ABSTRACT

Title of Dissertation: ZERO-KNOWLEDGE PROOFS FOR PROGRAMMABLE ANONYMITY, MODERATION, AND REPUTATION

Michael Allan Rosenberg
Doctor of Philosophy, 2024

Dissertation directed by: Professor Ian Miers and
Professor Jonathan Katz,
University of Maryland, Department of Computer Science

Anonymous credentials deal with a core tension of privacy-enhancing technologies (PET), namely the desire to participate in society versus the desire to remain anonymous. But despite decades of research, anonymous credential schemes have not received nearly as much general uptake as other PET such as end-to-end encryption. This is due, in part, to its high barriers of design and deployment.

Many existing anonymous credential schemes are constructed by first fixing notions of identity and what should selectively be revealed, and then designing towards that goal. This yields just-so schemes built on primitives like Pedersen commitments and blind signatures. But while these schemes are often efficient, they often require an expert redesign when the notion of identity changes, or the statement to selectively reveal changes (e.g., adding a range proof to a system that previously only permitted equality proofs). It is possible to flip the order of operations, i.e., to design a proof system and then let users program their own notions of identity and what they want to show. Concretely, using modern, general-purpose zero-knowledge proof schemes and their deep tooling, it is possible to design extensible solutions to the problems of identity, moderation and reputation.

In this dissertation, I present research which builds novel, extensible, and practical privacy-enhancing technologies from succinct noninteractive zero-knowledge proofs of knowledge (zk-

SNARKs). These works are: SNARKBlock—a scalable anonymous blocklisting scheme, zk-creds—a construction of anonymous credentials which are bootstrappable from existing government-issued documents, and zk-promises—a framework for asynchronous anonymous blocklisting and reputation which supports complex notions of reputation.

ZERO-KNOWLEDGE PROOFS FOR PROGRAMMABLE ANONYMITY, MODERATION, AND REPUTATION

by

Michael Allan Rosenberg

Dissertation submitted to the Faculty of the Graduate School of
the University of Maryland, College Park, in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
2024

Advisory Committee:

Professor Ian Miers, Co-chair (CMSC)

Professor Jonathan Katz, Co-chair (CMSC)

Professor Dana Dachman-Soled, Dean's Representative (ECE)

Professor Michelle Mazurek, Member (CMSC)

Professor Pratyush Mishra, Special Member (University of Pennsylvania)

Dedication

This dissertation is dedicated to my parents, whose unceasing support has carried me further than I'd ever believe. And to my friends, who I have spent more hours on the phone with than without, cooking, walking, and watching TV. Without you, I would have gone mad as a meat axe.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Since the beginning of the internet, it has been a tenet that your real-world identity doesn't matter in cyberspace. Dedicated early adopters who wanted to improve anonymity created tools such as remailers, open web proxies, and FTP drop servers. But, like anything else, these technologies turned out to have dual uses. Remailers were repurposed for spam, proxies were used to obfuscate the origin of cyber attacks, and FTP servers were used to distribute illegal material. Over time, most of these services have shut down or become more restrictive, but the need and desire for anonymity remains. This tension of anonymity and accountability is at the heart of modern privacy-enhancing technologies (PET) research. How do you allow people to communicate anonymously and with dignity while preventing malicious actors from flooding your services with spam or using them to spread harmful material?

The notion of *anonymous credentials* was developed to address this question. Broadly, an anonymous credential scheme is a protocol wherein an entity with a persistent identity interacts with some degree of anonymity with another entity. An anonymous credential is usually a cryptographic commitment to identifying data which can be used as the basis for accessing certain system. The committed information is considered private, and may be *selectively revealed* to specific recipients. It is not unlike how a public library works, in principle: new patrons must present ID with residency info to receive a card, but once issued, the card bears no identifying information. Thus no future

browsing or check-outs are linked to the patron's identity, and are instead *pseudonymous*—linking to an account number with no further information.

There is a long line of important and high-profile work in anonymous credentials [Cha81, Cha82, CL03, CHL05, BCC04, CHK+06, ASM06, SAB+19, San22]. By and large, these use Pedersen commitments [Ped92] to represent a credential and an interactive blind signature protocol (RSA [Cha82] or BBS+ [CDL16]) for issuance. Yet of all of these works, few have been deployed at scale, and those which have have had notable drawbacks. For example, Intel deploys Direct Anonymous Attestation (DAA) [San22] to handle enclave attestation and revocation for all SGX-enabled chips. But attestation certificate revocation has faced significant efficiency issues due to the fact that attestation time is linear in the size of the revocation list [vSSY+22, Appendix B].

An issue in common with these schemes is that they are highly bespoke—e.g., Coconut [SAB+19] only permits predicates which are systems of linear equations of the committed values—and are challenging to modify. Of course, it is possible to make more complicated credentials and predicates, such as computing a deterministic ElGamal encryption and showing set membership to prove you are a Signal group administrator [CPZ20], but the schemes are again bespoke and quite technical. This not only makes schemes hard to customize to user-specific cases, but it also makes them harder to iterate on, since they require expert knowledge to create.

A promising path forward is to lower barriers of entry, and use general-purposes zero-knowledge proof schemes for our PET use cases. Modern zero-knowledge succinct noninteractive proofs of knowledge (zkSNARKs) are fast, aggregatable, and have a large and still-growing ecosystem of tools that let non-experts define data structures and predicates for their own use cases [Ar21, Bow17a, MTBI+22, OBW22, Zok, Noi, Leo]. These algorithms and tooling can be leveraged to provide privacy-enhancing technologies which are not only efficient, but also modular enough to be hacked on by a programmer with moderate zero-knowledge experience, a significantly lower barrier than domain expert.

This dissertation. In this dissertation, I present three works that I have coauthored which bring to bear the flexibility of general-purpose zkSNARKs on problems relating to anonymous credentials

and accountability mechanisms more broadly.

Chapter 3 presents SNARKBlock [RMM22],[1] a protocol for zero-knowledge blocklisting with server-side verification that is logarithmic in the size of the blocklist. SNARKBlock is also the first approach to support ad-hoc, federated blocklisting: websites can mix and match their own blocklists from other blocklists and dynamically choose which identity providers they trust. Its core technical advancement, of separate interest, is the HICIAP zero-knowledge proof system, which addresses a common problem in privacy-preserving protocols: reusing previously computed zero-knowledge proofs without revealing that they are the same proofs. These techniques already exist for monolithic proofs (i.e., rerandomzing a Groth16 proof), but no efficient techniques exist for schemes which combine several underlying proofs in order to prove a conjunctive statement with a common hidden value shared between the statements (e.g., a commitment representing a credential). HICIAP resolves this problem by aggregating n Groth16 proofs with a common input into one $O(\log n)$-sized, $O(\log n)$-verification time conjunctive proof. Because HICIAP is zero-knowledge, repeated showings of the same aggregate or an updated aggregate are unlinkable even though the underlying Groth16 proofs are never recomputed.

Chapter 4 presents zk-creds [RWGM23],[1] an anonymous credentials scheme that uses general-purpose zero-knowledge proofs to (1) remove the need for credential issuers to hold signing keys: credentials can be issued to a bulletin board instantiated as a transparency log, Byzantine system, or even a blockchain; (2) convert existing identity documents into anonymous credentials without modifying documents or coordinating with their issuing authority; and (3) allow for flexible, composable, and complex identity statements over multiple credentials. Concretely, identity assertions using zk-creds take less than 150ms in a real-world scenario of using a passport to anonymously access age-restricted videos.

Chapter 5 presents zk-promises [SRM24], a framework that adds a notion of *callbacks* to a class of anonymous credentials schemes of which the Zcash *account model* [HBHW21] is perhaps the best known example. With this extension, an anonymous user can send a callback to a service

[1]Work supported by a National Defense Science and Engineering Graduate (NDSEG) Fellowship.

provider which, when called, will mutate the user's credential as defined in some zero-knowledge circuit. When the credential contains a reputation score, this naturally becomes an anonymous reputation scheme. This scheme, which is itself a generalization of anonymous blocklisting, is a fundamental departure from the anonymous blocklisting scheme in Chapter 3. zk-promises not only addresses the drawbacks of existing approaches to anonymous reputation, but also enables sophisticated moderation tools, including temporary bans, multidimensional reputation assessment, and dynamic rate limiting based on user reputation. Concretely, a user reputation proof in a scheme with these features takes less than 1s.

Chapter 2

Preliminaries

We provide common notation and constructions which will be used in the following chapters.

1 Notation

We write $x := z$ to denote variable assignment, $y \leftarrow\!\!\$\, S$ to denote sampling uniformly from a set S, and $x \leftarrow \mathscr{D}$ for sample from any distribution $\mathscr{D}$. We write $y \leftarrow \mathsf{A}(x)$ to denote sampling from a probabilistic algorithm A, and $y := \mathsf{A}(x;r)$ to denote the execution of A on input x, using random coins r. $\langle \mathsf{A}, \mathsf{B} \rangle$ denotes the transcript of a protocol run between algorithms A and B. We write $\bar{x} := x_1, x_2, \ldots$ to denote a variable-length list. The security parameter of our system is denoted by λ.

2 Zero-knowledge proofs

For an arbitrary, efficiently computable predicate P, we say that *a proof of knowledge of a relation* $R = \{(x;w) : P(x,w)\}$ with respect to an *instance* x is a proof of knowledge of the *witness* w such that $P(x,w)$ is satisfied. We will often refer to x as a *public input* and w as a *private input*, and we will use zero-knowledge proofs of knowledge (ZKPs) for various relations in order to hide w from the verifier.

A zero-knowledge proof scheme is defined by three algorithms:

$\mathsf{Setup}(P) \to \mathsf{crs}$ Receives a description of predicate P and returns a *common reference string* which will be used for proving and verifying.

$\mathsf{Prove}(\mathsf{crs}, x, w)$ Performs and interactive protocol which demonstrates $(x,w) \in R$.

$\mathsf{Vfy}(\mathsf{crs}, x)$ Interacts with Prove to verify that there exists a w such that $(x,w) \in R$.

A ZKP scheme has *perfect correctness* iff for every $(x,w) \in R$ and honestly computed crs, $\langle \mathsf{Vfy}(\mathsf{crs}, x), \mathsf{Prove}(\mathsf{crs}, x, w) \rangle$ succeeds. A ZKP scheme is *public-coin* iff the messages sent by the verifier are chosen uniformly at random, and independently of the prover's messages. We reproduce the definitions of knowledge soundness and perfect honest verifier zero-knowledge from [BMM+19] below.

Definition (Knowledge-sound argument)**.** *A public-coin argument* $\Pi = (\mathsf{Setup}, \mathsf{Prove}, \mathsf{Verify})$ *on a relation R is knowledge-sound with error* $\kappa(\lambda)$ *iff for all deterministic efficient (possibly dishonest) provers* P^**, there exists an efficient extractor* E *such that for all PPT adversaries* A*,*

$$\Pr\left[\mathsf{tr}\ \textit{accepts} \wedge (x,w) \notin R \;\middle|\; \begin{array}{c} \mathsf{crs} \leftarrow \mathsf{Setup}(1^\lambda) \\ (x, \mathsf{tr}) \leftarrow \langle \mathsf{P}^*, \mathsf{Verify} \rangle (\mathsf{crs}) \\ w \leftarrow \mathsf{E}^{P^*}(\mathsf{crs}, x, \mathsf{tr}) \end{array}\right] \leq \kappa(\lambda).$$

We say Π *is knowledge-sound iff it is knowledge-sound with negligible error.*

Definition (Perfect honest-verifier zero-knowledge (HVZK))**.** *Let* $\Pi = (\mathsf{Setup}, \mathsf{Prove}, \mathsf{Verify})$ *be an interactive argument of knowledge on a relation R. Let an adversary be a pair of PPT algorithms* $\mathsf{A} = (\mathsf{A}_0, \mathsf{A}_1)$ *such that* $\mathsf{A}_0(\mathsf{crs})$ *picks an instance, witness, and random coins* (x, w, ρ)*; and* $\mathsf{A}_1(\mathsf{tr})$ *decides whether a transcript is the result of a simulation or not.*

Π *is perfect honest-verifier zero-knowledge iff there exists an efficient simulator* Sim *such that*

for all adversaries $\mathsf{A} = (\mathsf{A}_0, \mathsf{A}_1)$,

$$\Pr\left[\begin{array}{r} \mathsf{crs} \leftarrow \mathsf{Setup}(1^\lambda) \\ (x,w) \in R \wedge \mathsf{A}_1(\mathsf{tr}) \;\middle|\; (x,w,\rho) \leftarrow \mathsf{A}_0(\mathsf{crs}) \\ \mathsf{tr} \leftarrow \mathsf{Sim}(\mathsf{crs},x;\rho) \end{array}\right]$$

$$= \Pr\left[\begin{array}{r} \mathsf{crs} \leftarrow \mathsf{Setup}(1^\lambda) \\ (x,w) \in R \wedge \mathsf{A}_1(\mathsf{tr}) \;\middle|\; (x,w,\rho) \leftarrow \mathsf{A}_0(\mathsf{crs}) \\ \mathsf{tr} \leftarrow \langle \mathsf{Prove}(\mathsf{crs},x,w), \mathsf{Vfy}(\mathsf{crs},x;\rho)\rangle \end{array}\right]$$

Finally, when a ZKP scheme is noninteractive, Prove algorithm produces a proof π, and Vfy accepts π as an input. We say a non-interactive zero-knowledge proof scheme is *succinct* if the runtime of Vfy is $O(\log|P|)$ where $|P|$ represents the size of the description of P. A succinct non-interactive ZKP scheme is called a *zkSNARK*.

3 Groups and pairings

We will work exclusively with prime-order groups and their associated scalar fields. Group elements are denoted with capital letters $G \in \mathbb{G}$, while field elements are lowercase $r \in \mathbb{F}$. *Vectors* are bolded: $\mathbf{A} \in \mathbb{G}^n$, and $\mathbf{r} \in \mathbb{F}^n$. We write $\mathbf{A}_{[:k]}$ to denote the first k elements of $\mathbf{A} \in \mathbb{G}^n$, and $\mathbf{A}_{[k:]}$ to denote the last $n-k$ elements. We say that a bilinear function $e : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$ is a *type-3 bilinear pairing* if there is no efficiently computable group homomorphism from $\mathbb{G}_2$ to $\mathbb{G}_1$. We say e is *degenerate* if there is a non-identity $G \in \mathbb{G}_1$ such that $e(G,H) = 1$ for all $H \in \mathbb{G}_2$. Following convention, we use additive notation for $\mathbb{G}_1$ and $\mathbb{G}_2$, and multiplicative notation for $\mathbb{G}_T$.

For vectors $\mathbf{A} \in \mathbb{G}_1^n$ and $\mathbf{B} \in \mathbb{G}_2^n$ and a bilinear pairing e, we write $\mathbf{A} * \mathbf{B}$ to denote the *inner pairing product* $\prod_{i=1}^n e(A_i, B_i)$. For vectors $\mathbf{A} \in \mathbb{G}^n$ and $\mathbf{r} \in \mathbb{F}^n$ we write $\mathbf{A}^{\mathbf{r}}$ to denote the *multi-scalar multiplication* (MSM) $\sum_{i=1}^n r_i A_i$, and write $\mathbf{r} \odot \mathbf{A}$ to denote the element-wise multiplication $(r_1 A_1, \ldots, r_n A_n)$. For a field element $x \in \mathbb{F}$, we denote $[x]_1 \coloneqq xG$ and $[x]_2 \coloneqq xH$, where G and H

are the canonical generators of $\mathbb{G}_1$ and $\mathbb{G}_2$, respectively.

4 Groth16

We briefly describe the trusted-setup zkSNARK scheme defined in [Gro16]. At a high level, given a description of an arithmetic circuit (over the scalar field of a pairing-friendly elliptic curve), a Groth16 proof proves that a circuit is satisfied by a set of public wires (instance elements) and private wires (witness elements).

Let $e : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$ be an efficiently-computable, non-degenerate, type-3 bilinear pairing, where $|\mathbb{G}_1| = |\mathbb{G}_2| = |\mathbb{G}_T|$ is a prime p and $p > 2^\lambda$. Let G be a generator of $\mathbb{G}_1$ and H be a generator of $\mathbb{G}_2$. We use $\mathbb{F}$ to denote the finite field $\mathbb{Z}/p\mathbb{Z}$. The Groth16 scheme defines five procedures: the base three necessary for any noninteractive zero-knowledge proof scheme, plus Prepare for preparing public inputs known in advance for use in verification, and Rerand for rerandomizing pre-existing proofs.

$\mathsf{Setup}(\mathsf{desc}) \to \mathsf{crs}$ Generates a common reference string for the given arithmetic circuit description. Specifically, this uniformly samples $\alpha, \beta, \gamma, \delta, x \leftarrow\!\!\$\ \mathbb{F}$ and computes $\mathsf{crs} = ([\sigma_1]_1, [\sigma_2]_2)$, where

$$\sigma_1 = \left(\begin{array}{c} \alpha, \beta, \delta, \{x^i\}_{i=0}^{n-1}, \left\{ \frac{\beta u_i(x) + \alpha v_i(x) + w_i(x)}{\gamma} \right\}_{i=0}^{\ell} \\ \left\{ \frac{\beta u_i(x) + \alpha v_i(x) + w_i(x)}{\delta} \right\}_{i=\ell+1}^{m}, \left\{ \frac{x^i t(x)}{\delta} \right\}_{i=0}^{n-2} \end{array} \right) \qquad \sigma_2 = (\beta, \gamma, \delta, \{x^i\}_{i=0}^{n-1}),$$

and where u, v, w, h, t are polynomials that are defined by the circuit, m is the total number of wires in the circuit, ℓ is the number of public input wires, n is the number of constraints that are used to represent the circuit, and $a_0 = 1$.

$\mathsf{Prove}(\mathsf{crs}, \{a_i\}_{i=0}^{\ell}, \{a_i\}_{i=\ell+1}^{m}) \to \pi$ Proves the circuit described by crs is satisfied, where $a_0, \ldots, a_\ell \in \mathbb{F}$ represent the circuit's public input wires and $a_{\ell+1}, \ldots, a_m \in \mathbb{F}$ represent the private wires.

Specifically, this samples $r, s \leftarrow\$ \mathbb{F}$ and computes $\pi := (A, B, C) = ([\eta]_1, [\theta]_2, [\iota]_1)$, where

$$\eta = \alpha + \sum_{i=0}^{m} a_i u_i(X) + r\delta \qquad \theta = \beta + \sum_{i=0}^{m} a_i v_i(X) + s\delta$$

$$\iota = \sum_{i=\ell+1}^{m} \frac{a_i(\beta u_i(X) + \alpha v_i(X) + w_i(X)) + h(X)t(X)}{\delta} + \eta s + \theta r - rs\delta.$$

$\mathsf{Prepare}(\mathsf{crs}, \{a_{i_j}\}_{j=1}^{t}) \to \hat{S}$ Aggregates any subset of public inputs into a single group element called a *prepared input*: $\hat{S} = \sum_{j=1}^{t} a_{i_j} W_{i_j}$, where W_i are the CRS values whose coefficient represents the value of the i-th wire of the circuit, i.e., $W_i = \left[\frac{\beta u_i(x) + \alpha v_i(x) + w_i(x)}{\gamma}\right]_1$.

$\mathsf{Vfy}(\mathsf{crs}, \pi, \{a_i\}_{i=0}^{\ell}) \to \{0,1\}$ Verifies the proof $\pi = (A, B, C)$ by checking the relation,

$$e(A, B) \stackrel{?}{=} e([\alpha]_1, [\beta]_2) \cdot e(C, [\delta]_2) \cdot e\left(\sum_{i=0}^{\ell} a_i W_i, [\gamma]_2\right).$$

Vfy permits any subset of the public inputs to be prepared as above, by simply using the precomputed sum in the last term.

$\mathsf{Rerand}(\pi) \to \pi'$ Rerandomizes the proof $\pi = (A, B, C)$ by sampling $\zeta, \omega \leftarrow\$ \mathbb{F}$ and computing

$$\pi' := (\zeta^{-1} A, \zeta B + \zeta\omega [\delta]_2, C + \omega A).$$

The output of Rerand is statistically indistinguishable from a fresh proof of the same underlying statement [BKSV20, Theorem 3].

Chapter 3

SNARKBlock: Federated Anonymous Blocklisting from Hidden Common Input Aggregate Proofs

Work coauthored with Mary Maller and Ian Miers, and published in the proceedings of IEEE S&P 2022 [RMM22].

1 Introduction

Moderation is a powerful tool for combating online harassment, trolling and spam messages. But banning an account on one platform has an obvious problem: it leaves the user free to post under other accounts and on other platforms. As a result, moderation tends towards stronger centralized *identity providers* (e.g., Facebook's real-name policy [Fac]) and the linking of disparate pseudonymous identities within and across platforms. Tying users' online speech to a centralized identity provider poses major problems for the decentralized web and user privacy, and can have a chilling effect on free speech.

Providing both privacy and moderation is a challenge: a *user* posting anonymously on a forum presents two problems to the forum operator, termed *service provider*: access control and revocation. First, because the user's identity is unknown at post submission, the service provider cannot verify

that the user is authorized to post (i.e., isn't blocked). Second, because the user's identity is not linked to the post, and posts are not linked together, the service provider cannot revoke the user's posting permissions (i.e., block the user) if their current post violates forum policies. Linking posts together raises privacy concerns that may be undesirable on a single forum and are intolerable if applied across the web.

1.1 Zero-knowledge proofs of blocklist non-membership

BLAC [TAKS10] introduces the first solution to anonymous blocklisting without a trusted third party. It provides users with long-term identities and allows them to prove, in zero-knowledge, that they are not on a blocklist.

The approach introduced by BLAC, which we formalize as a *zero-knowledge blocklist* (ZKBL), is conceptually simple. A user's identity is a random PRF key k signed by an identity provider to ensure Sybil resistance. Anonymous comments and posts are associated with a tag $\mathsf{tag} := \mathsf{Prf}_k(\mathsf{nonce})$. A blocklist $\mathscr{L}$ consists of tuples $(\mathsf{tag}, \mathsf{nonce})$ from offending posts. A user *attests* that they are not blocked by presenting a fresh $(\mathsf{tag}, \mathsf{nonce})$ pair and a zero-knowledge proof that (1) tag is computed honestly; (2) k is signed by a valid party; and (3) none of the blocklisted tags were generated by k, i.e., $\forall(\mathsf{tag}', \mathsf{nonce}') \in \mathscr{L} : \mathsf{Prf}_k(\mathsf{nonce}') \neq \mathsf{tag}'$. A user is blocked by placing an offending $(\mathsf{tag}, \mathsf{nonce})$ pair on $\mathscr{L}$.

At its core, a ZKBL is a specialized zero-knowledge proof on the PRF evaluation, tag inequalities, and identity signature. Both security and privacy depend, mainly, on the zero-knowledge proof. This gives ZKBLs their main advantage: because the proofs are over arbitrary ban lists, the system is ad-hoc. We do not need a central party to coordinate bans as in [BCD+17, CL02, LLX07, VB20], or worse, a trusted third party who can deanonymize users [Cha85, Cv91, BMW03]. Additionally, it is a major advantage for deployment when a ZKBL supports private federated identity, i.e., supports issuance where a user can receive an identity from a choice of identity providers, and service providers can choose which identity providers they will accept.

1.2 Existing ZKBLs are impractical for both clients and servers

Unfortunately, the current approach for ZKBLs [TAKS10] requires the server to do linear work in the size of the blocklist when verifying a non-membership attestation. If the size of the blocklist and the number of attestations per second is proportional to the number of total users, then the service-provider's workload grows *quadratically* as their site scales. This is costly under normal circumstances and can be a major denial of service vector if an attacker can make concurrent posts or obtain even a small number of Sybil accounts that are later banned.

Almost as problematically, proof sizes are also linear in the size of the blocklist. At 144B per list entry, a single non-membership proof for a 4M-entry blocklist would require a client to upload 549MiB of data over a residential or mobile connection.

1.3 Our contribution

We design, implement, and benchmark SNARKBLOCK, a protocol for zero-knowledge blocklists which improves on the state of the art by offering log-sized proofs and log-time verification. We show that this is indeed an improvement over other available succinct methods such as monolithic Groth16 proofs or recursive proofs.

Beyond improved performance for ZKBLs, SNARKBLOCK makes ZKBLs fully ad-hoc and resolves a privacy and organization problem with deployment. While ban lists can be operated by anyone, ZKBLs—like any ban system, anonymous or not—require Sybil-resistant identities. Existing ZKBLs assume a single trusted issuer for credentials.

SNARKBLOCK removes the need for a single centralized identity provider by allowing service providers to dynamically pick the identity providers that they support. This avoids coordination concerns and allows different providers to adopt different levels of Sybil resistance ranging from CAPTCHAs, to cryptocurrency payment, to real-world identity verification. Crucially, during attestation, the service provider learns only that the user's identity was issued by *some* party in their *accepted identity provider* set.

The core of SNARKBLOCK is a new type of zero-knowledge proof, called HIdden Common

Input Aggregate Proofs, or HICIAP (pronounced "high-chop"). HICIAP is a zero-knowledge proof that aggregates n Groth16 proofs [Gro16] (of the same underlying circuit) into a single $O(\log n)$-sized proof, and shows that the aggregated proofs all verify and all share a common input which is not revealed to the verifier. It is also possible to *link* multiple HICIAP proofs, showing in zero-knowledge that the hidden common inputs across aggregations are equal.

HICIAP addresses a common problem when using zkSNARKs in privacy-preserving protocols like SNARKBLOCK: repeated interactions can require costly proof recomputations to ensure unlinkability. When presenting a single Groth16 proof, the proof can be rerandomized between interactions, achieving anonymity without recomputaton. Unfortunately, when presenting multiple proofs about a common hidden input—e.g., adding proofs about subsequent state changes—we must recompute all proofs since the state-of-the-art approach is to commit to the hidden input and have all proofs be made with respect to that public commitment. Since the commitment is a persistent identifier, it and all proofs relying on it must be recomputed to achieve anonymity. HICIAP resolves this by supporting zero-knowledge aggregation of proofs with a common hidden input.

With this building block, SNARKBLOCK is able to compute *chunk proofs*—Groth16 proofs of non-membership in equally sized non-overlapping portions of the blocklist—and reuse them indefinitely in the cheaper process of HICIAP aggregation.

2 Intuition for a ZKBL construction

Zero-knowledge Succinct Non-interactive Arguments of Knowledge (zkSNARKs), appear to offer a path to ZKBLs with fast verification, but limitations on prover performance—a common problem for nearly all zkSNARKs—make this challenging. This is clear when one examines the costs of using Groth16, a zkSNARK scheme with notably fast verification times.

Existing zkSNARKs can only handle pieces of a blocklist. Producing a zero-knowledge proof of knowledge is, fundamentally, at least linear in the size of the input, i.e., the blocklist. But for Groth16 and other zkSNARKs, the concrete constants are high. Looking ahead, for a blocklist

of 256 entries, a single statement of non-membership is 63k constraints and takes 2.84s to prove. A blocklist of 2^{21} entries would yield a non-membership statement with 2^{29} constraints. But for Groth16, prover memory usage scales poorly in the size of the circuit: a 2^{29}-constraint proof requires 4TB of memory and takes 3 hours to compute, due to the implementation overhead of distributing proving over a 256-core cluster [WZC+18]. To use zkSNARKs for a ZKBL, we cannot have the prover do work linear in the size of the blocklist for each attestation.

Decomposing blocklists by chunk. We observe that blocklists tend to strictly grow over time. While total prover workload is inevitably linear in the size of the blocklist, this work does not need to be recomputed from scratch every time. By breaking the list up into non-overlapping *chunks* we can both reuse work and limit the amount of recomputation required when the list changes.

A zero-knowledge proof for consistency between chunks. A sequence of chunk non-membership proofs for a blocklist $\mathscr{L}$ poses three problems:

1. The server would need to verify $O(|\mathscr{L}|)$ chunks.
2. Reusing a chunk proof across blocklist non-membership attestations would identify the client.
3. A malicious client could use a different identity when proving non-membership in a specific chunk, avoiding a block in that segment of the blocklist.

To address the above problems, we need a *compact* proof that a sequence of chunk non-membership proofs verifies with respect to a single *hidden* identity. Further, that proof must be zero-knowledge to ensure that the chunk proofs can be safely reused across blocklist non-membership proofs.

Recursive Groth16 proofs are impractical. For some proof π, we say a *recursive proof* over π is a proof that π verifies with respect to some public input. It is feasible to use this technique for blocklist non-membership. Each chunk proof could prove non-membership and also recursively check consistency of the previous chunk proof. The challenge for recursion, though, is latency: assuming the blocklist changes frequently, each attestation would need to compute an updated chunk proof and a recursive step. On our benchmark system, a single recursive step for Groth16 takes 16.5s

to prove.[1] With recursion, the cost of computing a chunk proof would also increase by 5–7× because of overhead imposed by the elliptic curves that are required to support recursion. [CCDW20].[2] Looking ahead, one approach we take adds 1s of latency to attestation using a *buffer*, i.e., a list head consisting of smaller chunks.

Beyond recursive and aggregate proofs. We observe that recursion is not necessary to verify a sequence of chunk non-membership proofs. There is no intermediate state in our computation, rather we only require that all proofs must share the same private input. Recent advances in inner product proofs [BMM+21] give a succinct proof that n Groth16 zkSNARK proofs all verify. However, this aggregate approach has two critical shortcomings: it is not zero-knowledge and it does not ensure consistency.

A natural approach for consistency would be to commit to the hidden value and use it as a public input to each Groth16 proof. But if the same commitment is used across multiple anonymous attestations, it forms a persistent identifier. On the other hand, when a fresh commitment is used for each attestation, we must regenerate every chunk proof.

We use [BMM+21] as a starting point and have a single *public* input to each chunk proof, then blind it in the aggregate proof so it is not revealed to the verifier. The resulting scheme reuses the same blinders in multiple parts of the zero-knowledge protocol. This unusual property made proving the honest-verifier zero-knowledge property challenging.

3 Preliminaries

We now present preliminary notation and mathematical concepts that are specific to this work.

[1] This is the cost to in-circuit verify a proof with no inputs using MNT6-753 over MNT4-753.

[2] *Depth-1 recursion* using, e.g., BW6-761 over BLS12-377, would avoid some of these costs compared to the MNT4/6 cycle. However, any addition to the blocklist would necessitate recomputing the top-level proof that checks all n chunk proofs. At 3.9s per check on our benchmark system, this is not feasible.

$$R_{\mathsf{TIPP}} := \left\{ \left(\begin{array}{c} \mathsf{ck}_1, \in \mathbb{G}_2^n, \mathsf{ck}_2 \in \mathbb{G}_1^n, \\ \mathsf{com}_A, \mathsf{com}_B, \mathsf{agg}_{AB} \in \mathbb{G}_T, \mathbf{r} \in \mathbb{F}^n; \\ \mathbf{A} \in \mathbb{G}_1^n, \mathbf{B} \in \mathbb{G}_2^n \end{array} \right) : \begin{array}{c} \mathsf{com}_A = \mathbf{A} * \mathsf{ck}_1 \\ \bigwedge \mathsf{com}_B = \mathsf{ck}_2 * \mathbf{B} \\ \bigwedge \mathsf{agg}_{AB} = \mathbf{A}^{\mathbf{r}} * \mathbf{B} \end{array} \right\}$$

$$R_{\mathsf{MIPP}-k} := \left\{ \left(\begin{array}{c} \mathsf{ck}_1 \in \mathbb{G}_2^n, \mathsf{com}_C, \in \mathbb{G}_T, \\ \mathsf{agg}_C \in \mathbb{G}_1, \mathbf{r} \in \mathbb{F}^n; \\ \mathbf{C} \in \mathbb{G}_1^n \end{array} \right) : \begin{array}{c} \mathsf{agg}_C = \mathbf{C}^{\mathbf{r}} \\ \bigwedge \mathsf{com}_C = \mathbf{C} * \mathsf{ck}_1 \end{array} \right\}$$

$$R_{\mathsf{HMIPP}} := \left\{ \left(\begin{array}{c} \mathsf{ck}_1 \in \mathbb{G}_2^n, \mathsf{ck}_3 \in \mathbb{G}_2, \\ \mathsf{com}_C, \in \mathbb{G}_T, \mathsf{agg}_C \in \mathbb{G}_1, \\ \mathbf{r} \in \mathbb{F}^n; \\ \mathbf{C} \in \mathbb{G}_1^n, z \in \mathbb{F} \end{array} \right) : \begin{array}{c} \mathsf{agg}_C = \mathbf{C}^{\mathbf{r}} \\ \bigwedge \mathsf{com}_C = e\left([z]_1, \mathsf{ck}_3\right) \cdot (\mathbf{C} * \mathsf{ck}_1) \end{array} \right\}$$

Figure 3.1: Inner pairing relations. We directly use Bünz et al.'s definition of R_{TIPP} and $R_{\mathsf{MIPP}\text{-}k}$, and we use R_{HMIPP} to refer to the "hiding commitment" version of $R_{\mathsf{MIPP}\text{-}k}$. While R_{HMIPP} hides its group elements $\mathbf{C}$, R_{TIPP} takes $\mathbf{A}$ and $\mathbf{B}$ as public input. Turning these inputs into witnesses is one of the primary focuses of HICIAP.

3.1 Inner product proofs

Bünz et al. [BMM+21] introduce a proof system for various inner product relations. We will make use of the TIPP, MIPP_k, and HMIPP proof systems, whose relations are defined in Figure 3.1.

In short, R_{TIPP} is satisfied when $\mathbf{A}^{\mathbf{r}} * \mathbf{B} = \mathsf{agg}_{AB}$, $R_{\mathsf{MIPP}-k}$ is satisfied when $\mathbf{C}^{\mathbf{r}} = \mathsf{agg}_C$, and R_{HMIPP} is the same as $R_{\mathsf{MIPP}-k}$ except its commitment to $\mathbf{C}$ is hiding.

3.2 HICIAP

Since HICIAP is used extensively in the construction of SnarkBlock, we provide a brief overview of its functionality here. We defer discussion of these algorithms including their construction and security claims until Section 6.

HICIAP is a zkSNARK which aggregates multiple Groth16 proofs of the same relation. Of its aggregated proofs, it proves that (1) they verify with respect to verifier-supplied public inputs, and (2) they share a common public input element (which is hidden by the aggregate proof). In addition, HICIAP can *link* aggregate proofs: it can prove in zero-knowledge that the proofs in a set of HICIAP proofs all share the same common input element.

Formally, HICIAP consists of six procedures:

$\mathsf{GenCk}(n) \to (\mathsf{ck}, \mathsf{srs})$ Generates a commitment key $(\mathsf{ck}_1, \mathsf{ck}_2, \mathsf{ck}_3) \in \mathbb{G}_2^n \times \mathbb{G}_1^n \times \mathbb{G}_2$ and a (short) structured verification key srs which can be used, respectively, to prove and verify HICIAP aggregates of up to $n-2$ Groth16 proofs, where n is a power of 2.

$\mathsf{Com}(\mathsf{ck}, \hat{\mathbf{S}}) \to \mathsf{com}_{\mathsf{in}}$ Constructs a commitment to the prepared Groth16 public inputs $\hat{\mathbf{S}} \in \mathbb{G}_1^{n-2}$ as $\mathsf{com}_{\mathsf{in}} := \hat{\mathbf{S}} * \mathsf{ck}_{1,[:n-2]}$.

$\mathsf{Prove}((\mathsf{ck}, \mathsf{crs}), \hat{\mathbf{S}}, (a_0, \{\pi_i\}_{i=1}^{n-2})) \to (\hat{\pi}, o)$ Produces a succinct proof that each Groth16 proof π_i verifies w.r.t. the common witness element $a_0 \in \mathbb{F}$, the prepared input $\hat{S}_i \in \mathbb{G}_1$, and the given Groth16 CRS. Also produces an opening o to a commitment to a_0 contained inside $\hat{\pi}$. The opening is used in LinkProve.

$\mathsf{Vfy}(\mathsf{srs}, \hat{\pi}, \mathsf{com}_{\mathsf{in}}) \to \{0,1\}$ Verifies the given aggregate proof w.r.t. the committed public input. Alternatively, a set of prepared Groth16 inputs can be passed instead of $\mathsf{com}_{\mathsf{in}}$.

$\mathsf{LinkProve}(\{\hat{\pi}_i\}_{i=1}^t, (a_0, \{o_i\}_{i=1}^t)) \to \pi_{\mathsf{link}}$ Using the openings o_i, produces a proof that the given aggregate proofs share the witness element $a_0 \in \mathbb{F}$.

$\mathsf{LinkVfy}(\pi_{\mathsf{link}}, \{\hat{\pi}_i\}_{i=1}^t) \to \{0,1\}$ Verifies the link proof w.r.t. the given aggregate proofs.

We describe the design and security HICIAP in greater detail in Section 6.

4 Zero-knowledge blocklists

We now give our framework for zero-knowledge blocklists, taken directly from BLAC [TAKS10], but with modifications to support multiple identity providers and allow for additional precomputation.

4.1 Setting

A zero-knowledge blocklist allows users to attest that an *identity* issued by one of a set of *identity providers* is not in a *blocklist*. We now detail these concepts:

Identity. We use k to denote a user's private *identity*. A single user in the real world can hold arbitrarily many identities. In all cases, k will be a field element selected uniformly at random by the user. Other similar schemes refer to k as a user's "nym," "pseudonym," or "credential."

Identity providers. Blocking users fundamentally depends on identities being Sybil-resistant. Most approaches to blocklisting, including BLAC's approach to ZKBLs, assume a single issuer.

Here we formalize a more general version of ZKBLs that supports federated identity with privacy: each service provider is allowed to maintain its own list $\mathscr{I}$ of accepted identity providers, which we call the *AIP set*. Identity providers are responsible for ensuring Sybil resistance. The service provider is allowed to update this set over time, and should distribute it via the same channels it uses to distribute its blocklist. Federated identity also comes with a privacy guarantee: when a user proves that their identity was issued by an issuer in $\mathscr{I}$, they do not reveal which issuer it was.

