and Jan Camenisch, editors, *EUROCRYPT 2004*, volume 3027 of *LNCS*, pages 56–73, Interlaken, Switzerland, May 2–6, 2004. Springer, Heidelberg, Germany. doi:10.1007/978-3-540-24676-3_4. 18

[BBDE19a] Johannes Blömer, Jan Bobolz, Denis Diemert, and Fabian Eidens. Updatable anonymous credentials and applications to incentive systems. In Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz, editors, *ACM CCS 2019*, pages 1671–1685. ACM Press, November 11–15, 2019. doi:10.1145/3319535.3354223. 13, 27

[BBDE19b] Johannes Blömer, Jan Bobolz, Denis Diemert, and Fabian Eidens. Updatable anonymous credentials and applications to incentive systems. Cryptology ePrint Archive, Report 2019/169, 2019. https://eprint.iacr.org/2019/169. 13, 27

[BBS04] Dan Boneh, Xavier Boyen, and Hovav Shacham. Short group signatures. In Matthew Franklin, editor, *CRYPTO 2004*, volume 3152 of *LNCS*, pages 41–55, Santa Barbara, CA, USA, August 15–19, 2004. Springer, Heidelberg, Germany. `doi:10.1007/978-3-540-28628-8_3`. 6, 26

[Ben94] Josh Benaloh. Dense probabilistic encryption. In *Proceedings of the Workshop on Selected Areas of Cryptography*, pages 120–128, 1994. 6

[BF01] Dan Boneh and Matthew K. Franklin. Identity-based encryption from the Weil pairing. In Joe Kilian, editor, *CRYPTO 2001*, volume 2139 of *LNCS*, pages 213–229, Santa Barbara, CA, USA, August 19–23, 2001. Springer, Heidelberg, Germany. `doi:10.1007/3-540-44647-8_13`. 27

[BLS04] Dan Boneh, Ben Lynn, and Hovav Shacham. Short signatures from the Weil pairing. *Journal of Cryptology*, 17(4):297–319, September 2004. `doi:10.1007/s00145-004-0314-9`. 6, 7

[BLS12] Daniel J. Bernstein, Tanja Lange, and Peter Schwabe. The security impact of a new cryptographic library. In Alejandro Hevia and Gregory Neven, editors, *LATINCRYPT 2012*, volume 7533 of *LNCS*, pages 159–176, Santiago, Chile, October 7–10, 2012. Springer, Heidelberg, Germany. 5, 6

[BvJ+15] Daniel J. Bernstein, Bernard van Gastel, Wesley Janssen, Tanja Lange, Peter Schwabe, and Sjaak Smetsers. TweetNaCl: A crypto library in 100 tweets. In Diego F. Aranha and Alfred Menezes, editors, *LATINCRYPT 2014*, volume 8895 of *LNCS*, pages 64–83, Florianópolis, Brazil, September 17–19, 2015. Springer, Heidelberg, Germany. `doi:10.1007/978-3-319-16295-9_4`. 6

[CCs08] Jan Camenisch, Rafik Chaabouni, and abhi shelat. Efficient protocols for set membership and range proofs. In Josef Pieprzyk, editor, *ASIACRYPT 2008*, volume 5350 of *LNCS*, pages 234–252, Melbourne, Australia, December 7–11, 2008. Springer, Heidelberg, Germany. `doi:10.1007/978-3-540-89255-7_15`. 14, 18, 19, 27

[CDS94] Ronald Cramer, Ivan Damgård, and Berry Schoenmakers. Proofs of partial knowledge and simplified design of witness hiding protocols. In Yvo Desmedt, editor, *CRYPTO'94*, volume 839 of *LNCS*, pages 174–187, Santa Barbara, CA, USA, August 21–25, 1994. Springer, Heidelberg, Germany. `doi:10.1007/3-540-48658-5_19`. 19, 27

[CPY06] Seung Geol Choi, Kunsoo Park, and Moti Yung. Short traceable signatures based on bilinear pairings. In Hiroshi Yoshiura, Kouichi Sakurai, Kai Rannenberg, Yuko Murayama, and Shin ichi Kawamura, editors, *IWSEC 06*, volume 4266 of *LNCS*, pages 88–103, Kyoto, Japan, October 23–24, 2006. Springer, Heidelberg, Germany. 27

