SIGMA-RS: A Modular Approach for Keyed-Verification Anonymous Credentials

Michele Orrù
CNRS

Lindsey Tulloch
Tor Project, University of Waterloo

Victor Snyder-Graf
RISC Zero

Ian Goldberg
University of Waterloo

Abstract

We introduce a new software stack in Rust aimed at simplifying constructions and deployments of protocols based on modern anonymous credential systems.

The stack, called SIGMA-RS, through its layered design, abstracts cryptographic complexity while remaining flexible enough to support a range of credential schemes, proofs, and access policies. It emphasizes misuse resistance via type safety, domain separation, and prover-state discipline, and supports side-channel-aware constant-time strategies.

We evaluate practicality through re-implementations of Tor's Lox bridge distribution protocols and of user authentication in the Open Observatory for Network Interference.

1 Introduction

Anonymous credentials (AC), originally introduced by David Chaum [23], augment classical digital signatures with privacy: they allow a user to obtain certified attributes from an issuer and later prove statements about those attributes while minimizing information leakage. Keyed-verification credentials [22] (KVAC) refine this model by restricting verification to designated verifiers holding a secret key: instead of publicly verifiable proofs, credentials are checked using verifier-specific keys, yielding compact and efficient protocols while preserving the essential privacy properties of ACs. Restricting the model to a self-contained environment with only two parties (an issuer and a client) makes keyed-verification credentials a fast and stable primitive, particularly appealing for real-world deployments where access control, rate limiting, and pseudonymous authentication are required.

In recent years, keyed-verification credentials have attracted significant attention in real-world deployments, particularly on the web. In the last two years alone, seven Internet standards [42, 50, 51, 57, 59, 78, 88] focusing on anonymous credential systems have been proposed to the Internet Engineering Task Force (IETF), the body responsible for the technical standards on the Internet. They have been proposed as solutions for digital identity [33], replacements for third-party cookies [21, 55], and rate limiting [8]. These primitives are not only being adopted by large technology companies to strengthen privacy guarantees, but are also relied upon by open-source and free software initiatives such as Signal [81] and Tor [86] to preserve user privacy.

These proposals often re-create the same credential system, the same interactive proof, and re-implement the Fiat–Shamir transformation. A number of problems arise from this:

(i) *Duplicated effort*. Formalizations, security analyses, and implementations that could be shared are duplicated, at the expense of additional effort. Key implementation techniques (for performance, side-channel resistance, secure message encoding, etc.) are scattered across the literature, rarely captured in specifications, and inconsistently included in implementations.

(ii) *Attack surface*. The attack surface increases, and vulnerabilities that commonly appear in these protocols are likely to be repeated.

(iii) *Cryptographic agility*. Tight coupling of application logic within the cryptographic primitives makes transitions harder. Transitioning to credentials with post-quantum privacy is already of immediate concern [20].

1.1 Contribution

We propose a new software stack for keyed-verification credential protocols, consisting of the following layers:

- CMZ, a macro system for specifying anonymous authentication protocols via the CMZ [22] and μCMZ [68] KVAC schemes. The protocols are converted into specifications of some zero-knowledge proof statements.
- SIGMA-COMPILER, a macro system that converts zero-knowledge proof statements into Σ-protocol relations.
- SIGMA-PROOFS, a library that implements interactive and non-interactive Σ-protocols and compressed Σ-

```
// Credential definition
CMZ! { ArcCredential: m1 }

// Issuance protocol (m1 is hidden)
muCMZProtocol! { issue, ,
    Cred: ArcCredential { m1: H },
}

// Presentation protocol
muCMZProtocol! { present<nonce, @TAG, @G>,
    Cred: ArcCredential { m1: H }, ,
    G = (Cred.m1 + nonce) * TAG
}
```

```
// Credential definition
CMZ! { NymCredential: nym_secret }

// Issuance protocol (nym_secret is jointly computed)
muCMZProtocol! { issue, ,
    Cred: NymCredential { nym_secret: J },
}

// Presentation protocol
muCMZProtocol! { present<@NYM, @D>,
    Cred: NymCredential { nym_secret: H }, ,
    NYM = Cred.nym_secret * D
}
```

Figure 1: The credential systems described in the Internet Drafts`draft-yun-cfrg-arc` [88] and `draft-irtf-cfrg-bbs-per-verif-link` [50] (adapted for keyed-verification), in SIGMA-RS. The syntax is discussed in more detail in Section 7 and Appendix C.

protocols for linear relations, providing generic AND, OR, and threshold compositions in constant time.

- SPONGEFISH, a library that implements a range of Fiat–Shamir transformations for different settings.

We evaluate our stack's integration in two FLOSS projects: the Tor bridge distribution system and the Open Observatory of Network Interference. In Figure 1 we showcase how the stack can be used to simplify existing specifications.

The artifact accompanying this paper is released as open-source software and is publicly available at https://git-crysp.uwaterloo.ca/SigmaProtocol/sigma-rs-artifact.

1.2 Related work

Modular approaches at credential systems are well-known to theoreticians. The seminal work of Camenisch and Lysyanskaya [17] built core anonymous credential systems by composing signature schemes with zero-knowledge proofs that can speak the same algebra, and later Chase et al. [22] extended this approach to message authentication codes (MACs) for keyed-verification credentials. This core framework was later extended to support access policies: Camenisch et al. [16] provided a modular framework for generic access policies in publicly verifiable credentials, with modules such as selective disclosure, revocation, and pseudonyms; Orrù [68] brought this modular framework to keyed-verification credentials.

Anonymous Credential systems. Anonymous credential systems in the literature are for the most part monolithic [58], and so are many Internet Drafts [50, 51, 78]; often witness generation, the zero-knowledge proof system, and the Fiat–Shamir transformation are bundled in a single procedure.

Two notable exceptions are Dock Network's crypto library [56] and Emmy [87] (in Go, now abandoned), which experiment with implementing multiple cryptographic protocols and anonymous credential systems. In contrast, we provide a complete, production-oriented stack with a concise language for specifying application-level protocols, a standalone Fiat–Shamir layer designed for composition across proof systems, and explicit support for deployment requirements such as constant-time execution, standardized message formats, and widely used elliptic-curve implementations.

Σ-protocols. Most frameworks built around Σ-protocols are built with modularity in mind. de Valence [31, 32] built a zero-knowledge proof system for Maurer proofs; libscapi [47] uses them as a component for multi-party computation protocols; zksk [60] builds generic Σ-protocols with OR- and AND- composition. The idea of a compiler for linear statements was also previously investigated by Bangerter et al. [5]. We augment all of the above with compressed Σ-protocols, threshold composition, and a macro system that compiles complex statements into linear relations. The relations generated are compatible with the Σ-protocol and Fiat–Shamir Internet Drafts [69, 70].

Fiat–Shamir. The Fiat–Shamir transformation is often hard-coded within proof systems; however, there do exist libraries built around the idea of a stateful hash object that is then used by an interactive proof system. Notable examples are Signal's POKSHO [82], which focuses on compression functions; the SAFE framework [4], which focuses on algebraic permutation functions; and Hamburg's STROBE framework [48], which focuses on binary permutation functions. SPONGEFISH takes this forward by integrating different approaches, generic hash functions, circuit building, and grinding, while maintaining compatibility with the IETF specification [69].

2 Preliminaries

Commitment schemes. A commitment scheme lets one party commit to a value while keeping it hidden, yet guaranteeing it cannot be changed later. Let $\mathbb{G}$ be an additive group

of prime order p where discrete logarithm is hard, and G is a generator. Pedersen commitments [72] allow committing to a value $x \in \mathbb{Z}_p$ by sampling a *randomness variable* $r \leftarrow\!\$\, \mathbb{Z}_p$ and setting $C := xG + rH$, where G, H are *computationally independent* generators (group generators whose respective discrete logarithms are not known to the committer). We say that C is a Pedersen commitment, and (x, r) its opening, via which it is possible to check that C was indeed a commitment to x. Pedersen commitments have the important property that if the committer proves knowledge of x, y, r, s such that both $C = xG + rH$ and $C = yG + sH$, then it must be the case that $x = y$ and $r = s$.

Σ-protocols. We briefly recap Σ-protocols from Cramer [26] as described in Boneh–Shoup [10, §19.4]. The prover begins by sending a **commitment** message to the verifier. This message is generated using randomness and may internally store some state that will be needed later. The verifier responds with a random **challenge** chosen uniformly from a fixed challenge space. The prover computes a **response** using the witness, the previously stored state, and the challenge. Finally, the verifier checks the full **transcript** (commitment, challenge, and response) and outputs an accept or reject decision. A transcript is valid if it is accepted by the verifier.

Σ-protocols enjoy two security properties: special soundness (which guarantees that a witness must have been known by the prover if the transcript is valid), and honest-verifier zero-knowledge (which guarantees that no information about the witness is revealed).

Two well-known examples of Σ-protocols used in our construction are Schnorr's protocol [79], which proves knowledge of a discrete logarithm and Okamoto's protocol [65], for proving knowledge of the opening of a Pedersen commitment. These protocols and their associated extractors are standard and well documented in the literature.

Although Σ-protocols are interactive public-coin proofs, in practice they are almost always made non-interactive using the Fiat–Shamir transformation, which replaces the verifier's random challenge with the output of a cryptographic hash function, resulting in a single message, called the **non-interactive argument string**, that will be able to convince any verifier.

Keyed-Verification Credentials. A keyed-verification credential system involves two parties: an issuer and a client. The issuer generates a secret verification key together with public parameters that define the system. The public parameters are used by clients when interacting with the issuer, while the secret key is required to issue and verify credentials. The system involves two main protocols: issuance and presentation.

In the issuance protocol, the client interacts with the issuer to obtain a credential on a vector of attributes. The client inputs their attributes, and the issuer the secret key. At the end of the protocol execution, the client obtains a credential for their attributes, valid under the secret key. No information about the secret key is revealed to the client. Client attributes can be partially hidden, or jointly computed with the issuer.

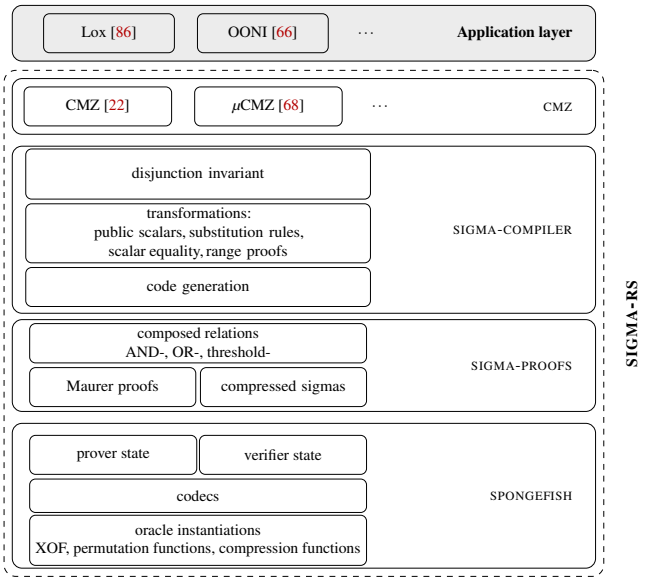


Figure 2: A high-level view of our software stack, highlighting the application layer and the constituent crates. The dashed outline groups the credential, compiler, proof and Fiat–Shamir layers into the main library.

In the presentation protocol, the client proves to the issuer that they possess a valid credential whose underlying attributes satisfy a given predicate. The client reveals no additional information about the attributes beyond the fact that the predicate holds. The issuer uses its secret key to verify the proof and outputs a single accept/reject bit.

A secure keyed-verification credential system must satisfy three high-level properties. *Correctness* ensures that honestly issued credentials can always be successfully presented for any predicate that the underlying attributes satisfy. *Anonymity* guarantees that protocol interactions with an honest client do not reveal the client's attributes, except for what is logically implied by the issuance and presentation predicate. *Extractability* means that for any successful issuance or presentation protocol, the client really must have had possession of a valid credential for those attributes. We point the curious reader to Orrù [68] for a formal, game-based treatment of these definitions.