Blocklists and session tuples. A ZKBL *blocklist* consists of pairs containing a *session nonce* nonce and *session tag* tag, where tag is bound to the user's identity by $\mathsf{tag} := \mathsf{Prf}_k(\mathsf{nonce})$ for some fixed pseudorandom function Prf. Blocklist entries can support *context binding* via structured auxiliary data. By computing nonce as $\mathsf{nonce} := \mathsf{H}(\mathsf{aux}, r)$ for some hash H, aux is bound to the attestation. This data can be used to, for example, bind attestation to an action (e.g., to prove that the blocked user is the action's author) or to a particular blocklist or policy (e.g., to enforce which lists a banned tuple can be transferred to).

Finally, in a departure from BLAC, we allow blocklists to be split into *chunks*—equally sized non-overlapping segments—whose sizes are decided by the service provider. Blocklists are chunked so that users can precompute non-membership proofs over individual chunks rather than the entire blocklist at once.

Formalizing non-membership proofs. A non-membership proof π_{zkbl} is a zero-knowledge proof

of three distinct properties:

1. *Issuance.* That the user's identity k is signed by an identity provider.
2. *Tag well-formedness.* That tag and nonce are honestly computed, i.e., $\mathsf{tag} = \mathsf{Prf}_k(\mathsf{nonce})$.
3. *Blocklist non-membership.* That the user's identity k did not generate any tuples already on a blocklist, i.e., $\forall(\mathsf{tag}', \mathsf{nonce}') \in \mathscr{L} : \mathsf{tag}' \neq \mathsf{Prf}_k(\mathsf{nonce}')$.

4.2 ZKBL functionality

A zero-knowledge blocklist consists of five algorithms.

CRS-Setup Generates system-wide parameters.

IdP-Keygen Generates a signing keypair $(\mathsf{sk}, \mathsf{pk})$ to be used by an identity provider for issuance.

Register Executes a protocol between a user and an identity provider. On success, the user obtains a signature of (a function of) their identity.

Sync Fetches the latest additions to a service provider's blocklist and then precomputes cryptographic material for them. Users periodically run this routine *offline*, i.e., when not attesting.

Attest Executes a non-interactive protocol in which a user authenticates to a service provider. First, the user constructs a session-specific tuple $(\mathsf{tag}, \mathsf{nonce})$ as $\mathsf{tag} := \mathsf{Prf}_k(\mathsf{nonce})$, where nonce is pseudorandom and optionally binds a context aux. This tuple can be used by the service provider to block the user at any point in the future by simply including it in the blocklist. The user then produces a zero-knowledge proof π_{zkbl} that proves well-formedness of the tuple and that their (signed) identity did not generate any tuples already on a blocklist. The session tuple and zero-knowledge proof are then sent to the service provider as $(\pi_{\mathsf{zkbl}}, \mathsf{tag}, \mathsf{nonce})$.

Verify Checks the validity of a user's attestation. A service provider accepts if and only if π_{zkbl} verifies with respect to the supplied session tuple $(\mathsf{tag}, \mathsf{nonce})$ and the service provider's blocklist $\mathscr{L}$, chunk size schedule, AIP set $\mathscr{I}$, and optional context-binding string aux.

Separately, we assume two non-cryptographic operations for blocklist management:

Blocklist-Add Adds a token to a blocklist.

Blocklist-Remove Removes a token from a blocklist.

We stress that the Add and Remove routines are distinct from the cryptographic scheme, and can be run by anyone. How parties decide to manage their blocklists is wholly orthogonal to the ZKBL construction.

BLAC as a ZKBL. The authors of BLAC construct their scheme using BBS+ signatures [BBS04] and a Camenisch-Shoup Σ-protocol [CS03]. Although not described as such, this is the same PRF approach we formalize here. BLAC's tag function is $\mathsf{nonce} \mapsto \mathsf{H}(\mathsf{nonce})^k$, and it is done in two steps, with the hash evaluation outside the zero-knowledge proof, and the exponentiation witnessed inside. Conceptually, the entire question for designing a practical ZKBL is how to co-design a PRF and zero-knowledge proof protocol to make an efficient non-membership proof.

4.3 Security requirements

Our desired security properties are taken from BLAC. For the complete definitions see [TAKS10]. Note the following aesthetic changes in our description: blocklistability encompasses misauthentication resistance; and anonymity is described as a distinguishability notion as opposed to a simulatability notion, which we believe better captures the actual security properties achieved by BLAC's game-based definition.

Blocklistability A coalition of dishonest service providers and users can only successfully authenticate to an honest service provider if that user holds a valid credential issued by an identity provider that is not included in the blocklist.

Non-Frameability A coalition of dishonest identity providers, service providers, and users cannot prevent an honest, non-blocklisted user from successfully authenticating with an honest service provider.

Anonymity A coalition of dishonest identity providers, service providers, and users cannot distinguish attestation transcripts associated with any two honest users. Further, no such coalition can link any given authentication transcript with the registration in which an identity provider issued the associated credential.

See Section 11.2 for a more detailed accounting of these properties.

5 SNARKBLOCK design and overview

The full design of SNARKBLOCK is detailed in Figure 3.3. The core relations are defined in Figure 3.2. In words, R_{isu} is satisfied when a (commitment to a) user's identity is signed by an issuer in the AIP set,[3] R_{tag} is satisfied when tag is computed correctly, and R_{chunk} is satisfied when a user did not produce any of the tags in a chunk.

We omit textual descriptions of the full set of algorithms and detail the two key ones: Sync and Attest.

Sync. Sync is the offline phase of attestation. During Sync a client fetches the most recent versions of the service provider's blocklist, chunk schedule, and AIP set. The client then precomputes Groth16 chunk proofs π_{chunk_i} of the relation $R_{\mathsf{chunk}}(k, \mathsf{chunk}_i)$ for every new chunk_i received from the service provider. The client also precomputes π_{isu}, by computing a Groth16 proof π_{isu} of $R_{\mathsf{isu}}((k, \mathscr{I}), (i^*, \sigma, r))$ where i^* is the chosen identity provider in the AIP set $\mathscr{I} = \{\mathsf{pk}_1, \ldots, \mathsf{pk}_\ell\}$, σ is the identity provider's signature of the identity commitment, and r is the randomness used to commit to k.

Attest. To attest to blocklist non-membership, the client must combine a series of proofs about the user's identity k. First the client computes fresh session tuple $(\mathsf{tag}, \mathsf{nonce})$ and proves it is well-formed with respect to k using a Groth16 proof π_{tag} for the relation $R_{\mathsf{tag}}(k, \mathsf{tag}, \mathsf{nonce})$.

Ideally, the client would combine π_{tag} with the precomputed π_{isu} and π_{chunk_i} proofs from Sync. But a single HICIAP instance only works for proofs over the same relation. Thus, π_{tag} and π_{isu}

[3] This need only be an EUF-CMA-secure signature scheme. For ease of implementability in a zero-knowledge circuit, we use Schnorr signatures.

$$R_{\mathsf{tag}} := \{(k, \mathsf{tag}, \mathsf{nonce};) : \mathsf{Prf}_k(\mathsf{nonce}) = \mathsf{tag}\}$$

$$R_{\mathsf{chunk}} := \left\{ (k, \mathsf{chunk};) : \bigwedge_{(\mathsf{tag},\mathsf{nonce}) \in \mathsf{chunk}} \mathsf{Prf}_k(\mathsf{nonce}) \neq \mathsf{tag} \right\}$$

$$R_{\mathsf{isu}} := \left\{ \left(\begin{array}{c} k, (\mathsf{pk}_i)_{i=1}^{\ell}; \\ i^*, \sigma, r \end{array} \right) : \begin{array}{c} 1 \leq i^* \leq \ell \\ \bigwedge \mathsf{Schnorr.Ver}_{\mathsf{pk}_{i^*}}(\mathsf{Com}(k,r), \sigma) \end{array} \right\}$$

$$R_{\mathsf{zkbl}} := \left\{ \left(\begin{array}{c} \mathscr{L}, \mathscr{I}, \mathsf{tag}, \mathsf{nonce}; \\ k, i^*, \sigma, r \end{array} \right) : \begin{array}{l} \bigwedge_{i=1}^{c} R_{\mathsf{chunk}}(k, \mathsf{chunk}_i) \\ \bigwedge R_{\mathsf{isu}}(k, \mathscr{I}; i^*, \sigma, r) \\ \bigwedge R_{\mathsf{tag}}(k, \mathsf{tag}, \mathsf{nonce}) \end{array} \right\}$$

Figure 3.2: R_{zkbl} is the relation which the attestation procedure in SNARKBLOCK attests to. $\mathscr{I}$ is the AIP set $\{\mathsf{pk}_1, \ldots, \mathsf{pk}_\ell\}$, and $\mathscr{L}$ is the set of chunks $\{\mathsf{chunk}_1, \ldots, \mathsf{chunk}_c\}$. Note that k is a public (rather than private) input to the three sub-relations R_{isu}, R_{tag}, and R_{chunk}. This is because the implementation of HICIAP requires that the hidden common input be a public input in the underlying Groth16 proof.

$\underline{\mathsf{IdPKeyGen}()}$
$(\mathsf{sk}, \mathsf{pk}) := \mathsf{Schnorr.KeyGen}()$
return $(\mathsf{sk}, \mathsf{pk})$

$\underline{\mathsf{RegU}(k)}$
$r \leftarrow\!\!\$\ \mathbb{F}$
$\mathsf{com} := \mathsf{Com}(k, r)$
return (com, r)

$\underline{\mathsf{RegS}(\mathsf{sk}, \mathsf{com})}$
$\sigma := \mathsf{Schnorr.Sign}_{\mathsf{sk}}(\mathsf{com})$
return σ

$\underline{\mathsf{CrsSetup}(n)}$
$\mathsf{crs}_{\mathsf{isu}} := \mathsf{Groth16.Setup}(R_{\mathsf{isu}})$
$\mathsf{crs}_{\mathsf{tag}} := \mathsf{Groth16.Setup}(R_{\mathsf{tag}})$
$\mathsf{crs}_{\mathsf{chunk}} := \mathsf{Groth16.Setup}(R_{\mathsf{chunk}})$
$(\mathsf{ck}, \mathsf{srs}) := \mathsf{HICIAP.GenCk}(n)$
return $(\mathsf{ck}, \mathsf{srs})$

$\underline{\mathsf{Attest}(k, \pi_{\mathsf{isu}}, \{\pi_{\mathsf{chunk}_i}\}_{i=1}^{c})}$
$\mathsf{nonce} \leftarrow\!\!\$\ \{0,1\}^{\lambda}$
$\mathsf{tag} := \mathsf{Prf}_k(\mathsf{nonce})$
$\pi_{\mathsf{tag}} := \mathsf{Groth16.Prove}(\mathsf{crs}_{\mathsf{tag}}, (k, \mathsf{tag}, \mathsf{nonce}), \cdot)$
$\hat{\pi}_{\mathsf{isu}} := \mathsf{HICIAP.Prove}((\mathsf{ck}, \mathsf{crs}_{\mathsf{isu}}), \mathscr{I}, (k, \{\pi_{\mathsf{isu}}\}))$
$\hat{\pi}_{\mathsf{tag}} := \mathsf{HICIAP.Prove}((\mathsf{ck}, \mathsf{crs}_{\mathsf{tag}}), (\mathsf{tag}, \mathsf{nonce}), (k, \{\pi_{\mathsf{tag}}\}))$
$\hat{\pi}_{\mathsf{chunk}} := \mathsf{HICIAP.Prove}\left(\begin{array}{c} (\mathsf{ck}, \mathsf{crs}_{\mathsf{chunk}}), \mathscr{L}, \\ (k, \{\pi_{\mathsf{chunk}_i}\}_{i=1}^{c}) \end{array} \right)$
$\pi_{\mathsf{link}} := \mathsf{HICIAP.LinkProve}(k, (\pi_{\mathsf{isu}}, \pi_{\mathsf{chunk}}, \pi_{\mathsf{tag}}), k)$
$\pi_{\mathsf{zkbl}} := (\pi_{\mathsf{link}}, \hat{\pi}_{\mathsf{isu}}, \hat{\pi}_{\mathsf{tag}}, \hat{\pi}_{\mathsf{chunk}})$
return $(\pi_{\mathsf{zkbl}}, \mathsf{tag}, \mathsf{nonce})$

$\underline{\mathsf{Sync}(\{\mathsf{chunk}_i\}_{i=c'}^{c}, \mathscr{I}, i^*, k, \sigma, r)}$
for $c' \leq j \leq c$:
  $\pi_{\mathsf{chunk}_j} := \mathsf{Groth16.Prove}(\mathsf{crs}_{\mathsf{chunk}}, (k, \mathsf{chunk}_j), \cdot)$
$\pi_{\mathsf{isu}} := \mathsf{Groth16.Prove}(\mathsf{crs}_{\mathsf{isu}}, (k, \mathscr{I}), (i^*, \sigma, r))$
return $\{\pi_{\mathsf{chunk}_1}, \ldots, \pi_{\mathsf{chunk}_c}\}$

$\underline{\mathsf{PrepBlocklist}(\{\mathsf{chunk}_i\}_{i=1}^{c})}$
for $1 \leq i \leq c$
  $\hat{S}_{\mathsf{chunk}_i} := \mathsf{Groth16.Prepare}(\mathsf{crs}_{\mathsf{chunk}}, \mathsf{chunk}_i)$
$\mathsf{com}_{\mathscr{L}} := \mathsf{HICIAP.Com}(\mathsf{ck}, \{\hat{S}_{\mathsf{chunk}_i}\}_{i=1}^{c})$
return $\mathsf{com}_{\mathscr{L}}$

$\underline{\mathsf{Vfy}(\pi_{\mathsf{zkbl}}, (\mathsf{tag}, \mathsf{nonce}), \mathscr{I}, \mathsf{com}_{\mathscr{L}})}$
$\hat{S}_{\mathsf{tag}} = \mathsf{Groth16.Prepare}(\mathsf{crs}_{\mathsf{tag}}, (\mathsf{tag}, \mathsf{nonce}))$
$\hat{S}_{\mathsf{isu}} = \mathsf{Groth16.Prepare}(\mathsf{crs}_{\mathsf{isu}}, \mathscr{I})$
return $\mathsf{HICIAP.LinkVfy}(\pi_{\mathsf{link}}, (\hat{\pi}_{\mathsf{isu}}, \hat{\pi}_{\mathsf{tag}}, \hat{\pi}_{\mathsf{chunk}}))$
$\bigwedge$ $\mathsf{HICIAP.Vfy}(\mathsf{srs}, \hat{\pi}_{\mathsf{isu}}, \{\hat{S}_{\mathsf{isu}}\})$
$\bigwedge$ $\mathsf{HICIAP.Vfy}(\mathsf{srs}, \hat{\pi}_{\mathsf{tag}}, \{\hat{S}_{\mathsf{tag}}\})$
$\bigwedge$ $\mathsf{HICIAP.Vfy}(\mathsf{srs}, \hat{\pi}_{\mathsf{chunk}}, \mathsf{com}_{\mathscr{L}})$

Figure 3.3: A pseudocode definition of the SNARKBLOCK system, using the HICIAP aggregation scheme (Figure 3.4).

are wrapped in HICIAP proofs $\hat{\pi}_{\mathsf{tag}}$ and $\hat{\pi}_{\mathsf{isu}}$ respectively, the π_{chunk_i} proofs are aggregated into a HICIAP proof $\hat{\pi}_{\mathsf{chunk}}$, and a linking proof π_{link} is used to show each aggregate is made with respect to the same identity k.

The client's output is thus $(\pi_{\mathsf{zkbl}}, \mathsf{tag}, \mathsf{nonce})$, where $\pi_{\mathsf{zkbl}} := (\hat{\pi}_{\mathsf{isu}}, \hat{\pi}_{\mathsf{tag}}, \hat{\pi}_{\mathsf{chunk}}, \pi_{\mathsf{link}})$.

Buffering recent blocklist additions and deletions. When a ban is added or removed from the blocklist, the user must redo the corresponding chunk proof. It is inevitable between Sync operations that some number of additions and deletions will occur, thus requiring recomputation during Attest and adding the corresponding amount of latency. The larger the chunk size, the higher the latency. While we can avoid this for deletions by having bans expire in batches, this is undesirable for additions—we want bans to take effect as soon as possible.

To avoid a tradeoff between chunk size and attestation latency, we have the tail of the list be a buffer of smaller chunks and have a separate instance of HICIAP aggregate them. Because the circuit is different from the circuit used for larger chunks, this optimization increases the number of distinct HICIAP proofs passed to the verifier, while decreasing the overall attestation time.

5.1 Security argument

Security of SNARKBLOCK depends on it correctly instantiating the PRF+Sig+ZKP paradigm using HICIAP. We state the theorem of security for SNARKBLOCK here and give a proof sketch in Section 11.2. This proof depends on the security of HICIAP as a building block, and so HICIAP is the main focus of our security analysis over subsequent sections and appendices.

Theorem 1 (SNARKBLOCK Security)**.** SNARKBLOCK *described in Figure 3.3 is blocklistable, anonymous and non-frameable provided that Groth16 and* HICIAP *proofs are knowledge-sound and subversion zero-knowledge; Schnorr signatures are unforgeable;* Prf *is pseudorandom; and* Com *is binding and hiding.*

5.2 Trusted setup

Our protocol and security proof assumes that a trusted party generates a CRS for each Groth16 circuit as well as for each HICIAP instance. The CRSs are similar, being of the form $sG, s^2G, \ldots$ for several bases. In most cases, service providers should be able to run the setup: assuming subversion resistance [Fuc17], a malicious CRS only undermines soundness, *not* privacy. If necessary, protocols [BGM17, BCG+15] for multiparty setup have been used for commercial cryptocurrencies such as Zcash [Rad21] and Filecoin, where failure would allow the forgery of billions of dollars. These also ensure subversion resistance.

6 HICIAP

We now introduce the core of SNARKBLOCK: HIdden Common-Input Aggregate Proofs (HICIAP), a novel zkSNARK scheme which we use to generate the zero-knowledge proof of blocklist non-membership π_{zkbl}.

Recall the purpose of HICIAP is to aggregate multiple Groth16 proofs of the same relation. Of its aggregated proofs, it proves that (1) they verify with respect to verifier-supplied public inputs, and (2) they share a common public input element (which is hidden by the aggregate proof). In the case of SNARKBLOCK, the relation is chunk non-membership, the verifier-supplied public inputs are the (prepared) blocklist chunks, and the common input element is the user's identity.

In addition, HICIAP can *link* aggregate proofs: it can prove in zero-knowledge that the proofs in a set of HICIAP proofs all share the same common input element. In the case of (unbuffered) SNARKBLOCK, there are three aggregate proofs that are linked: chunk non-membership, issuance, and tag well-formedness.

In this section, we provide intuition for the design of HICIAP and then describe each of its components in detail.

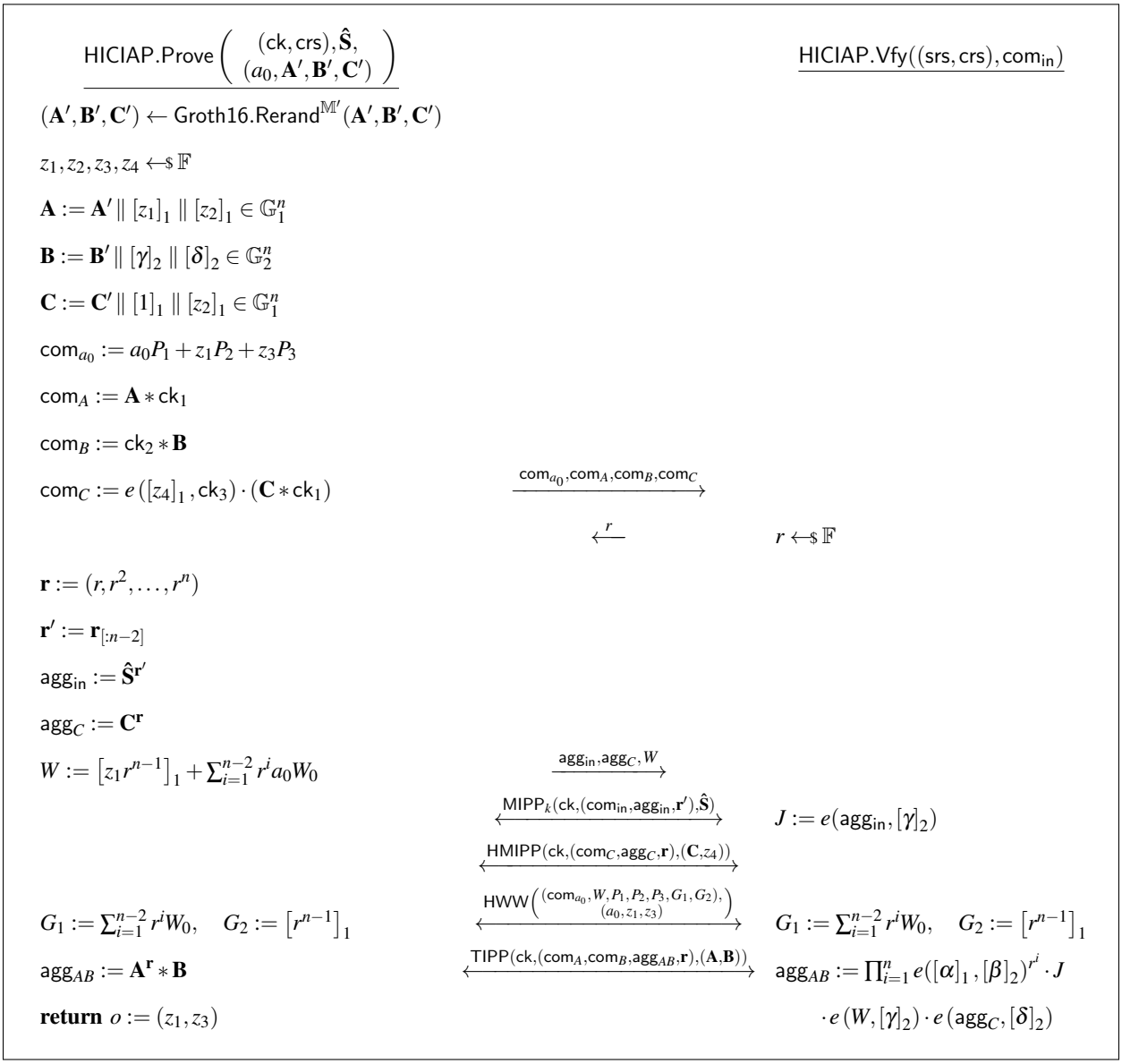


Figure 3.4: The (interactive) HICIAP protocol. Vfy accepts iff all subprotocols MIPP, HMIPP, HWW, and TIPP accept. The group elements $[\alpha]_1, [\beta]_2, [\delta]_2, [\gamma]_2, W_i = \left[\frac{\beta u_i(x) + \alpha v_i(x) + w_i(x)}{\gamma}\right]_1$ are supplied by crs. The values P_1, P_2, P_3 used to compute com_{a_0} form a Pedersen basis. The $\mathsf{Rerand}^{\mathbb{M}'}$ procedure is only executed on the indices in the (log-sized) masking set $\mathbb{M}' = \mathbb{M} \cup \{n-2\}$, where $\mathbb{M}$ is defined in the proof of Lemma 4. It is the identity function everywhere else.

6.1 Intution

To explain HICIAP, we start with a naïve verifier who takes a full (non-succinct and non-hiding) set of Groth16 proofs $\pi_i = (A_i, B_i, C_i)$ and checks that they verify with respect to a common public input. We transform this into a succinct zero-knowledge proof that vector commitments to $\mathbf{A}$, $\mathbf{B}$ and $\mathbf{C}$ contain proofs that verify with respect to a hidden input. For simplicity, we omit the blinding factors from discussion, and leave their detailed accounting to the proof of honest verifier zero-knowledge in Section 11.2.

The HICIAP verifier must be convinced that there is some hidden wire value $a_0 \in \mathbb{F}$ for which a set Groth16 proofs $\{(A_i, B_i, C_i)\}_{i=1}^{n-2}$ verify given a set of prepared public inputs $\hat{\mathbf{S}}$, i.e., for all $i = 1 \ldots, n-2$,

$$e(A_i, B_i) \stackrel{?}{=} e([\alpha]_1, [\beta]_2) \cdot e(C_i, [\delta]_2) \cdot e\left(a_0 W_0 + \hat{S}_i, [\gamma]_2\right).$$

Our first step is to combine the above $n-2$ equations into a single polynomial equation. Verifying this would still require the verifier to do linearly many equality checks, so the verifier picks a random $r \leftarrow\$ \mathbb{F}$ and evaluates the polynomial "in the exponent" at the random point. Combining these two steps, the new verifier equation is

$$\prod e(A_i, B_i)^{r^i} \stackrel{?}{=} \prod e([\alpha]_1, [\beta]_2)^{r^i} \cdot \prod e(C_i, [\delta]_2)^{r^i} \cdot \prod e\left(a_0 W_0 + \hat{S}_i, [\gamma]_2\right)^{r^i}.$$

By the Schwartz-Zippel lemma, equality here implies the equality of the initial $n-2$ equations with overwhelming probability. We now have one equality check.

The next step is to make the verifier oblivious to the hidden wire value a_0. To do that, we split the $e(a_0 W_0 + \hat{S}_i, [\gamma]_2)^{r^i}$ term in two. The prover sends W, a blinded version of $\sum r^i a_0 W_0$, to the verifier. It proves that W is computed correctly using an instance of the Σ-protocol HWW (described in Section 6.2). The verifier equation is now

$$\prod e(A_i, B_i)^{r^i} \stackrel{?}{=} \prod e([\alpha]_1, [\beta]_2)^{r^i} \cdot \prod e(C_i, [\delta]_2)^{r^i} \cdot e(W, [\gamma]_2) \cdot \prod e\left(\hat{S}_i, [\gamma]_2\right)^{r^i}.$$

For both succinctness and privacy, the prover cannot give the verifier every Groth16 proof. Instead, the prover gives only succinct commitments, $\mathsf{com}_A, \mathsf{com}_B, \mathsf{com}_C$ to the proof vectors $\mathbf{A}$, $\mathbf{B}$, $\mathbf{C}$, respectively. This requires the prover to calculate $\mathsf{agg}_{AB} := \prod e(A_i, B_i)^{r^i}$ and $\mathsf{agg}_C := \prod e(C_i, [\delta]_2)^{r^i}$ itself and send them to the verifier. Since these calculations can be expressed as inner product operations, the prover shows they are correct using instances of TIPP and HMIPP, respectively. The verifier equation is now

$$\mathsf{agg}_{AB} \stackrel{?}{=} \prod e([\alpha]_1, [\beta]_2)^{r^i} \cdot \mathsf{agg}_C \cdot e(W, [\gamma]_2) \cdot \prod e\left(\hat{S}_i, [\gamma]_2\right)^{r^i}.$$

This equation is now verifiable by the HICIAP verifier, but it is not fully succinct—the verifier must still do linear work in order to compute the products containing the r^i exponents. The verifier can avoid this for the term $\prod e([\alpha]_1, [\beta]_2)^{r^i}$ by simply using the geometric sum formula: $\sum_{i=0}^{n} r^i = (r^{n+1} - 1)/(r-1)$. The second optimization, due to Bünz et al. [BMM+21], moves the aggregation of the prepared inputs $\mathsf{agg}_{\mathsf{in}} := \sum r^i \hat{S}_i$ to the prover. The prover sends $\mathsf{agg}_{\mathsf{in}}$ and proves it was constructed correctly using an instance of MIPP. The verifier checks this with respect to $\mathsf{com}_{\mathsf{in}}$, which it can compute from public inputs independently of the proof-specific r values. With these two optimizations, the final verifier equation is

$$\mathsf{agg}_{AB} \stackrel{?}{=} e([\alpha]_1, [\beta]_2)^{\frac{r^{n+1}-1}{r-1}} \cdot \mathsf{agg}_C \cdot e(W, [\gamma]_2) \cdot e(\mathsf{agg}_{\mathsf{in}}, [\gamma]_2).$$

It is important to reiterate that, while this resembles HICIAP's verification equation,[4] the protocol outlined above is not zero-knowledge. W leaks a_0; agg_{AB}, com_A, and com_B leak parts of $\mathbf{A}$ and $\mathbf{B}$; and agg_C and com_C leak parts of $\mathbf{C}$. In order to achieve zero-knowledge, we blind all of these values using the explicit blinders $z_1, \ldots, z_4 \in \mathbb{F}$ and the implicit blinders in the Groth16.Rerand sub-procedure.

[4] For clarity's sake, however, the Vfy algorithm in Figure 3.4 does not include the geometric sum formula, or any other arithmetic optimizations.

6.2 HICIAP details

We now give the formal definitions of the HICIAP relations and procedures. The HICIAP relation for a fixed n, where n is a power of two, is defined to be

$$R_{\mathsf{HICIAP}} = \left\{ \left(\begin{array}{c} \mathsf{ck}, \mathsf{crs}, \mathsf{com}_{\mathsf{in}}, \{\hat{S}\}_{i=1}^{n-2}; \\ a_0, \{\pi_i\}_{i=1}^{n-2} \end{array} \right) : \begin{array}{r} \bigwedge_{i=1}^{n-2} \mathsf{Groth16.Vfy}(\mathsf{crs}, \pi_i, (a_0, \hat{S}_i)) \\ \bigwedge \mathsf{com}_{\mathsf{in}} = \hat{\mathbf{S}} * \mathsf{ck}_1 \end{array} \right\}.$$

The associated protocol is given in Figure 3.4, and is made non-interactive by applying the Fiat-Shamir transform [FS87].

Note that Prove outputs an opening $o = (z_1, z_3)$ of com_{a_0}. This opening is used for linkage proofs, which show that multiple HICIAP proofs share the same a_0. Formally, this relation is

$$R_{\mathsf{link}} = \left\{ \left(\{\hat{\pi}_i\}_{i=1}^{t}; a_0, \{o_i\}_{i=1}^{t} \right) : \bigwedge_{i=1}^{t} \mathsf{com}_{a_0}^{(i)} = a_0 P_1 + z_1^{(i)} P_2 + z_3^{(i)} P_3 \right\},$$

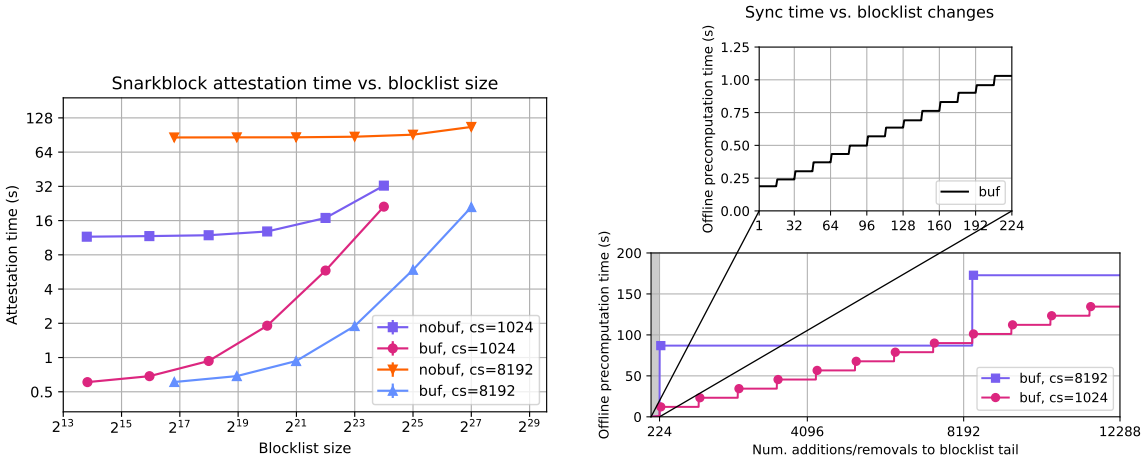
where $\mathsf{com}_{a_0}^{(i)}$ comes from $\hat{\pi}_i$, $(z_1^{(i)}, z_3^{(i)})$ come from o_i, and $P_1, P_2, P_3 \in \mathbb{G}_1$ is a Pedersen basis shared by all HICIAP instances. The LinkProve and LinkVfy algorithms are constructed using the generic Σ-protocol framework defined by Camenisch and Stadler [CS97]. The protocol is described and proven secure in Section 11.1.

For the last relation, recall from the intuition that the value W in HICIAP proofs must be proven to represent the value a_0 on the wire W_0 and no more (i.e., it must not allow the prover to cancel other wire values). We call this the *hidden wire well-formedness* (HWW) relation:

$$R_{\mathsf{HWW}} := \left\{ \left(\begin{array}{c} (U, V, \{G_i\}_{i=1}^{5} \in \mathbb{G}_1; \\ w, x, y \in \mathbb{F} \end{array} \right) : \begin{array}{r} U = wG_1 + xG_2 + yG_3 \\ \bigwedge V = wG_4 + xG_5 \end{array} \right\}$$

Like Link, the HWW proof system is a Σ-protocol constructed using the Camenisch-Stadler framework. The protocol is also described and proven secure in Section 11.1.

We claim that HICIAP is a zkSNARK for the R_{HICIAP} relation. The proofs of the below theorems



(a) Online costs for proving non-membership vs. blocklist size

(b) Precomputation cost vs. additions/removals to the blocklist

Figure 3.5: Client-side performance for SNARKBLOCK

can be found in Section 11.2.

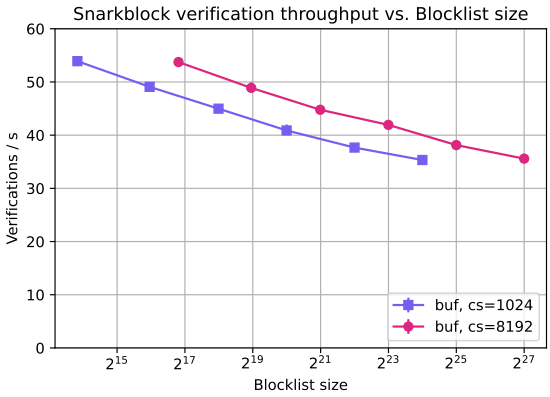
Theorem 2 (HICIAP Soundness)**.** HICIAP *on* $n-2$ *proofs is knowledge-sound against algebraic adversaries under the* $\mathbb{G}_1$-DL, n-ASDBP, *and* $2n$-SDH *assumptions.*

Theorem 3 (HICIAP Perfect Honest Verifier Zero Knowldege)**.** *The* HICIAP *protocol is perfect HVZK, provided that* $n \geq 16$.

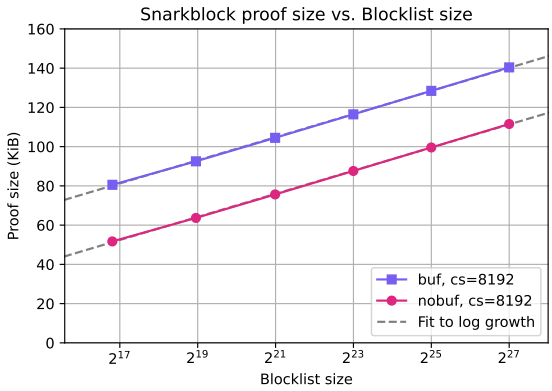
The requirement that $n \geq 16$ is to ensure that there is enough randomness in the set of Groth16 proofs to produce statistically independent HICIAP proofs. This, and the requirement that n be a power of 2 is not a barrier to usage, since proofs (and their prepared public inputs) can be duplicated arbitrarily many times to pad the input of the HICIAP algorithms.

7 Implementation and evaluation

We now detail the design and evaluation of SNARKBLOCK.



(a) SNARKBLOCK attestation verifications per second vs. blocklist size.



(b) SNARKBLOCK attestation proof size vs. blocklist size. Varying chunk size only shifts the x-axis.

Figure 3.6: Server-side performance for SNARKBLOCK

7.1 Implementation and setup

Hardware. All benchmarks were performed on a desktop computer with a 2021 Intel i9-11900KB CPU with 8 physical cores and 64GiB RAM running Ubuntu 20.04 with kernel 5.11.0-40-generic.

Code. SNARKBLOCK consists of 4.3k lines of Rust code[5] using the Arkworks [Ar21] zkSNARK crates and Rayon for parallelization where possible. The Criterion-rs crate was used for all benchmarks and statistics.

Statistics. Performance measurements are for medians and include error bars indicating the 95% confidence interval. These are not visible: over all benchmarks, the maximum relative standard error of the median is 1.6%.

Instantiating cryptographic primitives. We set $\lambda = 128$. We use BLS12-381 [Bow17b] for our Groth16 and HICIAP proofs, and Jubjub [ZCa19] for Schnorr signatures. We use hash functions $\mathsf{H}_1, \mathsf{H}_2, \mathsf{H}_3$ for identity registration, tag computation, and Schnorr signatures, respectively. Specifically, the commitment scheme used for R_{isu} is $\mathsf{Com}(m,r) := \mathsf{H}_1(r\|m)$ and the PRF scheme used for R_{chunk} is $\mathsf{Prf}_k(m) := \mathsf{H}_2(k\|m)$. Each H_i is a domain-separated instantiation of the Poseidon family [GKK+19], configured to be compatible with BLS12-381 and a 128-bit security level (i.e., $\alpha = 5$ and capacity $= 1$).

[5]The HICIAP crate source code can be found at https://github.com/rozbb/hiciap and the SNARKBLOCK crate source code can be found at https://github.com/rozbb/snarkblock.