[CS97] Jan Camenisch and Markus Stadler. Proof systems for general statements about discrete logarithms. Technical report, 1997. 14, 20

[Dam00] Ivan Damgård. Efficient concurrent zero-knowledge in the auxiliary string model. In Bart Preneel, editor, *EUROCRYPT 2000*, volume 1807 of *LNCS*, pages 418–430, Bruges, Belgium, May 14–18, 2000. Springer, Heidelberg, Germany. `doi:10.1007/3-540-45539-6_30`. 19, 27

[DAR15] Jesus Diaz, David Arroyo, and Francisco B. Rodriguez. libgroupsig: An extensible C library for group signatures. Cryptology ePrint Archive, Report 2015/1146, 2015. `https://eprint.iacr.org/2015/1146`. 4

[DDG+20] Fynn Dallmeier, Jan P. Drees, Kai Gellert, Tobias Handirk, Tibor Jager, Jonas Klauke, Simon Nachtigall, Timo Renzelmann, and Rudi Wolf. Forward-secure 0-RTT goes live: Implementation and performance analysis in QUIC. In Stephan Krenn, Haya Shulman, and Serge Vaudenay, editors, *CANS 20*, volume 12579 of *LNCS*, pages 211–231, Vienna, Austria, December 14–16, 2020. Springer, Heidelberg, Germany. doi:10.1007/978-3-030-65411-5_11. 6

[Deca] Decentralized and Distributed Systems Research Lab at EPFL. Cothority. https://github.com/dedis/cothority. 7

[Decb] Decentralized and Distributed Systems Research Lab at EPFL. Kyber. https://github.com/dedis/kyber. 7

[EFLL12] Yael Ejgenberg, Moriya Farbstein, Meital Levy, and Yehuda Lindell. SCAPI: The secure computation application programming interface. Cryptology ePrint Archive, Report 2012/629, 2012. https://eprint.iacr.org/2012/629. 8

[FHS19] Georg Fuchsbauer, Christian Hanser, and Daniel Slamanig. Structure-preserving signatures on equivalence classes and constant-size anonymous credentials. *Journal of Cryptology*, 32(2):498–546, April 2019. doi:10.1007/s00145-018-9281-4. 26

[FKDL13] Cédric Fournet, Markulf Kohlweiss, George Danezis, and Zhengqin Luo. ZQL: A compiler for privacy-preserving data processing. In Samuel T. King, editor, *USENIX Security 2013*, pages 163–178, Washington, DC, USA, August 14–16, 2013. USENIX Association. 8

[FS87] Amos Fiat and Adi Shamir. How to prove yourself: Practical solutions to identification and signature problems. In Andrew M. Odlyzko, editor, *CRYPTO'86*, volume 263 of *LNCS*, pages 186–194, Santa Barbara, CA, USA, August 1987. Springer, Heidelberg, Germany. doi:10.1007/3-540-47721-7_12. 13, 19, 27

[GPSW06] Vipul Goyal, Omkant Pandey, Amit Sahai, and Brent Waters. Attribute-based encryption for fine-grained access control of encrypted data. In Ari Juels, Rebecca N. Wright, and Sabrina De Capitani di Vimercati, editors, *ACM CCS 2006*, pages 89–98, Alexandria, Virginia, USA, October 30 – November 3, 2006. ACM Press. Available as Cryptology ePrint Archive Report 2006/309. doi:10.1145/1180405.1180418. 27

[Leg] Legion of the Bouncy Castle Inc. Bouncy Castle Crypto APIs. https://www.bouncycastle.org/java.html. 5

[liba] libhydrogen - Hydrogen: A lightweight, secure, easy-to-use crypto library suitable for constrained environments. https://github.com/jedisct1/libhydrogen. 6

[libb] libsodium - Sodium: A modern, portable, easy to use crypto library. https://github.com/jedisct1/libsodium. 5, 6

[LKF+19] Wouter Lueks, Bogdan Kulynych, Jules Fasquelle, Simon Le Bail-Collet, and Carmela Troncoso. zksk: A library for composable zero-knowledge proofs. In *Proceedings of the 18th ACM Workshop on Privacy in the Electronic Society (WPES@CCS)*, pages 50–54, 2019. 8