3 Architecture and system design

Figure 2 summarizes the stack and the section layout that follows: each layer corresponds to a crate, and the paper proceeds from the bottom of the stack upward. At the base, the Spongefish/Fiat–Shamir layer provides prover and verifier state machines, codecs, and oracle instantiations over standard hash or permutation primitives. The sigma-proofs layer builds on this to compose protocols (AND-, OR-, and

threshold relations) and to offer compressed Σ-protocols. The sigma-compiler layer enforces the disjunction invariant, applies a sequence of invariant-preserving transformations (public scalars, substitution rules, scalar equality, and range proofs), and produces code for concrete protocols. On top, the credentials layer instantiates concrete schemes such as CMZ14 and μCMZ, which are then consumed by applications like Lox and OONI outside the SIGMA-RS crate boundary.

This layered structure keeps each crate independently usable while supporting a clean integration path: applications can depend on the credential layer, or reuse lower-level components directly, and the compiler and proof layers remain available for protocols beyond credentials.

4 The SPONGEFISH crate

The Fiat–Shamir transformation turns a public-coin interactive proof into a non-interactive argument by replacing verifier challenges with a cryptographic hash output. Since its introduction [40], the term has come to encompass a family of heuristics rather than a single, precisely defined primitive.

In the proof-systems literature, the *canonical* Fiat–Shamir transformation defines the i-th challenge as the output of a random oracle queried on the instance $\mathbb{x}$ and all prover messages $\alpha_1, \ldots, \alpha_i$ sent so far. Although this transformation is proven secure in the random oracle model and widely used in theory, it is rarely implemented directly in practice.[1]

In practice, engineers do not have access to ideal random oracles over arbitrary domains, but instead rely on standard primitives such as compression functions (e.g., SHA-256 [63], BLAKE2 [75]) and permutation functions (e.g., SHA-3 [64], Poseidon [45, 46]).

While programming languages expose safe APIs for fixed- and variable-length hash outputs (e.g., `digest::Digest` and `digest::ExtensibleOutput` in Rust), a substantial gap remains between these interfaces and a correct Fiat–Shamir instantiation. The latter requires managing prover and verifier state and securely encoding and serializing protocol messages into the hash domain. These steps are typically left to the implementer, and ad-hoc solutions in deployed systems have led to numerous vulnerabilities, as surveyed in Appendix A.

Multiple Fiat–Shamir transformations. SPONGEFISH provides the following Fiat–Shamir transformations:

- a transformation from extendable-output functions (XOFs), which implements the canonical XOF Fiat–Shamir transformation. This is instantiated for SHAKE-128 [64] and is the simplest, most immediate transformation that matches the ongoing RFC [69];
- a transformation from generic permutations (over any domain), via the Duplex Sponge Fiat–Shamir transformation [24]. This transformation can be particularly useful in light of more recent algebraic hash functions that are "friendly" to zero-knowledge circuits;
- a transformation from "legacy" hash functions inteфaces that absorb arbitrary-length inputs and output fixed-length digests, via the POKSHO transformation [82].

The library defines an interface for generic absorb/squeeze operations called `DuplexSpongeInterface`, implemented by all above transformations (and potential custom, novel ones). The interface supports custom prover and verifier state.

Mapping to/from the hash function. Providing a general interface for any hash function, over any domain, highlights another issue often disregarded in both academic papers as well as concrete implementations: prover messages must be encoded into the hash function domain securely, and verifier messages must be mapped to the challenge domain preserving the challenge distribution.

SPONGEFISH provide two interfaces for mapping complex types to/from the hash function domain: `Encoding<T>` maps a type `T` into a sequence of elements in the alphabet of the hash functions (bytes or field elements; any type with a zero element and size known at compile time), and `Decoding<U>` maps a set of hash function outputs into a type `U`. `Encoding` implementations must provide a prefix-free injective map into the random oracle domain, and `Decoding` implementations must preserve the uniform distribution. These requirements cannot be enforced at the type system level (as Rust's compiler is not powerful enough), but they are scoped and highlighted as they are crucial for soundness and honest-verifier zero-knowledge.

Encoding and malleability. Incorrect or non-injective encodings of prover messages in the Fiat–Shamir transformation can lead to malleable proofs and concrete soundness failures. For example, in DSA-style constructions, distinct commitment messages K and $-K$ map to the same hash input, resulting in identical challenges and enabling malleability attacks. Similarly, the reference implementation of hash-to-curve [38] does not enforce canonical representatives, leading to proofs where multiple encodings of the same group element yield the same challenge. Such issues have been exploited in the past. [1]

To prevent these attacks, the function used to absorb messages into the Fiat–Shamir transcript must be injective and prefix-free, and this is enforced by the `Encoding` trait.[2]

While verifier challenges in the interactive protocol are uniform elements of $\mathbb{Z}_p$, hash functions output uniformly distributed bit strings. Correct decoding from a hash output to the challenge space is done via the `Decoding` interface.

`Encoding` and `Decoding` are implemented for all built-in integer types and byte arrays; with specific feature flags, we also support algebraic objects from common algebraic li-

[1] The limitations of Fiat–Shamir are well known [6, 43]. Recent work derives Fiat–Shamir-like transformations from standard assumptions using correlation-intractable hash functions [18], but these constructions remain impractical. We therefore focus on random oracle instantiations.

[2] In the literature, this property is referred to as *simulation extractability* [76] for proof systems and *strong unforgeability* [36] for signatures.

braries such as `p3-koala-bear`, `p3-baby-bear`, `risc0-ff`, `cruve25519-dalek`, `k256`, `ark-ff`, `pallas`, `bls12-381`, and more. Whenever different libraries re-implement the same object, we add interoperability tests.

Prover and verifier state. The non-interactive prover and verifier are responsible not only for deriving public coins, but also for serializing and parsing the non-interactive argument string.

In practice, several Fiat–Shamir vulnerabilities have stemmed from inconsistencies between the serialized proof and the transcript absorbed by the hash function [25,41]. Common implementations store prover messages in a *Proof* structure while separately hashing them, making it easy to omit messages or hash them in the wrong order. Such mistakes have led to concrete soundness failures [41] and ordering attacks identified by Ciobotaru et al. [25].

In SPONGEFISH, the prover state owns both the public coins and proof serialization, while the verifier state mirrors this by parsing the argument string from the same transcript, preventing these classes of bugs at the API level.

Domain separation and cross-protocol attacks. In addition to the prover messages, the Fiat–Shamir transformation requires three additional components: the *protocol identifier*, that uniquely identifies the non-interactive protocol being built (e.g., "`sigma-proofs_Shake128_P256`"); the *session identifier*, corresponding to session and sub-session identifiers in universal composability (e.g., the URL and timestamp "`ooni.org/userauth 723310200`"); and the *instance* being proven (e.g., the linear relation being proven). The developer is in charge of making sure they are chosen appropriately, but SPONGEFISH provides a helper struct `DomainSeparator` that allows the user to input strings for each of them, and processes them into a format compatible with the IETF specification [69].

5 The SIGMA-PROOFS crate

In his PhD thesis, Cramer [26] introduced the notion of Σ-protocols as a 3-message, special sound and special honest-verifier zero-knowledge proof system. Maurer proofs [26,61] are an elegant Σ-protocol for proving knowledge of the pre-image of a group morphism $\mathbf{M} : \mathbb{Z}_p^n \to \mathbb{G}^m$:

$$\mathcal{R} := \left\{ (\vec{x}, \vec{X}) : \mathbf{M} \cdot \vec{x} = \vec{X} \right\} \quad . \tag{1}$$

These protocols have very strong security properties: they are unconditionally special sound and statistically zero-knowledge which can be seen as proving knowledge of $(x_1, \ldots, x_n)$ in a *set* of equations jointly proven together:

$$\begin{aligned} X_1 &= x_1 \cdot \mathbf{M}_{1,1} + \cdots + x_n \cdot \mathbf{M}_{1,n} \quad , \\ &\vdots \\ X_m &= x_m \cdot \mathbf{M}_{m,1} + \cdots + x_n \cdot \mathbf{M}_{m,n} \quad . \end{aligned}$$

Schnorr relations are a trivial example with $n = m = 1$ and $\mathbf{M} = [G]$, for proving knowledge of x such that $X = xG$. More involved linear relations are possible:

Example 1 (Chaum–Pedersen)**.** *In Chaum–Pedersen the prover shows knowledge of x, r, s such that C and D are Pedersen commitments to the same x under independent randomness. In this case, the morphism can be represented as:*

$$C = xG + rH \quad , \qquad D = xG + sH \quad .$$

Variants of the above have been widely used in the VRF literature (Jarecki et al. [49]), and the anonymous token literature (Davidson et al. [30, 71]).

Other more complex relations, such as affine relations with constant terms or quadratic equations, can still be reduced to this form. In fact, satisfiability of any arithmetic circuit (and thus any NP relation) can be proven with this relation family.

Example 2 (Single-bit range proof)**.** *In a single-bit range proof, the prover shows knowledge of a Pedersen commitment $C = bG + rH$ to a bit b. This can be done via the Chaum–Pedersen relation, proving:*

$$C = bG + rH \quad , \qquad C = bC + sH \quad .$$

In fact, substituting the second equation into the first one yields $C = bC + sH = b(bG + rH) + sH = b^2G + (br + s)H$ and so $b = b^2 \implies b \in \{0, 1\}$.

From an implementation perspective, the group element matrices are often sparse, so we use a Yale/CSR-style representation [13] instead of dense matrices. In SIGMA-PROOFS, the relation is expressed in terms of typed indices (`ScalarVar` for witness scalars and `GroupVar` for group elements), and each constraint stores only the nonzero terms as a list of weighted pairs (`ScalarVar`, `GroupVar`).

Relations are built through the `LinearRelation` interface. This type allows for allocating group and scalar variables, representing symbols in the relation. Using Rust arithmetic operator implementations, statements such as `x * G` are used to construct constraints.

```
let mut relation = LinearRelation::new();

let x_var = relation.allocate_scalar();
let G_var = relation.allocate_element();
let P_var = relation.allocate_eq(x_var * G_var);

let x = Grp::Scalar::from(42);
relation.set_element(G_var, Grp::generator());
relation.set_element(P_var, x * Grp::generator());
```

Public constants are represented using a unit scalar term with an explicit weight, so affine offsets do not introduce extra witness variables. The canonicalization step moves those constants to the left-hand side; e.g., $C = 2G + rH$ becomes

$(C - 2G) = rH$, so the runtime only sees linear combination statements. The resulting canonical relation is thus a list of sparse terms, that can be efficiently represented as follows: encode the number of equations, then for each equation the number of terms and their indices, followed by the serialized group elements.

A trait `SigmaProtocol` exposes the three-move structure of Σ-protocols explicitly. It fixes associated types for the commitment, prover state, response, witness, and challenge, and requires `prover_commit`, `prover_response`, and `verifier` methods. Implementations also define serialization and deserialization for each proof component plus a `protocol_identifier` and an `instance_label` so Fiat–Shamir transcripts can bind to the protocol and instance.

In Appendix B we describe how `LinearRelation` may also be used for producing succinct arguments of knowledge (SNARKs, in the style of bulletproofs [14]), through the more recent line of literature of compressed Σ-protocols [3].

5.1 Protocol composition

Two parameterized implementations of `SigmaProtocol` are provided: `CanonicalLinearRelation<G>` implements a Σ-protocol conforming to the draft IETF standard for linear relations over the group `G`, and `ComposedRelation<G>` implements standard AND/OR/threshold composition following Cramer, Damgård, and Schoenmakers [27]. `ComposedRelation` represents a tree of sub-protocols with nodes `Simple`, `And`, `Or`, and `Threshold(t, ...)`. Each node implements `SigmaProtocol` by delegating to its children. For AND, the prover commits and responds for every child under the same challenge. For OR, the prover selects one branch for which it has a witness (the *real* branch) and simulates all other branches. In particular, the prover assigns random challenges to the simulated branches. Upon receiving the verifier's challenge e, the prover sets the remaining challenge for the real branch so that the sum of all per-branch challenges equals e. The prover then sends the first $n-1$ challenges, and the responses for every branch; the verifier recomputes the missing challenge from the sum and verifies each branch transcript.