	Client Attestation	Server Verification	Proof Size	
BLAC [TAKS10]	$2nM_{\mathbb{G}_1}$	$(2n+4)M_{\mathbb{G}_1}+2P$	Abs.	$(3n+12)\lvert\mathbb{F}\rvert+(n+3)\lvert\mathbb{G}_1\rvert$
			Real	$528\text{B}+n\cdot 144\text{B}$
SNARKBLOCK	$(197+10c)M_{\mathbb{G}_1}$ $+(160+10c)M_{\mathbb{G}_2}+2M_{\mathbb{G}_T}$ $+(244+15c)P$	$25M_{\mathbb{G}_1}+38P$ $+(46+10\log_2(c))M_{\mathbb{G}_T}$	Abs.	$8\lvert\mathbb{F}\rvert+29\lvert\mathbb{G}_1\rvert+14\lvert\mathbb{G}_2\rvert$ $+(48+10\log_2 c)\lvert\mathbb{G}_T\rvert$
			Real	$29.3\text{KiB}+\log_2(c)\cdot 5.6\text{KiB}$

Legend: n = Blocklist length, c = Num. chunks, $M_{\mathbb{G}}$ = Var. base MSM in $\mathbb{G}$, P = Pairing op., $\lvert\mathbb{G}\rvert$ = Size of group elem., $\lvert\mathbb{F}\rvert$ = Size of scalar field elem.

Table 3.1: BLAC and SNARKBLOCK operation counts and proof sizes. SNARKBLOCK operation counts assume a fully synchronized client and an unbuffered blocklist. The top subcell in the Proof Size column represents abstract element counts. The bottom subcell represents the byte count when instantiated with BLS12-381.

7.2 Evaluation

Benchmarks are given in Figures 3.5 and 3.6. Lines marked *buf* were benchmarked using a size-14 buffer of 16-element chunks. Lines marked *nobuf* used no buffer. The *cs* parameter refers to chunk size.

Figure 3.5a gives the time clients take to attest to non-membership on a blocklist that has recently changed. Specifically, this is the time it takes for a user to recompute the last Groth16 chunk proof; compute HICIAP proofs over the blocklist, buffer (if the buffer exists), issuance, and tag well-formedness proofs;[6] and compute the link proof over those. Separately, Figure 3.5b gives the offline computation a client must do as a function of the number of additions/removals to the blocklist (e.g. per day). This includes syncing chunks and precomputing its Groth16 issuance and tag well-formedness proofs for the next attestation.

Figure 3.6 gives proof sizes and throughput for server verification. These graphs, which are semi-log scale, show that, unlike previous work, SNARKBLOCK proofs scale logarithmically with the number of elements in the blocklist.

8 Discussion

We now discuss real-world performance and possible extensions.

8.1 Is SNARKBLOCK practical?

Attestation latency. How long can attestation take in practice? A client that computes an attestation in the background while a user drafts their post or comment adds no latency to the user's workflow. When the expected time to write a comment is lower than attestation time (e.g., writing a tweet), then the comment must be queued and posted by the client software when attestation is complete. While this is acceptable in many cases, it renders SNARKBLOCK impractical for real-time chat when blocklists are large.

Operating Costs. SNARKBLOCK can be used when (1) logging in to a pseudonymous session, or (2) posting or commenting anonymously. The latter puts more load on a server. We use it as an estimate for worst-case performance costs.

English language Wikipedia had 2 edits per second in 2021 [Wika] and Reddit had 64 comments per second in 2020 [Red20]. Estimating from event logs for June through October 2020, English language Wikipedia has about 2k bans per day, of which 250 (12.5%) were permanent. Assuming a similar ban rate, Reddit has at least 8k permanent bans per day and perhaps 32k temporary.

An Amazon EC2 c5.4xLarge costs about $10 USD per day if reserved for a year.[7] For a blocklist of 2^{24} entries, SNARKBLOCK handles at least 35 attestations per second. At Reddit's scale, deployed in the more resource-intensive attestation-per-comment mode, SNARKBLOCK costs on the order of $20 per day when pessimistically assuming full EC2 retail pricing at scale. With generous allowances for CPU differences and virtualization overhead, SNARKBLOCK is at most $200 per day in the worst case. For reference, Facebook pays moderators in the US $120 a day [Sal19].

A final consideration is increased bandwidth usage by the server. A SNARKBLOCK attestation

[6] For speed, we combine R_{isu} and R_{tag} into a single circuit in our implementation. Thus there are only 2 proofs to link in an unbuffered SNARKBLOCK attestation.

[7] With 16 virtual Xeon CPUs and 30GB of memory, this is a decent analogue to our test system since in testing, SNARKBLOCK never exceeded 20 GB of memory for verification.

for a 4M-entry blocklist is 130KiB, at least two orders of magnitude larger than an average text comment it would accompany in the fully anonymous setting. 130KiB, however, is dwarfed by the size of image and video files uploaded to many service providers. Moreover, inbound traffic is typically a small fraction of total traffic for web services. So much so that on EC2, for example, it's free.

8.2 Client-side performance vs BLAC

SNARKBLOCK's main advantage over BLAC is logarithmic server-side scaling. Nonetheless, we briefly discuss client-side performance. The biggest problem for BLAC, surprisingly, is proof size. A blocklist with 4M bans yields proofs of 549MiB. In contrast, a SNARKBLOCK attestation is less than 200KiB for a 134 million entry list. On a 50Mbps connection, which is 5× the upstream bandwidth of the median US household [FCC20],[8] uploading a BLAC attestation would take 90s. Even if both have a 100Mbps fiber connection, SNARKBLOCK can compute and upload the attestation before a BLAC proof would upload.

What if we ignore proof size? Although Tsang et al. give benchmarks for BLAC, they are on 10+ year old hardware using the very dated PBC library [Lyn] for pairings. Luckily, Tsang et al. also characterize their system's performance in terms of group operations. In lieu of a reimplementation, we report these measurements and give the equivalent values for SNARKBLOCK in Table 3.1.

SNARKBLOCK pays an initial overhead in terms of upfront costs (e.g., the 244 pairings). The major advantage for SNARKBLOCK is that its operations are per chunk as opposed to per element. Ignoring constants, SNARKBLOCK is faster for proving whenever $2nM_{\mathbb{G}_1} > \frac{10nM_{\mathbb{G}_1}}{s} + \frac{10nM_{\mathbb{G}_2}}{s} + \frac{15nP}{s}$, where $s = n/c$ is the chunk size. Thus, as the blocklist size grows, SNARKBLOCK will outperform as long as $s > 5 + 12.5o$ where $o = P/M_{\mathbb{G}_1}$ is the overhead for pairings relative to $\mathbb{G}_1$ multiplications.[9] On our benchmark system, $\mathbb{G}_2$ multiplications are about 3 times $\mathbb{G}_1$, and pairings twice that. i.e.,

[8] FCC measurements are a trailing indicator. The latest report as of this writing, released in Sep. 2020 [FCC20], is for data as of Dec. 2018. For Oct. 2021, `Speedtest.net` reports its US users have upstream *averages* of 19.18Mbps for wired connections and 8.81Mbps for mobile.

[9] Since $M_{\mathbb{G}_2} < P$, we can approximate them as the same.

$o \approx 6$.

Unfortunately, giving a precise estimate for the transition point is impossible with only group operations: we need to compare runtimes to a full reimplementation of BLAC. Real-world performance will differ significantly from group operation counts due to parallelization and other optimizations. Indeed, SNARKBLOCK outperforms estimates based on group operations and benchmarked operation times.

SNARKBLOCK has one substantial cost that BLAC does not: SNARKBLOCK requires periodic sync computations for blocklist additions and removals. Per Figure 3.5, this is less than 200s for every 12k additions, with appropriate batching or buffering. For much faster churning lists, e.g., 64k additions per day, BLAC would have a large initial advantage by avoiding these recomputation costs. But at even 8k additions per day, the service provider will exceed 2^{22} bans within two years. At this point, under reasonable bandwidth assumptions, SNARKBLOCK will outperform BLAC.

8.3 Cold Start

One significant caveat for SNARKBLOCK is that a new user of a system with a pre-existing blocklist must do significant work to sync the entire blocklist and compute the chunk proofs.

One option is to leverage the issuance date for a user's identity and allow them to skip proving membership in blocklist chunks whose last entry is before they joined. This can be done directly now, albeit at the cost of leaking the user's approximate join time for, e.g., a particular forum. Specifically, a given service provider can use a custom CRS for their Groth16 chunk proofs. They can then, using the CRS trapdoor, give each new user non-membership chunk proofs for earlier portions of the blocklist. Crucially, proving with a trapdoor is constant-time, so this process is efficient.

We leave to future work the question of how to build a general trapdoor for cold start. In particular, it should be possible skip chunk proofs whose last entry was inserted before the issuance data of the user's identity.

9 Related work

For a full formalization of privacy preserving blocklists, we refer the reader to excellent SoK of Henry and Goldberg [HG11]. This also describes a number of interesting hybrid systems that can be constructed in a black-box way from either SNARKBLOCK or BLAC and allow for pruning of blocklists.

9.1 Blocklists

The work closest to ours is the ZKBL approach introduced in BLAC [TAKS10]. As discussed in prior sections, by replacing the zero-knowledge proofs in BLAC with our novel proving system HICIAP, we get a system that offers logarithmic verification time and proof size, rather than linear. Further, we extend the system to support federated identities.

Also close to our work is the windowed approach from PEREA [TAKS08], also by the authors of BLAC. In PEREA, users are issued a finite number of one-time-use identity tickets for use during a revocation window, e.g., one month. To complete an action, a user must prove none of those tickets are in the blocklist. A user computes the same proof to get the next set of tickets. Verification time is proportional to size of the revocation window, not the total size of the blocklist. It has a number of drawbacks for broad deployment on the web:

1. Issuing users a small number of tickets is feasible for individual low-volume sites, but the limit would apply to all sites in a federated system.

2. The approach is inherently centralized. All blocklists must be registered with the single identity provider to ensure non-membership before reissuing identities.

3. Service providers must react quickly to ban users, since bans expire once the user gets new identities. The exact time depends on configuration; PEREA gives the example of a 1-hour window for a site like Wikipedia.

Finally, a number of systems provide weaker anonymity. One line of work relies on a trusted

third party to revoke anonymity, e.g, [Cha85, Cv91, BMW03]. Another approach is to leverage blind signatures to remove the linkage between, e.g., an IP address, and the pseudonym, e.g., [JKTS07, TKCS11, LH10]. These schemes only provide pseudonymity, allowing the linking of pseudonymous posts across different platforms. In contrast, SNARKBLOCK provides anonymity and does not trust a third party to safeguard user identities.

9.2 Zero-knowledge proofs

Our HICIAP protocol consists of a non-membership proof and a proof that a revocation tag has been computed correctly. Bayer and Groth design a non-membership proof [BG13] with logarithmic proof size and no trusted setup, but they have (quasi-)linear prover and verifier costs. Non-membership proofs can also be constructed in groups of unknown order [CL02, BCFK19], and have constant verifier time and prover time. However, it is not obvious how to apply these techniques to a blocklist without requiring a finite number of tickets per user as in PEREA.

An alternative and thus far unexplored direction for proving blocklist non-membership is recent advances in recursive zero-knowledge proofs using techniques first introduced by Bowe et al. for Halo [BGH19]. Halo-like schemes, formalized in [BCMS20] as *accumulator schemes*, have been extended to a wider variety of polynomial commitment schemes in [BDFG21]. These use Bulletproofs [BBB+18] as a building block, which introduces a linear verification time component to the constructions. This cost is typically small if the individual computation step is small, but leads to a different set of design tradeoffs than recursive proofs with fully succinct verifiers. Bünz et al. [BCL+21] and Kothapalli et al. [KST21] improve upon these results.

One key challenge to using Halo-like techniques is the concrete cost of recursion. With SNARKBLOCK, aggregation costs are less than 8× the cost of native verification, keeping online costs low. For Halo-like systems, these costs depend heavily on the exact approach taken.

10 Appendix: Deferred definitions

We will use the definition of the Discrete Log (DL) assumption from [FKL18] and the definitions of the q-ASDBP and q-SDH assumptions from [BMM+21].

Assumption (DL)**.** *Let* $\mathsf{GGen}(1^\lambda)$ *be a nondeterministic function which outputs a group* $\mathbb{G}$ *of prime order* $p > 2^\lambda$ *and a generator* G*.*

The Discrete Logarithm *assumption holds relative to* GGen *iff for any PPT algorithm* A*,*

$$\Pr\left[xG = H \mid (\mathbb{G}, G) \leftarrow \mathsf{GGen}(1^\lambda), H \leftarrow\!\$\, \mathbb{G}, x \leftarrow \mathsf{A}(\mathbb{G}, G, H)\right] = \mathrm{negl}(\lambda).$$

Assumption (q-ASDBP)**.** *Let* $\mathsf{GGen}(1^\lambda)$ *be a nondeterministic function which outputs 3 groups* $\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T$ *of prime order* $p > 2^\lambda$*; a bilinear pairing* $e : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$*; and generators* G, H *for* $\mathbb{G}_1$ *and* $\mathbb{G}_2$*, respectively. Let* pp *denote* $(\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, G, H, e)$*.*

The q-Auxiliary Structured Double Pairing assumption holds relative to GGen *iff for any PPT algorithm* A*,*

$$\Pr\left[\begin{array}{c} (A_0, \ldots, A_{q-1}) \neq \mathbf{1}_{\mathbb{G}_1} \\ \bigwedge \prod e\left(A_i, \left[\beta^{2i}\right]_2\right) = 1 \end{array} \;\middle|\; \begin{array}{c} \mathsf{pp} \leftarrow \mathsf{GGen}(1^\lambda) \\ \beta \leftarrow\!\$\, \mathbb{F} \\ (A_0, \ldots, A_{q-1}) \leftarrow \mathsf{A}(\mathsf{pp}, [\beta]_1, \{\left[\beta^{2i}\right]_2\}_{i=1}^{q-1}) \end{array}\right] = \mathrm{negl}(\lambda).$$

Assumption (q-SDH)**.** *Let* $\mathsf{GGen}(1^\lambda)$ *be a nondeterministic function which outputs 3 groups* $\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T$ *of prime order* $p > 2^\lambda$*; a bilinear pairing* $e : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$*; and generators* G, H *for* $\mathbb{G}_1$ *and* $\mathbb{G}_2$*, respectively. Let* pp *denote* $(\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, G, H, e)$*.*

The q-Strong Diffie Hellman assumption holds relative to GGen *iff for any PPT algorithm* A*,*

$$\Pr\left[A = \left[(\beta - a)^{-1}\right]_2 \;\middle|\; \begin{array}{c} \mathsf{pp} \leftarrow \mathsf{GGen}(1^\lambda) \\ \beta \leftarrow\!\$\, \mathbb{F} \\ (a, A) \leftarrow \mathsf{A}(\mathsf{pp}, [\beta]_1, [\beta]_2, \ldots, [\beta^q]_1, [\beta^q]_2) \end{array}\right] = \mathrm{negl}(\lambda).$$

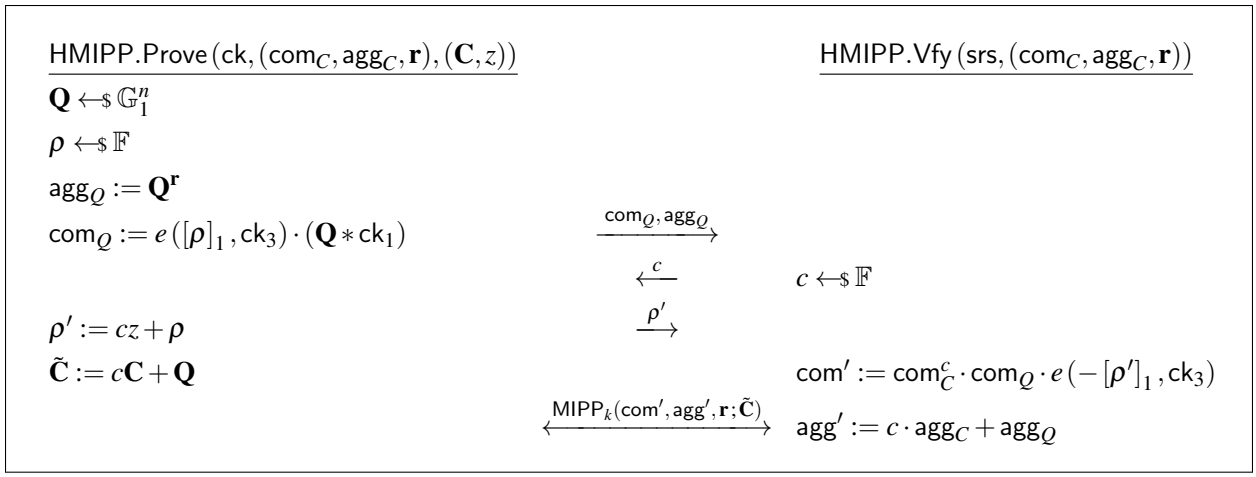


Figure 3.7: The HMIPP protocol

11 Appendix: Deferred proofs

11.1 Subprotocols

In this section we state and prove theorems of security of the subprotocols used in HICIAP and SnarkBlock, namely HMIPP, HWW, and HICIAP.Link.

Recall R_{HMIPP} is a zero-knowledge ("hiding") version of the $R_{\mathsf{MIPP}-k}$ relation. Both relations are detailed in Figure 3.1, and their proof systems are given in [BMM+21]. In Figure 3.7 we restate the R_{HMIPP} proof system, and we prove its knowledge-soundness below.

Lemma 4. *The* HMIPP *protocol is perfect HVZK. (Lemma 5 in [BMM+21])*

Lemma 5. HMIPP *on n elements is knowledge-sound against algebraic adversaries under the* n-ASDBP *and* $2n$-SDH *assumptions.*

Proof. By Theorem 6 of [BMM+21], under the n-ASDBP and $2n$-SDH assumptions, there is a polynomial-time extractor $\mathsf{E}_{\mathsf{MIPP}-k}$ for $\mathsf{MIPP}_k(\mathsf{ck}, \mathsf{com}_C, \mathsf{agg}_C, \mathbf{r}, \mathbf{C})$ which extracts $\tilde{\mathbf{C}}$ such that $\tilde{\mathbf{C}} * \mathsf{ck}_1 = \mathsf{com}'$ and $\tilde{\mathbf{C}}^{\mathbf{r}} = \mathsf{agg}'$.

Our extractor $\mathsf{E}_{\mathsf{HMIPP}}$ is defined as follows: send a random c_1 as the verifier in the protocol would; receive ρ' and denote it ρ_1; use $\mathsf{E}_{\mathsf{MIPP}-k}$ to extract $\tilde{\mathbf{C}}_1$ from MIPP_k; rewind and send a fresh c_2; receive ρ_2; use $\mathsf{E}_{\mathsf{MIPP}-k}$ to extract $\tilde{\mathbf{C}}_2$; output $z := (c_1 - c_2)^{-1}(\rho_1 - \rho_2)$ and $\mathbf{C} := (c_1 - c_2)^{-1}(\tilde{\mathbf{C}}_1 - \tilde{\mathbf{C}}_2)$.

We refer to the agg' from the first execution by agg'_1 and from the second execution by agg'_2. We notate com'_1 and com'_2 similarly. Since $\mathsf{E}_{\mathsf{MIPP}-k}$ guarantees $\tilde{\mathbf{C}}_1^{\mathbf{r}} = \mathsf{agg}'_1$ and $\tilde{\mathbf{C}}_2^{\mathbf{r}} = \mathsf{agg}'_2$,

$$\mathbf{C}^{\mathbf{r}} = (c_1 - c_2)^{-1}(\tilde{\mathbf{C}}_1 - \tilde{\mathbf{C}}_2)^{\mathbf{r}} = (c_1 - c_2)^{-1}(\mathsf{agg}'_1 - \mathsf{agg}'_2) = \mathsf{agg}_C.$$

Further, since $\mathsf{E}_{\mathsf{MIPP}-k}$ guarantees $\tilde{\mathbf{C}}_1 * \mathsf{ck}_1 = \mathsf{com}'_1$ and $\tilde{\mathbf{C}}_2 * \mathsf{ck}_1 = \mathsf{com}'_2$,

$$\begin{aligned}
&e([z]_1, \mathsf{ck}_3) \cdot (\mathbf{C} * \mathsf{ck}_1) \\
&\qquad = e([z]_1, \mathsf{ck}_3) \cdot ((\tilde{\mathbf{C}}_1 - \tilde{\mathbf{C}}_2) * \mathsf{ck}_1)^{(c_1-c_2)^{-1}} \\
&\qquad = e([z]_1, \mathsf{ck}_3) \cdot (\mathsf{com}'_1 \cdot (\mathsf{com}'_2)^{-1})^{(c_1-c_2)^{-1}} \\
&\qquad = e([z]_1, \mathsf{ck}_3) \cdot ((\mathsf{com}_C^{c_1} \cdot \mathsf{com}_Q \cdot e(-[\rho_1]_1, \mathsf{ck}_3)) \\
&\qquad\qquad \cdot (\mathsf{com}_C^{-c_2} \cdot \mathsf{com}_Q^{-1} \cdot e([\rho_2]_1, \mathsf{ck}_3)))^{(c_1-c_2)^{-1}} \\
&\qquad = e([z]_1, \mathsf{ck}_3) \cdot \mathsf{com}_C \cdot e(\left[-(\rho_1 - \rho_2)(c_1 - c_2)^{-1}\right]_1, \mathsf{ck}_3) \\
&\qquad = e([z]_1, \mathsf{ck}_3) \cdot \mathsf{com}_C \cdot e(-[z]_1, \mathsf{ck}_3) \\
&\qquad = \mathsf{com}_C
\end{aligned}$$

as desired. □

Recall the hidden wire well-formedness relation R_{HWW} described in Section 6

$$R_{\mathsf{HWW}} := \left\{ \left(\begin{array}{c} (U, V, \{G_i\}_{i=1}^5 \in \mathbb{G}_1 ; \\ w, x, y \in \mathbb{F} \end{array} \right) : \begin{array}{c} U = wG_1 + xG_2 + yG_3 \\ \bigwedge V = wG_4 + xG_5 \end{array} \right\}$$

We write this relation's proof system in Figure 3.8 and prove it knowledge-sound and honest-verifier zero-knowledge below.

Lemma 6. HWW *is perfect HVZK.*

Proof. We define a simulator as follows: sample $r_1, r_2, r_3, c \leftarrow\!\!\$\ \mathbb{F}$ uniformly; compute $\mathsf{com} := (r_1 G_1 + r_2 G_2 + r_3 G_3 + cU, r_1 G_4 + r_2 G_5 + cV)$; output $(\mathsf{com}, r_1, r_3, r_3, c)$. This is perfectly indistin-

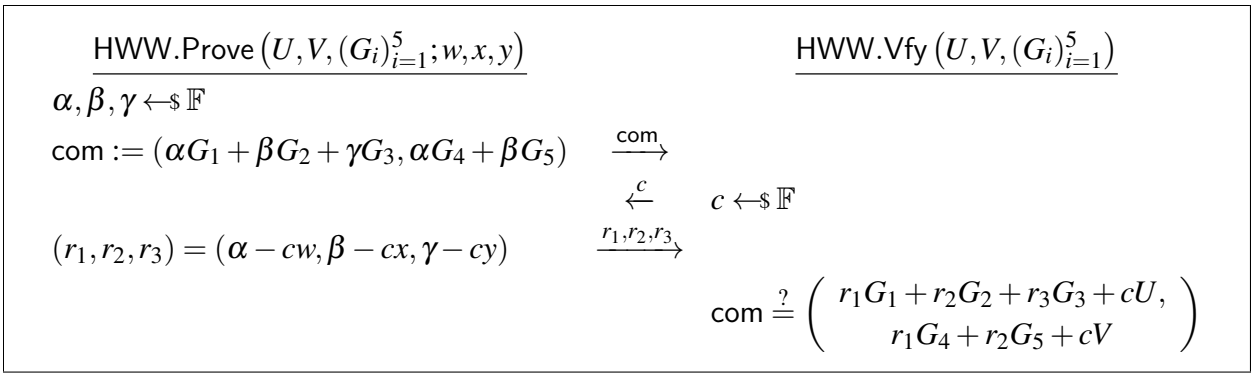


Figure 3.8: The Hidden Wire Well-formedness sigma protocol

guishable from a real transcript: in a real transcript, r_1, r_2, r_3 are uniform and independent because of the blinders α, β, γ; c is uniform and independent by definition; and com is uniquely determined by these values. In the simulator, r_1, r_2, r_3 and c are uniform and independent by construction and com is uniquely determined by these values. □

Lemma 7. HWW *is knowledge-sound.*

Proof. We define an extractor $\mathsf{E}_{\mathsf{HWW}}$ as follows, aborting on verification error: send $c \leftarrow\$ \mathbb{F}$ to the prover; receive r_1, r_2, r_3; rewind and send a fresh $c' \leftarrow\$ \mathbb{F}$; receive r'_1, r'_2, r'_3. Finally, $\mathsf{E}_{\mathsf{HWW}}$ outputs

$$w := (r_1 - r'_1)(c' - c)^{-1} \qquad x := (r_2 - r'_2)(c' - c)^{-1} \qquad y := (r_3 - r'_3)(c' - c)^{-1}.$$

Since $\mathsf{E}_{\mathsf{HWW}}$ did not abort, i.e., both runs verified, it is the case that $U = wG_1 + xG_2 + yG_3$ and $V = wG_4 + xG_5$. □

Recall the HICIAP.Link relation R_{link}

$$R_{\mathsf{link}} = \left\{ \left(\{P_i\}_{i=1}^3, \{\mathsf{com}_{a_0}^{(i)}\}_{i=1}^t; a_0, \{z_1^{(i)}, z_3^{(i)}\}_{i=1}^t \right) : \bigwedge_{i=1}^t \mathsf{com}_{a_0}^{(i)} = a_0 P_1 + z_1^{(i)} P_2 + z_3^{(i)} P_3 \right\}.$$

We write this relation's proof system in Figure 3.9 and prove it knowledge-sound and honest-verifier zero-knowledge below. Like HWW, this is a Σ-protocol. Thus, its proofs of security are nearly identical to those of HWW.

Lemma 8. HICIAP.Link *is perfect HVZK.*

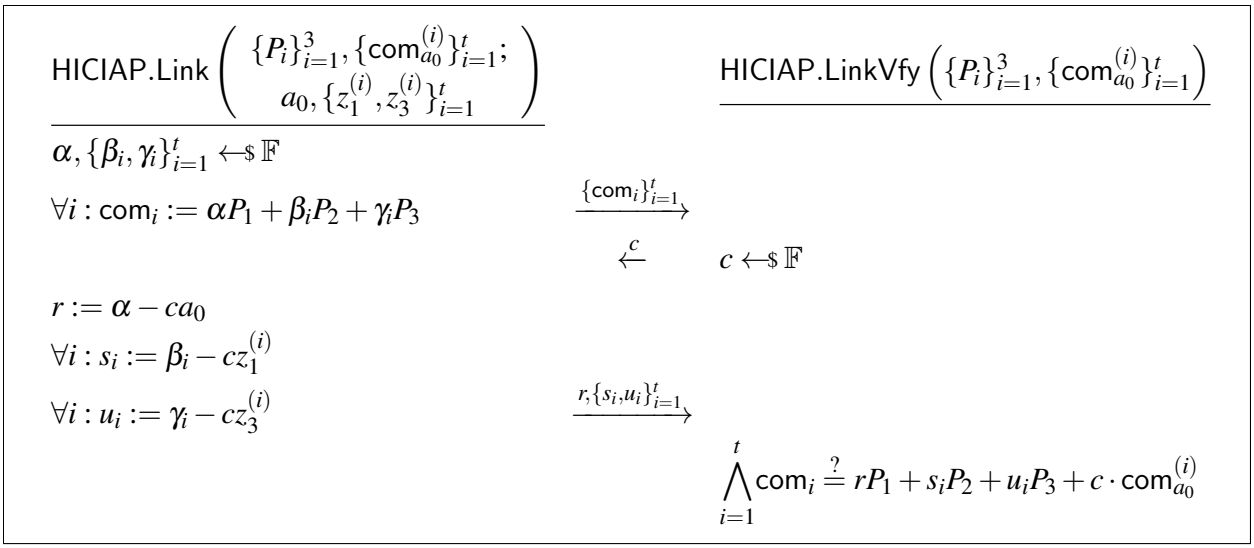


Figure 3.9: The HICIAP.Link sigma protocol

Proof. We define a simulator as follows: sample $r, s_i, u_i, c \leftarrow\$ \mathbb{F}$ uniformly for all $i = 1, \ldots, t$; for all i, compute $\mathsf{com}_i := rP_1 + s_i P_2 + u_i P + c \cdot \mathsf{com}_{a_0}^{(i)}$; output $(\{\mathsf{com}_i\}_{i=1}^t, c, r, \{s_i, u_i\}_{i=1}^t)$. This is perfectly indistinguishable from a real transcript: in a real transcript, all r, s_i, u_i, c are uniform and independent because of the blinders $\alpha, \beta_i, \gamma_i$; c is uniform and independent by definition; and each com_i is uniquely determined by these values. In the simulator, all r, s_i, u_i, c are all uniform and independent by construction, and com is uniquely determined by these values. □

Lemma 9. HICIAP.Link *is knowledge-sound.*

Proof. We define an extractor $\mathsf{E}_{\mathsf{link}}$ as follows, aborting on verification error: send $c \leftarrow\$ \mathbb{F}$ to the prover; receive $r, \{s_i, u_i\}_{i=1}^t$; rewinds and send a fresh $c' \leftarrow\$ \mathbb{F}$; receive $r', \{s_i', u_i'\}_{i=1}^t$. Finally, $\mathsf{E}_{\mathsf{link}}$ outputs

$$a_0 := (r - r')(c' - c)^{-1} \qquad z_1^{(i)} := (s_i - s_i')(c' - c)^{-1} \qquad z_3^{(i)} := (u_i - u_i')(c' - c)^{-1}$$

for all i. Since $\mathsf{E}_{\mathsf{link}}$ did not abort, i.e., both runs verified, it is the case that $\mathsf{com}_i = a_0 P_1 + z_1^{(i)} P_2 + z_3^{(i)} P_3$ for all i. □

11.2 Main theorems

In this section we prove the main theorems of SNARKBLOCK and HICIAP.

Theorem 1 (SNARKBLOCK Security)**.** SNARKBLOCK *described in Figure 3.3 is blocklistable, anonymous and non-frameable provided that Groth16 and* HICIAP *proofs are knowledge-sound and subversion zero-knowledge; Schnorr signatures are unforgeable;* Prf *is pseudorandom; and* Com *is binding and hiding.*

Proof Sketch. **Blocklistability.** Let A be an adversary that breaks blocklistability. Then A generates a verifying attestation $(\pi_{\mathsf{zkbl}}, \mathsf{tag}, \mathsf{nonce})$. Either an extractor can output k such that $\mathsf{tag} = \mathsf{Prf}_k(n)$; or $\hat{\pi}_{\mathsf{tag}}$ is a forgery for HICIAP and we cannot extract a verifying Groth16 statement and proof k, π_{tag} breaking knowledge soundness; or π_{tag} is a forgery for Groth16 and $\mathsf{tag} \neq \mathsf{Prf}_k(\mathsf{nonce})$ breaking knowledge soundness.

Either an extractor can output a verifying signature σ under some identity providers public key pk_{i^*} on the message $\mathsf{com} = \mathsf{Com}(k, r)$ the same k and some r; or $\hat{\pi}_{\mathsf{link}}$ is a forgery for Link breaking knowledge soundness; or the adversary can find $\mathsf{com} = \mathsf{Com}(k', r')$ for different k', r' breaking binding; or $\hat{\pi}_{\mathsf{isu}}$ is a forgery for HICIAP and we cannot extract a verifying Groth16 proof π_{isu} for k breaking knowledge soundness; or $((k, \mathscr{I}), \pi_{\mathsf{isu}})$ is a forgery for Groth16 breaking knowledge soundness. If σ is a verifying signature then either pk_{i^*} authenticated com at some point, or we break unforgeability of the signature scheme.

If σ has been authenticated by pk_{i^*} that gets blocked then $(\mathsf{tag}, \mathsf{nonce})$ gets added to $\mathscr{L}$. If A later generates a verifying attestation with respect to the same σ then either $\hat{\pi}_{\mathsf{link}}$ is a forgery for Link breaking knowledge soundness; or the adversary can find $\mathsf{com} = \mathsf{Com}(k', r')$ for different k', r' breaking binding; or $\hat{\pi}_{\mathsf{chunk}}$ is a forgery for HICIAP and we cannot extract verifying Groth16 proofs π_{chunk_j} for k breaking knowledge soundness; or π_{chunk_j} is a forgery for Groth16 for some j and $\mathsf{Prf}_k(\mathsf{nonce}^*) = \mathsf{tag}^*$ for some $(\mathsf{nonce}^*, \mathsf{tag}^*) \in \mathscr{L}$ breaking knowledge soundness; or σ is never associated with a blocked session and A does not break blocklistability.

Non-Frameability. If an adversarial identity provider prevents an honest user from authenticating

then they must get some $(\mathsf{nonce}, \mathsf{tag})$ added to $\mathscr{L}$ such that $\mathsf{tag} = \mathsf{Prf}_k(\mathsf{nonce})$ for an honest user's k. By the pseudorandomness of Prf and the anonymity of SNARKBLOCK, the probability that they guess any such tag is negligible.

Anonymity. We claim that the transcript between an honest user and any number of identity providers and service providers is uncorrelated. By the hiding of Com we have that com reveals no information about k (and uses distinct r each registration). By the zero-knowledge of HICIAP we have that π_{zkbl} reveals no information (even to the identity providers). nonce is chosen uniformly at random for each session associated with k. By the pseudorandomness of Prf, tag is indistinguishable from random for users that don't know k and thus reveals no information about k. Thus the scheme is anonymous. □

Theorem 2 (HICIAP Soundness)**.** HICIAP *on* $n-2$ *proofs is knowledge-sound against algebraic adversaries under the* $\mathbb{G}_1$-DL*,* n-ASDBP*, and* $2n$-SDH *assumptions.*

Proof. We wish to show that, there exists an efficient HICIAP extractor $\mathsf{E}^{\mathsf{P}^*}_{\mathsf{HICIAP}}(\mathsf{ck}, \mathsf{crs}, \hat{\mathbf{S}})$ which outputs a witness $(a_0, \mathbf{A}', \mathbf{B}', \mathbf{C}')$ such that for all $i = 1, \ldots, n-2$,

$$\mathsf{Groth16.Vfy}\left(\mathsf{crs}, (A'_i, B'_i, C'_i), (a_0, \hat{S}_i)\right).$$

By Theorem 3 of [BMM+21], there is an efficient extractor $\mathsf{E}_{\mathsf{TIPP}}$ for $\mathsf{TIPP}(\mathsf{com}_A, \mathsf{com}_B, \mathsf{agg}_{AB}, \mathbf{r})$ which extracts $(\mathbf{A}, \mathbf{B})$ such that $\mathsf{com}_A = \mathbf{A} * \mathsf{ck}_1$, $\mathsf{com}_B = \mathsf{ck}_2 * \mathbf{B}$, and $\mathsf{agg}_{AB} = \mathbf{A}^{\mathbf{r}} * \mathbf{B}$. By Theorem 6 of [BMM+21], there is an efficient extractor $\mathsf{E}_{\mathsf{MIPP}-k}$ for $\mathsf{MIPP}_k(\mathsf{com}_{\mathsf{in}}, \mathsf{agg}_{\mathsf{in}}, \mathbf{r}')$ which extracts $\hat{\mathbf{S}}'$ such that $\hat{\mathbf{S}}' * \mathsf{ck}_{1,[:n-2]} = \mathsf{com}_{\mathsf{in}}$ and $(\hat{\mathbf{S}}')^{\mathbf{r}'} = \mathsf{agg}_{\mathsf{in}}$. By Lemma 5, there is an efficient extractor $\mathsf{E}_{\mathsf{HMIPP}}$ for $\mathsf{HMIPP}(\mathsf{com}_C, \mathsf{agg}_C, \mathbf{r})$ which extracts $(\mathbf{C}, z_4)$ such that $\mathbf{C}^{\mathbf{r}} = \mathsf{agg}_C$ and $\mathsf{com}_C = e(z_4 G, \mathsf{ck}_3) + (\mathbf{C} * \mathsf{ck}_1)$. By Lemma 7, there is an efficient extractor $\mathsf{E}_{\mathsf{HWW}}(\mathsf{com}_{a_0}, W, P_1, P_2, P_3, G_1, G_2)$ for HWW which extracts (a_0, z_1, z_3) such that $\mathsf{com}_{a_0} = a_0 P_1 + z_1 P_2 + z_3 P_3$ and $W = a_0 G_1 + z_1 G_2$.

Let P^* be a probabilistic prover with fixed randomness and unknown probability ε of producing an argument that accepts. We define an extractor $\mathsf{E}^{\mathsf{P}^*}_{\mathsf{HICIAP}}(\mathsf{ck}, \mathsf{crs}, \hat{\mathbf{S}})$, which extracts the witness

$(a_0, \mathbf{A}, \mathbf{B}, \mathbf{C})$, as follows.

First, run HICIAP with a random $r \leftarrow\!\!\$\, \mathbb{F}$, and run all the subprotocols honestly. Note that, by the definition of knowledge soundness, if P^* does not produce an accepting transcript tr on the first run, the extractor is allowed to exit early with $(\mathsf{tr}, \perp)$. If tr is accepting, rewind to the point after r is chosen and run $\mathsf{E}_{\mathsf{MIPP}-k}$, $\mathsf{E}_{\mathsf{HMIPP}}$, $\mathsf{E}_{\mathsf{HWW}}$, and $\mathsf{E}_{\mathsf{TIPP}}$ to extract $(a_0, z_1, z_3, z_4, \mathbf{A}, \mathbf{B}, \mathbf{C}, \hat{\mathbf{S}}')$. Finally, output $(a_0, \mathbf{A}_{[:n-2]}, \mathbf{B}_{[:n-2]}, \mathbf{C}_{[:n-2]})$.

Note that $\mathsf{E}_{\mathsf{HICIAP}}$ algorithm is efficient, since its runtime is at most the sum of the runtimes of $\mathsf{E}_{\mathsf{HMIPP}}$, $\mathsf{E}_{\mathsf{HWW}}$, and $\mathsf{E}_{\mathsf{TIPP}}$, which are assumed to be efficient.