[Mau09] Ueli M. Maurer. Unifying zero-knowledge proofs of knowledge. In Bart Preneel, editor, *AFRICACRYPT 09*, volume 5580 of *LNCS*, pages 272–286, Gammarth, Tunisia, June 21–25, 2009. Springer, Heidelberg, Germany. 13

[MEK+10] Sarah Meiklejohn, C. Christopher Erway, Alptekin Küpçü, Theodora Hinkle, and Anna Lysyanskaya. ZKPDL: A language-based system for efficient zero-knowledge proofs and electronic cash. In *USENIX Security 2010*, pages 193–206, Washington, DC, USA, August 11–13, 2010. USENIX Association. 8

[Mic] Microsoft Research, Carnegie Mellon University, INRIA, MSR-INRIA joint center. Project Everest. `https://project-everest.github.io`. 6

[Möl01] Bodo Möller. Algorithms for multi-exponentiation. In Serge Vaudenay and Amr M. Youssef, editors, *SAC 2001*, volume 2259 of *LNCS*, pages 165–180, Toronto, Ontario, Canada, August 16–17, 2001. Springer, Heidelberg, Germany. `doi:10.1007/3-540-45537-X_13`. 9

[Ngu05] Lan Nguyen. Accumulators from bilinear pairings and applications. In Alfred Menezes, editor, *CT-RSA 2005*, volume 3376 of *LNCS*, pages 275–292, San Francisco, CA, USA, February 14–18, 2005. Springer, Heidelberg, Germany. `doi:10.1007/978-3-540-30574-3_19`. 26

[Pai99] Pascal Paillier. Public-key cryptosystems based on composite degree residuosity classes. In Jacques Stern, editor, *EUROCRYPT'99*, volume 1592 of *LNCS*, pages 223–238, Prague, Czech Republic, May 2–6, 1999. Springer, Heidelberg, Germany. `doi:10.1007/3-540-48910-X_16`. 6

[PBP+20] Marina Polubelova, Karthikeyan Bhargavan, Jonathan Protzenko, Benjamin Beurdouche, Aymeric Fromherz, Natalia Kulatova, and Santiago Zanella-Béguelin. HACLxN: Verified generic SIMD crypto (for all your favourite platforms). In Jay Ligatti, Xinming Ou, Jonathan Katz, and Giovanni Vigna, editors, *ACM CCS 20*, pages 899–918, Virtual Event, USA, November 9–13, 2020. ACM Press. `doi:10.1145/3372297.3423352`. 5, 6

[Ped92] Torben P. Pedersen. Non-interactive and information-theoretic secure verifiable secret sharing. In Joan Feigenbaum, editor, *CRYPTO'91*, volume 576 of *LNCS*, pages 129–140, Santa Barbara, CA, USA, August 11–15, 1992. Springer, Heidelberg, Germany. `doi:10.1007/3-540-46766-1_9`. 26

[PS16] David Pointcheval and Olivier Sanders. Short randomizable signatures. In Kazue Sako, editor, *CT-RSA 2016*, volume 9610 of *LNCS*, pages 111–126, San Francisco, CA, USA, February 29 – March 4, 2016. Springer, Heidelberg, Germany. `doi:10.1007/978-3-319-29485-8_7`. 6, 7, 20, 26

[PS18] David Pointcheval and Olivier Sanders. Reassessing security of randomizable signatures. In Nigel P. Smart, editor, *CT-RSA 2018*, volume 10808 of *LNCS*, pages 319–338, San Francisco, CA, USA, April 16–20, 2018. Springer, Heidelberg, Germany. `doi:10.1007/978-3-319-76953-0_17`. 11, 12, 26

[SBM+19] Miha Stopar, Manca Bizjak, Jolanda Modic, Jan Hartman, Anze Zitnik, and Tilen Marc. emmy - trust-enhancing authentication library. In Weizhi Meng, Piotr Cofta, Christian Damsgaard Jensen, and Tyrone Grandison, editors, *Trust Management XIII - 13th IFIP WG 11.11 International Conference, IFIPTM 2019, Copenhagen, Denmark, July 17-19, 2019, Proceedings*, volume 563 of *IFIP Advances in Information and Communication Technology*, pages 133–146. Springer, 2019. URL: `https://doi.org/10.1007/978-3-030-33716-2_11`, `doi:10.1007/978-3-030-33716-2_11`. 8