For threshold t-of-n, the prover performs a t-of-n secret sharing of the global challenge. Concretely, it samples a random degree-$(n-t)$ polynomial whose constant term is the global challenge and evaluates it at n distinct points to derive per-branch challenges. The prover transmits a compressed representation consisting of the $n-t$ polynomial coefficients, from which the verifier reconstructs all branch challenges during verification.

5.2 Constant time strategy

Any operations over secret data must be constant-time [54]. In the case of keyed-verification credentials, this includes the zero-knowledge verifier, because verification uses the server's secret key. Constant-time group arithmetic alone is insufficient: operation ordering and data-dependent memory access can also create timing side-channels.

This can manifest in several forms. In range proofs, the Hamming weight or bit decomposition of the witness can influence the number of group operations performed. In OR- and threshold compositions, the witness determines which branches correspond to true statements and which are simulated; the prover algorithm and the simulator are distinct procedures with distinct timing signatures, and so a naïve implementation therefore leaks information about the witness.

For linear relations as described in Section 5, achieving constant time is comparatively straightforward: the prover reduces to evaluating a fixed group morphism. In our implementation, the morphism underlying `CanonicalLinearRelation` is evaluated using only constant-time group operations and witness-independent memory access patterns.

For OR-proofs, we proceed as follows. First, the witness is checked against all branches, and the resulting validity bits are stored. Then, for every branch, both the prover algorithm and the simulator are executed unconditionally. The actual commitment is selected using constant-time conditional swaps based on the validity bits. This ensures that the sequence of executed operations and memory accesses is independent of which branch is true.

In threshold proofs, the prover must select exactly t-out-of-n true branches in order to perform polynomial interpolation. Selecting these t elements is witness-dependent, and performing it naïvely will leak which branches are true. The solution is *oblivious compaction*: given a secret selection bitmask over an array of n items, compact the selected items into the first t positions without revealing which items were selected.

A direct oblivious compaction can be implemented in $O(n^2)$ time by scanning and conditionally swapping all pairs. More efficient fully oblivious algorithms are known: in particular, Sasy et al. [77] give an $O(n \log n)$ algorithm with deterministic memory access patterns and control flow, originally developed in the context of trusted execution environments (TEEs). We rely on this style of oblivious compaction to implement threshold Σ-protocols with witness-independent execution traces.

We test our implementation for constant-timedness using the statistical techniques of DudeCT [74]. This approach assesses the runtime behavior of the implementation on a particular machine, and is a useful litmus test. It is complementary to, although generally weaker than, guarantees provided by symbolic execution based techniques such as Binsec [28] which are more difficult to apply and require a timing model for the underlying machine.

6 The SIGMA-COMPILER crate

The next level up in our architecture is the SIGMA-COMPILER crate. This crate provides a *compile-time macro* for easy declarative definitions of Σ-protocols. As a compile-time macro, none of the machinery described in this section has to be shipped to servers or clients; this is particularly important when the ZKP code is bundled into a small app or WebAssembly (wasm) binary (see Section 8).

The syntax is similar to that of de Valence's `zkp` crate [32], described in Section 1.2, but with a more expressive language. A simple example proves the morphism from Example 1:

```
sigma_compiler!{ dleq<Grp>,
    (x, rand r, rand s),
    (C, D, cind G, cind H),
    C = x*G + r*H,
    D = x*G + s*H,
}
```

Data types. This macro invocation (indicated by `!` in Rust) creates a Rust module called `dleq` implementing a Σ-protocol over the group `Grp` (where `Grp` implements Rust's `PrimeGroup` trait).[3]

The protocol involves three scalars (`x`, `r`, `s`) of type `Scalar`, and four group elements (`C`, `D`, `G`, `H`) of type `Point`. The protocol will prove the listed statements (implicitly connected by AND).

Each listed `Scalar` can be optionally *tagged* with one or more of the tags `pub`, `rand`, or `vec`. `pub` means that the `Scalar` is public; this can be used for public parameters to the protocol, such as the limits of ranges, or other constants that can appear in the statements. Any `Scalar` not marked as `pub` is assumed to be private to the prover, and the verifier will learn nothing about that value other than what is implied by the truth of the statements. `rand` means that the `Scalar` is a uniform random `Scalar`; these are typically used as randomizers in Pedersen commitments. A `rand` variable cannot be `pub`. `vec` means that the variable represents a vector of `Scalar`s, as opposed to a single `Scalar`. The number of entries in the vector can be set at run time, when the Σ-protocol is executed.

Similarly, each listed `Point` can be optionally tagged with one or more of the tags `cind` or `vec`. All `Point` variables are considered public, but expressions evaluating to a `Point` may be public or private; for example, multiplying a private `Scalar` by a public `Point` yields a private `Point` expression. (In general, an expression is public if all of its components are public, and private if any of its components is private.) `Point`s tagged with `cind` are computationally independent (see Section 2). Typically, these elements would be generators of the group, generated with a hash-to-group function [39]. `Point`s marked `cind` are typically the bases used in Pedersen commitments. `vec` means that the variable represents a vector of `Point`s, as opposed to a single `Point`. The number of entries in the vector can be set at run time, when the Σ-protocol is executed.

Statements. The most basic statement is the *linear combination statement*. This kind of statement proves that a `Point` (or an expression evaluating to a public `Point`) is a linear combination of other `Point`s. The coefficients on the `Point`s are arithmetic expressions involving `Scalar`s, integer constants, and the operations `+`, `-`, `*`, with the restriction that two private subexpressions cannot be multiplied. For example, if capitalized variables represent `Point`s, `a` is a public `Scalar`, and `x` and `y` are private `Scalar`s, then `(a-1)*C = 2*(a+1)*x*G - 5*(y-2)*H` is a valid linear combination statement. The basic linear combination statement is the only type of statement supported by the lower-layer SIGMA-PROOFS crate, and so one of the roles of the SIGMA-COMPILER crate is to compile more complex statement types into linear combination statements.

The statements in the `sigma_compiler` macro can be combined with arbitrarily nested AND, OR, and THRESH combiners, forming a tree structure called the *statement tree*, where leaves are statements, and interior nodes are AND/OR/THRESH combiners. As mentioned above, a list of statements with no explicit combiner is considered to be combined with AND.

Compilation. The module output by the macro contains: (i) a struct `Instance` containing all the `Point`s and public `Scalar`s specified in the macro invocation; (ii) a struct `Witness` containing all the private `Scalar`s specified in the macro invocation; (iii) a function `prove` that consumes an `Instance` and a `Witness`, and outputs a proof as a byte array (`Vec<u8>`); (iv) a function `verify` that consumes an `Instance` and a proof, and outputs a `Result` indicating success or failure.

The stages of the `sigma_compiler` macro are (i) rewrite the statement tree so that it satisfies the *disjunction invariant*; (ii) apply *transformations* to the statement tree to convert complex statements into basic linear combination statements, preserving the disjunction invariant; (iii) output Rust code that uses the SIGMA-PROOFS API to implement a prover and a verifier for the resulting statement tree of linear combination statements with AND/OR/THRESH combiners.

6.1 The disjunction invariant

The disjunction invariant is a technical condition on the structure and variables of the statement tree that prevents leakage from shared commitments across OR branches. It states that no private `Scalar` may appear both inside and outside any branch of a disjunction (an OR or THRESH combiner). For example, if capital letters are `Point` variables, and lowercase letters are private `Scalar` variables, then the following statement trees all violate the disjunction invariant:

[3] If the optional group specifier `<Grp>` is omitted, `sigma_compiler` defaults to using a group literally called `G`.

- `OR (D = x*A + s*B, E = x*A + t*B)`
- `AND (C = x*G + r*H,`
  `OR (D = x*A + s*B, E = y*A + t*B))`
- `OR (D = x*A + s*B,`
  `AND ( E = x*A + r*B, F = y*A + t*B))`

In contrast, the following one satisfies the invariant:

- `THRESH (2, C = y*A + s*B, D = z*A + u*B,`
  `AND ( E = x*A + r*B, F = x*A + t*B))`

A naive ZKP implementation might produce a single commitment for each private variable in the statement tree. However, each branch of a disjunction will receive its own challenge, and responding to two different challenges for the same commitment reveals the private witness. This is the reason for the disjunction invariant.

The `sigma_compiler` macro converts a statement tree that violates the disjunction invariant to a logically equivalent one that satisfies it as follows. For each private `Scalar` it scans the statement tree to see if it appears in leaves in more than one disjunction branch (including the *root disjunction branch*, which is the set of nodes in the tree outside of all disjunctions). If so, it (i) finds or inserts a Pedersen commitment to the variable in the root disjunction branch,[4] (ii) renames all instances of the variable in each non-root disjunction branch to a name unique to that disjunction branch, and (iii) adds a Pedersen commitment *with the same commitment value as the one in step (i)* to the renamed variable, with a fresh randomness variable in each disjunction branch in which it appears.

Take the second violating example in the list above. Step (i) finds the existing Pedersen commitment `C = x*G + r*H` in the root disjunction branch; step (ii) renames the other instance of `x` to, say, `x1`; step (iii) adds a Pedersen commitment `C = x1*G + z*H` to the disjunction branch now containing `x1` using the AND combiner. This results in a statement tree equivalent to the original, which does satisfy the disjunction invariant:

- `AND (C = x*G + r*H,`
  `OR (AND (D = x1*A + s*B, C = x1*G + z*H),`
  `E = y*A + t*B))`

The observation is that, by the definition of the computationally independent `Point`s used in the Pedersen commitments, if the prover proves that `C = x*G + r*H` and also `C = x1*G + z*H` (with the *same* `C`), then it must be the case that `x=x1`; this binds together the values of the original and renamed variables.

While previous work [60] recognized the problem of a private variable appearing both inside and outside a disjunction, that system simply rejects such a statement, whereas our SIGMA-COMPILER crate rewrites the statement automatically.

[4] Finding and using an existing commitment improves the efficiency of the ultimate proof, saving both proof size and runtime.

6.2 Transformations

After converting the statement tree into an equivalent one that satisfies the disjunction invariant, the `sigma_compiler` macro applies *transformations* to handle statement types other than linear combination statements. Linear combination statements about `Point`s are the only kinds of statements that SIGMA-PROOFS can handle. The SIGMA-COMPILER transformations provide an easy way to specify statements about `Scalar`s (the values committed to in Pedersen commitments, for example).

`Scalar` substitution. The first transformation that is applied is that of *substitution*. A substitution statement looks like `x = y + 1`, where the left side of the `=` is a private `Scalar` variable, and the right side is any expression that evaluates to a `Scalar` (that does not itself include the left-side variable, directly or after other substitutions). The disjunction invariant guarantees that any private `Scalar`s in the substitution statement can *only* appear in the same disjunction branch as the substitution statement itself.

The transformation applied for each substitution statement is as follows: (i) wherever the variable on the left side appears in the statement tree, replace it with the expression on the right side; (ii) replace the substitution statement itself with the dummy statement `true`, which represents a statement that is trivially true. After all of the substitution statements have been processed, *prune* the expression tree by removing `true` statements from AND nodes, converting AND nodes with a single child to just that child, and similar processing for disjunction nodes. Importantly, if the disjunction invariant was true before applying the transformation, it remains true after, since the private `Scalar`s in the substitution statement can all appear only in the disjunction branch the substitution statement itself was in.

Instead of performing a substitution, relationships between private `Scalar`s could alternately be handled by adding a constraint equation. The benefit of using the substitution technique is that it *reduces* the size of the witness (and thus the proof), while the alternate method would *increase* the size of the proof.

An example application of the substitution transformation can be seen in Figure 3.