To prove the claimed relations hold, first note that the commitment $\mathsf{com}_{\mathsf{in}} = \hat{\mathbf{S}} * \mathsf{ck}_{1,[:n-2]}$ is computationally binding under the $(n-2)$-ASDBP assumption, and so, with overwhelming probability, $\hat{\mathbf{S}}' = \hat{\mathbf{S}}$.

It remains to show that, with overwhelming probability, the extracted witness satisfies the Groth16 verification condition. That is, for all $i = 1, \ldots, n-2$,

$$e(A_i, B_i) = e([\alpha]_1, [\beta]_2) \cdot e(C_i, [\delta]_2) \cdot e\left(\hat{S}_i, [\gamma]_2\right).$$

The commitments $\mathsf{com}_A, \mathsf{com}_B, \mathsf{com}_C$ are computationally binding under the n-ASDBP assumptions. Further, since P_1, P_2, P_3 are unrelated by assumption, the Pedersen commitment com_{a_0} is computationally binding by the DL hardness assumption on $\mathbb{G}_1$. Thus, with overwhelming probability, the formal product being evaluated in TIPP is the one committed to by $\mathsf{com}_A, \mathsf{com}_B, \mathsf{com}_C$, i.e.,

$$\begin{aligned}\prod_{i=1}^{n} e(A_i, B_i)^{x_i} = \prod_{i=1}^{n-2} e(a_0 W_0, [\gamma]_2)^{x_i} \cdot \prod_{i=1}^{n} e([\alpha]_1, [\beta]_2)^{x_i} \cdot \prod_{i=1}^{n-2} e\left(\hat{S}_i, [\gamma]_2\right)^{x_i} \\ \cdot \prod_{i=1}^{n} e(C_i, [\delta]_2)^{x_i} \cdot e([z_1]_1, [\gamma]_2)^{x^{n-1}}\end{aligned}$$

Then by the Schwartz-Zippel lemma, the above relation holds with probability at least $1 - n/p$. Since the above equality directly implies the Groth16 verification condition, the theorem is proved. □

Theorem 3 (HICIAP Perfect Honest Verifier Zero Knowldege)**.** *The* HICIAP *protocol is perfect*

HVZK, provided that $n \geq 16$.

Proof. A HICIAP proof consists of the values:

$$\mathsf{com}_{\mathsf{in}}, \mathsf{com}_{a_0}, \mathsf{com}_A, \mathsf{com}_B, \mathsf{com}_C, \mathsf{agg}_{\mathsf{in}}, \mathsf{agg}_C, W, \mathsf{tr}_{\mathsf{MIPP}\text{-}k}, \mathsf{tr}_{\mathsf{HMIPP}}, \mathsf{tr}_{\mathsf{HWW}}, \mathsf{tr}_{\mathsf{TIPP}}.$$

We construct a simulator that knows a Groth16 simulation trapdoor τ to crs and which can choose the verifier's randomness in advance, such that the simulated transcript is indistinguishable from an honest prover's transcript. The simulator will also use the simulators described in Lemmas 4 and 6 which generate transcripts for subprotocols HMIPP and HWW,

$$\mathsf{Sim}_{\mathsf{HMIPP}}(\mathsf{ck}, \mathsf{com}_C, \mathsf{agg}_c, \mathbf{r}) \rightarrow \mathsf{tr}_{\mathsf{HMIPP}}$$
$$\mathsf{Sim}_{\mathsf{HWW}}(\mathsf{com}_{a_0}, W, P_1, P_2, P_3, G_1, G_2) \rightarrow \mathsf{tr}_{\mathsf{HWW}}.$$

The simulator is given the Groth16 prepared public inputs $\hat{\mathbf{S}}$ and behaves as follows.

1. The simulator computes the first prover message $\mathsf{com}_{a_0}, \mathsf{com}_A, \mathsf{com}_B, \mathsf{com}_C$. It chooses randomness $a_0, z_1, z_2 \leftarrow\!\$\, \mathbb{F}$ and $\mathsf{com}_{a_0} \leftarrow\!\$\, \mathbb{G}_1$ and $\mathsf{com}_C \leftarrow\!\$\, \mathbb{G}_T$. It runs $(A'_i, B'_i, C'_i) = \mathsf{Sim}_{\mathsf{Groth16}}(\mathsf{crs}, \tau, (a_0, \hat{S}_i))$ for $1 \leq i \leq n-2$. It sets

$$\mathbf{A} := \mathbf{A}' || [z_1]_1 || [z_2]_1 \qquad \mathsf{com}_A := \mathbf{A} * \mathsf{ck}_1$$
$$\mathbf{B} := \mathbf{B}' || [\gamma]_2 || [\delta]_2 \qquad \mathsf{com}_B := \mathsf{ck}_2 * \mathbf{B}$$
$$\mathbf{C} := \mathbf{C}' || [1]_1 || [z_2]_1 .$$

2. The simulator computes the first verifier message honestly and chooses $r \leftarrow\!\$\, \mathbb{F}$ randomly.

3. The simulator uses $\hat{\mathbf{S}}$ to construct $\mathsf{agg}_{\mathsf{in}}$ and $\mathsf{tr}_{\mathsf{MIPP}\text{-}k}$ honestly.

4. The simulator computes the second prover message (agg_C, W) honestly as $\mathsf{agg}_C := \mathbf{C}^{\mathbf{r}}$ and $W := [z_1 r^{n-1}]_1 + \sum_{i=1}^{n-2} r^i a_0 W_0$ for $\mathbf{r} = (r, r^2, \ldots, r^n)$.

5. The simulator generates a transcript $\mathsf{tr}_{\mathsf{HMIPP}}$ for the HMIPP protocol by running $\mathsf{tr}_{\mathsf{HMIPP}} := \mathsf{Sim}_{\mathsf{HMIPP}}(\mathsf{crs}, \mathsf{com}_C, \mathsf{agg}_c, \mathbf{r})$

6. The simulator generates a transcript $\mathsf{tr}_{\mathsf{HWW}}$ for the HWW protocol by running $\mathsf{tr}_{\mathsf{HWW}} := \mathsf{Sim}_{\mathsf{HWW}}\left(\mathsf{com}_{a_0}, W, P_1, P_2, P_3, \sum_{i=1}^{n-2} r^i W_0, [r^{n-1}]_1\right)$

7. The simulator computes agg_{AB} honestly and generates a transcript $\mathsf{tr}_{\mathsf{TIPP}}$ by running $\mathsf{tr}_{\mathsf{TIPP}} := \mathsf{TIPP}(\mathsf{ck}, (\mathsf{com}_A, \mathsf{com}_B, \mathsf{agg}_{AB}, \mathbf{r}), \mathbf{A}, \mathbf{B})$.

We will show that the simulator's transcript is indistinguishable from an honest prover's. We look at the distribution of each of the proof components.

The MIPP optimization. We first note that this optimization is simulated perfectly, since it involves no witness values. Specifically, $\mathsf{com}_{\mathsf{in}}$, $\mathsf{agg}_{\mathsf{in}}$, and $\mathsf{tr}_{\mathsf{MIPP}\text{-}k}$ are simulated perfectly, since both the prover and simulator have access to the Groth16 public prepared inputs $\hat{\mathbf{S}}$.

The first prover message. We look at com_{a_0}, com_A, com_B, and com_C. In the real prover execution: com_{a_0} is distributed uniformly at random because it is randomized by z_3; com_A is distributed uniformly at random because it is randomized by z_2; com_B is distributed uniformly at random because it is randomized by B'_{n-2}; com_C is distributed uniformly at random because it is randomized by z_4. In the simulated execution: com_{a_0} is chosen uniformly at random; com_A is distributed uniformly at random because it is randomized by z_2; com_B is distributed uniformly at random because it is randomized by B'_{n-2}; com_C is chosen uniformly at random. Thus both the provers and the simulators first messages are distributed randomly and are indistinguishable.

The second prover message. We second look at agg_C, W. In the real prover execution: agg_C is distributed uniformly at random because it is randomized by C'_{n-2}; W is distributed uniformly at random because it is randomized by z_1. In the simulated execution: com_{a_0} is chosen uniformly at random; agg_C is distributed uniformly at random because it is randomized by C'_{n-2}; com_W is distributed uniformly at random because it is randomized by z_1. Thus both the provers and the simulators second messages are distributed randomly and are indistinguishable.

The hidden MSM argument. We see that $\mathsf{tr}_{\mathsf{HMIPP}}$ generated by the prover and simulator are indistinguishable by the zero-knowledge of HMIPP (Lemma 4).

The HWW argument. We see that $\mathsf{tr}_{\mathsf{HWW}}$ generated by the prover and simulator are indistinguishable by the perfect zero-knowledge of HWW (Lemma 6).

The TIPP argument. In order to argue that $\mathsf{tr}_{\mathsf{TIPP}}$ generated by the prover and simulator are indistinguishable we must look at the rerandomizations of each (A_i, B_i, C_i). The bulk of the following argument consists of demonstrating that enough values in the HICIAP protocol are independent and uniformly distributed. To do this, we associate each iid uniform blinding factor to at most one transcript variable. One thing to be careful about here is enforcing the "at most one" requirement.

Following [HKR19] we define a masking set $\mathbb{M}$ of size $O(\log_2 n)$ that defines a position of randomized values that will ensure the transcripts appear random in the recursion. We track the parts of the TIPP transcript which are functions of $\mathbf{A}, \mathbf{B}, \mathsf{ck}_1, \mathsf{ck}_2$ (where we let $\mathbf{A}$ represent $\mathbf{r} \odot \mathbf{A}$ and ck_1 represent $\mathbf{r}^{-1} \odot \mathsf{ck}_1$ for simplicity). In each round of the TIPP protocol (of which there are $\log n$), the prover sends six values:

$$\mathsf{com}_{LA} := \mathbf{A}_{[:h]} * \mathsf{ck}_1 \qquad \mathsf{com}_{RA} := \mathbf{A}_{[h:]} * \mathsf{ck}_1$$

$$\mathsf{com}_{LB} := \mathsf{ck}_2 * \mathbf{B}_{[:h]} \qquad \mathsf{com}_{RB} := \mathsf{ck}_2 * \mathbf{B}_{[h:]}$$

$$\mathsf{agg}_{LR} := \mathbf{A}_{[:h]} * \mathbf{B}_{[h:]} \qquad \mathsf{agg}_{RL} := \mathbf{A}_{[h:]} * \mathbf{B}_{[:h]}$$

The verifier sends a challenge x, which defines the prover's values for the next round:

$$\mathbf{A}' := \mathbf{A}_{[:h]} + x \cdot \mathbf{A}_{[h:]} \qquad \mathsf{ck}'_1 := \mathsf{ck}_{1,[:h]} + x^{-1} \cdot \mathsf{ck}_{1,[h:]}$$

$$\mathbf{B}' := \mathbf{B}_{[:h]} + x^{-1} \cdot \mathbf{B}_{[h:]} \qquad \mathsf{ck}'_2 := \mathsf{ck}_{2,[:h]} + x \cdot \mathsf{ck}_{2,[h:]}$$

Note that a randomized A_i value in round k will yield a uniform value of A_j in round $k+1$, where $j \equiv i \pmod{2^{k-1}}$, and similarly for B_i.

With 6 proof elements in each round, we need to ensure there are at least 6 randomizers per round,

and that one unique randomizer appears in each proof element. We divide them as 3 randomizers in $\mathbf{A}$ (to randomize $\mathsf{com}_{LA}, \mathsf{com}_{RA}, \mathsf{agg}_{LR}$) and 3 in $\mathbf{B}$ (to randomize $\mathsf{com}_{LB}, \mathsf{com}_{RB}, \mathsf{agg}_{RL}$). We define the masking set

$$\mathbb{M} = \{2^k, 2^k+1\}_{k=2}^{\ell-1} \cup \{2^k-1\}_{k=2}^{\ell-1},$$

where $\ell = \log_2(n)$. The two sets making up $\mathbb{M}$ are non-overlapping. Note that because $\log_2(n) \geq 4$ we have that $\mathbb{M}$ also does not overlap with the blinders B'_{n-2} or C'_{n-2}.

For the components agg_{LR} and agg_{RL} in the TIPP argument, we must use the fact that with overwhelming probability, none of the components of a Groth16 proof (A_i, B_i, C_i) equals 0. This implies that the rerandomization is a uniform proof of the same statement, and also contains no zeros.

With this in mind we argue that $\mathbb{M}$ is sufficient to randomize the distribution of the com_{LA}, com_{RA}, agg_{LR} components of TIPP. To see this, observe that in round k with verifier challenges $x_0, \ldots, x_{k-1}$,

$$\begin{aligned}
\mathsf{com}_{LA} &= \prod_{\mathbf{b}\in\{0,1\}^{\ell-k-1}} e\left(\sum_{\mathbf{s}\in\{0,1\}^k} A_{(\mathbf{s},0,\mathbf{b})} f_{k,\mathbf{x}}(\mathbf{s}), \sum_{\mathbf{s}\in\{0,1\}^k} \mathsf{ck}_{A,(\mathbf{s},1,\mathbf{b})} f_{k,\mathbf{x}^{-1}}(\mathbf{s}) \right) \\
\mathsf{com}_{RA} &= \sum_{\mathbf{b}\in\{0,1\}^{\ell-k-1}} e\left(\sum_{\mathbf{s}\in\{0,1\}^k} A_{(\mathbf{s},1,\mathbf{b})} f_{k,\mathbf{x}}(\mathbf{s}), \sum_{\mathbf{s}\in\{0,1\}^k} \mathsf{ck}_{A,(\mathbf{s},0,\mathbf{b})} f_{k,\mathbf{x}^{-1}}(\mathbf{s}) \right) \\
\mathsf{agg}_{LR} &= \prod_{\mathbf{b}\in\{0,1\}^{\ell-k-1}} e\left(\sum_{\mathbf{s}\in\{0,1\}^k} A_{(\mathbf{s},0,\mathbf{b})} f_{k,\mathbf{x}}(\mathbf{s}), \sum_{\mathbf{s}\in\{0,1\}^k} B_{(\mathbf{s},1,\mathbf{b})} f_{k,\mathbf{x}^{-1}}(\mathbf{s}) \right),
\end{aligned}$$

where $f_{k,\mathbf{x}}(\mathbf{s}) := \prod_{j=0}^{k-1} \left(s_j x_{k-j-1} + 1 - s_j \right)$ for $\mathbf{s} \in \{0,1\}^k$ and $A_{\mathbf{b}}$ represents A_i when $\mathbf{b}$ is the binary representation of the integer i. Thus

- $A_{\mathbf{0},0,1,\mathbf{0},1}$ is included in com_{LA} in the kth round and corresponds to the blinder $2^{\ell-k-2}+1$
- $A_{\mathbf{0},1,\mathbf{0}}$ is included in com_{RA} in the kth round and corresponds to the blinder $2^{\ell-k-1}$
- $A_{\mathbf{0},0,\mathbf{1}}$ is included in agg_{LR} in the kth round and corresponds to the blinder $2^{\ell-k-1}-1$

Denote

$$R_{LA,k,\boldsymbol{b}} := \sum_{\mathbf{s}\in\{0,1\}^k} \mathsf{ck}_{A,(\mathbf{s},1,\mathbf{b})} f_{k,\mathbf{x}^{-1}}(\mathbf{s}) \qquad R_{LR,k,\boldsymbol{b}} := \sum_{\mathbf{s}\in\{0,1\}^k} B_{\mathbf{s},1,\mathbf{b}} f_{k,\mathbf{x}^{-1}}(\mathbf{s})$$

$$R_{RA,k,\boldsymbol{b}} := \sum_{\mathbf{s}\in\{0,1\}^k} \mathsf{ck}_{A,(\mathbf{s},0,\mathbf{b})} f_{k,\mathbf{x}^{-1}}(\mathbf{s}).$$

Observe that with overwhelming probability, each of $\{R_{LR,k-1,(\mathbf{0},0,\mathbf{1})}\}_{k=1}^{\ell-1}$ are non-zero, depend non-trivially on x_{k-1}, and have no dependence on x_k Thus the $R_{LR,0,(0,\mathbf{0},1)},\ldots,R_{LR,\ell-2,(0,\mathbf{0},1)}$ are pairwise independent and ensure that: the $A_{0,0,\mathbf{1}}$ term in $\mathsf{agg}_{LR,2}$ (denoting the second round's agg_{LR}) is randomized by x_1 and thus is independent from the $A_{0,0,\mathbf{1}}$ terms in $\mathsf{agg}_{LR,1}$; the $A_{0,0,0,\mathbf{1}}$ term in $\mathsf{agg}_{LR,3}$ is randomized by x_2 and thus is independent from the $A_{0,0,0,\mathbf{1}}$ terms in $\mathsf{agg}_{LR,1}$ and $\mathsf{agg}_{LR,2}$; etc. Thus $A_{\mathbf{0},0,\mathbf{1}}$ perfectly blinds agg_{LR} except with negligible probability.

By the same argument, the $R_{LA,k,(\mathbf{0},0,1,\mathbf{0},1)}$ terms are pairwise independent and the $R_{LB,k,(\mathbf{0},1,\mathbf{0})}$ terms are pairwise independent ensuring independence between the $A_{\mathbf{0},0,1,\mathbf{0},1}$ terms in $\mathsf{com}_{LA,k}$ and $A_{\mathbf{0},1,\mathbf{0}}$ terms in $\mathsf{com}_{LB,k}$ respectively. Thus $A_{\mathbf{0},0,1,\mathbf{0},1}$, $A_{\mathbf{0},1,\mathbf{0}}$ terms perfectly blind com_{LA}, com_{RA} except with negligible probability.

By a symmetric argument we see that $B_{\mathbf{0},0,1,\mathbf{0},1}$, $B_{\mathbf{0},1,\mathbf{0}}$, $B_{\mathbf{0},0,\mathbf{1}}$ perfectly blinds com_{LB}, com_{RB}, agg_{RL} except with negligible probability.

We now consider the penultimate round (round $\ell-1$). Here the proof elements com_{LA}, com_{LB}, com_{RA} and com_{RB} of both the honest prover and simulator take the form

$$\mathsf{com}_{LA} = e(A_{\ell-1,1}, H_2) \qquad \mathsf{com}_{RA} = e(A_{\ell-1,2}, H_1),$$

$$\mathsf{com}_{LB} = e(G_1, B_{\ell-1,2}) \qquad \mathsf{com}_{RB} = e(G_2, A_{\ell-1,1})$$

for $\mathsf{ck}_{1,\ell-1} = (H_1, H_2)$ and $\mathsf{ck}_{2,\ell-1} = (G_1, G_2)$. Thus the proof elements

$$\mathsf{agg}_{LR} = e(A_{\ell-1,1}, B_{\ell-1,2}) \qquad \mathsf{agg}_{RL} = e(A_{\ell-1,2}, B_{\ell-1,1})$$

are uniquely determined given com_{LA}, com_{LB}, com_{RA} and com_{RB}. Hence they are sampled from the same distribution. □

Chapter 4

zk-creds: Flexible Anonymous Credentials from zkSNARKs and Existing Identity Infrastructure

Work coauthored with Jacob White, Christina Garman, and Ian Miers, and published in the proceedings of IEEE S&P 2023 [RWGM23].

1 Introduction

Privacy-preserving identification is an apparent contradiction in terms: one cannot both wish to simultaneously identify themselves and stay private. But this is increasingly necessary on today's internet. For example, Australia, the EU, and the UK age-restrict access to some video content, requiring identification via a credit card or photo of an official ID to access it [Goob, Gooa]. The tracking and data exposure risks raised by such requirements can be eliminated with privacy-preserving cryptography: anonymous credentials allow a user to assert that they meet some access criteria, e.g., are over 18, without revealing anything else about themselves, linking their viewing habits to their identity, or even linking distinct video views together. Beyond this narrow application, anonymous credentials could be extended to complex identity statements—for example, checking

residency for accessing online library resources or petitioning local elected representatives[1]—and the composition of credentials such as the pairing of a vaccine card with a photo ID.

While the subject of extensive academic work [Cha85, CL03, CL04, BCKL08, BL13, GGM14, CDHK15, SAB+19], anonymous credentials have thus far seen little deployment.[2] In large part, this is because most existing systems are designed with a number of assumptions about identity that, while suitable for advancing a body of cryptographic knowledge, produce designs that can be difficult to actually deploy in real-world identity systems.

Existing anonymous credential schemes make, at a minimum, some subset of the following simplifying assumptions: there is a single issuer for a given identity property (e.g., date of birth); when there are multiple issuers for a property, the property formats are compatible; there exist reputable authorities that are able and (more importantly) willing to be responsible for holding signing keys, verifying identity properties, and running sophisticated cryptographic protocols for issuing anonymous credentials; all attribute formats needed for a credential are known in advance; and the set of authorities for a given identity attribute or credential can be enumerated at the time one instantiates the system.

In this paper, we build `zk-creds`, a flexible, issuer-agnostic anonymous credential toolkit for complex identity statements. The general-purpose proving functionality supported by zkSNARKs gives us the flexibility to address most of these challenges. From this approach, we automatically gain support for ad-hoc composition of credentials and access criteria, and issuance by (threshold) signatures. In many cases, we can even remove the need for trusted anonymous credential issuers entirely by instantiating a *bulletin board* to track issued credentials, no longer requiring issuers to hold signing keys or other secrets. Concretely, this work contributes the design and implementation of a toolkit for flexible privacy-preserving credentials, and builds two example applications.

[1] New York State provides such a platform with no privacy guarantees [New].

[2] Notable exceptions for human uses of credentials are limited trials of CL-sigs with Idemix [CV02] and, although it is not a full-fledged anonymous credential scheme, Privacy Pass [DGS+18]. Intel also makes use of DAA [BCC04] for device attestation, and a MAC variant of anonymous credentials is also being used for private groups in Signal [CPZ20].

1.1 Past work and real-world limitations

Several approaches have tried to address the limitations of anonymous credentials, focusing primarily on the problems of finding and trusting issuers.

Distributing issuance via multiple issuers. To reduce the trust needed in issuers, schemes have explored threshold issuance [SAB+19] and support for multiple issuers [CL01]. While this improves the situation if there are multiple willing issuers, it does not address the potential scarcity of issuers who are willing or able to deploy novel (or any) cryptography. Nor does it provide a means to reconcile the differing identity document formats or use cases multiple issuers would have.

Decentralizing issuance by removing signing keys. In Decentralized Anonymous Credentials [GGM14], credentials are maintained in some form of transparency log which can either be centralized and audited, distributed across cooperating parties, or operated in a decentralized fashion by a Byzantine system or blockchain. While this approach removes one obstacle to credential issuance by avoiding signing keys, the concrete protocol has performance and operational limitations. For example, the protocol requires that all clients have the full list of issued credentials, and does not address any of the other complexities of real use cases.

The messy reality of identity claims. We now return to our initial example: an anonymous credential to allow access to age-restricted videos and prevent tracking of browsing habits. In theory, whichever authority issues identity cards in a country can also issue anonymous digital credentials to everyone of age. But in practice, a number of problems arise when attempting to deploy such a scheme with existing anonymous credentials.

First, there is not a single source of identity documents (e.g., the US has 50+ drivers license issuers) and few might wish to participate due to the burden of deploying new technology. Fewer still can be trusted to secure the requisite signing keys for issuing credentials.

Second, requirements will change. What started as a token for being over 18 will need to support other age checks—under 12, over 21, over 65—necessitating more complex credentials, access criteria, and potentially credential revocation and reissuance.

Third, each ID issuer will, by default, form its own anonymity set. Even for "multi-authority"

schemes designed to avoid this, differences in data fields can distinguish populations:[3] a foreign diplomat accessing age-restricted content in their host country may be distinguished from a resident using a local ID.

Fourth, new identity documents need to be integrated as they emerge to avoid access equity issues, and these documents may have differing formats. For example, many cities now issue IDs in part for undocumented residents [IDN].

Finally, even for something as conceptually simple as "of age," identity statements are not necessarily simple: in the event age limits differ between jurisdictions, a video platform needs to check where the viewer is located, and IP geolocation may be insufficient (e.g, in the case of Tor or a VPN). Credentials can directly encode a home address but, even for physical credentials, this does not work in practice: people move and do not update their IDs, and as a result need to provide alternative proofs of address. Supporting this privately requires composing credentials for, e.g., age and residency.

Minimizing trust when issuing credentials. Current (even non)-anonymous credential protocols assume the same party verifies claimed identity properties and signs cryptographic credentials. This requires finding a single party who is trusted for two different (and not necessarily related) tasks: one who is both capable of verifying identity attributes and competent to manage signing keys. The linking of these two roles is often unnecessary and complicates deployment.

First, many uses of anonymous credentials do not use identity attributes which must be verified by a trusted party to issue a credential. Looking ahead, we describe an issuer-less Privacy Pass-like construction [DGS+18] where Sybil-resistant anonymous tokens are issued by making a blockchain payment. This has no trusted parties—neither for verifying that the user is not a Sybil, nor for signing a credential.

Second, even when we must trust some entity to verify identity attributes (e.g., a passport issuer for a user's date of birth), it is not necessary to trust an additional party to hold key material for a novel cryptographic scheme. Looking ahead, we offer the minimal trust assumptions in many

[3]Consider something as simple as date formats: Japanese Drivers Licenses give birth year relative to eras that correspond to the reign of the emperor. However, they use the Gregorian calendar for months and days.

such cases by replacing signing with a list maintained by a centralized party or a distributed bulletin board.

Third, even where there is a trusted party who both verifies identity attributes and issues credentials, trusting a party to maintain a list is safer than trusting them to secure signing keys. In existing anonymous credential schemes, compromise of issuing keys is frequently undetectable and rollback requires rekeying and reissuing. With issuance via a list, compromise is detectable and easily reversible. This is true even if the list is maintained by a single party.

All of the aforementioned issues can be addressed by a scheme that is flexible, dynamically adaptable to new use cases post-deployment, minimizes the need to find new trusted parties, and can support complex access criteria that are agnostic to the issuer or credential format.

1.2 Our contribution

We introduce `zk-creds`, a toolkit for privacy-preserving authentication protocols and anonymous credentials that offers flexible identity assertions and does not need trusted issuers. A key contribution of `zk-creds` over previous works is the usage of general-purpose zero-knowledge proofs rather than bespoke proof systems over blind signatures.

The switch to general-purpose zero-knowledge proofs as the basis for anonymous credentials, instead of blind signatures, is a paradigm shift: rather than imagining a subset of use cases and designing custom protocols for each while balancing cryptographic tradeoffs, `zk-creds` gets full privacy and full expressivity even after a protocol is designed and deployed. A single scheme, built with `zk-creds`, is adaptable to shifting requirements without requiring the development of new custom cryptographic protocols. Moreover, application-specific logic can be defined and modified in simple programming languages via a number of publicly available tools [MTBI+22, Ar21, Bow17a] with the instantiation of the scheme handled by the compiler.

General-purpose zero-knowledge proofs enable `zk-creds` to support flexible and composable access criteria. `zk-creds` not only allows users to privately show that their credential(s) meet some arbitrary access criteria check, but it also allows these criteria to be defined at any time (even

after system setup or credential issuance), by any party, and composed dynamically as *gadgets*. This flexibility allows `zk-creds` to meet the reality of real-world authentication mechanisms: requirements can dynamically change at any time, as can use cases and even identity issuers.

The second major contribution of `zk-creds` is its support of existing government identity infrastructure without modification or collaboration. Using general-purpose zero-knowledge proofs, we can convert the digital (non-anonymous) identity information that is increasingly included in national identity cards and passports into anonymous credentials. This *zk-supporting-documentation* allows us to provide a digital analogue of walking into a US Department of Motor Vehicles and presenting existing identity documents to get a driver's license, *without* exposing any information to the issuer. We implement an end-to-end example of this paradigm, using a zero-knowledge proof over the data in unmodified NFC-enabled US passports to create credentials for accessing age-restricted videos.

As a third contribution, `zk-creds` supports publicly verifiable credentials. Because the issued credential list can be maintained by a public system (i.e., a transparency log or blockchain) and each credential can include zk-supporting-documentation justifying its issuance, the set of all issued credentials is publicly auditable. This is not possible when credentials can be surreptitiously issued via signing keys.[4] As such, we need only trust the issuer to add credentials (and their supporting documentation) to the list, and any compromise or malfeasance is detectable and reversible.

To summarize, in this paper we design, build, and benchmark `zk-creds` which:

- drastically improves performance over existing decentralized schemes via reusable proofs where ShowCred takes < 150ms;
- supports existing physical identity documents (e.g., passports) without modification via zk-supporting-documentation;
- provides support for flexible and composable gadgets that can be combined to express complex access criteria checks even after system setup;
- allows for public auditability of issued credentials, without harming anonymity;

[4]We note, however, that while the credentials are fully auditable, identity statements require inherent trust in something. If the identity infrastructure, e.g., US passports, is not trusted, then we can make no guarantees.

- provides (of independent interest), blind Groth16, a novel mechanism for privately linking together multiple zero-knowledge proofs in a way that enables proof rerandomization and reuse; and
- includes a full application for age-restricted video access with cloning resistance, using existing passports for issuance.

Non-goals: On-chain verification of credentials. This work constructs flexible credentials that can be *issued* without a central party holding a signing key (although we also support signature-based issuance). This should not be confused with a different area of both industrial and academic work (see e.g., [RPX+22]), which considers *verification* of existing (i.e., centrally issued) anonymous credentials by a smart contract. The question for on-chain verification of anonymous credentials is not how to remove centralized issuance, but simply how to minimize the cost of verification given the extreme cost of smart contract execution. Reducing verification costs is typically done by batching verification inside a zero-knowledge proof[5] and is generically applicable to any anonymous credential scheme, including the ones proposed here.

2 Overview

`zk-creds` is a system for issuing credentials to users and privately showing that a credential meets access criteria.

2.1 Example application and credential lifecycle

A *credential* is a commitment to a set of attributes (e.g., name, date of birth, etc.). A credential is *issued* (see Figure 4.1) when it is made a leaf in a Merkle tree. We call the set of leaves the *issuance list*. Optionally, protocol designers can require *zero-knowledge supporting-documentation* that the attributes match some (existing) document without revealing any additional information. In our implemented example (see Section 7), this consists of a zero-knowledge proof that the attributes in

[5]This is, in essence, a specialized zk-rollup [But].

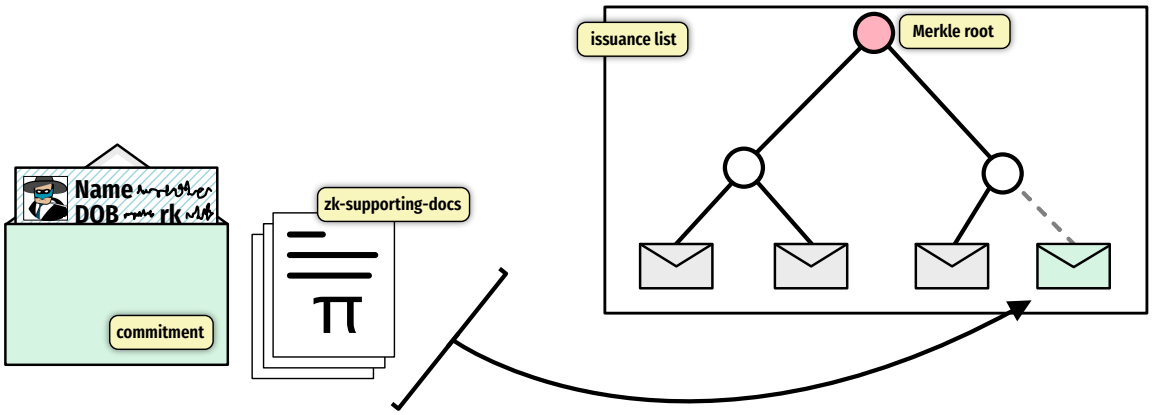


Figure 4.1: A credential in `zk-creds` is issued by adding it to a Merkle tree after (optionally) presenting zk-supporting-documentation to justify issuance.

the credential match a US passport.

We emphasize that, while our example trusts an existing identity document issuer—the passport authority—it requires no additional trusted parties, no existing parties to take on additional trusted roles, or even modification of the issuer. With standard anonymous credentials, we would need to also trust the security of credential signing keys, likely held by an additional party. In our example, we need only some way of maintaining a list of issued credentials.

Clients, once issued a credential, *show* a credential to gain access to some resource, as shown in Figure 4.2. The client presents a non-interactive zero-knowledge proof that: 1) they have a credential (a commitment) in the Merkle tree of issued credentials and 2) the attributes meet some *access criteria*.

Crucially, the zero-knowledge proof hides which credential is used, the credential's attributes, and the details of how the access criteria were met. In our example, the client uses a credential containing their birth date to show they are over 18 and gain access to a website. The verifier learns only that the client is over 18, not their identity, which credential they used, their exact birth date, or any of the other attributes in their credential.

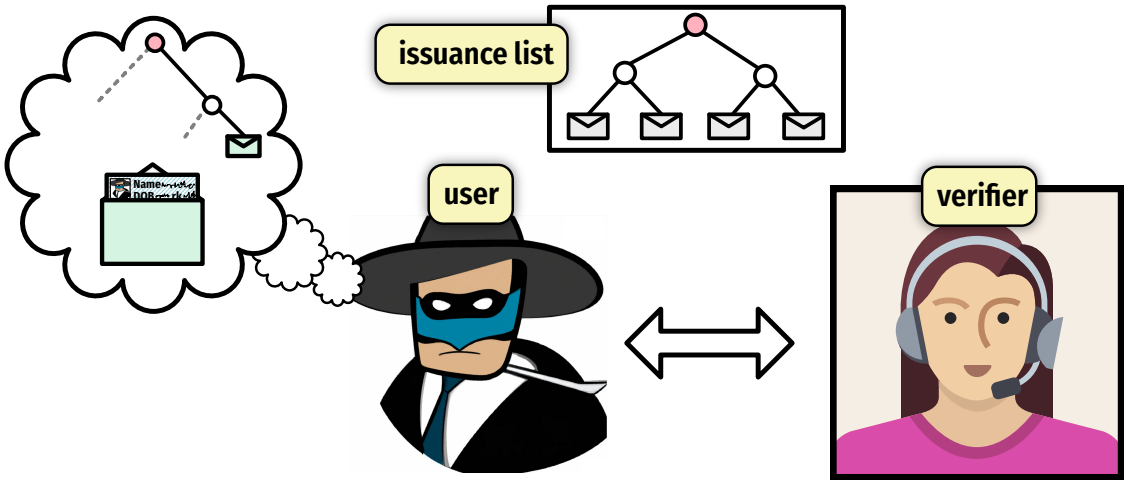


Figure 4.2: To show a credential in `zk-creds`, the prover uses knowledge of their credential opening and the position of the credential in the issuance list to construct a zero-knowledge proof. The verifier need only know the issuance list root.

2.2 Design features

As illustrated by our example application, our approach to building anonymous credentials has a number of important features.

Flexible access criteria. Application developers can define arbitrary access criteria at any point in the lifetime of the system. We support common features from the anonymous credential literature, such as hidden-attribute credentials, inequality or expiry checks, rate limiting, and cloning resistance where violating the rate limit (e.g., by sharing a credential with others) results in the credential's identification and revocation. And because we support the efficient encoding of access criteria as an NP relation, we can easily support more complex criteria than existing schemes, such as a proof of residency in a given municipality to access ebooks from a local public library.

Auditable issuance. Credential issuance can be publicly auditable as well: e.g., in our passport example anyone can download the list of credentials and, with zk-supporting-documentation, verify both that a credential was issued and why. Even without such documentation, all issued credentials are visible and issuance can be investigated. In contrast, it is impossible to enumerate, let alone audit, every credential signed with a given key.

Flexible credential management. Because credential issuance is simply a matter of list manage-

ment, credential issuance is flexible. In many cases, we need not find a trusted party at all: a simple bulletin board is sufficient, as is a blockchain. In other cases, a central party can maintain the list without needing to be trusted to secure signing keys.

Signature-issued credentials. Separately, in cases where there is a party who is trusted to issue correct credentials without public auditability and is trusted, willing, and able to secure signing keys, our implementation of `zk-creds` also supports issuing credentials via signatures. In this case, there is no issuance list. When feasible, this leads to faster credential shows and removes the overhead of managing a witness to list membership.

Witness management. Using Merkle trees for credential issuance requires the user to maintain an up-to-date witness to the credential's membership in an issuer's list. Periodically, the user can ask the issuer for an updated witness. Looking ahead, this also lets any user update their witness by downloading logarithmic-sized updates from the tree's *frontier* and a constant number of Merkle roots (see Section 13).

Revocation. Many existing approaches require expensive asymmetric cryptographic operations for each revocation. Some schemes, like EPID [BL09], require each credential show to perform work linear in the number of revoked private keys. Other schemes use, e.g., RSA accumulators, which require recomputing accumulator witnesses per revocation at cost linear in the number of revocations. And, while more efficient accumulators exist [BBF19], such techniques have not been used for revocation, to the best of our knowledge. In contrast, each revocation in `zk-creds` only requires removing the credential from its Merkle tree, incurring only logarithmic costs in the number of issued credentials. This captures revocation of the credential where the holder's public identity is known, and so called *private key revocation* where a stolen or leaked credential is banned.

3 Preliminaries

3.1 Merkle trees

In `zk-creds` we use Merkle trees T to represent set membership. The root of a tree T is denoted T_{root}. A Merkle *forest* F is a set of Merkle tree roots. Merkle trees have the following functionality:

$T.\mathsf{Insert}(v) \to T'$ Inserts the value v into the next free leaf in T and returns the modified tree.

$T.\mathsf{Remove}(v) \to T'$ Removes v from the tree (if present) and returns the modified tree.

$T.\mathsf{AuthPath}(v) \to \theta$ Creates an *authentication path* θ that proves that $v \in T$. The size of θ is proportional to the height of the tree.