[Sec] Secure Information and Communication Technologies (SIC), Institute for Applied Information Processing and Communication (IAIK) of Graz University of Technology. ECCelerate: Core crypto toolkits. `https://jce.iaik.tugraz.at/products/core-crypto-toolkits/eccelerate`. 4, 5

[Sha79] Adi Shamir. How to share a secret. *Communications of the Association for Computing Machinery*, 22(11):612–613, November 1979. 27

[Shi] MITSUNARI Shigeo. mcl: A portable and fast pairing-based cryptography library. `https://github.com/herumi/mcl`. 4, 5, 9

[STV+16] Ewa Syta, Iulia Tamas, Dylan Visher, David Isaac Wolinsky, Philipp Jovanovic, Linus Gasser, Nicolas Gailly, Ismail Khoffi, and Bryan Ford. Keeping authorities "honest or bust" with decentralized witness cosigning. In *2016 IEEE Symposium on Security and Privacy*, pages 526–545, San Jose, CA, USA, May 22–26, 2016. IEEE Computer Society Press. `doi:10.1109/SP.2016.38`. 7

[SW05] Amit Sahai and Brent R. Waters. Fuzzy identity-based encryption. In Ronald Cramer, editor, *EUROCRYPT 2005*, volume 3494 of *LNCS*, pages 457–473, Aarhus, Denmark, May 22–26, 2005. Springer, Heidelberg, Germany. `doi:10.1007/11426639_27`. 27

[Wat08] Brent Waters. Ciphertext-policy attribute-based encryption: An expressive, efficient, and provably secure realization. Cryptology ePrint Archive, Report 2008/290, 2008. `http://eprint.iacr.org/2008/290`. 7, 27

[Wat11] Brent Waters. Ciphertext-policy attribute-based encryption: An expressive, efficient, and provably secure realization. In Dario Catalano, Nelly Fazio, Rosario Gennaro, and Antonio Nicolosi, editors, *PKC 2011*, volume 6571 of *LNCS*, pages 53–70, Taormina, Italy, March 6–9, 2011. Springer, Heidelberg, Germany. `doi:10.1007/978-3-642-19379-8_4`. 7, 27

[ZBPB17] Jean Karim Zinzindohoué, Karthikeyan Bhargavan, Jonathan Protzenko, and Benjamin Beurdouche. HACL*: A verified modern cryptographic library. In Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *ACM CCS 2017*, pages 1789–1806, Dallas, TX, USA, October 31 – November 2, 2017. ACM Press. `doi:10.1145/3133956.3134043`. 5, 6, 7

A Implemented Schemes

In the following we list the concrete schemes that are implemented in Cryptimeleon.

- **Accumulators**:
  - Nguyen's dynamic accumulator [Ngu05]
- **Commitment schemes**:
  - Pedersen's commitment scheme [Ped92]
- **Digital signature schemes**:
  - Pointcheval's & Sanders' short randomizable signature scheme [PS16] and the variant secure under SDH [PS18].
  - The BBS+ signature [ASM06] (a multi-message variant of BBS signatures [BBS04]).
  - Structure-preserving signatures on equivalence classes [FHS19].
- **Encryption schemes**:
  - ElGamal
  - Attribute-based:

        * Waters' ciphertext-policy attribute-based encryption scheme [Wat11, Wat08].
        * Goyal et al.'s key-policy attribute-based encryption scheme [GPSW06]
    - Identity-based:
        * Fuzzy identity-based encryption [SW05]
        * Identity based encryption from the Weil pairing [BF01]

- **Key encapsulation mechanisms (KEM)**: We implemented KEMs based on the encryption schemes of this library, e.g. KEMs for [Wat11, GPSW06, SW05] and ElGamal.

- **Secret sharing schemes**:
    - Shamir's secret sharing scheme [Sha79] and its tree extension

- **Zero-knowledge proof of knowledge**:
    - Generalized Schnorr proofs (Section 4).
    - Range proofs [CCs08].
    - Proofs of partial knowledge [CDS94].
    - Damgård's transformation [Dam00] and the Fiat-Shamir heuristic [FS87]

- **Group signature scheme**:
    - Traceable group signature scheme [CPY06].

- **Incentive system schemes**:
    - Incentive system based on updatable anonymous credentials [BBDE19a, BBDE19b].