Public `Scalar` equality. While substitution statements assert values for, and relationships between, private `Scalar`s, it is sometimes convenient to be able to do the same for *public* `Scalar`s. A public `Scalar` equality statement is of the form `a = b + 1`, where the left side of the `=` is a public `Scalar` variable, and the right side is an expression evaluating to a public `Scalar`. The public `Scalar` equality transformation handles these kinds of statements in two ways. In the first case, if the statement appears in the root disjunction branch (that is, the ZKP is asserting the truth of the statement unconditionally), then the statement is simply removed from the statement tree, and code is queued to be emitted (in the code

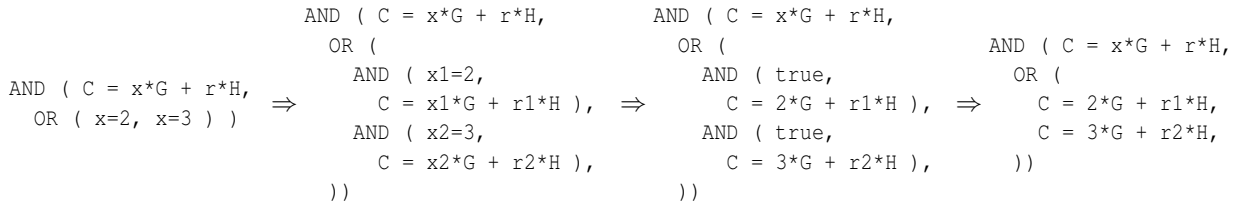


Figure 3: Compiling an example statement tree proving that the value `x` committed to in the Pedersen commitment `C` is either 2 or 3. The first step enforces the disjunction invariant; the second step applies the substitution transformation; the third step prunes the expression tree. The resulting expression tree consists only of linear combination statements, and can be directly compiled into Rust code that uses the SIGMA-PROOFS API.

generation phase below) for both the prover and the verifier to simply check that the given public variable (which will appear in the `Instance` struct available to both of them) has the correct value.

In the second case, the public `Scalar` equality statement appears somewhere within a disjunction. Here, the statement is not guaranteed to be true (as a different branch of the disjunction may be the true branch). In this case, the transformation converts the public `Scalar` equality statement into a linear combination statement by multiplying both sides by one of the `cind Point`s (e.g., the generator `G` from a Pedersen commitment). For example, the above statement might be converted to `a*G = (b+1)*G`. Because this statement involves only public `Scalar`s and public `Point`s, it does not increase the size of the proof.

`Scalar` range proofs. The third kind of transformation is the *range transformation*. A range statement is of the form `(lo..hi).contains(x)`, where `lo` and `hi` are expressions evaluating to public `Scalar`s, and `x` is a `LinScalar` (that is, a private `Scalar`, or a linear function of one, like `y-3`). As is standard in Rust, the notation `lo..hi` is a range that *includes* `lo` but *excludes* `hi`. The Rust notation `lo..=hi` can also be used to include both endpoints, which is treated as `lo..(hi+1)`. Note also that although `lo` and `hi` must be public, they need not be constant; they may depend on public `Scalar`s whose values will only be known at run time, not at compile time.

The first step of the transformation is to subtract `lo` from both `hi` and `x`, converting the above example statement to `(0..(hi-lo)).contains(x-lo)`. The transformation initializes a *basic statement list* to empty. It then looks for an existing Pedersen commitment to the `LinScalar` `x-lo` (or indeed to any `LinScalar` for the same private `Scalar` as `x`). If it finds one, the transformation will use it below. Otherwise, the transformation inserts a new Pedersen commitment to `x-lo` into the basic statement list. For the sake of example below, suppose the (found or generated) Pedersen commitment is `D = (x-lo)*G + t*H`. The next (and key) step in the transformation uses the following observation:

Remark 1. *Let u, n, and x be integers. If* $2^n < u \leq 2^{n+1}$*, then* $0 \leq x < u$ *iff there exist bits* $b_0, b_1, \ldots, b_n \in \{0,1\}$ *such that* $x = b_0 + 2b_1 + 4b_2 + \ldots + 2^{n-1}b_{n-1} + (u - 2^n)b_n$.

The next observation is that if `C = b*G + r*H` is a Pedersen commitment to a value `b`, we can prove that `b` is a bit (that is, is either 0 or 1) by adding to the statement tree the single statement `C = b*C + s*H` (for a fresh `Scalar` variable `s`), as in Example 2.

Therefore, the next step in the transformation is to create fresh `vec Scalar` variables, say `b`, `r`, and `s`, and a fresh `vec Point` variable, say `C`, and insert into the basic statement list the two statements `C = b*G + r*H` and `C = b*C + s*H`. These are statements about `vec` variables, which are interpreted pointwise; e.g., the first statement asserts that `C[i] = b[i]*G + r[i]*H` for all `i`. The transformation also queues code to be emitted in the code generation phase to compute the length of these vectors based on the (known at run time) value of `hi-lo`, and for the prover to compute the correct bit vector `b`, randomness vectors `r` and `s`, and commitment vector `C`. The value `r[0]` will also be chosen so that `D = C[0] + 2*C[1] + ... +` 2^{n-1}`*C[n-1] + (hi-lo-`2^n`)*C[n]`.

As a small optimization, rather than *proving* the above linear combination of values for `D`, both the prover and the verifier will *compute* `C[0]` to be `D - (2*C[1] + ... +` 2^{n-1}`*C[n-1] + (hi-lo-`2^n`)*C[n])`. In this way, some complexity in the proof is eliminated, and also `C[0]` need not be transmitted as part of the proof.

Finally, the range statement is replaced in the statement tree with the AND of the (two or three, depending on whether a Pedersen commitment to `x-lo` was found or generated) statements in the basic statement list.

`Scalar` non-equality. The final transformation supported by SIGMA-COMPILER is `Scalar` non-equality. A `Scalar` non-equality statement looks like `x != a + 1`, where the left side is a `LinScalar`, and the right side is an expression evaluating to a public `Scalar`.

The first step of the transformation is to subtract the right side from the left side, converting the statement to (in the above example) `x-(a+1) != 0`. The `sigma_compiler` macro then, as above, looks for a Pedersen commitment to any `LinScalar` with the same private `Scalar` variable as

the one in the non-equality statement, or generates a new Pedersen commitment if one is not found. Continuing the above example, suppose the Pedersen commitment `C = x*G + r*H` is found; then the macro computes that `C-(a+1)*G` is a Pedersen commitment to `x-(a+1)`.

The observation used by this transformation is that if we have a commitment `D = v*G + r*H` to a `Scalar` expression `v`, then we can prove that `v != 0` by adding the single statement `G = y*D + s*H` for fresh `Scalar` variables `y` and `s`. This works because substituting the definition of `D` into the latter equation yields `G = y*(v*G + r*H) + s*H = (y*v)*G + (y*r+s)*H`, so it must be the case that `1 = y*v`, and so `y` cannot be 0.

The action of the transformation is then to replace the non-equality statement `v != 0` with either the single statement `G = y*D + s*H` (if it found a Pedersen commitment `D` to `v`, or could compute one from a commitment it did find), or the AND of a fresh Pedersen commitment `D` to `v` and the above statement `G = y*D + s*H`.

6.3 Code generation

After performing the transformations, the statement tree will consist entirely of linear combination statements, combined with AND, OR, and THRESH combiners. The first step in code generation is to *flatten* the statement tree. This action "inlines" the children of any AND that is itself the child of another AND. For example, `AND ( s1, AND ( s2, s3 ), OR (s4, s5) )` would get flattened to `AND ( s1, s2, s3, OR (s4, s5) )`, where each of the `s`i is a statement.

The `sigma_compiler` macro converts a flattened statement tree T into Rust code that produces a `SigmaProtocol`.

The code for the module created by the macro will include the definitions of the `Instance` and `Witness` structs, the above code to generate the `SigmaProtocol` (which will be called by `prove` and `verify`), and the `prove` and `verify` functions themselves on the `SigmaProtocol`.

7 The CMZ crate

The top layer of our software stack is the CMZ crate. Like the SIGMA-COMPILER crate below it, this crate provides compile-time macros. These macros allow a programmer to easily specify KVAC protocols, which get transformed (at compile time) to invocations of the `sigma_compiler` macro, which get themselves transformed into code that directly uses the SIGMA-PROOFS API.

To demonstrate the flexibility of our layered approach, we implemented two different KVAC systems: CMZ14 [22] and μCMZ [68]. From the programmer's point of view, the API we expose for either choice is basically identical; the main difference is simply the name of the macro to be invoked (`CMZ14Protocol` vs. `muCMZProtocol`). Other KVAC choices could be implemented here as well, such as Barki et al.'s [7] or μBBS [68], which would also present the same API to the programmer, and would similarly internally compile down to invocations of the `sigma_compiler` macro.

The CMZ crate exposes two macros: one to declare credential *types* and one to declare credential *protocols*. The first is extremely simple. The following code creates a credential type called `Wallet`, with two attributes `randid` and `balance`, and a second credential type `Item`, with two attributes `serialno` and `price`, both using the `PrimeGroup` `Grp` (as before, defaulting to a group called `G` if not specified):

```
CMZ! { Wallet<Grp>: randid, balance }
CMZ! { Item<Grp>: serialno, price }
```

This code will create Rust structs `Wallet` and `Item`, each containing fields for their respective attributes (of type `Option<Scalar>`[5]), a field for the MAC, and implementations for our `CMZCredential` trait, which handles key generation and management, and MAC creation and verification.

KVAC protocols are performed between a *client* and an *issuer*. The issuer is the only party that can either create or verify the validity of credentials. Each protocol defined by the CMZ crate's second macro can, in one action, (i) present zero or more credentials already held by the client (of the same or different credential types), (ii) issue zero or more new credentials to the client (of the same or different types), and (iii) prove in zero knowledge statements involving any of the attributes of any of the shown or to-be-issued credentials.

The protocol macro invocation will also state how to handle each attribute in each credential. Attributes in presented credentials can be (H)idden from the issuer, (R)evealed to the issuer, or be (I)mplicitly known by both the client and the issuer. Attributes in issued credentials can have those specifications, but also be (S)et by the issuer, or (J)ointly created by the client and the issuer. In joint creation, as used by Lox [86] to generate credential unique identifiers for example, the issuer learns no information about the value of the credential, and the client has no influence over its value.

As an example, the code to create a protocol to spend some of the balance in a `Wallet` on an `Item` could be:

```
muCMZProtocol! { wallet_spend<max_balance>,
    [ W: Wallet { randid: R, balance: H },
      I: Item { serialno: H, price: H } ],
    N: Wallet { randid: J, balance: H },
    (0..=max_balance).contains(N.balance),
    W.balance = N.balance + I.price
}
```

This macro invocation creates a protocol called `wallet_spend`, with a public `Scalar` parameter `max_balance`. The first argument is a list of two credentials to show: a `Wallet` called `W`, and a `Item` called `I`. The

[5] `Option` in Rust denotes partial value; i.e., a value that may either contain a `Scalar` or contain nothing (`None`), making the absence of a value explicit and type safe.

randid attribute of W will be revealed to the issuer during the protocol (to prevent double spending; i.e., presenting an already-spent credential), while the balance attribute, and both attributes of I, will be hidden from the issuer (but statements about them may be proved in zero knowledge).

The next argument to the macro are the credentials to issue, in this case the single credential N, of type Wallet. The randid attribute of N will be jointly created, and the balance attribute will be hidden from the issuer.

The remaining arguments form a statement tree, with the same syntax as in the SIGMA-COMPILER crate. In this case, the protocol proves that the balance in the issued Wallet is computed correctly, and has not gone negative.

The generated API. Like the sigma_compiler macro, the CMZ protocol creation macro creates a Rust module containing all of the data structures and code needed to execute the credential protocol. Continuing with the above wallet_spend example, the name of the module would be wallet_spend. The module would contain definitions for the following structs: (i) Params, which encapsulates the public parameters to the protocol (in this example, max_balance); (ii) Request, which is the (single) message sent from the client to the issuer in the protocol; (iii) Reply, which is the (single) message sent from the issuer to the client in the protocol; (iv) ClientState, encapsulating the client state that the client needs to keep between the time the Request is sent and the Reply is received. The Request, Reply, and ClientState structs are opaque to the programmer, but are serializable so that they can be stored or transmitted. If the protocol specifies zero credentials to issue, then Reply and ClientState are empty and unneeded; the resulting protocol is non-interactive in that case, with just a single Request going from the client to the issuer.