3.2 Groth16 Linkage

We describe a high-level interface that allows us to construct a (blinded) *linkage* proof over Groth16 proofs. This allows one to show that a hidden collection of Groth16 proofs $\pi_1, \ldots, \pi_k$ all share some subset of *hidden common inputs* $\mathbf{x}^*$, not known to the verifier. Concretely, this proof system proves that

$$\bigwedge_{i=1}^{k} \mathsf{G16.Verify}(\mathsf{crs}_i, \pi_i, (\mathbf{x}^*, \mathbf{x}_i))$$

where $\mathbf{x}_i$ are the non-hidden public inputs (i.e., public inputs known to the verifier). See Section 15 for the full description and security proofs of LinkG16. For `zk-creds`, however, it suffices to specify the functionality:

$\mathsf{LinkG16.Link}(\mathbf{x}^*, \{\mathsf{crs}_i, \pi_i\}_{i=1}^k) \to \pi_{\mathsf{link}}$ Constructs a zero-knowledge proof of the above relation, with respect to hidden common inputs $\mathbf{x}^*$.

$\mathsf{LinkG16.LinkVerify}(\pi_{\mathsf{link}}, \{\mathsf{crs}_i, \mathbf{x}_i\}_{i=1}^k) \to \{0,1\}$ Verifies the above statement with respect to the given public inputs and Groth16 CRSs.

3.3 Cryptographic assumptions

We state the security properties of the above schemes and the cryptographic assumptions necessary to achieve them. For brevity, we defer the definitions of the specific assumptions to the cited references.

Groth16 is perfectly zero-knowledge and weak white-box simulation-extractable against algebraic adversaries under $q-\mathsf{dlog}$ and a linear independence assumption [BKSV21]. We also assume this result holds under the common Groth16 substitution $\gamma = 1$. We use Poseidon [GKK+19] to instantiate a hash for Merkle trees, as well as commitments. Finally, we assume that the key-prefixed Poseidon hash function, used to instantiate the gadgets in Section 5.3, is a PRF. As Poseidon is a sponge construction, prefixing is secure. Separately, see Section 17 for an alternate instantiation using Pedersen hashes and perfectly hiding commitments.

4 Definitions

4.1 Security definitions

Security definitions are given by an ideal functionality in Figure 5.3, corresponding to the usual security properties of anonymous credentials: unforgeability, correctness and unlinkability. It also implies an additional security property, session binding: shows of a credential are inherently bound to the channel or session in which they are presented, thus preventing replay attacks.

Threat model. The ideal functionality corresponds to the following general threat model. We assume all issuers, verifiers, and (almost all) users are malicious and can collude. We inherit the standard requirement that, for anonymity, there must be at least two honest users with valid issued credentials. We also assume that there exists a reliable mechanism for parties to agree on the list of issued credentials.

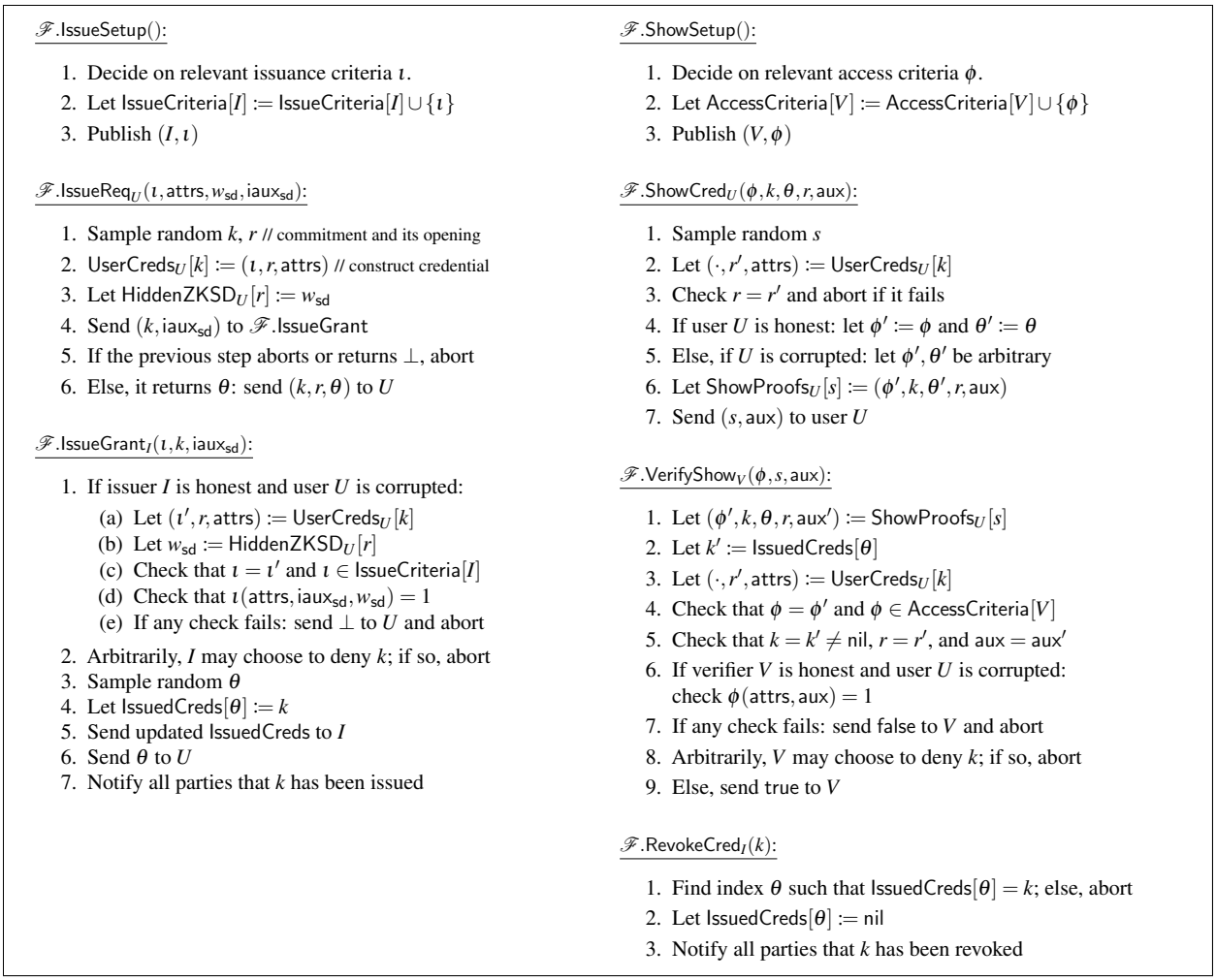


Figure 4.3: An ideal functionality $\mathscr{F}$ for `zk-creds`.

4.2 Anonymous credentials

We give a generic overview of the data structures and algorithms which our scheme instantiates.

Let a *credential* be the commitment $k := \mathsf{Com}(\mathsf{nk}, \mathsf{rk}, \mathsf{attrs}; r)$ where nk is the *pseudonym key*, a private random value used to generate persistent pseudonyms; rk is the *rate key*, a private random value used to generate rate-limit tokens; $\mathsf{attrs} \in \mathscr{A}$ is an arbitrary set of public and hidden *attributes*; and r is the commitment randomness. Note that the values within the credential remain private by the hiding property of commitment schemes, with the exception of attribute information revealed by the user.

We say that an issuer I *issues* a credential if it appears on I's credential list CL. Looking ahead, while the credential list may be instantiated in many ways (e.g. an accumulator, or Merkle tree), we later instantiate this as a Merkle forest and a list of corresponding Merkle trees $\mathsf{CL} := (F, \overline{T})$, with an *authentication path* θ providing membership attestation. Every issuer has some *issuance criteria* ι that the requester must meet in order to have their k issued, e.g., that the birth date in k matches a signed digital passport. Over an issuer-authenticated channel, the requester (running $\mathsf{IssueReq}$) sends k to the issuer with some *zk-supporting-documentation* sd, e.g., a Groth16 proof or a digital signature, that convinces the issuer of the criteria. The issuer runs $\mathsf{IssueGrant}$ and, upon success, adds k to their list and returns an authentication path θ attesting to its issuance.

Next, let the list of all possible *access criteria* be $\Phi := \{\phi \mid \phi : \mathscr{A} \to \{0,1\}\}$ which can be defined dynamically by users, verifiers, or even third-parties for an application using `zk-creds`, even after system instantiation. Over an anonymous channel, a user *shows* a credential (running $\mathsf{ShowCred}$) by presenting a zero-knowledge proof that they have a valid issued credential (with θ as a private witness) whose attributes in witness w satisfy the verifier's access criteria. The proof must also be bound to some session context, aux. The verifier runs $\mathsf{VerifyShow}$ and, upon success, grants access to the user.

An anonymous credential system with zk-supporting-documentation can then be defined by the following algorithms (where the subscript U, I, or V denotes that the user, issuer, or verifier runs the algorithm, respectively):

$\mathsf{Setup}(1^{1^n}) \to \mathsf{pp}$ Generates the system parameters.

$\mathsf{IssueSetup}(\mathsf{pp}) \to \iota$ Establishes the *public* attribute fields and issuance criterion ι for obtaining a credential.

$\mathsf{ShowSetup}(\mathsf{pp}) \to \phi$ Establishes a new access criterion $\phi \in \Phi$ for showing a credential.

$\mathsf{IssueReq}_U(\mathsf{pp}, \iota, \mathsf{attrs}, w_{\mathsf{sd}}, \mathsf{iaux}_{\mathsf{sd}}) \to (k, \mathsf{sd})$ Creates and requests a credential k with zk-supporting-documentation sd under issuance criteria ι.

$\mathsf{IssueGrant}_I(\mathsf{pp}, \iota, \mathsf{CL}, k, \mathsf{sd}) \to (\mathsf{CL}', \theta)$ Decides whether to grant a user the requested credential k; if so, adds it to the list CL and returns its issuance attestation θ.

$\mathsf{ShowCred}_U(\mathsf{pp}, \phi, \mathsf{CL}, k, \theta, w, \mathsf{aux}) \to (\pi_{\mathsf{link}}, \mathsf{aux})$ Shows that an issued k satisfies access criteria ϕ.

$\mathsf{VerifyShow}_V(\mathsf{pp}, \phi, \mathsf{CL}, \pi_{\mathsf{link}}, \mathsf{aux}) \to b$ Validates a credential show, including the current session context in aux.

$\mathsf{RevokeCred}_I(\mathsf{pp}, \mathsf{CL}, k) \to \mathsf{CL}'$ Revokes a credential.

5 Construction

$\underline{\mathsf{Setup}(1^{1^n}, h, n)}$:

1. Choose bilinear group bg w/ large prime order p
2. Let desc_T be a circuit w/ public inputs (T_{root}, k), where T is a Merkle tree of height h. It asserts that there is an auth path θ attesting to $k \in T$
3. Let desc_F be a circuit w/ public inputs $(T_{\mathsf{root}}, k, F)$, where F is a forest of n Merkle trees. It asserts that the tree is in the forest: $(T_{\mathsf{root}} = F_1) \vee \ldots \vee (T_{\mathsf{root}} = F_n)$
4. Compute $\mathsf{crs}_T \coloneqq \mathsf{G16.Setup}(\mathsf{bg}, \mathsf{desc}_T)$
5. Compute $\mathsf{crs}_F \coloneqq \mathsf{G16.Setup}(\mathsf{bg}, \mathsf{desc}_F)$
6. Let $\mathsf{pp} \coloneqq (\mathsf{bg}, \mathsf{crs}_T, \mathsf{crs}_F)$
7. Return pp

$\underline{\mathsf{IssueSetup}(\mathsf{pp})}$:

1. Decide on issuance criteria $(\iota_{\mathsf{zk}}, \iota_{\mathsf{pub}})$ for supporting docs
2. Let desc be a circuit w/ public input $\mathsf{iaux}_{\mathsf{zk}}$ asserting: $\iota_{\mathsf{zk}}(\mathsf{attrs}, \mathsf{iaux}_{\mathsf{zk}}) = 1 \wedge k$ opens to $(\mathsf{nk}, \mathsf{rk}, \mathsf{attrs})$
3. Compute $\mathsf{crs}_\iota \coloneqq \mathsf{G16.Setup}(\mathsf{bg}, \mathsf{desc})$
4. Return $(\mathsf{crs}_\iota, \iota_{\mathsf{pub}})$

$\underline{\mathsf{ShowSetup}(\mathsf{pp})}$:

1. Decide on access criteria ϕ for satisfying attributes
2. Let desc be a circuit w/ public inputs $(T_{\mathsf{root}}, k, \mathsf{aux})$ which asserts: $\phi(\mathsf{nk}, \mathsf{rk}, \mathsf{attrs}, \mathsf{aux}) = 1 \wedge k$ opens to $(\mathsf{nk}, \mathsf{rk}, \mathsf{attrs})$
3. Compute $\mathsf{crs}_\phi \coloneqq \mathsf{G16.Setup}(\mathsf{bg}, \mathsf{desc})$
4. Return crs_ϕ

$\underline{\mathsf{IssueReq}(\mathsf{pp}, \mathsf{crs}_\iota, \mathsf{attrs}, w_{\mathsf{sd}}, (\mathsf{iaux}_{\mathsf{zk}}, \mathsf{iaux}_{\mathsf{pub}}))}$:

1. Sample r, nk, rk // commitment nonce, pseudonym key, rate key
2. Commit $k \coloneqq \mathsf{Com}(\mathsf{nk}, \mathsf{rk}, \mathsf{attrs}; r)$
3. Let $w \coloneqq (w_{\mathsf{sd}}, r, \mathsf{nk}, \mathsf{rk}, \mathsf{attrs})$ // collect the witnesses
4. Prove $\pi_\iota \coloneqq \mathsf{G16.Prove}(\mathsf{crs}_\iota, (k, \mathsf{iaux}_{\mathsf{zk}}), w)$
5. Let $\mathsf{sd}_{\mathsf{zk}} \coloneqq (\pi_\iota, \mathsf{iaux}_{\mathsf{zk}})$ // collect the supporting docs
6. Let $\mathsf{sd}_{\mathsf{pub}} \coloneqq \mathsf{iaux}_{\mathsf{pub}}$
7. Send $(k, \mathsf{sd}_{\mathsf{zk}}, \mathsf{sd}_{\mathsf{pub}})$ to issuer
8. Receive the root of the modified credential tree T' and a Merkle auth path θ attesting to $k \in T'$
9. Store $(\mathsf{nk}, \mathsf{rk}, r, \mathsf{attrs}, \theta)$

$\underline{\mathsf{IssueGrant}(\mathsf{pp}, (\mathsf{crs}_\iota, \iota_{\mathsf{pub}}), (F, \overline{T}), k, (\mathsf{sd}_{\mathsf{zk}}, \mathsf{sd}_{\mathsf{pub}}))}$:

1. Parse $(\pi_\iota, \mathsf{iaux}_{\mathsf{zk}}) \coloneqq \mathsf{sd}_{\mathsf{zk}}$
2. Check $\iota_{\mathsf{pub}}(\mathsf{sd}_{\mathsf{pub}})$ // check public supporting docs
3. Check $\mathsf{G16.Verify}(\mathsf{crs}_\iota, \pi_\iota, (k, \mathsf{iaux}_{\mathsf{zk}}))$ // check ZK SD
4. If either check fails, reject issuance and abort
5. Else, choose T from forest F and let $T' \coloneqq T.\mathsf{Insert}(k)$
6. Let $\theta \coloneqq T'.\mathsf{AuthPath}(k)$ // θ attests to $k \in T'$
7. Store T' and update the forest F
8. Send θ to user U

$\underline{\mathsf{ShowCred}(\mathsf{pp}, \overline{\mathsf{crs}}, (F, T_{\mathsf{root}}), k, \theta, \{w_i, \mathsf{aux}_i\}_{i=1}^k)}$:

1. Parse $(\{\mathsf{crs}_{\phi_i}\}_{i=1}^k, \mathsf{crs}_T, \mathsf{crs}_F) \coloneqq \overline{\mathsf{crs}}$ and k's $T_{\mathsf{root}} \in F$
2. For all $i = 1, \ldots, k$, compute the access criteria proof: $\pi_{\phi_i} \coloneqq \mathsf{G16.Prove}(\mathsf{crs}_{\phi_i}, (T_{\mathsf{root}}, k, \mathsf{aux}_i), w_i)$
3. Prove $\pi_T \coloneqq \mathsf{G16.Prove}(\mathsf{crs}_T, (T_{\mathsf{root}}, k), \theta)$ // tree
4. Prove $\pi_F \coloneqq \mathsf{G16.Prove}(\mathsf{crs}_F, (T_{\mathsf{root}}, k, F), \mathsf{nil})$ // forest
5. Let $\overline{\pi} \coloneqq (\{\pi_{\phi_i}\}_{i=1}^k, \pi_T, \pi_F)$ // collect the proofs
6. Prove $\pi_{\mathsf{link}} \coloneqq \mathsf{LinkG16.Link}((T_{\mathsf{root}}, k), \{\mathsf{crs}_i, \pi_i\}_{i=1}^{k+2})$
7. Send $(\pi_{\mathsf{link}}, \{\mathsf{aux}_i\}_{i=1}^k)$ to verifier V

$\underline{\mathsf{VerifyShow}(\mathsf{pp}, \overline{\mathsf{crs}}, F, \pi_{\mathsf{link}}, \{\mathsf{aux}_i\}_{i=1}^k)}$:

1. Parse $(\{\mathsf{crs}_{\phi_i}\}_{i=1}^k, \mathsf{crs}_T, \mathsf{crs}_F) \coloneqq \overline{\mathsf{crs}}$
2. Let $(\mathsf{aux}_{k+1}, \mathsf{aux}_{k+2}) \coloneqq (\mathsf{nil}, F)$ // collect auxiliary inputs
3. Check $\mathsf{LinkG16.LinkVerify}(\pi_{\mathsf{link}}, \{\mathsf{crs}_i, \mathsf{aux}_i\}_{i=1}^{k+2})$
4. Upon success, accept. Else, reject

$\underline{\mathsf{RevokeCred}(\mathsf{pp}, (F, \overline{T}), k)}$:

1. Find $k \in T$ within forest F; if k not found, abort
2. Let $T' \coloneqq T.\mathsf{Remove}(\mathsf{k})$
3. Store T' and update the forest F
4. Return T'

Figure 4.4: `zk-creds` Construction. NB: Although the inputs (T_{root}, k) are public in all Groth16 proofs in ShowCred, they are hidden from the verifier by LinkG16. Also note that any necessary updates to the auth path θ or credential list $(F, \overline{T})$ are handled out-of-band.

`zk-creds` assumes there is a list of issued credentials maintained by either a trusted party, some Byzantine system, or a blockchain. Our scheme provides three sets of functionalities: issue, show, and revoke. Through the issuance process, a user, Alice, convinces the issuance mechanism she should be given a credential. Once her credential is put on the issuance list, Alice can then use the credential to show she meets some access criteria. Conceptually, using a credential involves two steps: (1) a membership proof that the credential is on the issuance list, and (2) a proof that the committed attributes meet some access criteria. Finally, an issuer is able to revoke a credential if need be by simply removing it from the list.

We can realize this paradigm in different ways and using different set-membership techniques such as an RSA accumulator, purpose built zero-knowledge schemes [ZBK+22], or even using signatures of issuance.

In our construction of `zk-creds`, we realize membership proofs using Merkle forests, a new approach that allows developers to trade a slight increase in verification time and witness data for a large reduction in proving time. For access criteria checks, we provide developers with a set of *gadgets*. Gadgets can be composed to form complex access criteria checks. Finally, we tie these components together with a new blind Groth16 proof, of potentially independent interest, that lets us show multiple Groth16 proofs shared the same blinded input without—as in commit-and-prove—creating a persistent identifier. This allows us to reuse the membership proof across multiple credential shows without the reused proofs being tracked.

We now give details on our specific instantiation and describe the full construction in Figure 4.4.

5.1 Merkle forests

Rather than using a single Merkle tree to accumulate credentials and prove membership, `zk-creds` uses a forest of Merkle trees. The membership proof attests to two parts: $k \in T$ for some Merkle tree T, and $T_{\mathsf{root}} \in F$ where F is the forest of Merkle trees containing issued credentials. Compared to a single Merkle tree, the Merkle forest approach gives us a tunable tradeoff between proving time and verification time. Shorter Merkle trees can drastically reduce proving costs—up to 50%, or

143ms (see Section 13). Furthermore, since forest membership is a simple OR-proof over Merkle tree roots, the cost of a larger forest is negligible to the prover and allows for a much larger list. Also, we note that, for the size of the forests we consider, the additional verification cost is trivial (137μs).

Witness management. Showing a credential in `zk-creds` requires knowing the witness (a.k.a., authentication path) θ attesting to issuance in its Merkle tree. θ must be updated as credentials are added or removed from the tree. In a naïve construction, a user might download newly-issued credentials to update the tree. However, this requires the user to construct and maintain a local copy of their entire Merkle tree, which is often impractical. On the other extreme, users could periodically query an issuer or list manager to provide the updated θ. However, this uniquely identifies the credential and strongly correlates with subsequent shows, posing a large privacy risk (especially in low-use deployments). We introduce better constructions leveraging Merkle forests in Section 13.

5.2 Blind Groth16

The membership proof is the most costly part of ShowCred. Looking ahead, it takes 460ms to complete. While the access criteria check must be redone for every show in many cases—for example, rate-limited shows include a token that uniquely identifies reuse—the membership proof does not change unless more credentials are issued.

We use a blind Groth16 linkage proof to combine a membership proof with (perhaps multiple) access proofs. Blind Groth16 lets us reuse an already computed membership proof in multiple shows without breaking privacy. Furthermore, it expands the functionality of the system by supporting the easy composition of access criteria: without this ability to compose access criteria, system designers would need to either: 1) dynamically generate circuit parameters for gadgets as they are needed, 2) determine in advance all the gadgets they will support, or 3) generate the circuit parameters for every combination of gadgets that could be used.

Concretely, blind Groth16 lets us prove that a number of Groth16 proofs are all made with respect to the same credential *without* revealing the credential. At a high level, the algorithm works

as follows. First, it prepares the public inputs (here, k and its Merkle root) shared by the underlying proofs. For each proof, it then blinds a copy of the prepared input, and blinds the proof in a way that cancels with the blinded input. Finally, it proves that all the blinded inputs are consistent with each other. After canceling the blinding factors, the verification equation is identical to the typical Groth16 verification equation. For more detail, see the description of LinkG16 in Section 15.

5.3 Gadgets

Since ShowCred supports arbitrary statements, verifiers have the flexibility to add and remove helpful subcircuits, or *gadgets*, from their protocol. In fact, rather than embedding gadgets in existing circuits, verifiers can make use of the structure of ShowCred to create a separate proof for each gadget and link them together. The benefit to this kind of customization is twofold: users can precompute and cache standalone gadget proofs separately from other access criteria proofs, and verifiers are freed from having to define custom circuits and generate the CRSs.

Gadgets are arbitrary NP relations which can capture nearly any conceivable identity check. We now describe some gadgets that serve as building blocks for `zk-creds`-based systems. Recall that rk denotes the rate key, used for generating rate-limit tokens, and nk denotes the pseudonym key, used for deriving uniform but linkable tokens.

Linkable show Reveals a pseudonym $\mathsf{Prf}_{\mathsf{nk}}(\mathsf{ctx})$ that persists across interactions in a given context ctx, but is unlinkable to any use of the credential in other contexts. For example, a single Sybil-resistant credential could be used for creating unlinkable accounts across sub-forums within a single site, such as Discord servers or subreddits.

Rate limiting Limits users to performing ShowCred only N times per epoch, for some verifier-chosen rate limit N. Every ShowCred, the user produces a pseudorandom token $\mathsf{tok} = \mathsf{Prf}_{\mathsf{rk}}(\mathsf{epoch} \| \mathsf{ctr})$, reveals epoch, and proves that ctr is less than N.

Cloning resistance Performs the same function as rate limiting, but deanonymizes rate violators. The technique was introduced by Camenisch et al. [CHK+06] (Section 5.2). Every run of

ShowCred, the user receives a nonce from the verifier and sends two tokens:

$$\mathsf{tok}_1 = \mathsf{Prf}_{\mathsf{rk}}(\mathsf{epoch} \| \mathsf{ctr})$$

$$\mathsf{tok}_2 = \mathsf{id} + H(\mathsf{nonce}) \cdot \mathsf{Prf}'_{\mathsf{rk}}(\mathsf{epoch} \| \mathsf{ctr})$$

where id is an identifying attribute (e.g., credential hash). As above, ShowCred proves the tokens are constructed correctly. If one of these shows is reused, tok_1 will be repeated, but tok_2 will be distinct, giving the verifier two instances of the tok_2 equation and two unknown variables: id and $\mathsf{Prf}'_{\mathsf{rk}}(\mathsf{epoch} \| \mathsf{ctr})$. Solving the equation for id identifies the credential holder. Note that if id is the credential (or its hash), then the cloned credential can be revoked immediately by removing it from the issuance list.

Expiry Opens an attribute e in the credential and proves that $e > \mathsf{today}$, i.e., that the credential is not yet expired.

Session binding Gives the verifier the ability to reject replayed ShowCred proofs by binding a verifier-chosen nonce, or *session context*, to every ShowCred. This can be done by including an empty proof that takes the nonce as the public input[6].

Join Allows credentials to be composed by *joining* them along some common attribute(s), such as full name or address. This is either done inside a single ShowCred, or between separate ShowCreds by publicly committing to the common attribute(s).

5.4 Signature-issued credentials

One of the major advantages of `zk-creds` is that we do not need issuers who are trusted to hold signing keys and can instead use a transparency log or blockchain to issue credentials. However, if this feature is not needed, we can build a traditionally issued anonymous credential scheme that retains `zk-creds`' other features and flexibility. Because our gadget-based approach is flexible, we

[6]This binds the nonce to the Groth16 proof, assuming some basic properties about the circuit [BKSV21].

can replace the membership-check gadget with one that instead verifies a signature. This means zk-creds, like Coconut [SAB+19], also supports credentials that are issued by signing under a standard signature (e.g., ECDSA or Schnorr) either by a single party or by a threshold of parties via a threshold signature scheme such as FROST [KG21]. Moreover, we can compose credentials issued via a list with ones issued via signatures.

5.5 Additional features

Notably, our construction of zk-creds also allows for the construction of protocols with several features previously only available in dedicated schemes.

Hidden issuer We can completely hide the identity of a credential issuer, e.g., in situations where leaking where a credential came from can cause significant harm to privacy. While this is not a new notion in the literature, very few existing schemes support this hidden issuer property. zk-creds supports this inherently, as ShowCred can be performed with respect to synthetic lists created by concatenating lists maintained by different issuers, thus hiding the issuer. One drawback though is that multiple issuers will likely issue different credential formats.

Hidden credential type zk-creds can also be configured to hide the credential type which is both independently useful as well as necessary to fully support hidden issuers. In zk-creds, credentials with different attributes can be padded to the same size post-issuance, and then used interchangeably for and efficiently verified over an access criterion. This is accomplished by constructing a new circuit that is the OR of the criteria on the individual credentials.

Delegation Issuance authority is delegatable in zk-creds. Authority to issue a credential can be shown via zk-supporting-documentation that is itself the show of another credential. Moreover, because the proof in zk-supporting-documentation is general-purpose, the delegation process can constrain attributes in the credential being issued. For example, we could define a credential for an authority that can only be delegated three layers deep by having a hidden attribute of

delegation level decremented each time. Credential attributes can be selectively delegated as well.

5.6 NIZK setup

Groth16 requires a one-time trusted setup to generate a set of parameters called a common reference string (or CRS) for each statement (a.k.a., circuit). Once this CRS is generated, it can be used throughout the lifetime of the system to prove different instances of the statement.

Distributed setup for Groth16 CRSs is a solved problem via multiparty computation setup *ceremonies* [BGM17, BCG+15, KMSV21] that need only two honest parties. These have been run with hundreds of users and used to secure billions of dollars in cryptocurrency. These protocols are efficient and produce *subversion-resistant zero-knowledge proof systems* [Fuc18]—systems which ensure that, even if all parties in a setup are malicious, the proofs are still zero-knowledge and user privacy is unaffected.

Independently, as mentioned previously, we have also sought to minimize the impact of this CRS on the flexibility of `zk-creds`. By utilizing blind Groth16, a system designer does not need to decide on and pre-generate CRSs for all possible combinations of access criteria they wish to support and can instead just generate CRSs on a per-gadget basis.

5.7 Security argument

We argue that `zk-creds` is secure under computationally-bounded adversaries performing *static* corruption of users, verifiers, and even issuers who can collude arbitrarily with other parties. Without loss of generality we consider the security of our protocol under a single issuer, since any corrupted issuer can potentially corrupt the credential list; see Section 8 for a discussion of various methods to mitigate and distribute trust in a multi-issuer setting. Let the ideal functionality $\mathscr{F}$ represent the algorithms defined in Figure 5.3.

The simulator $\mathscr{S}$ first runs Setup and gets the necessary trapdoors for the simulation and extraction of zero-knowledge proofs in Setup, IssueSetup, and ShowSetup. $\mathscr{S}$ maintains a table

mapping credentials in its simulated real-world protocol to their ideal counterparts in $\mathscr{F}$ and, when necessary, updates the mapping when it needs to create a real-world object for an ideal-world one (or vice versa).

Because all messages between parties in our security game are accompanied by zero-knowledge proofs, the simulator can extract on all adversarially generated messages (e.g., for IssueReq or ShowCred), look up the corresponding ideal-world credentials in its table, and proxy the requests to the ideal functionality. Similarly, for any honest interactions in the ideal functionality, the simulator can model the adversary's view of the real-world protocol by simulating the zero-knowledge proofs with respect to random commitments and, in the case of rate limiting, PRF outputs.

We note that care must be taken in two cases. First, both honest and corrupted issuers can deny credential issuance, so the simulator's table needs to be updated only on successful issuance. Second, both honest and corrupted parties can revoke credentials, so the simulator must also synchronize revocations.

The security of `zk-creds` rests on the assumption that Groth16 is perfectly zero-knowledge and simulation-extractable (in the Algebraic Group Model); that linkage proofs are sound and zero-knowledge under the Random Oracle Model and dlog (see Section 15); that the Poseidon hash function is collision-resistant and a secure PRF and; for other variants of our scheme, that Schnorr signatures are unforgeable under dlog, Pedersen hashes are collision-resistant under dlog, and Pedersen commitments are computationally binding under the same assumption and perfectly hiding. Then, the adversary's view of the real-world protocol is indistinguishable from a simulated view where honest parties use the ideal functionality $\mathscr{F}$. Therefore, the instantiation of `zk-creds` given in Figure 4.4 is secure against malicious adversaries who engage in static corruption.

6 Implementation and evaluation

We now detail the evaluation of `zk-creds`.

Client-opt. (C) / Server-opt. (S)	ShowCred C	ShowCred C (full)	ShowCred S	VerifyShow C	VerifyShow S	VerifyShow S (batch)	Proof Size C	Proof Size S
Simple Possession	5ms	465ms	450ms	3ms	1.5ms	1.8 verifs/ms	744B	192B
Expiry	53ms	526ms	461ms	5ms			1064B	
Linkable Show	43ms	494ms	457ms					
Rate Limiting	60ms	507ms	461ms					
Clone Resistance	90ms	542ms	530ms					

Table 4.1: Gadget microbenchmarks using Poseidon hashes for two versions of `zk-creds`. Membership proofs are done on a Merkle forest of size 2^{31} (tree height = 24, #trees = 2^8). The first configuration (C) minimizes client-side proving cost; the second configuration (S) maximizes server throughput. ShowCred (full) gives the cost of including membership recomputation while showing a credential. VerifyShow (batch) gives throughput for verifying a set of 100 proofs. We emphasize that all verification numbers are *single-threaded*, allowing for efficient concurrent processing.

Client-opt.	ShowCred	VerifyShow	Proof Size
Simple Possession	2ms	2ms	424B
Expiry	50ms	5ms	744B
Linkable Show	41ms		
Rate Limiting	58ms		
Clone Resistance	88ms		

Table 4.2: Benchmarks for `zk-creds` configured for minimizing client side proving cost, using signature-based issuance. Unlike the client-optimized benchmarks in Table 4.1, which uses list-based membership, there is no distinction between a partial or full ShowCred, since the membership proof (a signature verification proof) never has to be recomputed.

6.1 Code and setup

Hardware. All benchmarks were performed on a desktop computer with a 2021 Intel i9-11900KB CPU with 8 physical cores and 64GiB RAM (mostly unused in our experiments) running Ubuntu 20.04 with kernel 5.11.0-40-generic.

Code. `zk-creds` consists of 7.6k lines of Rust code[7] and relies on the Arkworks [Ar21] zkSNARK framework. For benchmarks and statistics, we used the Criterion-rs crate. In addition, we modified an existing Android passport scanner app to extract intermediate cryptographic values from the passport and dump them to a JSON file.[8] The Rust code uses the dump file format for all its passport proofs.

Statistics. Each figure and plot shows the median runtime of 100 executions. Over all experiments,

[7] Code repository: https://github.com/rozbb/zkcreds-rs
[8] Code repository: https://github.com/rozbb/zkcreds-passport-dumper

the maximum estimated relative standard error of the median is 1.8%. For completeness, our plots include error bars indicating the 95% confidence interval, though they are not visible due to the low error.

Instantiating cryptographic primitives. We set $\lambda = 128$. We compute Groth16 proofs over the BLS12-381 curve [Bow17b]. The collision-resistant hash function used in all Merkle trees are domain-separated instances of the Poseidon hash function [GKK+19], configured to be compatible with BLS12-381 and a 128-bit security level ($\alpha = 3$ and capacity $= 1$). We instantiate Com using key-prefixed Poseidon as well. Separately, in Section 17, we give benchmarks for an instantiation with provably secure Pedersen hashes and commitments.

6.2 Microbenchmarks

In this section we measure performance of various common gadgets, with reasonable parameter choices. Recall the performance of `zk-creds` depends on credential list size, attribute size, criteria complexity, and number of standalone gadget proofs. We measure the effects of these parameters in greater detail in Section 12. We also measure the sizes of the associated proving and verifying keys in Section 18.

Table 4.1 gives a summary of `zk-creds`'s performance for common usage scenarios. We assume a setting where 2^{31} credentials have been issued. The first variant (C) minimizes client-side proving cost by separating the Merkle forest proof, allowing for reuse across shows. A basic show takes 5ms to produce and 3ms to verify, assuming precomputation of the Merkle tree and forest membership proofs. More complex statements like rate-limited credentials with clone resistance take 90ms to show and 5ms to verify. If the Merkle tree membership proof is not precomputed, then the full show takes an additional 460ms but verification time is unchanged.

The Simple Possession benchmark shows the prover has an attribute-less credential on a list and proves no predicates. This maps to a use case such as possessing a valid access card, as that is often sufficient to enter a building. The remaining benchmarks build on Simple Possession, adding their own predicate to the set of linked proofs. For example, Expiry proves possession, but also bears a

single attribute and proves that it has a value less than some timestamp.

Separately, we give an alternate variant of `zk-creds` in Table 4.1 (S) which is optimized for verification latency and server throughput. The server-optimized construction combines the Merkle membership and criteria check circuits into a single monolithic zkSNARK without proof reuse. As a result, clients pay approximately the full ShowCred cost every time, but since proofs are a single Groth16 proof rather than a linkage proof, they can be batch-verified by the server at 1.8 verifications per millisecond per core. We note it may be possible to batch verify the non-optimized scheme as well, but throughput would be lower as there are at least three times as many proofs.

Finally, to demonstrate the full flexibility of our approach, we also provide a signature-based variant of `zk-creds` in Table 4.2, where credentials are issued in a more traditional signature-based manner by a trusted issuer or threshold quorum of issuers. We implemented a signature gadget for checking Schnorr (and therefore FROST threshold [KG21]) signatures and measured the proving time to be 129ms (compare to 460ms for tree-based issuance).

Memory usage. Peak memory usage for any of our experiments was 824 MB. Since prover memory usage is a function of circuit size, this peak was, as expected, for the server optimized configuration with the larger monolithic circuit. Separately, the client optimal approach has a peak usage of 800 MB, which arose when proving LinkG16.

7 Case studies: `zk-creds` as a toolkit

We design, implement, and benchmark two full scenarios for `zk-creds` using credentials derived from existing government identity infrastructure without any modifications or coordination. Many government identity documents now include the ability to perform various authentication protocols (e.g, the German and Estonian [Fed, e-E] smart-card enabled national IDs). For our applications, we use US passports, which contain a signed digital copy of the passport's basic data in an NFC-readable chip. Our applications validate that `zk-creds` can be used as a toolkit by application developers to support privacy-preserving identity in realistic applications with complex, compound

access criteria.

7.1 Digital passport data

Over 150 countries issue passports with an NFC-readable chip which is standardized in ICAO Doc. 9303 [Rea, ICA]. We are interested in the first two *data groups* on the chip. Data group 1 (DG_1) contains the textual info available on the passport's data page: name, issuing state, date of birth, and passport expiry. Data group 2 (DG_2) contains a JPEG-encoded image of the passport holder's face. The ICAO standard also requires the immutable part of the chip's contents to be signed by the issuing state. For example, every US passport has an RSA PKCS#1 v1.5 signature under a known US State Department public key.

zk-supporting-documentation for passports. While we could just reveal the signed passport to the credential issuer, this 1) requires the issuer be trusted to maintain the confidentiality of sensitive information, and 2) is wholly incompatible with issuance via a bulletin board or blockchain. Instead, we design and implement a zero-knowledge proof that the attributes of a credential commitment match the signed contents of DG_1 and DG_2.

Parsing the contents of an e-passport in zero-knowledge is non-trivial: the signature is not just over DG_1 and DG_2, but the *econtent hash*, which is calculated over the mostly variable-length data groups $\mathsf{DG}_1, \ldots, \mathsf{DG}_{16}$. Variable length inputs are particularly challenging to parse with a fixed-size zero-knowledge circuit. Our zero-knowledge proof is made possible by realizing that the econtent hash is actually a tree hash, roughly of the form

$$H(H(H(\mathsf{DG}_1)\|H(\mathsf{DG}_2)\|\ldots\|H(\mathsf{DG}_{16}))\ldots).$$

We do not care about the contents of DG_3 through DG_{16}, so their hashes can be used directly as witnesses to the proof. $H(\mathsf{DG}_2)$ is the image hash, which can either be hidden as a witness or revealed by showing all of DG_2.[9] Since DG_1 contains the attributes we care about, we must parse

[9] In reality, DG_2 contains slightly more than the bare image; it also has the image's creation and expiry dates.

DG_1 inside the zero-knowledge proof. Luckily, since DG_1 mirrors the content of the passport's Machine Readable Zone (MRZ), it is fixed-length. The proof then hashes the data group digests along with other fixed-length values until it has computed the econtent hash. We avoid the cost of checking the RSA signature by simply revealing it and the econtent hash.[10] Computing this proof takes less than 2 seconds.

Why zk-proofs over passports are insufficient. A folklore solution for anonymous credentials is to replace the signature with a zero-knowledge proof of signature possession. Consider, for example, Cinderella, [DFKP16]. Cinderella elegantly converts existing X.509 certificates into anonymous credentials via an intricate zero-knowledge proof that validates the X.509 certificate and its entire certificate chain. Assuming identity documents with X.509 certificates, this is seemingly a viable approach for proof of age.

Zero-knowledge proof of possession of an existing signed document is both insufficient in our case and, more subtly, unnecessarily complex. A proof over a certificate or indeed any existing credential is insufficient: it may not include the necessary data and that data cannot be hidden from the issuer. This is a challenge for, e.g., the random seed we use for cloning resistance. As a result, we must issue our own standalone credentials and can, at best, use existing certificates as supporting documentation to justify issuance. But a zero-knowledge proof for supporting documentation has a subtly different goal from the folklore solution; while we must hide the attributes in the certificate and issued credential (e.g., name and date of birth), we can leak other information such as who issued the credential (e.g., the US State Department). While this could uniquely identify a user during a credential show, revealing the same information during issuance poses no such risk: issuance is by definition unlinkable to show anyway. This drastically simplifies the zero-knowledge proof and removes unnecessary complexity.

[10] Revealing the econtent hash reveals the identity of the requester to the passport authority during issuance. However, since ShowCred proofs are unlinkable to issuance, this is not a problem. Moreover, it is possible to hide the hash and verify the RSA signature in the zkSNARK, at a cost of about 868k constraints using the approach from [KZM+16], or 3.6s in our benchmarking environment.

7.2 Instantiating an issuance bulletin board

An instantiation of `zk-creds` requires a publicly accessible bulletin board to distribute the credential list, as well as parties running our software. We stress that `zk-creds` can be deployed either on a blockchain or (at much lower cost) a more centralized transparency log; we choose to instantiate the bulletin board here as a smart contract on an Ethereum Virtual Machine (EVM)-compatible blockchain to demonstrate that full decentralization is feasible and because, unlike many other consensus systems in development or deployment, the EVM has comparatively robust development tooling and documentation.

The smart contract stores a list of credentials and corresponding zk-supporting-documentation—together, referred to as an *issuance request*—as well as the current roots of the Merkle forest. An issuer issues a credential by posting the full issuance request to the smart contract. This allows any external auditor to download the full list, reconstruct a local copy of the Merkle forest, then verify in zero-knowledge that each credential was validly issued. While any party (including the user) can audit the full list themselves if desired, they need not do so if they trust another party (e.g., an issuer or auditor) to promptly update and verify the credential list for them. Periodically, users request their credential's authentication path and the updated root from the bulletin board or auditing party. To perform VerifyShow, a verifier only needs to retrieve the current Merkle tree roots F from the bulletin board.

Furthermore, we must prevent DoS attacks from blocking or flooding additions to the bulletin board. Our prototype instantiation assumes a smart contract operator who is authorized to add to the bulletin board contract on behalf of the issuer. An alternative solution, allowing for operator-free setup, is to verify all zk-supporting-documentation and Merkle tree root computations within the EVM smart contract itself. While this is feasible [Torb], verifying proofs and Merkle tree updates on-chain without extensive optimizations is expensive. Another solution is to instead support on-chain proof verification with an optimistic rollup [Eth]: bulletin board additions include a deposit which is burned if, after the smart contract evaluates a challenge, it determines that the proofs or computed Merkle tree roots are invalid. While the challenge still requires costly computation, this

is not paid for during issuance.

We implement our smart contract in Solidity and the requisite client-side scripts to post and retrieve data in `web3.js`. Note that we can deploy our contract to any EVM-compatible chain, such as Avalanche's C-Chain or Ethereum itself. EVM contract operations are measured in *gas*, which roughly acts as a complexity-weighted count of the EVM instructions used. Posting each issuance request costs 576,808 gas, while posting a Merkle root costs 78,355 gas. The price of gas and the underlying currency it is priced in is highly volatile. As of early June 2022, Ethereum gas prices range from about 20 to 50 Gwei (a Giga-wei is 10^{-9} ETH), and 1 ETH is about $1800 USD; as such, posting an issuance request costs about $20–$50 USD. Gas prices are proportionally similar for Avalanche but, at $25 USD per token, actual costs would be 70 times smaller.

7.3 Credentials from existing identity infrastructure

Given a construction of zk-supporting-documentation for a passport and a choice of bulletin board in an Ethereum smart contract, application developers can now readily build access control schemes. Once the credentials are issued, multiple developers can independently rely on them by either composing existing identity gadgets or defining new ones.

Issuance. We provide a toolchain to convert a passport into an anonymous credential. An Android app extracts the NFC passport data and a separate program converts it into a credential containing the holder's nationality, full name, and date of birth (dob); a rate key (rk); the passport expiry date (expiry); and the hash of the image of the holder's face (facehash). Separately the program computes the zk-supporting-documentation that this credential is correct with respect to the signature and econtent hash.

The IssueReq payload sent to the issuer consists of the signed econtent hash, credential, and zk-supporting-documentation proof. IssueGrant verifies the proof and econtent hash signature. Upon success, the issuer adds the credential to their list and returns a Merkle authentication path.

Scenario 1: Viewing age-restricted content on the internet. Age-restricted content is common on the internet. For example, in Switzerland, the EU, and the UK, YouTube requires users to

upload an image of their ID or credit card in order to prove their age [Gooa]. In this scenario we demonstrate the feasibility of `zk-creds` for proving age without revealing any personally identifying information.

Our `zk-creds` toolkit has three features that are crucial to building a real-world feasible age verification credential. First, it can be used without coordination with existing identity infrastructure. Second, it can readily support other identity credentials, provided they indicate date of birth and are signed. Third and, most crucially, it can create credentials that are clone-resistant (via the gadget in Section 5.3) with easy revocation of cloned credentials. This last point is essential: while a zero-knowledge proof over a passport is itself an anonymous credential,[11] practical usage demands cloning protections. And cloning resistance requires a rate key to be bound to the credential and kept secret from the issuer. Existing identity documents (such as a passport) lack such a key. `zk-creds` allows composition of identity without coordinating with the passport issuer or any trusted party to add such information.

Given issued credentials via passports, building a privacy-preserving age verification scheme with `zk-creds` is straightforward and requires no new cryptography: website developers need simply define the issuers they will accept[12] and construct the access criteria they need using gadgets. For this scenario, the only issuer is our passport-based issuer, and the access criteria being proved are age, expiry, and non-cloning. Concretely, the access predicate is:

$$\mathsf{dob} \leq \mathsf{today} - 18\mathrm{yrs} \wedge \mathsf{expiry} > \mathsf{today} \wedge \mathsf{CloneResistance}(\mathsf{rk}, \mathsf{nonce}, \mathsf{tok}_1, \mathsf{tok}_2, \ldots)$$

where $\mathsf{CloneResistance}$, nonce, tok_1, and tok_2 are as described in Section 5.3.

Table 4.3 gives performance numbers. Given a credential, it takes Alice 143ms to show a website she is over 18 (and 602ms when she must recompute her membership proof). The server can verify her assertion in 5ms. If we wish to optimize for server verification time or throughput, we can switch to `zk-cred`'s server-optimized construction and achieve 1.5ms verification times and

[11] When augmented to hide the passport signature.

[12] This defines which credential list they use or defines a new list as some subset of existing ones.

	IssueReq	IssueGrant	ShowCred	ShowCred (full)	VerifyShow
Age-restricted vid.	1.97s	2ms	143ms	602ms	5ms
Entering a bar	1.97s	2ms	98ms	557ms	5ms

Table 4.3: `zk-creds` case study benchmarks. IssueReq is the time to convert a passport into a credential using zk-supporting-documentation. ShowCred is the time it takes a user to prove they are over 18. All other parameters are the same as in Table 4.1.

1.8 verifications per ms per core. Extrapolating from the server-optimized benchmarks in Table 4.1, proving times would increase to approximately 595ms.

Scenario 2: A cryptographer walks into a bar. To purchase alcohol in the United States, one needs to show photo ID and proof they are at least 21 years old. But showing a driver's license reveals name, sex, weight, and date of birth. And if the license's barcode is scanned [ACL12], additional information is revealed, potentially including whether the holder is insulin-dependent, hearing-impaired, developmentally disabled, or, surprisingly, a sexual predator [Veh].

We now build a system for in-person age verification coupled with photographic verification. Importantly, our goal in this scenario is *selective disclosure* and not anonymity. Anonymity in an in-person setting is often not possible or even desired. Rather, what we want is privacy: the ability to control what information is revealed and limit it to what is necessary—the patron's photo and the fact that they have an unexpired ID with a birth date making them of drinking age.

We envision a hypothetical system where bar patrons have an ID-wallet application on their phone. The app, using ShowCred, presents an identity assertion (e.g., via a QR code or NFC) to an app on a bouncer's phone which checks the assertion with VerifyShow and displays the user's photo along with whether they are over 21. In contrast to scanning the user's driver's license, this reveals only the minimal information necessary. The necessary access predicate is:

$$\mathsf{dob} \leq \mathsf{today} - 21\mathrm{yrs} \wedge \mathsf{expiry} > \mathsf{today} \wedge \mathsf{facehash} = H(\mathsf{face})$$

Table 4.3 gives concrete costs for local computations. To show that a patron is over 21 takes 98ms in the common case and 557ms if membership proofs must be recomputed. As before, verification is 5ms in either case.

8 Extensions and applications

We now discuss extensions to `zk-creds` and additional applications of our approach.

Other signed identity sources. As shown by our e-passport example, `zk-creds` can transform legacy identity sources into anonymous credentials if there is a digitally signed component. This raises an interesting question about what parts of existing identity and credential infrastructure include such signatures. For example, digital diplomas for many US universities include digital signatures over the diploma holder's name, degree, and institution. Many emails are signed with DKIM which, while problematic in many contexts [SPG21], could be a source of identity or membership in an organization. New York's Excelsior Pass for COVID vaccination contains the holder's name, birth date, and a signature.[13] Other existing digital protocols may contain a signature that establishes ownership of a resource (e.g., a phone number in an eSIM or virtual SIM card) or identity (e.g., Apple's digital driver's license features).

Complex access criteria. We have discussed conceptually simple access control criteria such as "my credential is not expired," or "I am of age," perhaps with a cryptographically complex mechanism for clone resistance. However, real-world access criteria can be far more complex. `zk-creds` provides a way to deal with such criteria without requiring coordination with identity issuing authorities for every custom access check that must be implemented.

An example of this comes in the form of online petitions and discussions. New York State has an online portal for discussion and petition which asks a user for their address to match them with the appropriate state senator [New]. While this check does not seem to be enforced, one could imagine both wanting to enforce this constituency check and allowing constituents to leave non-identifying comments. Similarly, some online resources, such as ebooks from the New York Public Library, are limited to city residents; enforcing this currently requires in-person registration for a library card to present proof of address, and opens up a (hypothetical) risk of tracking online reading habits [Ame06].

Geolocating an address to the bounds of, e.g., a city council district, however, is not simple.

[13] This was obtained by scanning the pass's QR code, whose contents are a W3C Verifiable Credential [W3C].

The computation is, by the standards of credential schemes, complex, and involves converting the address to a location and then performing a point-in-polygon check.[14] For a small number of fixed boundaries (e.g., federal congressional districts), one might imagine avoiding the problem by issuing identity documents with this information included. But even in cases where the identity issuer would cooperate, coordinating all geocoding restrictions one might want to realize (e.g., anonymous discussion boards for a school zone, a neighborhood, or even specific apartment building) is impractical and may cause credential sizes to blow up.

Because `zk-creds` supports general purpose zero-knowledge proofs, geocoding restrictions are made more feasible with Groth16 gadgets: even if the Groth16 proof for the gadget is expensive, the resident or an outsourced prover avoids recomputing it every show. After the first time, the proof can be reused arbitrarily, until the user's Merkle tree is updated by a new issuance.

Sybil-resistant IDs from email or money. Internet services currently prevent multiple account registration (Sybils) by requiring the user to consume a (hopefully) scarce resource, such as money (via a micropayment), attention (via a CAPTCHA), and possession of, e.g., phone number or email address.

`zk-creds` provides several avenues for Sybil-resistant credentials with anonymity. Credentials can be issued based on signed identity documents (e.g., a passport, as demonstrated in Section 6) with the signature as a uniqueness check. Similarly, `zk-creds` can thwart Sybils via cryptocurrency: a simple smart contract issues credentials if and only if a small fee is paid.

Finally, and perhaps most surprisingly, we can use possession of a valid email address as a Sybil-resistance mechanism without the use of a trusted third party or cooperation with the email provider. A DomainKeys Identified Mail (DKIM) header, which appears on all outgoing mail of most modern email providers, contains a signature from the email provider. By embedding the credential issuance request in the email body, we get an externally verifiable proof of possession of an email address that can be used to issue Sybil-resistant accounts. This allows us to leverage the Sybil resistance mechanisms used by Gmail, for example.

[14] Indeed, geolocation is the correct way to prove residency. For example, the borders of New York City's 10 city council districts are defined by 1.8 megabytes of geospatial vector bounds [NYC].

These short- and long-lived IDs can be reused and rate-limited across actions, similar to the functionality of Privacy Pass [DGS+18], which issues one-time-use anonymous access tokens for every CAPTCHA a user completes.

Oracle- and self-issued credentials. A number of academic works and industrial systems have emerged to address the so-called "oracle problem": how does a consensus scheme such as a blockchain come to agreement about real world events?

One class of solutions [EJN] relies on incentive systems and the ability to challenge the veracity of data. These approaches, if viable, could be used to issue anonymous credentials based on public online reputation data (e.g., Twitter follower count, Reddit karma, etc.). Crucially, because `zk-creds` forces all issued credentials to be on a public list, any malfeasance by an issuer could be detected and punished.

An orthogonal approach is the creation of notaries who attest to data on third party servers. DECO [ZMM+20] proposes a 2PC protocol between a client and a *notary* that authenticates data retrieved over a TLS connection from a third-party server. DECO also allows users to construct zero-knowledge proofs to only selectively disclose HTTP response contents. These features would allow a user to obtain a credential for their name and address by, e.g., logging into their utility provider and retrieving the bill. Moreover, in the event the user has trustworthy secure hardware, they can *self-issue* the credential by attesting the hardware ran this notary check itself.

We note, however, that such schemes inherently rely on assumptions (either non-collusion or the security of trusted hardware) that become increasingly infeasible as the value of the credential goes up. But, for example, it may be viable as a simple Sybil prevention or anti-spam mechanism.

Composable credentials. When new use cases for existing credentials emerge, they often require the combination of two different credentials. Take, for example, US-issued vaccine cards. Because these contain a person's name, but not a photo, COVID-19 vaccine mandates frequently required restaurants and bars to ask customers for a vaccine card *and* a photo ID with a matching name. `zk-creds` supports this type of post-hoc composition: two credentials both containing a field for, e.g., full name, can be jointly shown using the Join gadget described in Section 5.3.

9 Related work

Anonymous credentials derive from a long line of work, starting with Chaum [Cha85], and subsequently seeing numerous extensions [Cha85, CL01, CL03, CL04, CHK+06, BCKL08, CG08, BL13, GGM14, CDHK15, SAB+19]. While showing a credential initially allowed little more than (unlinkably) presenting a signed token connected to a user's pseudonym, the schemes were generalized and extended to provide more sophisticated properties such as issuance of hidden attributes, rate-limiting, k-show, and efficient selective disclosure of attributes. Because it uses general-purpose zero-knowledge proofs, `zk-creds` can (and does) capture all of these properties as implemented.

One drawback to deploying the majority of these schemes is the requirement of a single, trusted issuer. As such, existing work has sought to solve this issuance problem. We briefly compare and contrast other approaches to addressing this.

Distributed issuance. In 2014, Garman et al. [GGM14] were the first to propose the notion of decentralized anonymous credentials. Our approach is directly inspired by their work.

While the approach of Garman et al. removes the assumptions of a single issuer and the need for issuers to hold keys, it both leaves open a number of essential questions for operating a real system and introduces new ones which we address. First, showing a credential requires users to 1) locally store the full credential list and 2) compute proofs which take time linear in the number of issued credentials for every show (even for static credential lists). In contrast, `zk-creds` develops a new approach and new cryptography (blind Groth16) to allow proof reuse and fast credential showing. And, via the use of Merkle trees, `zk-creds` requires users to store only logarithmic-sized witness data to compute credential shows. Second, while Garman et al. suggest the possibility of more complex features, they do not implement them. More importantly, the set of bespoke sigma protocols they use does not provide for the composition of credentials and identity attributes or support complex identity statements. By developing a protocol based on general-purpose zkSNARKs, we do. Lastly and most significantly, the work does not answer the question of how the decision to issue a credential could be made without disclosing sensitive

information to the issuer (i.e., the very problem zk-supporting-documentation addresses). This also makes it impossible to audit credentials that are issued via Garman et al.'s construction, in marked contrast to `zk-creds`.

Threshold issuance. The Coconut [SAB+19] anonymous credential scheme addresses the issuance challenge via threshold signatures. In Coconut, credentials are issued by n static parties under the assumption $t > 1/2$ of them are honest. The scheme is clever and achieves efficiencies on par with single-issuer schemes. However, while threshold issuance increases the security of a scheme by requiring an attacker to corrupt more parties, it only addresses the scarcity of issuers if we have an abundance of parties who are willing to issue credentials but, for whatever reason, no individual one is trusted. In many settings, it is a challenge to find even a single party who both 1) is empowered to make identity statements and 2) is willing and able to run cryptographic infrastructure even if, by fiat, we trust them. Moreover, Coconut only supports selective disclosure of attributes in a credential, not complex zero-knowledge proofs over attributes. It does not meet our design goals of flexibility or dynamic generation of access criteria.

Finally, we note that `zk-creds`, as shown by the version given in Table 4.2, supports credential issuance via threshold Schnorr schemes (e.g., FROST [KG21]), so we do capture the functionality of Coconut.

Decoupling issuance from identity verification. More broadly, another line of work, starting with TLSNotary [TLS], considers convincing third parties of the correctness of data. DECO [ZMM+20] extended this protocol to support TLS 1.3 and used zero-knowledge proofs for selective disclosure (e.g., the party learns a bank account balance is over a threshold, but not the balance itself or the account holder's identity). Applying this to the anonymous credential setting, one could use it to separate finding a cryptographic issuer for credentials from the process of verifying entitlement to a credential. Such approaches are complementary to `zk-creds` and we consider them as an extension in Section 8. Without such integrations, however, such an approach would still require at least one trustworthy party to be willing to run a highly available web service that holds live signing keys.

Decentralized identity (DID). We note another area of research on decentralized identity which

has been the subject of both academic and industrial work, including some standardization. While `zk-creds` is compatible with these goals inasmuch as it allows issuance without a party holding keys, such works are largely orthogonal: decentralized identity as an area considers who makes decentralized identity assertions, while our work considers how to transform some identity assertion (centralized or otherwise) into an anonymous credential without introducing additional trusted parties.

10 Conclusion and future work

The approach we develop here—a switch from blind signatures to zero-knowledge proofs as the foundation for anonymous credentials—implies several avenues for further work. In this section, we enumerate a few immediate consequences of this paradigm and future research areas.

Instantiating `zk-creds` with improved zero-knowledge proofs. We have instantiated `zk-creds` with Groth16. The `zk-creds` approach we develop, however, is proof system-agnostic. As such, instantiating the `zk-creds` paradigm with other proof systems, such as ones without trusted setup (e.g., [BBHR18, Gro16, BBB+18]) or with universal setup (e.g., [MBKM19, GWC19, CHM+20]), is entirely possible. For the monolithic construction, such a change is a drop-in replacement. If we wish to support precomputation of separate membership proofs, as in our client-optimized scheme, we must either adapt blind Groth16 to the new proof system or take an alternative approach, e.g., recursive proofs. The choice of proof system is also tied to the choice of accumulator scheme.

Instantiating `zk-creds` with alternative accumulator schemes or primitives. We have instantiated `zk-creds` using Merkle trees. However, as with proof schemes, the same approach generalizes to other accumulator mechanisms such as RSA accumulators [GGM14], polynomial commitments [KZG10], Verkle trees [Kus18], or perhaps special-purpose schemes for zk-SNARKs [ZBK+22]. Again, for many such accumulators, this is a simple black-box replacement of the membership zkSNARK. Instantiating `zk-creds` with alternative accumulators will offer different tradeoffs for witness size, witness update requirements, witness computation cost, and ac-

cumulator verification cost (and hence zkSNARK proving time). A particularly exciting prospect is the co-design of accumulator schemes and zkSNARKs to achieve drastically improved performance.

Co-design or co-selection of zero-knowledge proofs and credential schemes. Similarly, one could instantiate either our existing version of `zk-creds`, or a different construction, with different cryptographic primitives. For example, one might replace Poseidon with a newer, circuit-optimized hash function, perhaps making use of low-degree gates in proving systems like Plonk [GWC19].

Batched verification/verification in a smart contract via proof composition. One interesting question is how to reduce the cost of verifying many credential shows. Zebra [RPX+22] considers this problem for batch verifying classic blind signature based anonymous credentials in the context of smart contracts where computation is very costly. In Section 6.2, we give basic benchmarks using batched pairings for verification of our server-optimized construction. But it is also possible to construct a zero-knowledge proof that batches many shows into one proof that has sublinear (in the number of shows) and potentially constant verification cost. For example, SnarkPack [GMN22], which can dropped in to aggregate shows in our server optimized version, quotes a logarithmically scaling batching mechanism with verification costs of 8192 proofs in 163ms. The design of better aggregation mechanisms, potentially co-designed with a `zk-creds` style scheme, is an interesting avenue for future work.

Instantiating `zk-creds` with alternative (post-quantum) primitives. While we have instantiated `zk-creds` with primitives such as Groth16, as mentioned previously, the flexibility and modularity of our system easily allows for the swapping of any primitives with those that fulfill the necessary requirements. For example, one could easily build a post-quantum version of `zk-creds` by instantiating it with post-quantum secure primitives and zero-knowledge proofs such as [AHIV17, BBHR19].

11 Acknowledgements

We would like to thank Mary Maller for the idea and sketch of the LinkG16 proof system, Daniel Benarroch for starting the discussion of Merkle Forests, and Jeffrey Burdges for pointing out that passports contain signed credentials. Both were productive conversations at Real World Crypto 2019. We would like to thank the anonymous reviewers and shepherd for their helpful comments.

Michael Rosenberg's work was supported by the National Defense and Engineering Graduate (NDSEG) Fellowship. Jacob White and Christina Garman's work was partially supported by NSF grants CNS-1816422 and CNS-2047991. Ian Miers's work was supported in part by a Facebook Research Award. Image credits [Lit].

12 Appendix: Performance across parameter choices

We expand on the microbenchmarks in Section 6.2 by investigating how `zk-creds` scales with respect to various parameters.

Recall that showing a credential requires proving: 1) the credential is in the list of issued credentials, 2) the relevant attributes are part of the credential, 3) the attributes meet the access criteria being shown, 4) each of the above is about the same data (linkage proof). Thus, the performance of `zk-creds` depends on the number of issued credentials (via 1 above), the size of the attribute (via 2), and the complexity of the access criteria (via 3).

Membership benchmarks. Recall that a membership proof consists of a proof of credential membership in a tree, followed by a proof of membership of that tree in a forest. In Figure 4.6, we show the performance of proving membership as the shape of the forest changes. For a fixed number of total leaves, we find the size of the forest (and, consequently, height of its trees) that minimizes membership proving time. This results in a 50% (143ms) speedup in the best case. Further, the verifier pays nothing for this optimization, since all public inputs are prepared in advance and reused for all verifications.

In Table 4.4, we show the time to compute a proof of membership in the credential list as the

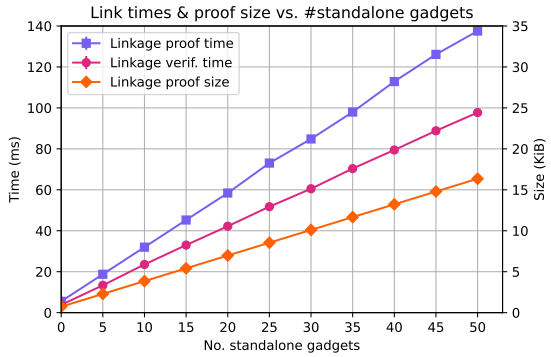


Figure 4.5: Costs of linkage proofs as #standalone gadgets varies.

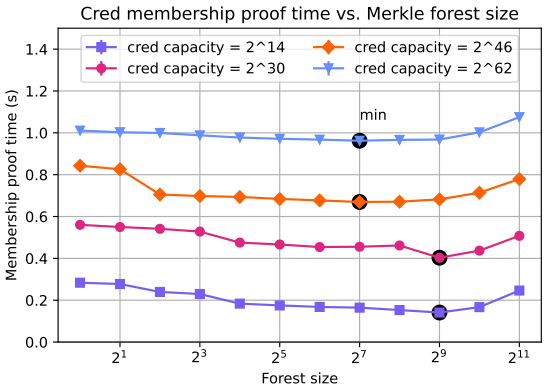


Figure 4.6: Costs of proving tree then forest membership, as number of trees in forest varies.

#leaves	2^{14}	2^{30}	2^{46}	2^{62}
Issuance proof time	141ms	403ms	670ms	962ms

Table 4.4: Cost of a proof of credential list membership as the size of the list varies.

list size varies. This represents the baseline cost of the issuance portion of any ShowCred call. The benchmark consists of one Groth16 proof of tree membership plus one Groth16 proof of forest membership. For a fixed number of leaves, the tree height is chosen using the optimal parameters from the experiments in Figure 4.6.

Linkage benchmarks. In Figure 4.5, we plot the size, proving time, and verification time of linkage proofs as the number of standalone gadget proofs varies. Every additional gadget adds 330B to the proof size.

13 Appendix: Merkle forests and witness management

Showing a credential requires proving membership in a Merkle tree. Since the list of issued credentials is dynamic, users must somehow maintain an up-to-date witness to their credential's membership in the tree. A naïve approach requires $O(N)$ storage[15] and $O(N)$ communication: clients store the entire tree and retrieve every addition. In this section, we attempt to solve the issues introduced in Section 5.1, iteratively building up techniques to optimize Merkle trees and

[15]This is trivially reduced to $O(\log N)$ storage per credential, as the client can update witnesses in place.

their membership proofs in the interests of both efficiency and privacy. First, we address common problems with using Merkle forests which motivate this discussion and clarify the assumptions our constructions make. Second, in Section 13.2 we look at various interoperable batching techniques for Merkle trees, allowing issuers to summarize *prior* Merkle tree witness updates for specific clients based on when they last received an updateand/or defer *future* credential issuance until many credentials can be added in a single batch to the Merkle tree(s). Third, in Section 13.3 we look at ways to minimize correlation attacks against privacy due to users requesting then immediately showing their credential's membership proof, and suggest replacing update requests with a universal broadcast of all relevant updates.

We conclude our discussion of Merkle tree optimizations with a unified Merkle forest construction in Section 13.4 which balances the competing tradeoffs of 1) expected number of membership proof updates, 2) privacy, and 3) communication overhead. We accomplish this by combining the aforementioned techniques into a unified approach which leverages the unique structure of Merkle trees and authentication paths. Concretely we find that, compared to a single Merkle tree of similar capacity, our construction requires fewer membership proof updates over time for *all* users of older credentials, at the expense of only a small-constant logarithmic increase in communication overhead between issuers and users.

13.1 Merkle Trees in Practice: Overview and Motivations

In `zk-creds`, users must frequently produce membership proofs across the lifetime of a credential to show that it has been issued on a credential list. Doing so requires a user to obtain some new information about the list—the *frontier*—with which users can update the witness to their credential's issuance.

A common approach to allow proving membership on a list is to instantiate it as as a full binary Merkle tree with capacity N, allowing anyone with the credential and its $O(\log N)$-sized witness (i.e., the *authentication path* θ) to verify that it is a leaf node included in the computation of the list's Merkle root. Naïve Merkle tree-based instantiations might require the user to manage a local

copy of the entire Merkle tree themselves, or otherwise delegate the task of updating and distributing the list to some *list manager* who is only trusted to not perform DoS attacks against the system. In real systems, however, keeping up-to-date with the current state of an entire Merkle tree is a non-trivial problem: credential lists can be fast-growing or large; users' devices may be particularly resource-constrained and thus unable to process or even store the entire list at once (e.g., envisoning its usage on a mobile phone or domain controller);[16] users who become inactive for long periods of time will need to request much of the updated list; and users who miss any part of the update would struggle to return to a consistent state.

Moreover, since `zk-creds` is designed such that an honest user's credential remains unlinkable even when a dishonest majority of malicious parties collude, our credential list instantiation should support strong unlinkability and privacy in practice as well. However, we observe that a user updating an authentication path through *any* kind of request to a third-party will correlate closely with the subsequent usage of that credential, since doing so is necessary to maintain a valid membership proof. This leaves users vulnerable to correlation attacks regardless of whether the updates sent back to the user identify the credential.[17] Correlation attacks especially pose a problem if collusion occurs between an issuer and verifier or if the authentication path update identifies or reduces the anonymity set of the requesting credential. If adversaries can build plausible profiles to identify credentials within and across various shows in this manner, this has serious implications on unlinkability and user privacy. We will address these various problems in the remainder of this section.

Hereafter, assume that issued credentials can only be added to the Merkle tree; list entries will never change in-place. For simplicity, assume that credentials are only added from left to right. Lastly, assume that Section 13.2–13.3 uses the conventional construction for Merkle forests, where each Merkle tree in the forest has fixed height n and capacity $N := 2^n$; see Section 13.4 for design decisions and optimizations that can arise when these assumptions are relaxed.

[16] For reference, storing a full 32-height Merkle tree with 256-bit hashes requires about 137GB.

[17] Practical session privacy protections are also insufficient. Even anonymous communication networks such as Tor are (perhaps inherently) vulnerable to de-anonymization via various correlation attacks [DMMK18, Tora].

13.2 Batching issued credentials

Requesting summaries of newly issued credentials. Observe that, if a user has a valid Merkle authentication path (i.e., witness) θ attesting to their credential's issuance at time t, not all $n+1$ nodes in θ will usually need updating by time $t' > t$. Instead, the user only needs a *summary* of all $\leq n+1$ Merkle tree nodes which have been added since time t. While users still reveal identifying information about the credential when updating their authentication paths with this approach—reducing the anonymity set to anywhere from $N/2$ to 1 credentials depending on the request—this approach is somewhat better at both mitigating privacy risks and reducing communication overhead as well compared to the naïve approach of requesting the entire θ each time.

Deferring credentials to issue. One of the more effective techniques to make credential list updates more efficient is to allow the issuer to update the credential list in *batches* according to a specified epoch. Instead of immediately adding a validated k to a Merkle tree T (as with IssueGrant in Figure 4.4), the issuer could, for example, define a regular interval Δ_I to wait before issuing all valid credentials they collected at once. For example, if Alice's credential is being issued in this manner, she must wait at most Δ_I seconds before she can present a valid membership proof for her issued credential.[18] After the batched credentials are added, Alice can obtain its initial witness as appropriate. Note that an issuer can decide on an appropriate batching policy based on usability and efficiency tradeoffs.

Synchronizing Merkle tree state. More generally, we could consider some agreed-upon list manager (perhaps distinct from the issuer) who determines the current state of the Merkle tree for the purposes of membership verification. Even if the issuer themselves is unwilling or unable to batch in the manner described above, the list manager can *snapshot* the state of the Merkle forest

[18] Indeed, many real-world credentials already require a waiting period. For example, a US passport can take 5–11 weeks to validate and 1–2 days to ship before it can be used [U.S].

according to a specified epoch, e.g. at a regular interval Δ_ℓ, and distribute this snapshot as the "current" agreed-upon state of the Merkle forest.

For example, suppose that the latest snapshot occurred at time t and that snapshots are published every Δ_ℓ seconds. First, the user updates their authentication path θ to be valid as of time t using the latest snapshot. If the user and verifier agree that the latest snapshot is treated as current, then θ is automatically up-to-date; otherwise, the user now only needs to manage updates to θ between times t and $t+\Delta_\ell$.

In either case, this approach can be particularly useful for users who have been offline and unable to update their authentication path for many epochs, or who have corrupted their local copy and need to recover the state of the credential list; snapshots of the credential list allow the user to more easily remain up-to-date.

These techniques for batching issued credentials are mostly agnostic to the list's instantiation. As we will soon see, however, this technique is amenable to other approaches where Merkle trees are used.

13.3 Broadcasting frontier nodes

Instead of the user *requesting* specific updates to frontier nodes in θ, as in Section 13.2, consider an alternate approach where a list manager proactively *broadcasts* the relevant frontier update(s) to all users. Each addition to the Merkle tree requires the list manager to broadcast at most $1+\log_2 N$ nodes to all users, i.e. all updated nodes (as relevant to existing credentials) that changed after adding one or more credentials. If left unbatched, this requires broadcasting a small-constant $O(N \log N)$ number of nodes to each user across the lifetime of the system.

However, one can reduce the number of nodes to stream by summarizing a sequence of newly issued credentials. As a simple example, if an issuer issued 2^b credentials by placing them in a binary Merkle tree, it suffices to broadcast a single path of at most $n-b+1$ nodes from the updated root of the tree to the subroot of *all* new batched credentials. With this, the user of any previously-issued credential can parse any updated nodes they need for their authentication path θ

to be current up to that batch. Since broadcasted tree updates now only occur once per batch instead of once per credential, this helps decrease communication overhead by a factor of at least 2^b.

Together, summarizing batches of issued credentials and broadcasting frontiers preserves the instantiation's unlinkability across credential usages. Notably, users no longer need to send identifying information about their credential to update its authentication path, since the broadcasted frontier nodes contain every relevant update to the Merkle tree for all users. Furthermore, proactively broadcasting the frontier updates to users significantly reduces the ability for adversaries to correlate credential usage, as users no longer even need to reveal via request when they want to update their authentication path.

While this approach comes at the cost of additional overhead and system complexity compared to the approach in Section 13.2, it is especially suitable for applications where credential lists are relatively static and credentials do not need to be issued immediately. This approach is also better for many users compared to the naïve approach of downloading the entire Merkle tree, especially when using resource-constrained devices. Similar to standard Merkle-based constructions, the user only needs to store a logarithmic number of nodes (roughly $\log_2 N + c$, where c is a small constant) to process the stream of frontier nodes and update θ. Furthermore, while broadcasting all Merkle tree updates results in $O(N \log N)$ frontier nodes (more precisely, at most $\sum_{i=1}^{N} 1 + \log_2 N = N \cdot (n+1)$) being sent to each user across the lifetime of the Merkle tree, the stream can be processed periodically and in only a single pass using a logarithmic-sized buffer.

13.4 Optimizing Merkle forests

In this subsection, we propose a novel construction which leverages many of the techniques discussed above to simultaneously minimize witness update frequency(and thus proof computation costs), privacy risks (via update request interactivity), and user storage costs at the expense of a moderate increase in communication overhead. We believe that this provides an amenable tradeoff between privacy, efficiency, and usability for many use-cases leveraging Merkle trees. In particular,

users can (eventually) rely solely on broadcasted frontier nodes to update their authentication path θ, the Merkle forest scales effectively as the number of credentials grows, and every user will be required to update their credentials' proofs of membership less often compared to a naïve Merkle tree construction of comparable capacity.

Consider an alternate Merkle forest instantiation $F := (r_1, r_2, \ldots, r_n)$ where each Merkle root r_i corresponds to a full binary Merkle tree T_i *with height* i, and where n is the current number of roots in F. As such, we define the capacity of F as $N := \sum_{i=1}^{n} 2^i = 2^{n+1} - 2$.

The Merkle forest approach below relies on two crucial observations. First, *any* approach based on balanced Merkle forests will have smaller tree heights than a single Merkle tree of comparable capacity (a single Merkle tree with capacity 2^{n+1} has height $n+1$). Second, we can leverage the fact that, with multiple Merkle trees, the authentication path of a credential in tree T_i will only change when other credentials are added to that *same* tree T_i. Modifications to any other tree $T_j \neq T_i$ in the forest F will only invalidate the user's forest-membership proof (which is a simple OR proof) but not their tree-membership proof, allowing reuse of the latter proof via LinkG16.

Growing Merkle forests. Assume without loss of generality that the user knows the representation of the broadcasted frontier updates, and (for simplicity of analysis alone) that issuance and authentication path updates are left unbatched. We also assume, as before, that credentials are not modified in-place once added. We first define a *forest-packing algorithm* describing how a list manager or issuer adds newly issued credentials to Merkle forest F:

0. Initialize F with $n_0 > 0$ sparse binary Merkle trees, where each tree T_i has height i and all leaves/entries are currently empty (i.e., the hash of nil). That is, F initially contains $n := n_0$ empty trees.
1. Populate the entries in F with newly issued credentials as desired (e.g., from left to right, starting with the smallest available tree) until F is full.
2. Compose all now-filled trees $T_1, \ldots, T_n$ into a single tree T_{n+1} with height $n+1$, arranging each T_i into a sequence of right subtrees. Initialize the remaining 2 leftmost entries in T_{n+1} as empty and add r_{n+1} to F.