The module defines a function called prepare. This function is run by the client to start the protocol. It takes as input the Params (if any), the credentials that are presented in the protocol (W and I in this example), as well as *incomplete* credentials that are to be issued (N). An incomplete credential would definitely not yet have its MAC filled in, and would also be missing any attributes marked as (S)et by the issuer or (J)ointly created, since the client would not yet know the values those attributes will eventually have. The prepare function returns a Request and a ClientState. The client will serialize the Request and send it to the issuer.

The module also defines a function called handle. This function is run by the issuer upon receipt of a Request. The handle function takes as input the received Request, and two callbacks, fill_creds and authorize. The handle function will read the request, and use it to fill in the revealed attributes from the shown and issued credentials (in this case, just W.randid). The hidden attributes from the credentials will be set to None, as will the implicit, set by issuer, and joint creation attributes. It is the job of the fill_creds callback to (i) generate a Params struct appropriate to the request; (ii) set the values of the implicit and set by issuer attributes (if any) for each shown and issued credential; (iii) set the private keys for each credential.

The handle function will then check that the credentials shown by the client are all valid, and verifies the proof of the statements given in the macro invocation. If so, handle will call the authorize callback, which can do any final application-specific checks on the credentials, such as checking for double spending. If authorize returns successfully, then handle will issue the credentials to be issued (in this case, the new Wallet credential). It will return a Reply and copies of the shown and issued credentials (but the attributes not visible to the issuer will still be None). The issuer will serialize the Reply and send it to the client.

The client's ClientState has a method called finalize that consumes a Reply, and returns completed versions of the credentials to be issued (N in this example).

As another example, in Appendix C we demonstrate how to use the CMZ crate to implement *rate limiting* [8, 15, 52, 83].

8 Applications in censorship circumvention

One use case of anonymous credential systems is establishing a notion of trust among anonymous users. In this setting, users may be reluctant to provide any kind of proof of identity in exchange for access since most suitable forms are either issued, or else easily tampered with, by bodies closely connected to those that impose internet censorship. It is therefore important to be strategic in balancing the availability and openness of anonymous credential systems against the risk of enumeration or DoS attacks from censors or otherwise malicious users. To address this, credential attributes, such as creation time and measurements of a credential holder's behaviour, can be used to determine the trustworthiness of a user while maintaining anonymity. This can help to safeguard highly effective tools for users with an established level of trust.

However, it can be difficult for the benefits of implementing and maintaining anonymous credential systems to outweigh the complexity. The modular software stack that we have presented in this work, which allows a developer to abstract away the low-level complexity of an anonymous credential system, along with the expressiveness in the statements we have developed as part of the SIGMA-COMPILER layer, reduces the overhead involved in developing and maintaining these systems, making them much more accessible.

Here we describe our re-implementation of an existing censorship resistance anonymous credential system, Lox [86], and describe the advantages of using our system over the previous implementation. We also describe a new censorship resistance anonymous credential system that we implemented for the Open Observatory of Network Interference (OONI) [66].

8.1 Lox

Lox is a censorship-resistant bridge distribution system developed by Tulloch and Goldberg [86] to address the bridge distribution problem, where secret resources ("bridges") intended for genuine users are distributed in an environment where there is a high likelihood they may be intercepted and blocked by censors first. Lox reduces the ability of a censor to enumerate bridges by prioritizing distribution of bridges through the social graph of users who have accumulated trust over time.

Tor [34], an overlay network that enables anonymous communication over the internet, provides bridges to help censored users connect to the free and open internet. Tor has adopted Lox [84] to help address the bridge distribution problem for Tor bridges and deployed Lox in an alpha version of the Tor browser; it is currently undergoing further development prior to a testing release.

Integration with Tor has led to some changes from the original Lox implementation. Lox was originally implemented using the CMZ14 algebraic MAC anonymous credential scheme by Chase et al. [22]. Recently, Orrù [68] has shown that statistical anonymity and improvements to issuer efficiency can be achieved using their μCMZ scheme and the Tor developers integrated these changes into Lox. To perform the zero-knowledge proofs, Lox protocols were implemented using de Valence's zkp crate [32]; however, that crate has not been updated since Lox was originally developed and many of the libraries it depends on have become outdated, leading to Tor developers maintaining a fork of the zkp crate that can be updated along with Lox. Finally, to address the logistics of deploying to numerous users, privacy-preserving key rotation protocols for some Lox credential keys were created to limit the storage requirements for spent ID attributes over time.

Overview of the credential extension. We used the SIGMA-RS crates described in this work to re-implement the protocols in the Lox library, forking the project from its current state as part of the Lox workspace maintained by Tor [84]. Our re-implementation preserves the functionality of the original system described by Tulloch and Goldberg [86] as well as the additional functions added for integration with Tor.

We compared the lines of code required to implement each of the Lox protocols with the SIGMA-RS crates against the original implementation, which used the zkp crate to implement both the CMZ [22] and μCMZ [68] versions of the protocols. We see that the switch to μCMZ reduced the code required to express the same proofs, especially for the issuance proofs. However, with our `muCMZProtocol` macro, we are able to define the same behaviour for both client and issuer proofs in a single compact statement, reducing the code significantly for every protocol as shown in Table 1. We have included in Appendix D, a comparison of the lines of code for just the proof declarations of each protocol as well as code snippets for the proofs from one of the μCMZ protocols for both the zkp crate implementation compared with using the SIGMA-RS crates. Our crates also eliminate overhead code required to enable communication between a client and server and greatly reduce the complexity involved with building anonymous credentials systems. This is particularly evident with range proofs.

Table 1: Lines of code required for a developer to implement each of the Lox protocols, including request and issuance proofs, using the zkp crate versus our SIGMA-RS crates. We see the reduction in code from CMZ14 to μCMZ using the zkp library but a much larger reduction in code for all Lox protocols using SIGMA-RS.

Lox Protocol	**Lines of code** (LOC)		
	zkp crate		**SIGMA-RS**
	CMZ14	**μCMZ**	**μCMZ**
Open Invitation	287	241	121
Trust Promotion	436	419	147
Trust Migration	398	320	129
Level Up	818	698	195
Issue Invitation	730	459	134
Redeem Invitation	495	387	115
Check Blockage	271	247	147
Blockage Migration	514	398	131
Update Invite	439	285	111
Update Credential	556	323	117

We orchestrated tests of each of the Lox protocols using a Rust native client and server as well as a wasm client and Rust native server and collected the time required to create a Lox request and handle the response (on the client side) and generate a response on the server side in each of these runtime environments. The tests were run on a Framework 16 laptop with an AMD Ryzen 9 7940HS processor supporting AVX512 instructions. Each test was run 10 times for each protocol, for each environment. The results in Table 2 show timings that are reasonable for modern web applications such as Tor for each of the protocols in both environments. They are also comparable to the original Lox implementation. We also ran tests to collect the request and response sizes in bytes, displayed in Appendix D, which are also comparable to the original Lox implementation [84].

8.2 OONI

The Open Observatory of Network Interference (OONI) [66] is a free software project that operates a global measurement platform aimed at documenting internet censorship while protecting the safety of its volunteers. The "OONI probe" app exists as a free mobile and desktop app, available for download from the OONI website and from the Google Play and Apple App stores. OONI probe works by launching a background process that periodically attempts to connect to a set

```
// Open registration protocol
muCMZProtocol! {open_registration, ,
  UAC: UserAuthCredential { nym_id: J,
    age: S, measurement_count: I},
}

// Update protocol
muCMZProtocol!(update,
  Old: UserAuthCredential { nym_id: H, age: H,
    measurement_count: H},
  New: UserAuthCredential { nym_id: H, age: H,
    measurement_count: H},
  Old.nym_id = New.nym_id,
  Old.age = New.age,
  Old.measurement_count = New.measurement_count
);
```

```
// Submission protocol
muCMZProtocol!(submit<min_age_today, max_age,
    min_measurement_count, max_measurement_count,
    @DOMAIN, @NYM>,
  Old: UserAuthCredential { nym_id: H, age: H,
    measurement_count: H},
  New: UserAuthCredential { nym_id: H, age: H,
    measurement_count: H},
  Old.nym_id = New.nym_id,
  Old.age = New.age,
  New.measurement_count = Old.measurement_count + 1,
  NYM = Old.nym_id * DOMAIN,
  (min_age_today..=max_age).contains(Old.age),
  (min_measurement_count..=max_measurement_count)
    .contains(Old.measurement_count)
);
```

Figure 4: The OONI credential protocols, as implemented in SIGMA-RS

Table 2: Average time (and standard deviation) to create Lox requests and responses; i.e., to create and verify proofs for the Lox client and server for a Lox authority with a bridge table size of 30 bridges.

Protocol	Time (ms)					
	zkp Library + μCMZ			SIGMA-RS + μCMZ		
	Client		Server	Client		Server
	Native	Wasm	Native	Native	Wasm	Native
Open Invitation	0.53 (0.07)	2.2 (0.9)	0.380 (0.007)	0.360 (0.004)	2.1 (0.7)	0.368 (0.002)
Trust Promotion	2.1 (0.5)	9 (1)	2.792 (0.007)	1.60 (0.08)	7.0 (0.8)	2.5 (0.3)
Trust Migration	1.127 (0.005)	4.9 (0.7)	0.822 (0.006)	1.017 (0.002)	4.9 (0.9)	0.741 (0.006)
Level Up	3.3 (0.3)	13 (1)	2.4 (0.6)	2.44 (0.01)	11.0 (0.9)	1.99 (0.01)
Issue Invite	1.94 (0.01)	7 (1)	1.24 (0.01)	1.7 (0.2)	8 (1)	1.19 (0.02)
Redeem Invite	1.62 (0.02)	6.2 (0.9)	1.131 (0.007)	1.279 (0.007)	6.0 (0.8)	1.033 (0.005)
Check Blockage	0.650 (0.008)	3.0 (0.8)	1.30 (0.01)	0.716 (0.009)	2.9 (0.6)	0.647 (0.003)
Blockage Migration	1.49 (0.06)	5.5 (0.7)	1.1 (0.2)	1.289 (0.006)	6.2 (0.8)	0.922 (0.005)
Update Invite	0.836 (0.008)	3.8 (0.9)	0.525 (0.004)	0.9 (0.4)	4 (1)	0.57 (0.09)
Update Credential	1.117 (0.008)	4.8 (0.6)	0.686 (0.007)	1.054 (0.005)	5.0 (0.7)	0.717 (0.006)

Table 3: Time required for registration, submission, and update operations of the OONI protocol, measured on a Framework 16 laptop with an AMD Ryzen 9 7940HS processor supporting AVX512 instructions (Native) and an iPhone 16 running iOS 26.2 (iOS).

Protocol	Time (ms)		
	Client		Server
	Native	iOS	Native
Registration	0.33 (0.06)	0.29 (0.03)	0.32 (0.08)
Submit	2.3 (0.2)	2.1 (0.1)	2.3 (0.2)
Update	0.76 (0.07)	0.68 (0.01)	0.6 (0.1)

of websites, determined by OONI, then sending the connectivity results back to OONI to be aggregated into connectivity measurements for specific regions. Unlike traditional systems, OONI cannot rely on persistent identifiers, accounts, or IP-based controls, as these could expose users to significant risk. At the same time, OONI must establish trust in submitted measurements and mitigate faulty or malicious data. This tension motivates OONI's adoption of anonymous credentials: a cryptographic mechanism that allows probes to prove properties such as long-term participation or contribution volume without revealing identity or location, or enabling tracking across networks. [67]

Evaluation of the credential extension. Our OONI user-authentication system equips each Probe installation with a reusable μCMZ credential whose attributes encode (i) a per-installation secret `nym_id`, (ii) an issuance date `age` (stored as a Julian day), (iii) a monotone counter `measurement_count`.