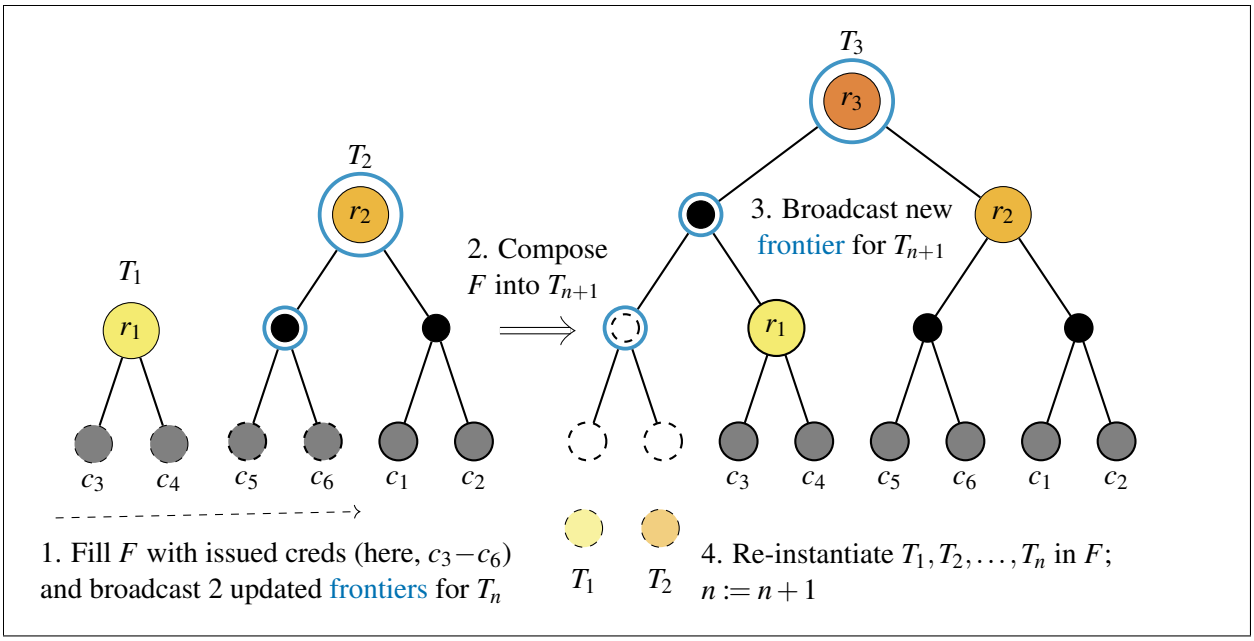


Figure 4.7: A single iteration of the forest-packing algorithm with $n_0 = 1$ trees in F at initialization and $n = 2$ trees as of Step 1. Dashed nodes represent hashes of empty entries or a root of an empty subtree; grey leaf nodes are hashed entries (i.e., issued credentials); blue-circled nodes are frontier nodes. Note that, invariably, the forest with n_0 trees requires no frontier because of Step 0 in the witness-update algorithm.

3. Immediately, and after any future addition to the 2 leftmost entries from Step 1, broadcast the *frontier* to T_{n+1} containing its "leftmost" intermediate nodes and the root r_{n+1} that updated.

4. Re-instantiate $T_1, \ldots, T_n$ and their corresponding roots in F as empty, then repeat from Step 1 with $n := n+1$.

Updating authentication paths. For clarity, we also define a *witness-update algorithm* which describes how a user tracks changes to their issued credential's authentication path θ as F grows. Suppose that the user can only reliably track the current *roots* of each Merkle tree F, and does so out-of-band.[19] Then:

0. The user first requests to issue a credential. If valid, the issuer adds it to some Merkle tree T_i in F, allowing the user to obtain their initial witness/authentication path θ. *Initially*, a user must track updates to θ as appropriate until T_i is full.

[19] Even for massive Merkle forests of capacity, say, $2^{33} - 2 = 8.6$ billion credentials, this only requires tracking and storing 32 hash values (i.e. for 256-bit digests, 1 KB).

1. Once the tree T_i holding the credential is filled, the user waits until all trees in F are completely filled; only the roots in $F \setminus \{T_i\}$ may change after this point, so θ remains a valid Merkle path to the credential.
2. The user listens for a broadcast indicating that their credential has been composed into the new tree T_{n+1} (see above). They scan in a single pass for the relevant frontier node(s) which, in conjunction with the old θ and (now sub-)roots in F, form the new authentication path for T_{n+1}; the user updates θ accordingly and marks that F has grown with $n := n+1$.
3. Thereafter, their credential's tree will invariably have 2 empty entries; listen for broadcast(s) indicating that an new credential changed the relevant frontier nodes, and update θ accordingly. Repeat from Step 1.

To see how a user (say, Alice) can correctly compute her current authentication path θ in this construction, consider the lifetime of her issued credential. Without loss of generality, consider her credential to be initially committed to by the i-th Merkle tree T_i (of n) in the Merkle forest F. Alice is initially responsible for updating or requesting θ herself, doing so until the tree T_i that her credential resides in is full. Thereafter, her credential will be composed into the largest tree of the forest, and after this point it suffices for Alice to track changes to θ using frontier updates alone to keep her membership proof updated. More precisely, any user with a credential in the (now sub-)trees $T_1, \ldots, T_n$ can update its authentication path θ to be current for the tree of height $n+1$ by scanning for the frontier nodes in the Merkle path which compute from the subroot (i.e., the old r_i) up to the current Merkle root r_{n+1}, also considering the roots of the subtrees that were just composed into T_{n+1}. By repeating this process for updates to this largest tree (invariantly, at most two more times), it follows by induction on the number of trees in the Merkle forest F that the user can correctly update to a valid authentication path θ across the lifetime of the credential, and can do so by using only broadcasted frontier nodes and the forest's roots after the initial tree is filled.

Comparison to Merkle tree. One of the primary benefits of this Merkle forest approach is that it allows users to reuse their authentication path more often than the approaches using fixed-height Merkle trees, greatly reducing how often a user must recompute expensive tree membership proofs.

For all users of older credentials which have already been composed into the largest tree in F, each user will only have to update θ at most three times per new tree. As such, this only requires users of older credentials to perform $O(\log N)$ updates (excluding updates while in the initial tree; see below) for the remaining lifetime of the Merkle forest. Furthermore, broadcasting a global frontier eliminates the need for users of older credentials to interact with other parties to update authentication paths, providing strong privacy guarantees beyond the short-term.

Even for users of newly issued credentials whose credentials are still in their initial tree T_i, this Merkle forest approach requires fewer authentication path updates compared to a standard Merkle tree of near-identical capacity (i.e. $T_1, \ldots, T_n$ vs. T_{n+1}). In the worst case, where a credential is initially added to tree T_n, this requires a credential to update its authentication path at most 2^n times as other added credentials update its Merkle root (compared to 2^{n+1} for a standard Merkle tree). However, depending on how the issuer distributes the newly issued credentials across the variable-height trees in F, users might expect a reduction from the worst case of 2^n initial updates. For example, assuming an average case where a credential has a uniform probability of being added to any given leaf node (i.e., average number of updates for each tree weighted by tree capacity in the forest), in expectation a user would need to perform $\sum_{i=1}^{n} \frac{2^i}{N} \cdot 2^{i-1} = \frac{1}{3}(2^n + 1)$authentication path updates in the initial tree. As such, the Merkle forest approach requires 2–3$\times$ fewer initial updates (before batching) compared to a single Merkle tree of near-identical size.

And, while the reduced number of updates comes at the expense of increased communication costs to broadcast to all users, the tradeoff is still competitive compared to standard Merkle trees in some cases. Since at most three frontiers will be broadcasted for each additional tree T_{i+1} added to a Merkle forest F as it grows from n_0 to $n \geq n_0$ trees, and each frontier contains i nodes, the system must broadcast at most $\sum_{i=n_0+1}^{n} 3i = \frac{3}{2}(n - n_0)(n + n_0 + 1)$ frontier nodes to each user across its lifetime. Concretely, assuming 256-bit hash digests and that the Merkle trees in the forest start with height $n_0 = 1$ and end with height $n := 32$ (i.e., 8.6 billion credentials), processing the frontier would require broadcasting at most 51 KB to each user across the lifetime of the system.

Overall, the issuer or list manager must broadcast a small-constant $O(\log^2 N)$ number of nodes

to each user, compared to sending $O(\log N)$ nodes per user request as with the naïve Merkle tree approach. Furthermore, since only 3 broadcasts are required per tree with increasing height i, the rate at which broadcasts occur will decrease exponentially as F grows.

Distributing newly issued credentials. We emphasize that, if using the Merkle forest packing approach, the issuer or list manager has power to add credentials in an arbitrary manner to any tree $T_1, \ldots, T_n$ of variable height in F by default. This asymmetry in initial tree sizes has important implications on computational fairness and efficiency for users wishing to mitigate proof re-computations for their newly issued credentials, even outside an adversarial setting. We leave further discussion on the implications and importance of asymmetry in Merkle forests to future work.

14 Appendix: Revocation of issued credentials

In certain situations, some party might wish to revoke an already-issued credential, invalidating its authentication path θ using mechanisms beyond simple checks for, e.g., expiry. A naïve approach for credential revocation would be to replace its leaf node in the Merkle tree with a hash of nil, as was done with RevokeCred for fixed-height Merkle forests in Figure 4.4. However, in-place revocation such as this comes with significant drawbacks. For example, naïve revocation would cause many of the optimizations mentioned before (especially batched requests and the forest-packing algorithm) to longer work. However, other revocation mechanisms for Merkle trees may allow `zk-creds` to support both revocation and the Merkle tree optimizations described above.

On the surface, anonymous credentials and X.509 certificate often employ similar methods for managing membership on a list. As such, one might be tempted to use the idea of certificate revocation lists (CRLs), as defined in RFC 5280 [CSF+08], for *anonymous* credentials as well. This would avoid many issues with deleting the credentials in-place on the membership list. However, there are important consequences to using CRL-based methods for revocation.

Perhaps most importantly, for revocation to be possible RFC 5280 requires certificates to have some form of persistent identifier; this is not possible for all types of anonymous credentials,

however. More precisely, a CRL-compatible anonymous credential must contain a unique, persistent, and privacy-preserving identifier id (e.g., a credential hash or committed attribute). Additionally, either the user must specifically request for their credential to be revoked, or a verifier/auditor must be able to de-anonymize a misused credential (e.g. through violation of cloning resistance) to revoke it.

A CRL-based revocation approach for anonymous credentials would likely also come with many of the same practical issues as with CRLs for certificates. For example, strict granularity and consistency of updates is required for all parties to maintain a secure and synchronized revocation list. Additionally, trusting a party to manage a credential revocation list may re-introduce trust assumptions that some central authorities' signing keys are kept secret and are used honestly (see [CSF+08], Section 3.3). We leave a more thorough investigation of `zk-creds`-compatible revocation techniques to future work.

15 Appendix: LinkG16

We now describe and prove the security of the LinkG16 proof system.

15.1 Goal

The purpose of LinkG16 is to show that k Groth16 proofs over heterogeneous circuits $\mathsf{crs}_1, \ldots, \mathsf{crs}_k$ all share the same first t public inputs $\{a_0, \ldots, a_{t-1}\}$ *without* revealing the inputs. That is, given k Groth16 proofs $\pi_1, \ldots, \pi_k$, we wish to construct a zero-knowledge proof of the following relation:

$$R_{\mathsf{linkg16}} = \left\{ \begin{array}{c} (\{\mathsf{crs}_i, \hat{S}_i\}_{i=1}^{k}; \{a_j\}_{j=0}^{t-1}, \{\pi_i\}_{i=1}^{k}) : \\ \bigwedge_{i=1}^{k} \mathsf{G16.Verify}(\mathsf{crs}_i, \pi_i, \hat{S}_i + \sum_{j=0}^{t-1} a_j W_j^{(i)}) \end{array} \right\}$$

where $W_j^{(i)}$ represents the wire W_j in crs_i, and $\hat{S}_i = \sum_{j=t}^{\ell} a_j W_j^{(i)}$ is the verifier-known prepared input for the i-th proof.

15.2 Construction

We define the new proof system below.

$\mathsf{LinkG16.Link}(\{a_j\}_{j=0}^{t-1}, \{\mathsf{crs}_i, \pi_i\}_{i=1}^{k}) \to \pi_{\mathsf{link}}$ Sample values $z_1, \ldots, z_k \leftarrow\!\!\$\, \mathbb{F}$ for blinding. For each i, commit to the shared inputs, $U_i := z_i\,[\delta]_1^{(i)} + \sum_{j=0}^{t-1} a_j W_j^{(i)}$. Let π_{eqwire} be an EqWire discrete-log equality proof (described in Section 16) that the U_i commit to the same a_j values,

$$R_{\mathsf{eqwire}} = \left\{ \begin{array}{c} (\{U_i, \mathsf{crs}_i\}_{i=1}^{k}; \{a_j\}_{j=0}^{t-1}, \{z_i\}_{i=1}^{k}) : \\ \bigwedge_{i=1}^{k} U_i = z_i\,[\delta]_1^{(i)} + \sum_{j=0}^{t-1} a_j W_j^{(i)} \end{array} \right\}.$$

Rerandomize the underlying proofs in place, $\pi_i := \mathsf{G16.Rerand}(\mathsf{crs}_i, \pi_i)$, then blind the proofs,

$$\pi_i' := (A_i, B_i, C_i - [z_i]_1).$$

The final output is

$$\pi_{\mathsf{link}} := (\pi_{\mathsf{eqwire}}, \{U_i, \pi_i'\}_{i=1}^{k}).$$

$\mathsf{LinkG16.LinkVerify}(\pi_{\mathsf{link}}, \{\mathsf{crs}_i, \hat{S}_i\}_{i=1}^{k}) \to \{0,1\}$ Check π_{eqwire} using EqWire.Verify. Then unpack each π_i' into (A_i', B_i', C_i'). For each $i = 1, \ldots, k$, check

$$e(A_i', B_i') \stackrel{?}{=} e([\alpha]_1^{(i)}, [\beta]_2^{(i)}) \cdot e(C_i', [\delta]_2^{(i)}) \cdot e(U_i + \hat{S}_i, H).$$

where $\hat{S}_i$ is the Groth16 prepared public input for circuit i.

15.3 Proofs

Theorem 10 (Correctness)**.** *If* G16.Prove *and* LinkG16.Link *are honestly computed, then* LinkG16.LinkVerify *succeeds.*

Proof. We show that the LinkVerify equation above holds for all i. For legibility, we omit the index i in the proof. Suppose π_{link} is computed honestly, i.e., that all U' and (A', B', C') are well-formed

and that the underlying Groth16 verification equations holds on the corresponding (A,B,C). First, we note that, since C' and U were computed honestly,

$$\begin{aligned}
&e(C',[\delta]_2)\cdot e(U,H)\\
&= e(C-[z]_1,[\delta]_2)\cdot e(\textstyle\sum a_j W_j + z[\delta]_1, H) && \text{Expanding}\\
&= e(C,[\delta]_2)\cdot e(-[z]_1,[\delta]_2)\cdot e(z[\delta]_1,H)\cdot e(\textstyle\sum a_j W_j,H) && \text{Bilinearity}\\
&= e(C,[\delta]_2)\cdot e(\textstyle\sum a_j W_j,H). && \text{Bilinearity}
\end{aligned}$$

Using this and the fact that $A'=A$ and $B'=B$, we see that the LinkVerify equation

$$e(A',B') = e([\alpha]_1,[\beta]_2)\cdot e(C',[\delta]_2)\cdot e(U+\hat{S},H)$$

holds if and only if

$$\begin{aligned}
&e(A,B)\\
&= e([\alpha]_1,[\beta]_2)\cdot e(C',[\delta]_2)\cdot e(U+\hat{S},H) && \text{Subst. } A',B'\\
&= e([\alpha]_1,[\beta]_2)\cdot e(C',[\delta]_2)\cdot e(U,H)\cdot e(\hat{S},H) && \text{Bilinearity}\\
&= e([\alpha]_1,[\beta]_2)\cdot e(C,[\delta]_2)\cdot e(\textstyle\sum a_j W_j,H)\cdot e(\hat{S},H) && \text{Above identity}\\
&= e([\alpha]_1,[\beta]_2)\cdot e(C,[\delta]_2)\cdot e(\hat{S}+\textstyle\sum a_j W_j,H) && \text{Bilinearity}
\end{aligned}$$

which is precisely the verification equation for the i-th underlying Groth16 instance. Since this equation holds by assumption, LinkVerify succeeds. □

Theorem 11. LinkG16 *is perfect HVZK.*

Proof. We define a simulator $\mathsf{Sim}_{\mathsf{linkg16}}$ with access to Groth16 trapdoors $\tau_1,\ldots,\tau_k$ as follows. For each i, sample $\bar{A}'_i,\bar{B}'_i,\bar{U}_i \leftarrow\!\!\$\ \mathbb{F}$ uniformly. Use the trapdoor τ_i to compute $C'_i \in \mathbb{G}_1$ as the unique group element which satisfies the i-th LinkVerify equation with respect to crs_i and public inputs[20]

[20]How does the verifier know a_j (for $j \geq t$) if it was only given the prepared input $\hat{S}$? It is merely for brevity that

$\{a_j\}_{j=t}^{\ell}$. Concretely,

$$\bar{C}_i' := \frac{\bar{A}_i'\bar{B}_i' - \alpha^{(i)}\beta^{(i)} - \bar{U}_i - \sum_{j=t}^{\ell} a_j W_j^{(i)}}{\delta^{(i)}}.$$

For all i, let $\pi_i' := (\left[\bar{A}_i\right]_1, [\bar{B}_i]_2, \left[\bar{C}_i\right]_1)$ and $U_i := [\bar{U}_i]_1$. Finally, let

$$\pi_{\mathsf{eqwire}} \leftarrow \mathsf{Sim}_{\mathsf{eqwire}}(\{\mathsf{crs}_i, U_i\}_{i=1}^{k}).$$

The output of $\mathsf{Sim}_{\mathsf{linkg16}}$ is $(\pi_{\mathsf{eqwire}}, \{U_i, \pi_i'\}_{i=1}^{k})$.

This is indistinguishable from the real world protocol. In the real world: each U_i is uniformly distributed due to the blinding values z_i; A_i', B_i' are uniformly distributed by the Groth16 proof procedure; and each C_i' is the unique group element which satisfies the i-th LinkVerify equation. Lastly, the simulated π_{eqwire} is indistinguishable from an honestly generated one due to the perfect HVZK of the EqWire protocol. □

Theorem 12. LinkG16 *is knowledge-sound.*

Proof. We define an extractor $\mathsf{E}_{\mathsf{linkg16}}$, aborting on verification error, as follows. By knowledge soundness of EqWire there exists an extractor $\mathsf{E}_{\mathsf{eqwire}}$ which extracts $\{a_j\}_{j=0}^{t-1}, \{z_i\}_{i=1}^{k}$ such that $U_i = z_i[\delta]_1^{(i)} + \sum a_j W_j^{(i)}$. For each $i = 1, \ldots, k$, $\mathsf{E}_{\mathsf{linkg16}}$ then reconstructs the underlying Groth16 proof

$$\pi_i = (A_i', B_i', C_i' + [z_i]_1).$$

$\mathsf{E}_{\mathsf{linkg16}}$ outputs $(\{a_j\}_{j=0}^{t-1}, \{\pi_i\}_{i=1}^{k})$. Since $\mathsf{E}_{\mathsf{linkg16}}$ did not abort, it is the case that, for each i,

$$\begin{aligned}
& e(A_i', B_i') \\
&= e([\alpha]_1^{(i)}, [\beta]_2^{(i)}) \cdot e(C_i', [\delta]_2^{(i)}) \cdot e(U_i + \hat{S}_i, H) && \text{LinkVerify} \\
&= e([\alpha]_1^{(i)}, [\beta]_2^{(i)}) \cdot e(C_i', [\delta]_2^{(i)}) \cdot e(z_i[\delta]_1^{(i)} + \sum a_j W_j^{(i)} + \hat{S}_i, H) && \mathsf{E}_{\mathsf{eqwire}}\text{ output} \\
&= e([\alpha]_1^{(i)}, [\beta]_2^{(i)}) \cdot e(C_i' + [z_i]_1, [\delta]_2^{(i)}) \cdot e(\hat{S}_i + \sum a_j W_j^{(i)}, H) && \text{Bilinearity}
\end{aligned}$$

LinkVerify is written to take $\hat{S}$. The verifier always knows the prepared input's constituent a_j values.

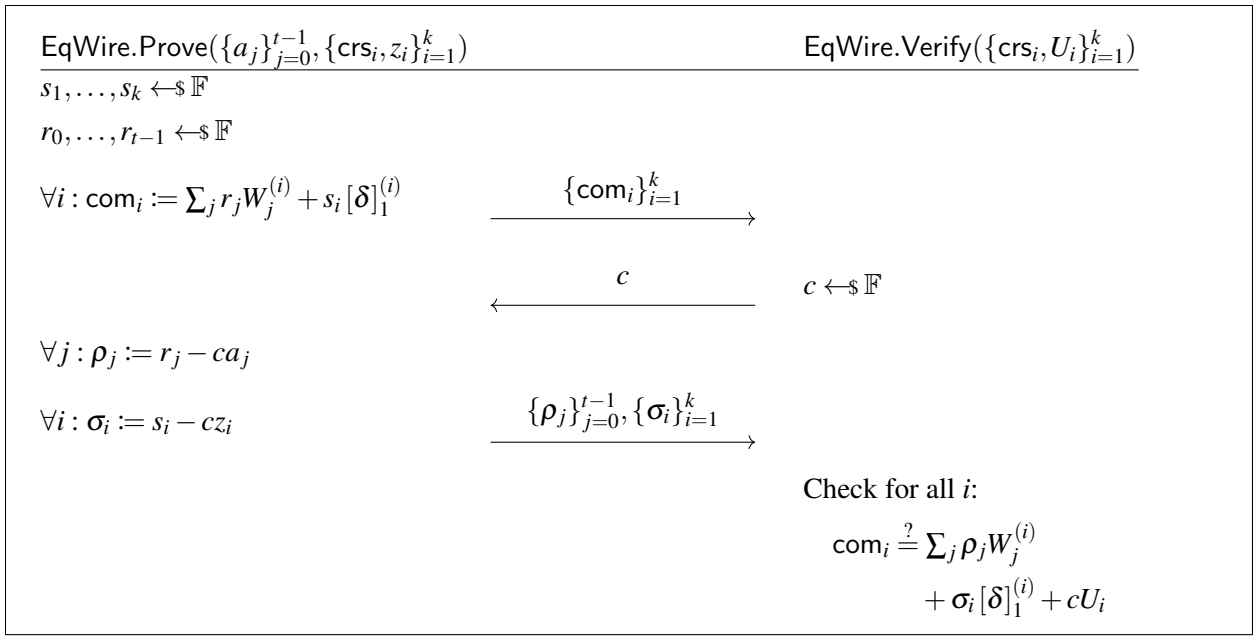


Figure 4.8: The EqWire protocol

which is precisely the verification equation for π_i. □

16 Appendix: EqWire

We now define and prove secure a proof system for the discrete-logarithm equality relation,

$$R_{\mathsf{eqwire}} = \left\{ \begin{array}{l} (\{U_i, \mathsf{crs}_i\}_{i=1}^{k}; \{a_j\}_{j=0}^{t-1}, \{z_i\}_{i=1}^{k}) : \\ \quad \bigwedge_{i=1}^{k} U_i = z_i [\delta]_1^{(i)} + \sum_{j=0}^{t-1} a_j W_j^{(i)} \end{array} \right\}.$$

The proof system is instantiated using a proof framework due to Camenisch and Stadler [CS97]. Concretely, it is the Fiat-Shamir transform of the protocol described in Figure 4.8.

Proofs

Theorem 13. *The* EqWire *protocol is knowledge-sound.*

Proof. We define extraction in the usual way for Camenisch-Stadtler sigma protocols. Let $\mathsf{E}_{\mathsf{eqwire}}$

be our extractor, aborting on verification failure. The extractor receives the commitments, and then picks challenge $c \leftarrow\$ \mathbb{F}$. It sends c and receives $\{\rho_j, \sigma_i\}_{i,j}$. The extractor then rewinds to pick a fresh $c' \leftarrow\$ \mathbb{F}$. It sends c' and receives $\{\rho'_j, \sigma'_i\}_{i,j}$. For all i and j, the extractor computes

$$a_j := \frac{\rho_j - \rho'_j}{c' - c} \qquad\qquad z_i := \frac{\sigma_i - \sigma'_i}{c' - c}$$

and outputs $(\{a_j\}_{j=0}^{t-1}, \{z_i\}_{i=1}^{k})$. Since the extractor did not abort, i.e., both runs passed verification, and the commitments did not change, it is the case that $U_i = z_i [\delta]_1^{(i)} + \sum_j a_j W_j^{(i)}$ for all i. □

Theorem 14. *The* EqWire *protocol is perfect HVZK.*

Proof. We define a simulator as follows: sample c and all σ_i, ρ_j uniformly from $\mathbb{F}$; for all i, compute $\mathsf{com}_i := \sum_j \rho_j W_j^{(i)} + \sigma_i [\delta]_1^{(i)} + c U_i$; output $(c, \{\mathsf{com}_i, \rho_j, \sigma_i\}_{i,j})$.

This is perfectly indistinguishable from a real transcript: all σ_i and ρ_j are uniform iid since they are blinded by s_i and r_j, respectively; c is uniform and independent by definition of honest-verifier; and all com_i are uniquely determined by these values. In the simulator, each σ_i and ρ_j is uniform iid by construction, and com_i is uniquely determined by these values. □

17 Appendix: Instantiation with Pedersen hashing

In Table 4.5 we give benchmarks using a different collision resistant hash for the Merkle tree. Specifically we use a Pedersen hash over the Jubjub curve.

18 Appendix: CRS sizes of evaluated circuits

We list in Table 4.6 the sizes of the proving and verifying keys for all the circuits evaluated in Sections 6 and 7.

Client-opt. (C)	ShowCred			VerifyShow			Proof Size	
Server-opt. (S)	C	C (full)	S	C	S	S (batch)	C	S
Simple Possession	5ms	784ms	699ms	4ms	1.5ms	1.8 verifs/ms	744B	192B
Expiry	98ms	875ms	796ms	6ms			1064B	
Linkable Show	104ms	879ms	837ms					
Rate Limiting	117ms	895ms	817ms					
Clone Resistance	139ms	916ms	812ms					

Table 4.5: Gadget microbenchmarks using Pedersen hashes for two versions of `zk-creds`. Membership proofs are done on a Merkle forest of size 2^{31} (tree height = 24, #trees = 2^8). The first configuration (C) minimizes client-side proving cost; the second configuration (S) maximizes server throughput. ShowCred (full) gives the cost of including membership recomputation while showing a credential. VerifyShow (batch) gives throughput for verifying a set of 100 proofs. We emphasize that all verification numbers are *single-threaded*, allowing for efficient concurrent processing.

	Proving key (MB)	Verifying key (KB)
Forest membership	0.3	12.8
Tree membership	10.1	0.6
Expiry	1.5	0.6
Linkable show	0.4	0.6
Rate limiting	1.3	0.8
Clone resistance	1.2	0.8
Age-restricted vid.	2.8	0.9
Entering a bar	2.5	0.6

Table 4.6: CRS sizes for our evaluated circuits

Chapter 5

zk-promises: Making Zero-Knowledge Objects Accept the Call, and Applications to Anonymous Reputation and Access Control

Work coauthored with Maurice Shih, Harikesh Kailad, and Ian Miers, and submitted to CCS 2024 [SRM24].

1 Introduction

Apple's private relay [App21], and the proliferation of VPNs deployed to hundreds of millions of users worldwide, highlight the need for accountability of anonymous clients. Take, as a simple example, banning an anonymous user for a Wikipedia edit. The client's account is a single bit of state—`isBanned`—and the server must update it—calling `setBanned()`— after detecting the vandalistic edit. For anonymity, the server cannot store the user account, requiring us to "outsource"' it to the client while somehow ensuring integrity, confidentiality, and obliviousness. User accounts, of course, are often more complex state machines than our example.

Inspired we explore a fundamental technical question with applications in federated reputation systems, stateful access control, and cryptocurrency: How can multiple parties interact with a private state machine stored by an anonymous user without compromising the state or leaking

access patterns?

The challenge of third party updates to private state, asynchronous negative feedback, and evading accountability. In our setting, some secret state is stored by an untrusted user. Our goal is to let third parties update this state while ensuring (1) state confidentiality, i.e., that the third party learns nothing about the state; (2) obliviousness in update and access, i.e., the third party cannot identify the user by linking accesses or updates; and (3) integrity, i.e., that updates conform to validation logic and access control. The core challenge in this setting is that state updates are asynchronous—the user is not guaranteed to be online when the state update is made. In addition, depending on application, users may have ample motive and opportunity to avoid an update. We refer to this as the *asynchronous negative feedback problem*.

Returning to our Wikipedia example: a user who makes an edit must show that their account is in a valid state in order to authenticate. Various anonymous credential systems support this. The problem comes when we need to update this state—e.g., when moderators score the quality of a Wikipedia edit or ban the user's account—after the user's session has ended. There is no direct way to update the user's state, since the user is anonymous and the location of the state is thus unknown. Finally, there is no incentive for the user to accept an indirect update, e.g., some signed statement posted on a server. This is in contrast to, e.g., an anonymous payment system, where users are financially incentivized to receive updates that increase their account balance.

Workarounds to the asynchronous negative feedback problem may be possible for simple cases, such as some escrow system to provide sufficient incentive to apply updates, but these offer limited functionality and are challenging to apply to complex logic. Since there is no one-size-fits-all approach, it is necessary to have a programmable protocol, The practical success of such a scheme requires flexibility.

For example, for anonymous banning, a forum moderator may have to respond differently to a user posting hate speech versus spam or explicit content. They may implement a three-strikes policy for specific infractions, or have a probation system when a second infraction triggers consequences.

Emulating trusted servers and Sybil resistance. For open enrollment platforms, Sybil resistance

consists of raising the resource costs for creating and using a Sybil account, e.g., by requiring a CAPTCHA during registration. While imperfect—a motivated attacker may pay the cost—it is essential to many systems. Bans, after all, are not effective if users can cheaply make new accounts.

Non-privacy-preserving systems, however, can go much further than privacy-preserving ones to raise the costs of Sybil attacks without placing undue costs on legitimate users. Services like StackExchange [Atw09] define arbitrary server-side logic tying allowed behavior to the account's history, e.g., rate-limiting the number of posts and, separately, the number of (possibly spam) links in posted question for new accounts, or gating moderator privileges behind a sequence of achievements. This substantially raises the cost of a Sybil attack, reducing the utility of a new account and requiring the attacker to commit additional resources to get an account in good standing. Using zk-promises, we can realize this approach for privacy preserving protocols without a trusted server by "outsourcing" this type of reputation and rate limiting logic to the client.

A promising but limited starting point. In PEREA [TAKS08], Tsang et al. give an elegant approach to the simplest version of the anonymous state update problem: anonymous blocklisting (née *blacklisting*). In PEREA, each post is accompanied by a pseudorandom ticket. The ticket is placed on a blocklist to ban the user. During authentication users prove in zero-knowledge that none of their tickets appear on a blocklist. With this protocol, PEREA avoids the pitfalls of previous approaches, such as trusted de-anonymization authorities or high initial computational costs [TAKS10].

Unfortunately, PEREA and successors [AK12, XF14], have substantial performance and functionality problems. Users are effectively stateless: a user's only private state is a fixed pseudorandom function key for generating tickets. Each authentication requires that a user regenerate a fixed k tickets from a global sliding *detection window* of size w and show they are unbanned.[1] This has two consequences:

1. Limited functionality. PEREA only supports checking if there is a ticket in the window.

[1] Even with a zero-knowledge proof which hides how many tickets the user checked—something PEREA and derivatives lack—there is no way to trust the user is honestly counting how many tickets it has open.

In subsequent work [AK12, XF14], tickets contain a single integer rating, users prove the sum of tickets in the window is below a threshold. But this does not provide the complex, programmable logic or state available in deployed moderation systems [ets].

2. Every user pays worst-case computation costs. For every authentication, regardless of the number of tickets a user used, they must perform $O(\omega_{\mathsf{auth}} \times w)$ work, where ω_{auth} is the rate at which the most active user uses the service and therefore produces tickets. In essence, every user is burdened with the workload of the worst-case user.

Our approach: zk-objects. We start, somewhat surprisingly, by repurposing techniques developed for privacy-preserving cryptocurrency payments and smart contracts. Starting with Zerocash [BCG+14] and commercial derivatives like Zcash, TornadoCash, and Railgun [HBHW21, PSS, rai], through to academic [KMS+16, BCG+20, XCZ+22] and commercial systems [Wil, ale] on privacy-preserving smart contract systems, there is a robust line of work which builds efficient protocols for privately manipulating Turing-complete state machines. We give a new view of this style of computation, which we refer to as the *zk-object model*.

In the zk-object model, method calls manipulate the object and return outputs. For integrity and replay/forking prevention, every object—even from different owners—is controlled by a global bulletin board or trusted server, which verifies zero-knowledge proofs of update correctness. For confidentiality, the bulletin board stores only cryptographic commitments to the objects. To hide access patterns and ensure anonymity, objects are not accessed directly or mutated in-place. Instead, updates are made through an oblivious copy-on-write approach, where a fresh commitment to the updated object is appended to the bulletin board along with the zero-knowledge proof the update results from a valid method call on the previous version of the object. To ensure obliviousness, rather than identifying the old committed object version directly, the proof shows that a secret previous version exists by, e.g., checking membership in a Merkle tree built over all entries on the bulletin board. The proof also reveals a *serial number* (a.k.a. a *nullifier* or *nonce*) to prevent replay of the previous object version in subsequent updates (e.g., a *double-spend* in payment systems). When these zero-knowledge proofs are instantiated with zkSNARKs, we can achieve Turing-complete

functionality and efficient verification by the bulletin board, giving us zk-objects with integrity, confidentiality, obliviousness, and atomicity.

Our contribution. We design, implement, and benchmark zk-promises, a protocol that augments the zk-object model with asynchronous oblivious callbacks.With zk-promises, we then build an example application for programmable anonymous blocking and moderation, complete with user reputation and more complex functionality such as reputation-dependent rate limiting.

Our approach is inspired by PEREA's ticket approach for simple bans, but overcomes a few key challenges to offer drastically improved performance, and generalize to arbitrary Turing-complete state machines. First, to achieve our performance requirements, we must describe a fixed-size state machine that allows users to incrementally iterate through an unbounded list of open callbacks, using zk-friendly data structures, while revealing nothing to the server and preventing skipping callbacks.[2] Second, we must support multiple callbacks and callers, while ensuring confidentiality of callback arguments, the authenticity of the caller, and the integrity (i.e., non-evasion) of the callback and the object itself. Since neither the client nor the caller are trusted,[3] we must carefully design a protocol that avoids malicious inputs, fault injections, and key misbinding issues. With these resolved, we achieve a scheme which permits arbitrary callbacks on user-defined zk-objects.

To summarize, in this paper we design, build, and benchmark zk-promises, which:

1. supports Turing-complete state machines and programmable logic for zk-objects with arbitrary callbacks;

2. adds asynchronous callbacks to the line of privacy-preserving payment and smart contract systems developed in [BCG+14, KMS+16, BCG+20];

3. yields an anonymous reputation and blocking system with significant performance and feature improvements over the state of the art, such as multi-dimensional reputation with support for arbitrary state-dependent logic such as probationary periods and reputation-dependent rate

[2] Naïvely using a zkSNARK for every such statement is insufficient: the verifier learns the length of the statement (and the statement itself), exposing how many tickets the user has.

[3] The client may evade calls and the caller may, e.g., provide a garbage callback to deadlock the client.

limiting for improved Sybil resistance; and

4. is concretely efficient, offering server verification times of less than 3ms and client-side authentication times of 1–10s in realistic scenarios.

1.1 Related work

There is a very large body of reputation and anonymous blocklisting schemes. Here, we constrain ourselves to systems that are anonymous (as opposed to relying on pseudonyms), do not need trusted third parties to de-anonymize and ban users, and allow asynchronous negative feedback. Token-based blocklisting schemes in the line of BLAC [TAKS10, AKS12, RMM22] require the user to do linear work in the number of bans/downvotes issued. Other schemes [TAKS08, AK12, XF14, MC23] have *global halting*, forcing a moderator to assess the oldest posts before they can move on to more recent posts. Others systems like k-TAA [TFS04] rely on one-time-use unlinkable tickets that users receive a fixed number of every epoch. To prove they are not banned, the user gives a ticket to the service provider. One difficulty with the scheme is synchronicity: a user must be online at the beginning of every epoch to receive their new tickets.

Finally, a common shortcoming of all of the above schemes is lack of flexibility. Reputation in all of these schemes is represented as a single integer, and the update function is represented, at best, by a linear function over recent ratings.

2 Preliminaries

2.1 Zero-knowledge proofs

A noninteractive zero-knowledge proof of knowledge (NIZKPok) is a representation of a statement "I know w such that $P(x,w)$" where x is the *instance* (or *public input*), w is the *witness*, and P is some efficiently computable predicate. We use the relation notation $R = \{(x,w) : P(x,w)\}$ to represent the set of such statements. A NIZKPoK is a tuple of algorithms:

$\mathsf{Setup}(P) \to \mathsf{srs}$ Receives a description of the predicate P and returns a *structured reference string* which will be used for proving and verifying.

$\mathsf{Prove}(\mathsf{srs}, x, w) \to \pi$ Computes a proof that $(x, w) \in R$.

$\mathsf{Verify}(\mathsf{srs}, x, \pi)$ Verifies π with respect to x, i.e., verifies that there exists a w such that $(x, w) \in R$.

2.2 zk-objects

We now describe the components of the zk-objects model.

Object. An object contains arbitrary state, e.g, payment account balances, reputation, ban status, account creation date, etc. In addition, an object contains a *serial number* (or *nullifier* or *nonce*)—a random string[4] which is revealed when the object is updated. By checking if transaction updating object contains a serial number that has already been revealed, we prevent stale object states from being replayed.