A new Probe registers via `open_registration`, where the server sets `age` to "today" and initializes `measurement_count` to 0, while `nym_id` is jointly created by the client and server. To upload a measurement, the Probe runs `submit`, proving in zero knowledge that it holds a valid credential, that `age` and `measurement_count` lie in coarse policy ranges, and that the next credential increments the counter by 1 while preserving `nym_id` and `age`. To prevent cross-network linkability, the Probe derives a network-local pseudonym `NYM` = `nym_id*DOMAIN`, where `DOMAIN` is a hash of `ooni.org/{CC}/{ASN}`; the server receives only the proof and the 32-byte compressed `NYM`. Finally, the update protocol `update` supports key rotation of the server keys, by re-issuing the same attributes without revealing them. We show our SIGMA-RS OONI protocol declarations in Figure 4.

In Table 3 we give an overview of the performance of our credential system in native code and as an iOS app. We see that protocol operations complete within a few milliseconds on both the client and server, with comparable performance across platforms and no significant overhead introduced by mobile execution.

9 Conclusion

We presented SIGMA-RS, a layered Rust software stack that makes keyed-verification anonymous credential protocols easier to specify, implement, and deploy for a wide range of access policies. Through case studies drawn from censorship circumvention systems, specifically Tor's Lox protocols and a new anonymous authentication mechanism for OONI, we demonstrate the effectiveness of this approach in simplifying the protocol specification and implementation complexity while preserving practical efficiency.

Acknowledgements

This research was undertaken, in part, thanks to funding from the Canada Research Chairs program (award CRC-2018-00135) and Agence Nationale de la Recherche (award ANR-24-CE39-6061-01A2C). This work benefited from the use of the CrySP RIPPLE Facility at the University of Waterloo. Part of this research was conducted at MIT while M. Orrù was hosted by Srini Devadas.

References

[1] Marcin Andrychowicz, Stefan Dziembowski, Daniel Malinowski, and Lukasz Mazurek. On the Malleability of Bitcoin Transactions. In *Financial Cryptography and Data Security, FC '15*, pages 1–18, 2015.

[2] Gal Arnon and Eylon Yogev. Towards a White-Box Secure Fiat-Shamir Transformation. In *45th Annual International Cryptology Conference, CRYPTO '25*, volume 16005, pages 27–56, 2025.

[3] Thomas Attema and Ronald Cramer. Compressed Σ-Protocol Theory and Practical Application to Plug & Play Secure Algorithmics. In *40th Annual International Cryptology Conference, CRYPTO '20*, 2020.

[4] Jean-Philippe Aumasson, Dmitry Khovratovich, Bart Mennink, and Porçu Quine. SAFE: Sponge API for Field Elements. 2023.

[5] Endre Bangerter, Stephan Krenn, Ahmad-Reza Sadeghi, and Thomas Schneider. YAZKC: Yet Another Zero-Knowledge Compiler. In *19th USENIX Security Symposium, USENIX Security '10, Poster Session*, pages 11–13, 2010.

[6] Boaz Barak. How to Go Beyond the Black-Box Simulation Barrier. In *42nd Annual Symposium on Foundations of Computer Science, FOCS 2001*, 2001.

[7] Amira Barki, Solenn Brunet, Nicolas Desmoulins, and Jacques Traoré. Improved Algebraic MACs and Practical Keyed-Verification Anonymous Credentials. In *23rd International Conference in Selected Areas in Cryptography, SAC '16*, 2016.

[8] Fabrice Benhamouda, Mariana Raykova, and Karn Seth. Anonymous Counting Tokens. In *29th International Conference on the Theory and Application of Cryptology and Information Security, ASIACRYPT '23*, 2023.

[9] David Bernhard, Olivier Pereira, and Bogdan Warinschi. How Not to Prove Yourself: Pitfalls of the Fiat-Shamir Heuristic and Applications to Helios. In *18th International Conference on the Theory and Application of Cryptology and Information Security, ASIACRYPT '12*, volume 7658, pages 626–643, 2012.

[10] Dan Boneh and Victor Shoup. A Graduate Course in Applied Cryptography, 2020. Available online https://toc.cryptobook.us/book.pdf.

[11] Jonathan Bootle, Andrea Cerulli, Pyrros Chaidos, Jens Groth, and Christophe Petit. Efficient Zero-Knowledge Arguments for Arithmetic Circuits in the Discrete Log Setting. In *35th Annual International Conference on the Theory and Applications of Cryptographic Techniques, EUROCRYPT '16*, volume 9666 of *Lecture Notes in Computer Science*, pages 327–357, 2016.

[12] Jonathan Bootle, Alessandro Chiesa, and Katerina Sotiraki. Sumcheck Arguments and Their Applications. In *41st Annual International Cryptology Conference, CRYPTO '21*, 2021.

[13] Aydin Buluç, Jeremy T. Fineman, Matteo Frigo, John R. Gilbert, and Charles E. Leiserson. Parallel sparse matrix-vector and matrix-transpose-vector multiplication using compressed sparse blocks. In *21st Annual ACM Symposium on Parallelism in Algorithms and Architectures, SPAA '09*, pages 233–244, 2009.

[14] Benedikt Bünz, Jonathan Bootle, Dan Boneh, Andrew Poelstra, Pieter Wuille, and Gregory Maxwell. Bulletproofs: Short proofs for confidential transactions and more. In *2018 IEEE Symposium on Security and Privacy, SP '18*, pages 315–334, 2018.

[15] Jan Camenisch, Susan Hohenberger, Markulf Kohlweiss, Anna Lysyanskaya, and Mira Meyerovich. How to win the clonewars: efficient periodic n-times anonymous authentication. In *Proceedings of the 13th ACM Conference on Computer and Communications Security, CCS '06*, 2006.

[16] Jan Camenisch, Stephan Krenn, Anja Lehmann, Gert Læssøe Mikkelsen, Gregory Neven, and Michael Østergaard Pedersen. Formal Treatment of Privacy-Enhancing Credential Systems. In *22nd International Conference in Selected Areas in Cryptography, SAC '15*, 2015.

[17] Jan Camenisch and Anna Lysyanskaya. A Signature Scheme with Efficient Protocols. In *Security in Communication Networks, Third International Conference, SCN '02*, 2002.

[18] Ran Canetti, Yilei Chen, Justin Holmgren, Alex Lombardi, Guy N. Rothblum, and Ron D. Rothblum. Fiat-Shamir From Simpler Assumptions. 2018.

[19] Ran Canetti, Oded Goldreich, and Shai Halevi. The random oracle methodology, revisited. *Journal of the ACM*, 51(4):557–594, 2004.

[20] Rutchathon Chairattana-Apirom, Nico Döttling, Anna Lysyanskaya, and Stefano Tessaro. Everlasting Anonymous Rate-Limited Tokens. In *31st International Conference on the Theory and Application of Cryptology and Information Security, ASIACRYPT '25*, volume 16250, pages 435–468, 2025.

[21] Melissa Chase, F. Betül Durak, and Serge Vaudenay. Anonymous Tokens with Stronger Metadata Bit Hiding from Algebraic MACs. In *43rd Annual International Cryptology Conference, CRYPTO '23*, 2023.

[22] Melissa Chase, Sarah Meiklejohn, and Greg Zaverucha. Algebraic MACs and Keyed-Verification Anonymous Credentials. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, CCS '14, pages 1205–1216. Association for Computing Machinery, 2014.

[23] David Chaum. Security Without Identification: Transaction Systems to Make Big Brother Obsolete. *Communications of the ACM*, 28(10):1030–1044, 1985.

[24] Alessandro Chiesa and Michele Orrù. A Fiat–Shamir Transformation from Duplex Sponges. In *Theory of Cryptography, TCC '25*, 2026.

[25] Oana Ciobotaru, Maxim Peter, and Vesselin Velichkov. The Last Challenge Attack: Exploiting a Vulnerable Implementation of the Fiat-Shamir Transform in a KZG-based SNARK. Cryptology ePrint Archive, Paper 2024/398, 2024.

[26] Ronald Cramer. *Modular Design of Secure yet Practical Cryptographic Protocols*. PhD thesis, 1997.

[27] Ronald Cramer, Ivan Damgård, and Berry Schoenmakers. Proofs of Partial Knowledge and Simplified Design of Witness Hiding Protocols. In *14th Annual International Cryptology Conference, CRYPTO '94, , Proceedings*, 1994.

[28] Lesly-Ann Daniel, Sébastien Bardin, and Tamara Rezk. Binsec/Rel: Symbolic Binary Analyzer for Security with Applications to Constant-Time and Secret-Erasure. *ACM Trans. Priv. Secur.*, 2023.

[29] Quang Dao, Jim Miller, Opal Wright, and Paul Grubbs. Weak Fiat-Shamir Attacks on Modern Proof Systems. In *44th IEEE Symposium on Security and Privacy, SP '23*, pages 199–216, 2023.

[30] Alex Davidson, Ian Goldberg, Nick Sullivan, George Tankersley, and Filippo Valsorda. Privacy Pass: Bypassing Internet Challenges Anonymously. 2018.

[31] Henry de Valence. merlin: Composable proof transcripts for public-coin arguments of knowledge. https://github.com/dalek-cryptography/merlin, 2019. [Accessed on December 2025].

[32] Henry de Valence. zkp: a toolkit for Schnorr proofs. https://github.com/dalek-cryptography/zkp, 2019. [Accessed on December 2025].

[33] Nicolas Desmoulins, Antoine Dumanois, Seyni Kane, and Jacques Traoré. Making BBS Anonymous Credentials eIDAS 2.0 Compliant. 2025.

[34] Roger Dingledine, Nick Mathewson, and Paul Syverson. Tor: The Second-Generation Onion Router. In *13th USENIX Security Symposium, USENIX Security '04*. USENIX Association, August 2004.

[35] Yevgeniy Dodis and Aleksandr Yampolskiy. A Verifiable Random Function with Short Proofs and Keys. In *Public Key Cryptography, PKC 2005*, 2005.

[36] Danny Dolev, Cynthia Dwork, and Moni Naor. Non-Malleable Cryptography (Extended Abstract). In *Proceedings of the 23rd Annual ACM Symposium on Theory of Computing*, 1991.

[37] fail0verflow. Console Hacking. *27th Chaos Communication Congress*, 2010.

[38] Armando Faz-Hernandez, Sam Scott, Nick Sullivan, Riad S. Wahby, and Christopher A. Wood. Hashing to Elliptic Curves. RFC 9380 reference Implementation, 2023.

[39] Armando Faz-Hernandez, Sam Scott, Nick Sullivan, Riad S. Wahby, and Christopher A. Wood. Hashing to Elliptic Curves. RFC 9380, 2023.

[40] Amos Fiat and Adi Shamir. How to Prove Yourself: Practical Solutions to Identification and Signature Problems. In *Advances in Cryptology, CRYPTO '86*, 1986.

[41] Solana Foundation. Post Mortem: ZK ElGamal Proof Program Bug. https://solana.com/news/post-mortem-may-2-2025, 2025. [Accessed on December 2025].

[42] Matteo Frigo and abhi shelat. The Longfellow Zero-knowledge Scheme. Internet-Draft draft-google-cfrg-libzk-01, Internet Engineering Task Force, 2025. Work in Progress.

[43] Shafi Goldwasser and Yael Tauman Kalai. On the (In)security of the Fiat-Shamir Paradigm. In *44th Symposium on Foundations of Computer Science, FOCS 2003*, 2003.

[44] Google LLC. longfellow: Implementation of the Google Zero-Knowledge library for Identity Protocols. https://github.com/google/longfellow-zk, 2026. v0.8.6.

[45] Lorenzo Grassi, Dmitry Khovratovich, Christian Rechberger, Arnab Roy, and Markus Schofnegger. Poseidon: A New Hash Function for Zero-Knowledge Proof Systems. In *30th USENIX Security Symposium, USENIX Security 2021*, 2021.

[46] Lorenzo Grassi, Dmitry Khovratovich, and Markus Schofnegger. Poseidon2: A Faster Version of the Poseidon Hash Function. In *14th International Conference on Cryptology in Africa, AFRICACRYPT '23*, 2023.

[47] Bar Ilan Cryptography Research Group. libscapi: Comprehensive Open Source Library for Secure Multiparty Computation. https://github.com/cryptobiu/libscapi, 2025.

[48] Mike Hamburg. The STROBE protocol framework. 2017.

[49] Stanislaw Jarecki, Aggelos Kiayias, and Hugo Krawczyk. Round-Optimal Password-Protected Secret Sharing and T-PAKE in the Password-Only Model. In *20th International Conference on the Theory and Application of Cryptology and Information Security, ASIACRYPT 2014*, 2014.

[50] Vasilis Kalos and Greg M. Bernstein. BBS per Verifier Linkability. Internet-Draft draft-irtf-cfrg-bbs-per-verifier-linkability-02, 2025. Work in Progress.

[51] Vasilis Kalos and Greg M. Bernstein. Blind BBS Signatures. Internet-Draft draft-irtf-cfrg-bbs-blind-signatures-02, 2025. Work in Progress.

[52] Jonathan Katz, Mariana Raykova, and Samuel Schlesinger. Anonymous Credentials with Range Proofs and Rate Limiting. https://docs.rs/crate/authenticated-pseudonyms/latest/source/design/Range.pdf, 2025.

[53] Dmitry Khovratovich, Ron D. Rothblum, and Lev Soukhanov. How to Prove False Statements: Practical Attacks on Fiat-Shamir. In *45th Annual International Cryptology Conference, CRYPTO '25*, volume 16005, pages 3–26, 2025.

[54] Paul C. Kocher. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In Neal Koblitz, editor, *Advances in Cryptology, CRYPTO '96*, volume 1109. Springer, 1996.

[55] Ben Kreuter, Tancrède Lepoint, Michele Orrù, and Mariana Raykova. Anonymous Tokens with Private Metadata Bit. In *40th Annual International Cryptology Conference, CRYPTO '20*, 2020.

[56] Dock Labs. docknetwork/crypto: Rust crypto library for data privacy tools. https://github.com/docknetwork/crypto, 2025.

[57] Watson Ladd. BBS for PrivacyPass. Internet-Draft draft-ladd-privacypass-bbs-01, Internet Engineering Task Force, 2024. Work in Progress.

[58] Anja Lehmann, Andrey Sidorenko, and Alexandros Zacharakis. Vision: A Modular Framework for Anonymous Credential Systems. Cryptology ePrint Archive, Paper 2025/1981, 2025.

[59] Tobias Looker, Vasilis Kalos, Andrew Whitehead, and Mike Lodder. The BBS Signature Scheme. Internet-Draft draft-irtf-cfrg-bbs-signatures-09, 2025. Work in Progress.

[60] 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 '19*, 2019.

[61] Ueli M. Maurer. Unifying Zero-Knowledge Proofs of Knowledge. In *Progress in Cryptology - Second International Conference on Cryptology in Africa, AFRICACRYPT '09*, Lecture Notes in Computer Science, 2009.

[62] Silvio Micali, Michael Rabin, and Salil Vadhan. Verifiable Random Functions. In *Proceedings of the 40th Annual Symposium on Foundations of Computer Science, FOCS '99*, 1999.

[63] National Institute of Standards and Technology. Secure Hash Standard. NIST FIPS PUB 180-2, U.S. Department of Commerce, 1995.

[64] National Institute of Standards and Technology. SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions. NIST FIPS PUB 202, U.S. Department of Commerce, 2015.

[65] Tatsuaki Okamoto. Provably Secure and Practical Identification Schemes and Corresponding Signature Schemes. In *12th Annual International Cryptology Conference, CRYPTO '92*, 1992.

[66] OONI. Open Observatory of Network Interference (OONI). https://ooni.org, November 2025. [Accessed on December 31, 2025] https://ooni.org.

[67] Michele Orrù. Probe Security Without Identification. https://ooni.org/post/2025-probe-security-without-identification/, February 2025. [Accessed on January 8, 2026] https://ooni.org/post/2025-probe-security-without-identification/.

[68] Michele Orrù. Revisiting Keyed-Verification Anonymous Credentials. In *Proceedings of the 2025 ACM SIGSAC Conference on Computer and Communications Security, CCS 2025, Taipei, Taiwan, October 13-17, 2025*, 2025.

[69] Michele Orrù. Fiat-Shamir Transformation. Internet-Draft draft-irtf-cfrg-fiat-shamir-01, Internet Engineering Task Force, 2025. Work in Progress.

[70] Michele Orrù and Cathie Yun. Interactive Sigma Proofs. Internet-draft, Internet Engineering Task Force, 2025.

[71] Tommy Pauly, Steven Valdez, and Christopher A. Wood. The Privacy Pass HTTP Authentication Scheme. RFC 9577, 2024.

[72] Torben P. Pedersen. Non-Interactive and Information-Theoretic Secure Verifiable Secret Sharing. In *11th Annual International Cryptology Conference, CRYPTO '91*, 1991.

[73] Thomas Pornin, Aleks Kircanski, Mason Hemmel, David Wong, Janet Ghazizadeh, Mathias Hall-Andersen, and Javed Samuel. Zcash Overwinter Consensus and Sapling Cryptography Review, 2019.

[74] Oscar Reparaz, Josep Balasch, and Ingrid Verbauwhede. Dude, is my code constant time? In *Design, Automation & Test in Europe Conference & Exhibition, DATE '17*, pages 1697–1702, 2017.

[75] Markku-Juhani O. Saarinen and Jean-Philippe Aumasson. The BLAKE2 Cryptographic Hash and Message Authentication Code (MAC). RFC 7693, 2015.

[76] Amit Sahai. Non-Malleable Non-Interactive Zero Knowledge and Adaptive Chosen-Ciphertext Security. In *40th Annual Symposium on Foundations of Computer Science, FOCS 1999*, 1999.

[77] Sajin Sasy, Aaron Johnson, and Ian Goldberg. Fast Fully Oblivious Compaction and Shuffling. In *ACM SIGSAC Conference on Computer and Communications Security, CCS '22*, 2022.

[78] Samuel Schlesinger and Thibault Meunier. Privacy Pass Issuance Protocol for Anonymous Credit Tokens. Internet-Draft draft-schlesinger-privacypass-act-00, 2025. Work in Progress.

[79] Claus-Peter Schnorr. Efficient Signature Generation by Smart Cards. *Journal of Cryptology*, 4(3):161–174, 1991.

[80] Eric Schorn and Ava Howell. O(1) Labs Mina Client SDK, Signature Library and Base Components Cryptography and Implementation Review, 2022.

[81] Signal Messenger, LLC. Technology Preview: Signal Private Group System. https://signal.org/blog/signal-private-group-system/, 2019. [Accessed on January 26, 2026].

[82] Signal Messenger, LLC. POKSHO: Proof of Knowledge, Stateful Hash Object. https://github.com/signalapp/libsignal/tree/main/rust/poksho, 2025. 0.7.0.

[83] Isamu Teranishi, Jun Furukawa, and Kazue Sako. k-Times Anonymous Authentication (Extended Abstract). In *10th International Conference on the Theory and Application of Cryptology and Information Security, ASIACRYPT '04*, volume 3329, pages 308–322, 2004.

[84] The Tor Project. Lox Workspace. https://gitlab.torproject.org/tpo/anti-censorship/lox, November 2024. [Accessed on January 2, 2026] https://gitlab.torproject.org/tpo/anti-censorship/lox.

[85] Trail of Bits. The Frozen Heart vulnerability in PlonK. https://blog.trailofbits.com/2022/04/18/the-frozen-heart-vulnerability-in-plonk/.

[86] Lindsey Tulloch and Ian Goldberg. Lox: Protecting the Social Graph in Bridge Distribution. *Privacy Enhancing Technologies*, 2023(1), 2023.

[87] XLAB d.o.o. Emmy: Library for zero-knowledge proof based applications (like anonymous credentials). https://github.com/xlab-si/emmy, 2025.

[88] Cathie Yun and Christopher A. Wood. Anonymous Rate-Limited Credentials. Internet-Draft draft-yun-privacypass-crypto-arc-00, Internet Engineering Task Force, 2025. Work in Progress.

A Attacks on the Fiat–Shamir transformation

Attacks on the Fiat–Shamir transformation can be partitioned into two classes: attacks to the random oracle model itself, and practical vulnerabilities.

The first is intrinsic to the model over which transformations are considered secure; they take the form of "diagonalization" attacks and, roughly speaking, involve computing the hash function inside the statement to be proven. The hash result is then used in order to "cancel out" the re-randomization applied by the challenge. Initially discovered by Canetti et al. [19], this class of attacks has more recently found application in some SNARKs for complex circuit relations; see Khovratovich et al. [53]. No general countermeasure is available for this attack; recently however progress in this direction has recently been made by Arnon and Yogev [2] relying on the relativized random oracle model. Concretely, current counter-measures in implementations (also used in Longfellow [44]) rely on making the hash function's circuit too deep to be expressed within the statement to be proven. For Σ-protocols for linear morphisms over prime-order elliptic curve groups with standardized hash functions, this restriction effectively precludes known instantiations of the attack; however, SPONGEFISH provides an extension for proof-of-work called `spongefish-pow` that mitigates this problem in the generic case.

The second class belongs to wrong instantiations of the transformation. Countless attacks have been found in this realm, and to name a few different sub-classes of this family:

(a) Forgetting the instance (also known as *weak* Fiat–Shamir transform) [9], or parts of the instance such as the generators [85], the index [29], or key material such as public keys [73];
(b) misplacing the order of the challenge [25], or swapping the order of prover/verifier messages [41];
(c) missing bound checks on values, or that the proof string is not malleable [80, p. 7];
(d) state restoration; that is, repeating the same prover message multiple times [37].

We attempt to mitigate all above attacks within the API itself. To address (a), both the prover and verifier state must be instantiated by explicitly providing an instance, forcing the implementor to handle weak Fiat–Shamir pitfalls. Concerning (b), the prover state internally serializes the non-interactive argument string, and the verifier state internally deserializes it. Putting the burden of proof serialization into the Fiat–Shamir transformation (as in the theoretical framework) removes the risk of mismatch/mis-ordering of messages between the serialized NARG string and the non-interactive protocol. Then, to address (c), we provide traits `Encoding` and `Decoding` that have explicit security requirements and are de-coupled from the proof system. We implement a collection of inter-operable decoding and encoding functions for the most popular algebraic libraries. Finally, to mitigate (d), the prover state has access to a private random-number generator (the prover's private coins), and cannot be copied. As a consequence, it is not possible to accidentally perform state-restoration for the prover state, and get two challenges for the same prover message.

B Compressed Sigma Protocols

In 2016, the seminal work of Bootle et al. [11] showed how, under the discrete logarithm and random oracle assumptions, it is possible to build very efficient zero-knowledge succinct arguments (zk-SNARKs) for linear relations over elliptic-curve groups. This has been later re-integrated in the Σ-protocol theory by Cramer & Attema [3], with a protocol for generic linear forms, called **compressed Σ-protocols**, and by Bootle et al. [12] as **sumcheck arguments** over generic moduli.

Roughly speaking, in a compressed Σ-protocol proving that $\vec{X} = \mathbf{M}\vec{x}$, in place of sending the response $\vec{r}$ satisfying $\mathbf{M}\vec{r} = \vec{R} := e \cdot \vec{X} + \vec{K}$, the prover will split the response vector into two segments of size $n/2$, $(\vec{r}_\mathrm{L}, \vec{r}_\mathrm{R}) = \vec{r}$ and will instead consider the two statements $\mathbf{M}_\mathrm{L} \cdot \vec{r}_\mathrm{L}$, $\mathbf{M}_\mathrm{R} \cdot \vec{r}_\mathrm{R}$, jointly proven together with a c as a **reduced witness** $\vec{r}_\mathrm{L} + c\vec{r}_\mathrm{R}$. The prover and verifier then recursively reduce again the proof statement until the length is trivial and the response can be sent. More precisely:

- If the witness size $n = 1$, the prover sends the response as in classical sigma protocols. The verifier checks $Mr = R$.[6]
- Else, the verifier will send a challenge c and the prover will send the *cross terms* $A := \mathbf{M}_\mathrm{R}\vec{r}_\mathrm{L}$, $B := \mathbf{M}_\mathrm{L}\vec{r}_\mathrm{R}$. The prover and verifier will recursively engage in proving the reduced statement of size $n/2$:

$$\vec{r}'\mathbf{M}' = R$$

  where $\mathbf{M}' := c\mathbf{M}_\mathrm{L} + \mathbf{M}_\mathrm{R}$, $\vec{r}' := \vec{r}_\mathrm{L} + c\vec{r}_\mathrm{R}$, and $R' := A + cR + B$.

We implement this class of proofs in an experimental branch, to show how our instance builder can also be used within more advanced proof systems.

C k-times anonymous authentication

A rate-limiting credential protocol, also known as *k-times anonymous authentication* in the literature [8, 15, 52, 83], typically divides time into *epochs* (say, one day). In each epoch, a given credential can be presented at most `max_pres` times. These presentations are *unlinkable*; that is, the issuer cannot tell whether two executions of the protocol are using

[6] In the case where the initial relation had $n = 1$, $R = ex + K$. Otherwise, this is recursively defined.

the same credential or different credentials, as long as the rate limit is adhered to.

One strategy for implementing rate limiting [15] is to have each credential contain a random key. When presenting the credential, the client outputs $V = f_{key}(pres_num)$, where f is a verifiable unpredictable function (VUF) or a verifiable random function (VRF) [62] that changes each epoch, $0 \leq pres_num < max_pres$, and the client issues a proof that V is computed correctly, and that $pres_num$ is in the correct range. The issuer checks the proof, and ensures that it never sees the same V twice in the same epoch.

We use the Dodis–Yampolskiy VUF $V = f_{key}(pres_num) = \frac{1}{key+pres_num}E$ [35], where E is a generator of $\mathbb{G}$ obtained by using a hash-to-group function [39] on an input containing the epoch number. The proof that V is computed correctly is simply proving that $E = (key + pres_num)V$, which is a linear combination statement, and we must also prove a range statement on $pres_num$. The resulting protocol can be compactly specified using the CMZ crate as:

```
CMZ! { Cred: key }
CMZ! { PresNum: pres_num }

muCMZProtocol! { pres_cred<max_pres,
    @Epoch_base, @VRF_output>,
  [ C: Cred { key: H },
    P?: PresNum { pres_num: H } ],
  ,
  Epoch_base = (C.key + P.pres_num)*VRF_output,
  (0..max_pres).contains(P.pres_num),
}
```

In the above, there are two small notations not introduced before. The `@` signs in the list of public parameters to the protocol indicate that the `Epoch_base` and `VRF_output` parameters are `Point`s instead of the default `Scalar`s. The `?` for the presented `PresNum` credential indicates that that credential is *incomplete*, and the issuer should not expect it to have a correct MAC, or to have been issued by the issuer at all. Such incomplete presented credentials in the CMZ crate allow the client to supply hidden attributes (such as `pres_num` in this example) that are not part of any issued credential, yet can appear in the statement tree that is proved in zero knowledge.

When performing this protocol, the client has their issued `Cred` credential, and creates their own `PresNum` incomplete credential, filling in the `pres_num` attribute with a value in the range `0..max_pres` not yet used in this epoch. The client also constructs a `Params` struct with the values `max_pres`, `Epoch_base` (the E value generated as above), and `VRF_output` (the V value generated as above). The client calls `prepare` with these inputs, yielding a `Request`. The client sends `VRF_output` and the `Request` to the issuer. The issuer checks that it has not seen the `VRF_output` before in this epoch; it computes `Epoch_base` as above, and it knows `max_pres`. It calls `handle` to consume the `Request` and verify the proof. If `handle` returns successfully, the client is accepted. As this protocol does not issue any credentials, there is no `Reply` or `finalize` step.

Table 4: Lines of code required for a developer to declare request and issuance proofs for Lox using the zkp crate's `define_proof` macro versus our `muCMZProtocol` macro. The zkp crate requires the definition of a client request proof as well as a separate issuance proof whereas using SIGMA-RS, the proof is defined once in a statement for both request and issuance. We see the reduction in code from CMZ14 to μCMZ using the zkp library but a much larger reduction in code for all Lox protocols using SIGMA-RS.

Lox Protocol	**Lines of code** (LOC)		
	zkp crate		**SIGMA-RS**
	CMZ14	**μCMZ**	**μCMZ**
Open Invitation	27	20	4
Trust Promotion	30	28	7
Trust Migration	42	30	8
Level Up	78	56	11
Issue Invitation	100	45	12
Redeem Invitation	56	36	7
Check Blockage	20	16	6
Blockage Migration	59	37	9
Update Invite	49	23	7
Update Credential	71	29	9

D Additional Lox protocol evaluations

In Section 8.1 we compared the lines of code required to implement each of the Lox protocols with the SIGMA-RS crates against the original implementation that used the zkp crate [32] to implement both the CMZ [22] and μCMZ [68] credential scheme. In Table 4 we drill down to show the size of the code each implementation requires simply for the declarations of the required zero-knowledge proofs. As a quick sample of the difference in complexity, we have included code snippets for the proofs from one of the μCMZ protocols using the zkp crate compared with using the SIGMA-RS crates in Figures 5 and 6 respectively.

In addition to the timings reported in Section 8.1, we also measured the request and response sizes in bytes for the μCMZ versions of both the original zkp implementation of Lox, and our re-implementation using SIGMA-RS, displayed in Table 5. The sizes collected for SIGMA-RS are comparable to the original Lox implementation [84], and are even slightly smaller for the most part.

```
define_proof! {
    requestproof,
    "Level Upgrade Request",
    (bucket, since, invremain, blockages, zbucket, zsince, zinvremain,
    zblockages, negzQ,
    zbucket_reach, negzQ_reach,
    s, id_client,
    g0, g1, g2, g3, g4, g5, g6, g7, g8,
    zg0, zg1, zg2, zg3, zg4, zg5, zg6, zg7, zg8,
    wg0, wg1, wg2, wg3, wg4, wg5, wg6, wg7, wg8,
    yg0, yg1, yg2, yg3, yg4, yg5, yg6, yg7, yg8,
    h0, h1, h2,
    zh0, zh1, zh2,
    wh0, wh1, wh2,
    yh0, yh1, yh2),
    (P, CBucket, CSince, CInvRemain, CBlockages, V, Xid, Xbucket, Xsince,
    Xinvremain, Xblockages,
    P_reach, CBucket_reach, V_reach, Xbucket_reach,
    CommitLoxBlind,
    CG0, CG1, CG2, CG3, CG4, CG5, CG6, CG7, CG8,
    CG0sq, CG1sq, CG2sq, CG3sq, CG4sq, CG5sq, CG6sq, CG7sq, CG8sq,
    CH0, CH1, CH2,
    CH0sq, CH1sq, CH2sq),
    (A) :
    // Blind showing of the Lox credential
    CBucket = (bucket*P + zbucket*A),
    CSince = (since*P + zsince*A),
    CInvRemain = (invremain*P + zinvremain*A),
    CBlockages = (blockages*P + zblockages*A),
    // Blind showing of the Bucket Reachability credential; note the
    // same bucket is used in the proof
    CBucket_reach = (bucket*P_reach + zbucket_reach*A),
    // User blinding of the Lox credential to be issued
    CommitLoxBlind = (s*A + id_client * Xid + bucket * Xbucket + blockages * Xblockages),
    // Prove CSince encodes a value at least LEVEL_INTERVAL
    // days ago (and technically at most LEVEL_INTERVAL+511 days
    // ago): first prove each of g0, ..., g8 is a bit by proving that
    // gi = gi^2
    CG0 = (g0*P + zg0*A), CG0sq = (g0*CG0 + wg0*A), CG0sq = (g0*P + yg0*A),
    CG1 = (g1*P + zg1*A), CG1sq = (g1*CG1 + wg1*A), CG1sq = (g1*P + yg1*A),
    CG2 = (g2*P + zg2*A), CG2sq = (g2*CG2 + wg2*A), CG2sq = (g2*P + yg2*A),
    CG3 = (g3*P + zg3*A), CG3sq = (g3*CG3 + wg3*A), CG3sq = (g3*P + yg3*A),
    CG4 = (g4*P + zg4*A), CG4sq = (g4*CG4 + wg4*A), CG4sq = (g4*P + yg4*A),
    CG5 = (g5*P + zg5*A), CG5sq = (g5*CG5 + wg5*A), CG5sq = (g5*P + yg5*A),
    CG6 = (g6*P + zg6*A), CG6sq = (g6*CG6 + wg6*A), CG6sq = (g6*P + yg6*A),
    CG7 = (g7*P + zg7*A), CG7sq = (g7*CG7 + wg7*A), CG7sq = (g7*P + yg7*A),
    CG8 = (g8*P + zg8*A), CG8sq = (g8*CG8 + wg8*A), CG8sq = (g8*P + yg8*A),
    // Then we'll check that CSince + LEVEL_INTERVAL*P + CG0 + 2*CG1
    // + 4*CG2 + 8*CG3 + ... + 256*CG8 = today*P by having the verifier
    // plug in today*P - (CSince + LEVEL_INTERVAL*P + 2*CG1 + 4*CG2
    // + ... + 256*CG8) as its value of CG0.

    // Prove CBlockage encodes a value at most MAX_BLOCKAGES (and at least MAX_BLOCKAGES-7)
    CH0 = (h0*P + zh0*A), CH0sq = (h0*CH0 + wh0*A), CH0sq = (h0*P + yh0*A),
    CH1 = (h1*P + zh1*A), CH1sq = (h1*CH1 + wh1*A), CH1sq = (h1*P + yh1*A),
    CH2 = (h2*P + zh2*A), CH2sq = (h2*CH2 + wh2*A), CH2sq = (h2*P + yh2*A)
    // Then we'll check that CBlockage + CH0 + 2*CH1 + 4*CH2 =
    // MAX_BLOCKAGES*P by having the verifier plug in MAX_BLOCKAGES*P -
    // (CBlockage - 2*CH1 - 4*CH2) as its value of CH0.
}

define_proof! {
    blindissue,
    "Level Upgrade Issuing",
    (x0, x0tilde, xlevel, xsince, xinvremain,  b),
    (P, BlindLoxQ, X0, Xlevel, Xsince, Xinvremain,
    Plevel, Psince, Pinvremain, CommitLoxBlind),
    (A, B):
    Xlevel = (xlevel*A),
    Xsince = (xsince*A),
    Xinvremain = (xinvremain*A),
    X0 = (x0*B + x0tilde*A),
    P = (b*A),
    BlindLoxQ = (b*CommitLoxBlind + x0 * P + xlevel * Plevel
                + xsince * Psince + xinvremain * Pinvremain)
}
```

Figure 5: The Lox "level up" protocol, as originally implemented using the zkp crate

```
muCMZProtocol! { level_up<credential_expiry, eligibility_max_age, max_blockage, today>,
    [ L: Lox { id: R, bucket: H, trust_level: R, level_since: H, invites_remaining: H, blockages: H },
    B: BucketReachability { date: R, bucket: H } ],
    N: Lox {id: J, bucket: H, trust_level: R, level_since: S, invites_remaining: I, blockages: H },
    (credential_expiry..=eligibility_max_age).contains(L.level_since),
    (0..=max_blockage).contains(L.blockages),
    B.date = today,
    B.bucket = L.bucket,
    N.bucket = L.bucket,
    N.trust_level = L.trust_level + 1,
    N.blockages = L.blockages,
}
```

Figure 6: The Lox "level up" protocol, as implemented using our SIGMA-RS crates

Table 5: Bytes sent and received for each Lox Protocol for a Lox authority with a bridge table size of 30 bridges.

Protocol	**Size** (B)			
	zkp crates + μCMZ		**SIGMA-RS + μCMZ**	
	Request	**Response**	**Request**	**Response**
Open Invitation	236	630	249	543
Trust Promotion	2120	2172	1677	2285
Trust Migration	744	328	845	241
Level Up	3080	360	2253	241
Issue Invite	1096	776	1101	401
Redeem Invite	1288	424	1037	241
Check Blockage	648	212	685	325
Blockage Migration	968	360	973	241
Update Invite	720	232	557	209
Update Credential	976	232	749	209