Object bulletin board. Objects must be stored in a way that permits a consistent global view of the same state. We thus require the existence of a global append-only log, called the *object bulletin board*. The bulletin board does not store objects directly, but rather commitments to objects. As a result, object owners need to store both the object's *opening*, i.e., its contents and the randomness used in its commitment. When discussing these constructions informally, we will refer to an object and its commitment interchangeably, assuming that all authorized parties have the commitment opening data.

The bulletin board needs to support efficient zero-knowledge set membership proofs. For the systems mentioned above, this is done via Merkle tree inside a zkSNARK. Proving membership of a leaf x in the Merkle tree with root r amounts to proving knowledge of an *authentication path*—a list of the siblings of x's ancestors—whose iterated hash equals r.

Updating an object. To update an object commitment obj, a user must submit a new object commitment obj' and a zero-knowledge proof π of the conjunction of the following statements. We

[4]Whether this is directly used as the nonce or materialized through a PRF and some key material depends on implementation.

let Φ be a predicate which determines validity of an update (e.g., a method mutating the object). The public inputs to the ZKP are a serial number sn, the (committed) object obj', and (the root of) the Merkle tree T.

1. $\mathsf{obj} \in T$
2. $\mathsf{sn} = \mathsf{obj.sn}$
3. $\Phi(\mathsf{obj}, \mathsf{obj}')$

To prevent forking, the bulletin board will only accept updates whose corresponding serial number has never been seen before.

2.3 Rerandomizable signature schemes

For our main construction, we will require a digital signature scheme with rerandomizable public keys. We say that $\Sigma = (\mathsf{Keygen}, \mathsf{SkToPk}, \mathsf{Sign}, \mathsf{Verify}, \mathsf{RerandPk}, \mathsf{RerandSk})$ is a rerandomizable signature scheme if (Keygen, SkToPk, Sign, Verify) is an ordinary signature scheme, and the following hold:

1. If pk is a valid public key, then $\mathsf{RerandPk}(\mathsf{pk}) \to (\mathsf{pk}', r)$ returns a fresh public key with randomness r such that pk' is computationally indistinguishable from a public key generated with Keygen.
2. If $(\mathsf{sk}, \mathsf{pk})$ is a valid keypair and (pk', r) is honestly generated by $\mathsf{RerandPk}(\mathsf{pk})$, then $\mathsf{RerandSk}(\mathsf{sk}, r) \to \mathsf{sk}'$ returns a secret key corresponding to pk', i.e., $\mathsf{SkToPk}(\mathsf{sk}') = \mathsf{pk}'$.

A simple example of rerandomizable signature schemes are EdDSA and ECDSA, where the keypair (x, P) can be rerandomized as (rx, rP), where r is a uniformly selected scalar. The resulting public key is perfectly indistinguishable from one generated with Keygen.

3 Overview

In this section, we introduce a simple example of zk-promises and its programming model for enforcing callbacks on arbitrary anonymous objects. We stress that the techniques described here have broader applications, as they permit arbitrary programmable logic and Turing-complete state machines with arbitrary callbacks.

We have an anonymous individual (the *user*) editing a page on Wikipedia (the *service provider*), where their account is a zk-object and callbacks are used to ban them or update their reputation. In this section, we give the programing model and, for ease of exposition, we phrase all actions imperatively, as if they were occurring on a mutable finite state machine. As we will see in the next section, every state transition and assertion in zk-promises is backed by a zero-knowledge proof on an append-only log.

Figure 5.1 gives pseudocode for our example `AnonUserRecord`. It stores both a multidimensional reputation score (e.g., containing separate scores for spam versus hateful content), state for a leaky-bucket-style rate limit on edits per day, and metadata to support callbacks. When making an edit to Wikipedia, users will prove they called `ShowAuthMakeCB` and provide the server with the generated callbacks. The server can later use these two callbacks to either update their reputation or ban them. Executing this method also proves to the server the following authorization checks passed.

1. The user's reputation, defined as the dot product of the their reputation with a public weight vector—their *projected reputation*—exceeds a specified threshold.

2. The user is within a rate limit for edits. To demonstrate the flexibility of zk-promises, we use a leaky bucket for the rate limit where the rate depends on the user's reputation.[5]

3. The user has, using the callback manager, recently scanned for all their open callbacks.

The reputation threshold and weighting are dynamically configurable via method arguments. As

[5]Edits require the bucket has remaining capacity. Once a user's bucket is full, they must wait for the bucket to partially drain.

these method arguments are public, the server can reject invalid parameters. Again, these features are picked to demonstrate the flexibility of zk-promises and its ability to handle arbitrary programs with branching, not just linear logic.

Note, the object defines its own access control. Users cannot alter data except by calling valid methods and, for example, `updateRep` method is restricted to Wikipedia.

3.1 Callbacks and their lifecycle

The core of zk-promises is a *callback*, a function which is used to modify the user record (here, `ban` and `updateRep`). Callbacks are first-class objects that can be passed around, transferred over the network, and stored, e.g., alongside an edit that is queued for moderation. Conceptually, they have a source that is making the callback (in our case Wikipedia moderators) and destination (the anonymous user).

Callback and authenticated origins. A key component of callbacks is an authentication origin. This prevents, e.g., a user from calling `updateRep` themselves to increase their reputation. Looking ahead, in our system, we assume the caller is identified by a public key, and the bulletin board verifies posted callbacks contain a signature under that key. Alternatively, callbacks could be generated by code executed by the bulletin board (e.g., in smart contracts), a trusted execution environment, or from the invocation of another zk-object.

Lifecycle and callback management. During its lifecycle, a callback is: *created* by the user, *called* by the service provider, and finally *ingested* by the user again.

Callbacks consist of pseudorandom tickets, which must be tracked to prevent evasion of asynchronous negative feedback. In zk-promises, we build a separate zk-object, called a callback manager, to create, track, and handle callbacks. As shown in Figure 5.2, it keeps a list `cbList` of all created callbacks. In order to get up to date, a user repeatedly calls `scanIncremental`, incrementally iterating over every callback on the list to check if it has been called or expired. When one full iteration of that loop completes, `lastFullScanTime` is set to the start time of the current loop. This implies a polling-like model where users assert they have scanned through all open

```
class AnonUserRecord:
  //Attributes are accessible via methods only
  private const instanceID
  private authorizedCaller // Pubkey of owner
  private cbMgr // For enforcing issued callbacks
  // Reputation can be a multidimensional vector of ints
  private rep = (0, 0, 0) // E.g, (hate,spam,quality)
  // We define a leaky bucket for rate limiting
  private leakyBucket = 0
  private lastPost = currentTime() // Used for leak rate

  AnonUserRecord(wikiPk): // Constructor
    instanceID = randomUID()
    this.cbMgr = CallbackManager(this)
    authorizedCaller = wikiPk // for access control.

  // Shows the user is allowed to edit and sends
  // callbacks to hold them accountable for that edit. As
  // arguments are public, servers can dynamically change
  // what weights, thresholds or cutoffs the client must
  // use.
  public ShowAuthorizedForEditandMakeCallbacks(
    curTime, repWeights, repThreshold, rateTreshold,
    cutOffTime
  ):
    this.leakyBucket -= this.calcDrain(curTime)
    // Authorization checks for edits.Checks reputation
    // is sufficient, edit is within rate limit, and that
    // all callbacks were handled as of cutOffTime.
    if this.rep.dotProd(repWeights) > repThreshold
     and this.leakyBucket < rateTreshold
     and this.cbMgr.lastFullScanTime >= cutOffTime:
      updateCb = this.cbMgr.makeCb(this.updateRep)
      banCb = this.cbMgr.makeCb(this.ban)
      this.leakyBucket++
      this.lastPost = curTime
      return (updateCb, banCb)
    else:
      return null // user isn't authorized

  public updateRep(caller, delta):
    // Users cannot update their own rep
    if caller == authorizedCaller:
      this.rep += delta

  public ban(): // no access check needed for ban
    this.rep = (-inf, -inf, -inf)

  private calcDrain(currentTime):
    if |rep| > 10 // threshold arbitrarily set to 10
      leakRate = 10/86400 // 10 auths per day in seconds
    else: // lower reputation gets 1 edit per day
      leakRate = 1/86400  //  1 auth per day in seconds
    elapsedTime = curTime - this.lastPost
    //cannot drain more than current bucket capacity.
    return min(leakRate * elapsedTime, this.leakyBucket)
```

Figure 5.1: Pseudocode for an anonymous user record with reputation, bans, and rate limiting.

```
class CallbackManager:
  private userObj               // Ptr to AnonForumUser
  private cbList = []           // List of callbacks
  private curCbIter = 0         // cbList iterator
  private scanStartTime = 0     // Current scan start
  private lastFullScanTime = 0 // Start of last full scan

  // Constructor
  CallbackManager(userObj)
    // Bind this cbManager to object
    this.userObj = userObj
    // Initialize the iterator
    this.curCbIter = this.cbList.begin()

  // Creates a new callback. Cannot run if
  // currently in the process of settling
  private makeCb(func):
    // This means we're not settling, ie not in a scan
    assert this.curCbIter == this.cbList.begin()
    ticket = randBytes(32);
    this.cbList.append(ticket, func)
    return ticket

  // Shows all callbacks were handled as of some time.
  public showSettledUpTo(cutoffTime):
    return this.lastFullScanTime >= cutoffTime

  // Incrementally scans the list of open callbacks
  // If a callback is on the bulletin board, its settled
  // If one is expired, its removed from the open list
  public scanIncremental(bulletin, curTime):
    if this.curCbIter == this.cbList.begin():
      // If starting a new scan, mark the start time
      this.scanStartTime = curTime

    // Take the next callback and see if its been called
    (ticket, func) = *this.curCbIter
    call = bulletin[ticket]
    //If the call is on the bulletin board, execute it
    if call not null:
      (caller, cbArgs) = call
      this.userObj.func(caller, cbArgs)
      // Mark the callback for deletion and move to next
      this.curCbIter.deleteAndIncr()
    else:
      // Move to next item in callback list
      this.curCbIter.incr()

    if this.curCbIter == this.cbList.end():
      // We have a new full complete scan, update times
      this.lastFullScanTime = this.scanStartTime
      this.curCbIter = this.cbList.begin()
```

Figure 5.2: Pseudocode for the callback manager, responsible for creating and settling callbacks.

callbacks as of some time tracked by the callback manager.

In our example, callbacks are represented by a random 32-byte *ticket* tik created by the user's zk-object invoking `makeCb` in `ShowAuthMakeCB`. The server, Wikipedia in our example, later calls the callback by placing the ticket on a bulletin board along with the method arguments. For example, a moderator, after reviewing the post, could decrement the anonymous users reputation by 3 via posting $(\mathsf{tik}, -3)$ to the bulletin board. The callback would then be ingested when the user scans through its open callbacks and is forced to run `updateRep(-2)`.[6]

3.2 Security objectives

Informally, the zk-object model has the following security properties:

Confidentiality An object's contents are not leaked beyond the amount a caller can deduce from their own calls.

Obliviousness An object update does not reveal which object was updated.

Integrity An object is only updated according to its programmed methods, and callbacks must be applied.

Atomicity There is one valid version of an object at a time and it cannot be rolled back (i.e., no forking or double spending).

For our example anonymous reputation application and the described functionality, these map to properties of existing schemes (see [TAKS10, TAKS08, AK12]). From confidentiality and obliviousness, we achieve anonymity, even after the user is banned (so called *backward anonymity* [HG11]). From integrity, we get the guarantee that our authorization logic is enforced and therefore get both authenticity—the server will only accept clients who meet the stated authorization checks, and *non-frameability*—no one other than the service provider can ban a user or alter their state.

[6] This must happen by the next time the user calls `ShowAuthMakeCB`, as that code requires `this.cbMgr.lastFullScanTime` $\geq$ `cutOffTime`

Readers will note, however, that this assumes the authorization logic describe in Figures 5.1 and 5.2 is correct, specifically, that the logic for enforcing callback evaluation works. Rather than defining generic functionality, we give a specific ideal functionality for anonymous reputation in our example application we construct in Section 6 as well as a proof sketch in Section 6.1.

4 Construction of zk-objects with callbacks

We now provide a more detailed description of zk-promises using primitives from the zk-object model. We begin by defining the algorithms for an extremely simple callback system. We then build features on top of it, including callback expiry, creation-calling unlinkability, and function argument authenticity and confidentiality. We will use these features to define the scheme we implement and benchmark in Section 7. A formal description of the final scheme can be found in Section 10.

4.1 Basic system

zk-promises is, at its core, a zk-object system, and thus carries with it all the same requirements (fresh serial numbers, persistent states in a bulletin board, zero-knowledge proofs over user predicates, etc.). In this section we augment zk-objects to support callbacks. In doing so, we ensure that all these base requirements are still met.

Data structures. There are two globally accessible data structures in zk-promises. $\mathsf{bb}_{\mathsf{obj}}$ stores every committed object obj on the bulletin board. $\mathsf{bb}_{\mathsf{cb}}$ stores every callback posted to the bulletin board. These structures permit efficient lookup of items inside zero-knowledge proofs. And, specifically for $\mathsf{bb}_{\mathsf{cb}}$, we must also support efficient non-membership proofs. Non-membership requires the bulletin board operator to commit to the complement of $\mathsf{bb}_{\mathsf{cb}}$.[7] For signature-backed bulletin boards, this requires a *bulletin board rollover*—a full recomputation of the complement set—whenever $\mathsf{bb}_{\mathsf{cb}}$ changes.

[7] Where the complement is represented as a set of ranges $[a,b)$ which partition the complement of the ticket space $\{0,1\}^{256} \setminus \mathsf{bb}_{\mathsf{cb}}$. The size of the representation of this set is roughly the size of $\mathsf{bb}_{\mathsf{cb}}$.

zk-promises permits the bulletin board manager(s) to represent passing time in whichever way they choose. This can mean a centralized server publishing a new bulletin board commitment every minute, or a blockchain-backed append-only log growing as block height grows.

Created callbacks are associated with a *ticket*, $\mathsf{tik} \in \{0,1\}^{256}$ that the service provider will eventually post to the bulletin board along with the arguments to the callback.

We will denote service providers by their ID spid. We will present algorithms which depend on service-provider-specific keypairs $(\mathsf{pk}_{\mathsf{spid}}, \mathsf{sk}_{\mathsf{spid}})$, but the structure of these keys is generic (later, we will describe an extension that uses $\mathsf{sk}_{\mathsf{spid}}$ to compute signatures).

Methods. zk-promises permits deployers to define multiple *methods*, functions which mutate the object, these methods can both create callbacks and be called by them. For example, as we saw in Section 3, it may be desirable for one callback creation method to enforce rate limiting on itself, while another creation method does not.

Finally, we describe how user-programmable functionality fits into the zk-promises. In our zero-knowledge proofs, we will use $\Phi_{\mathsf{meth}}(\mathsf{obj}, \mathsf{obj}', \ldots)$ to represent a valid transition from obj to obj' using method meth.

Algorithms. zk-promises consists of the following algorithms:

$\mathsf{Setup}(\Phi) \to \mathsf{pp}$ Takes a representation of zk-promises application-specific predicates and produces public parameters for our zero-knowledge proof scheme. This may be a trusted setup procedure as required by, e.g., the Groth16 zkSNARK [Gro16].

$\mathsf{ExecMethodAndCreateCallback}(\mathsf{pp}, \mathsf{obj}, \mathsf{pk}_{\mathsf{spid}}, \mathsf{meth}, x) \to (\mathsf{obj}', \pi, \mathsf{cbData}, \mathsf{aux})$ Invoked by a user with object obj for service provider with public key $\mathsf{pk}_{\mathsf{spid}}$, this executes the specified method and creates a callback ticket tik for the service provider. Additional public input is given in x. The function may modify obj. The resulting object obj', its zero-knowledge proof π, and any additional execution metadata cbData are sent to $\mathsf{bb}_{\mathsf{obj}}$. The same, plus some auxiliary data aux are sent to the service provider. In the most basic system, cbData contains tik and the object's serial number, and aux is empty.

$\mathsf{VerifyCreate}(\mathsf{pp}, \mathsf{sk}_{\mathsf{spid}}, \mathsf{obj}', \pi, \mathsf{cbData}, \mathsf{aux}) \to \{0,1\}$ Invoked by the service provider spid with secret key $\mathsf{sk}_{\mathsf{spid}}$, this verifies π with respect to obj' and cbData, and verifies that these values appear in $\mathsf{bb}_{\mathsf{obj}}$. This also optionally takes an auxiliary input aux.

$\mathsf{Call}(\mathsf{pp}, \mathsf{tik}, \mathsf{args})$ Invoked by a service provider, calls the callback represented by tik, with callback arguments args. These values are sent to $\mathsf{bb}_{\mathsf{cb}}$.

$\mathsf{VerifyCall}(\mathsf{pp}, \mathsf{tik}, \mathsf{args}, \mathsf{aux}) \to \{0,1\}$ Invoked by the manager of $\mathsf{bb}_{\mathsf{cb}}$, this verifies that the call $(\mathsf{tik}, \mathsf{args})$ (with optional caller-provided auxiliary input aux) is well-formed. The notion of well-formedness is application-defined.

$\mathsf{ScanOne}(\mathsf{pp}, \mathsf{obj}, x) \to (\mathsf{obj}', \pi, \mathsf{cbData})$ Invoked by a user with object obj, this takes one step in the callback list, checks on $\mathsf{bb}_{\mathsf{cb}}$ if the callback has been called, and, if so, executes it. Additional public input is given in x. The order of iteration through the list is application-specific. The resulting object obj', its zero-knowledge proof π, and the execution's metadata cbData are sent to $\mathsf{bb}_{\mathsf{obj}}$.

$\mathsf{VerifyMethodExec}(\mathsf{pp}, \mathsf{obj}', \pi, \mathsf{cbData}) \to \{0,1\}$ Invoked by the manager of $\mathsf{bb}_{\mathsf{obj}}$, this verifies π with respect to obj' and cbData. This is used to verify bulletin board submissions from ExecMethodAndCreateCallback and ScanOne. On verification success, $(\mathsf{obj}', \pi, \mathsf{cbData})$ is posted to $\mathsf{bb}_{\mathsf{obj}}$.

We now describe each algorithm in detail.

$\mathsf{ExecMethodAndCreateCallback}(\mathsf{pp}, \mathsf{obj}, \mathsf{pk}_{\mathsf{spid}}, \mathsf{meth}, x) \to (\mathsf{obj}', \pi, \mathsf{cbData}, \mathsf{aux})$. This algorithm performs two important functions: (1) it updates obj according to the given method; and (2) it creates a new callback entry appends it to the list of created callbacks obj.cbList. The resulting new object obj' is accompanied by a zero-knowledge proof that it was computed correctly.

We describe the update more formally. Let $\mathsf{pk}_{\mathsf{spid}}$ represent the public key of the service provider the user intends to give the callback to, and let x represent any additional public values, including the method meth' the user wishes to create a callback for. The user performs the following updates to its objects.

1. Create a fresh ticket $\mathsf{tik} \leftarrow\!\!\$\ \{0,1\}^{256}$
2. Set $\mathsf{obj}' := \mathsf{meth}(\mathsf{obj}, x)$
3. Set $\mathsf{obj}'.\mathsf{cbList} := \mathsf{obj}.\mathsf{cbList} \| \mathsf{entry}$ where $\mathsf{entry} = (\mathsf{tik}, \mathsf{meth}')$
4. Set $\mathsf{obj}'.\mathsf{sn}$ to be a fresh serial number

In upcoming sections, we will modify the first step to create tickets which depend on $\mathsf{pk}_{\mathsf{spid}}$.

The user then computes a zero-knowledge proof that the new object obj' is correctly computed. Let the public inputs of the proof be obj', x, and $\mathsf{cbData} := (\mathsf{entry}, \mathsf{obj}.\mathsf{sn})$. Let the new callback list entry be $(\mathsf{tik}, \mathsf{meth}')$ where $\mathsf{meth}' \in x$. The user proves the conjunction of the following statements:

zk-object bookkeeping:

The old object exists. $\mathsf{obj} \in \mathsf{bb}_{\mathsf{obj}}$

The serial number is revealed. $\mathsf{obj}.\mathsf{sn} = \mathsf{sn}$

The entry has been appended. $\mathsf{obj}'.\mathsf{cbList} = \mathsf{obj}.\mathsf{cbList} \| \mathsf{entry}$[8]

The predicate is satisfied. $\Phi_{\mathsf{meth}}(\mathsf{obj}, \mathsf{obj}', \emptyset, \mathsf{entry}, x) = 1$

The empty input corresponds to the fact that methods used for creation do not have external callers, and thus do not receive a separate args input, as seen later in ScanOne. We note zk-promises allows a choice of whether to reveal meth. We discuss the tradeoffs later in this section.

The user sends $(\mathsf{obj}', \pi, \mathsf{cbData})$ to the object bulletin $\mathsf{bb}_{\mathsf{obj}}$, and sends the same values, plus some auxiliary data aux, to the service provider.

VerifyCreate$(\mathsf{pp}, \mathsf{sk}_{\mathsf{spid}}, \mathsf{obj}', \pi, \mathsf{cbData}, \mathsf{aux}) \to \{0,1\}$**.** The service provider must check well-formedness of the callback created in ExecMethodAndCreateCallback. Otherwise, the callback may be uncallable, or callable by parties other than the service provider, or a duplicate callback that has already been called.

To verify well-formedness, the service provider with secret key $\mathsf{sk}_{\mathsf{spid}}$:

[8] We omit details for now on how precisely the callback list works. For now, it may be treated as a fixed-size list, where the callback manager retains a running index of unfilled slots. We extend this later in this section.

1. verifies that $(\mathsf{obj}', \pi, \mathsf{cbData})$ appears on $\mathsf{bb}_{\mathsf{obj}}$, i.e., that the callback was created;

2. verifies the proof π with respect to inputs cbData and obj'; and

3. extracts tik from cbData or aux, and verifies that it has never received tik in the past

The final check is to ensure the user does not attempt to double-create the same callback.

The service provider may perform additional checks on cbData and aux using its knowledge of $\mathsf{sk}_{\mathsf{spid}}$.

$\mathsf{Call}(\mathsf{pp}, \mathsf{tik}, \mathsf{args})$. Recall that to call a callback, the service provider must post some data to a bulletin board so that it may be handled asynchronously.

Formally, for a callback ticket tik and associated function arguments args, the algorithm $\mathsf{Call}(\mathsf{tik}, \mathsf{args})$ simply posts $(\mathsf{tik}, \mathsf{args})$ to $\mathsf{bb}_{\mathsf{cb}}$. We will later extend Call to sign and encrypt args.

$\mathsf{ScanOne}(\mathsf{pp}, \mathsf{obj}, x) \to (\mathsf{obj}', \pi, \mathsf{cbData})$. This algorithm reads the next entry of the list of issued callbacks, checks whether the callback has been called, and, if so, executes that method.

We describe the update more formally. Let x represent some additional public data and let $\mathsf{entry} = (\mathsf{tik}, \mathsf{meth})$ represent the next entry in the callback list $\mathsf{obj.cbList}$ (later in this section we give a concrete list traversal construction). If the entry has been called, i.e., $(\mathsf{entry.tik}, \mathsf{args}) \in \mathsf{bb}_{\mathsf{cb}}$, then the user performs the following updates to its object:

1. Set $\mathsf{obj}' \coloneqq \mathsf{meth}(\mathsf{obj}, \mathsf{args}, x)$

2. Delete tik from $\mathsf{obj}'.\mathsf{cbList}$

3. Set $\mathsf{obj}'.\mathsf{sn}$ to be a fresh serial number

If tik has not been called, then the user sets $\mathsf{obj}' \coloneqq \mathsf{obj}$ and skips (1) and (2).

The user must then compute a zero-knowledge proof that the new object obj' is correctly computed. Let the public inputs of the proof be obj' and $\mathsf{cbData} \coloneqq \mathsf{obj.sn}$. The user constructs a proof π of the conjunction of the following statements:

The old object exists. $\mathsf{obj} \in \mathsf{bb}_{\mathsf{obj}}$

The serial number is revealed. $\mathsf{obj.sn} = \mathsf{sn}$

entry *is the current entry.* $\mathsf{entry} = \mathsf{obj.cbList.nextUnscanned}()$

Callback was applied if called. If $(\mathsf{tik}, \mathsf{args}) \in \mathsf{bb}_{\mathsf{cb}}$ then $\Phi_{\mathsf{meth}}(\mathsf{obj}, \mathsf{obj}', \mathsf{entry}, \mathsf{args}, x) = 1$ and $\mathsf{obj}'.\mathsf{cbList} = \mathsf{obj.cbList.tail}()$

No-op if not called. If $\mathsf{tik} \notin \mathsf{bb}_{\mathsf{cb}}$, then $\mathsf{obj}' = \mathsf{obj}$

As in ExecMethodAndCreateCallback, the user sends $(\mathsf{obj}', \pi, \mathsf{cbData})$ to the object bulletin $\mathsf{bb}_{\mathsf{obj}}$. We note that the last two statements are the reason we require efficient zero-knowledge proofs of membership and non-membership in $\mathsf{bb}_{\mathsf{cb}}$.

In practice, ScanOne may be batched. A user may process 100 callbacks at a time, and produce a single zero-knowledge proof that all 100 were applied correctly.

VerifyMethodExec$(\mathsf{pp}, \mathsf{obj}', \pi, \mathsf{cbData}) \to \{0,1\}$**.** The object bulletin board must handle zk-object updates from ExecMethodAndCreateCallback and ScanOne. To do so, it does the same proof verification as VerifyCreate. Specifically, given $(\pi, \mathsf{cbData}, \mathsf{obj}')$ it verifies the zero-knowledge proof π with respect to its inputs cbData and obj'. In addition, as with any zk-object update, it must ensure that the serial number number is not repeated. That is, $\mathsf{cbData.sn} \notin S$, where S is the set of all serial numbers observed by the bulletin board operator. If all checks succeed, the operator posts $(\mathsf{obj}', \pi, \mathsf{cbData})$ to $\mathsf{bb}_{\mathsf{obj}}$.

VerifyCall$(\mathsf{pp}, \mathsf{tik}, \mathsf{args}, \mathsf{aux}) \to \{0,1\}$**.** The operator of the callback bulletin board must verify incoming calls $(\mathsf{tik}, \mathsf{args})$ before placing them on the bulletin board. In our basic construction, aux is empty, and there is nothing that requires verification. Later in this section, we will let aux be a digital signature, and condition acceptance on the signature's successful verification.

4.2 Unlinking Create and Call

Currently, a passive observer of the (possibly public) bulletin board sees tik, meaning it can correlate executions of ExecMethodAndCreateCallback and Call. While this does not identify the user, it may, e.g., leak what post was moderated.

To fix this, we slightly modify the zero-knowledge proof and public input of ExecMethodAndCreateCallback. The user replaces entry in cbData with a commitment $\mathsf{com}_{\mathsf{entry}} := \mathsf{Com}(\mathsf{entry}; s)$ for some randomness s, and opens the commitment in the proof. In addition, the auxiliary data sent to the service provider is now $\mathsf{aux} := (\mathsf{entry}, s)$. In VerifyCreate, the service provider checks that $\mathsf{com}_{\mathsf{entry}} = \mathsf{Com}(\mathsf{entry}; s)$.

The Call algorithm is unchanged. Since tik is not sent in the clear at any time before Call, there is no longer any event to link it to.

4.3 Expiry

Currently, there is no limit on the amount of time that can pass between creation and calling of a callback. This gives service providers the power to rate old and irrelevant posts for any reason. In addition, it requires the user to store all callbacks indefinitely until they are called. Over time, this makes a full scan computationally infeasible.

We solve both of these problems by associating an *expiry* with every callback. With expirable callbacks, users only have to store the tickets that have not expired, and calls are limited to the expiry period, e.g., at most one day after creation. We detail the changes to the base system that this feature entails.

ExecMethodAndCreateCallback now takes an expiry exp as an argument, and includes exp in entry.

In VerifyCall, the bulletin board operator does as before, but additionally stores the time t that the call was received. So each element of $\mathsf{bb}_{\mathsf{cb}}$ is now of the form $(\mathsf{tik}, \mathsf{args}, t)$.

ScanOne now takes as part of its public input x the current time c. When deciding whether to apply $\mathsf{entry} = (\mathsf{tik}, \mathsf{meth}, \mathsf{exp}) \in \mathsf{obj.cbList}$, they do as follows:

1. If tik was posted, i.e., $(\mathsf{tik}, \mathsf{args}, t) \in \mathsf{bb}_{\mathsf{cb}}$:

   (a) If $t <$ exp, the callback was called in time. The user applies it: $\mathsf{obj}' := \mathsf{meth}(\mathsf{obj}, \mathsf{args}, x)$ and deletes this entry from cbList.

(b) If $t \geq \mathsf{exp}$, the callback was called after expiry. The user ignores it $\mathsf{obj}' := \mathsf{obj}$ and deletes this entry from cbList.

2. If tik was not posted, i.e., $\mathsf{tik} \notin \mathsf{bb}_{\mathsf{cb}}$:

   (a) If $c \geq \mathsf{exp}$, the callback has already expired. The user ignores it $\mathsf{obj}' := \mathsf{obj}$ and deletes this entry from cbList.

   (b) If $c < \mathsf{exp}$, the callback is unposted and unexpired. The user ignores it $\mathsf{obj}' := \mathsf{obj}$ and leaves this entry cbList.

The user's zero-knowledge proof in ScanOne is also updated to reflect the above logic.

4.4 Authenticity and confidentiality for callback inputs

Our construction so far still has the following limitations: (1) callback arguments are sent in the clear in Call, meaning a passive adversary can, e.g., learn correlations between activity and moderation decisions; and (2) function arguments are not authenticated in any way, meaning that a malicious bulletin board provider or an active adversary can modify the callback arguments in a Call payload.

We would like to give service providers the ability to include encrypted, non-malleable arguments in their call. This must satisfy a few constraints simultaneously:

1. If a service provider posts a ticket and callback arguments, the arguments cannot be malleable.
2. If the arguments are malformed, the user must be able to reject the callback during ScanOne.
3. If the arguments are well-formed, the user cannot reject the callback during ScanOne.

We now describe how to add authenticity and confidentiality in a way that meets these goals.

Authenticity: Tickets are signature pubkeys. To bind tik to args, we interpret tik as a public key for a signature scheme. Surprisingly, this requires few modifications to our protocol overall, and no modifications to our zero-knowledge proofs.

Let Σ represent an EUF-CMA-secure rerandomizable signature scheme. We assume a public-key infrastructure for all service providers. That is, every service provider has an associated keypair $(\mathsf{pk}_{\mathsf{spid}}, \mathsf{sk}_{\mathsf{spid}})$, and a user, given spid, can discover $\mathsf{pk}_{\mathsf{spid}}$.

In ExecMethodAndCreateCallback, rather than selecting a random tik, the user computes $(\mathsf{tik}, r) \leftarrow \Sigma.\mathsf{RerandPk}(\mathsf{pk}_{\mathsf{spid}})$ and places r in aux for the service provider.

In VerifyCreate, the service provider computes $\mathsf{sk}'_{\mathsf{spid}} := \Sigma.\mathsf{RerandSk}(\mathsf{sk}_{\mathsf{spid}}, r)$ and checks that tik is well-formed, i.e., $\mathsf{tik} = \Sigma.\mathsf{SkToPk}(\mathsf{sk}'_{\mathsf{spid}})$.

In Call, instead of sending $(\mathsf{tik}, \mathsf{args})$, the service provider sends $(\mathsf{tik}, \mathsf{args}, \sigma)$, where $\sigma = \Sigma.\mathsf{Sign}_{\mathsf{sk}'_{\mathsf{spid}}}(\mathsf{args})$. This binds the arguments to the ticket.

In VerifyCall, instead of posting $(\mathsf{tik}, \mathsf{args})$ unconditionally, it now receives $\mathsf{aux} = \sigma$ and posts the call iff $\Sigma.\mathsf{Verify}_{\mathsf{tik}}(y, \sigma)$ is true.

Confidentiality: encrypted method augments and in-circuit decryption. To encrypt args, it suffices to have the caller simply compute and post $\mathsf{args} := \mathsf{Enc}_k(\mathsf{plaintextArgs})$, where k is an encryption key and Enc is a circuit-friendly CPA-secure encryption algorithm (CCA-security is not necessary, since args is now signed).

We must be careful about how k is determined, though. If we do not properly bind a ticket's associated symmetric encryption key to the ticket itself, a user could intentionally use the wrong key to decrypt arguments to the wrong value. On the other hand, if we require the client to prove the decrypted payload is well-formed, a malicious caller can deadlock a client by giving it an invalid ciphertext.

We thus include a separate encryption key with every entry in the callback list. This forces clients to use the correct key. Because callbacks are authenticated by the ticket holder, clients cannot tamper with the ciphertext. Further, because each ticket is bound to its encryption key, callers cannot use malformed ciphertext to deadlock the client. They can give a ciphertext with an ill-formed decryption, but this will merely result in a failed callback.

ExecMethodAndCreateCallback now generates a fresh encryption key k, and includes k in entry.

In Call, rather than letting args be plaintext, it uses its knowledge of k to compute args $=$

$\mathsf{Enc}_k(\mathsf{plaintextArgs})$.

In ScanOne, the user simply decrypts the payload in-circuit. That is, rather than using args directly, it uses $\mathsf{Dec}_k(\mathsf{args})$, where k comes from the current entry in obj.cbList.

4.5 Variable-length callback lists

The core of our extension to the zk-object model requires us to track issued callbacks via cbList. We must represent cbList in a way that gives efficient circuit operations for append, remove, and iteration. Naïve approaches do not work here: a large fixed-size array means a full scan requires traversing even the empty slots in the array. And a sparse Merkle has less than idea overhead, requiring $\log n$ hashes where n is the number of elements of the list, for a total of $O(n \log n)$ work for iteration. This leaves us, seemingly, needing to resort to expensive mechanisms for constant overhead zk-memory [FKL+21].

We observe that we do not need to support $O(1)$ random-access remove operations on cbList to scan and ingest callbacks. Since users must ensure all called callbacks are ingested, they must traverse the full list at some point. Thus, it suffices to support $O(1)$ removal *amortized, while traversing a list in order*, and $O(1)$ append for new callbacks. We define the representation of a list ℓ as

$$h_\ell := H(H(H(H(\ell_1), \ell_2), \ell_3) \dots \ell_n).$$

This supports $O(1)$ appending, since $\ell' := \ell \| x$ implies $h_{\ell'} = H(h_\ell, x)$. But it does not support efficient removal.

To support $O(1)$ removal when incrementally iterating through the list, we encode list traversal as a state machine in our user object, keeping track of the previous cbList as well as the new version that will be the result of a complete scan. Removal consists of *not* adding the value to the new list. We note we can readily make this incremental logic handle multiple entires in the list at a single time and thus create a batch proof. The complete zero-knowledge relation for variable-length cbList is provided in Section 10.

4.6 Private and public method identifiers

So far we have not specified whether the method being called in ExecMethodAndCreateCallback or ScanOne is hidden from passive adversaries or not. We describe how to implement either choice, and discuss tradeoffs.

If method IDs are private, then Φ is a single predicate that is used for every operation, and takes in meth as an argument. While this provides method privacy, it means that a user performing a ScanOne operation must prove a circuit whose size is the sum of all the method sizes.

Public method IDs can be used to mitigate this. That is, we may let each Φ_{meth} be a separate circuit, and have every ExecMethodAndCreateCallback and ScanOne also post meth in the clear to the bulletin board.

Finally, we note is possible to trade off method anonymity with performance by grouping different methods together into the same circuit.

5 Programming in the zk-object model

Given the basic construction, we now describe some applications and altenrative configurations.

Authorization logic. First, because the zk-object model supports Turing-complete state machines with arbitrary state, we can implement a variety of applications. zk-objects readily represent accounts with stateful access control policies spanning almost any authorization logic. As hinted at in Figure 5.2, we can program rate limits, where the rate depends on the client's current state. These give much more flexibility than existing techniques which enforce a fixed limit on a counter that resets, e.g., daily, (see, e.g., [CHK+06]) and applies equally to both new and well-established accounts. . Similarly, we can support logic for temporary bans, or a probation policy where an additional infraction, while in a probation state, results in a ban. Finally, because we can build Turing complete state machines, state itself is unbounded, we could make a zk-object store a client's entire history of actions and defining arbitrary logic on top that.

We are also not limited to purely anonymous access. zk-objects can allow both anonymous

and pseudonymous authentication for different pseudonyms across various domains,[9] where an infraction on one website affects the account backing all of them.

Non-anonymous callbacks, revocation, and privileges. . Because the callback management mechanism is itself a zk-object, we can also realize non-anonymous callbacks, where the object is anonymous, but callbacks are placed to a fixed identifier.[10] Given accounts issued to a known identity, e.g., an employee, this would allow for updates or revocation (e.g, in the case of promotion or employment termination).

Callback ordering. Similarly, we can change the semantics of callbacks themselves. We can force callbacks to be evaluated in the order they are issued. Or in the order they expire. The later would reduce the clients need to scan for open callbacks, at the cost of blocking settlement of subsequent calls. We could, similarly, imagine different tiers of calls with different priorities and scanning interval requirements.

Callbacks from TEEs, public smart contracts, other zk-objects . Currently we restrict the source of a callback to a public key, where the bulletin boards a signature over the callback and its arguments under the key. However, other options are possible. For example, the signing key could be controlled by a TEE, guaranteeing that callbacks could only be made by programs under certain conditions. Similarly, we could restrict callbacks to be called by a specific program ran by the bulletin board (e.g., a smart contract instance on a blockchain). Finally, we could restrict callbacks to coming from other zk-objects. Here we would require the bulletin board to, instead of a signature, verify a proof that the call came from the election of a method on another zk-object. In this case, we could imagine calls are either bound to a class of objects or to a specific instance id. This would require some form

Any reputation system must have a mechanism to prevent users from rating themselves. We note this can be achieved via a strict allow list, whereby Φ is hard-coded with a set of signing public

[9] Pseudonyms could be stored in the object directly via a mapping from domain to pseudonym, or derived via a PRF evaluation.

[10] Naively, such callbacks would be single-use. To allow multiple calls, we could, e.g., postfix the callback with a counter and force the client to retrieve the next counter value.

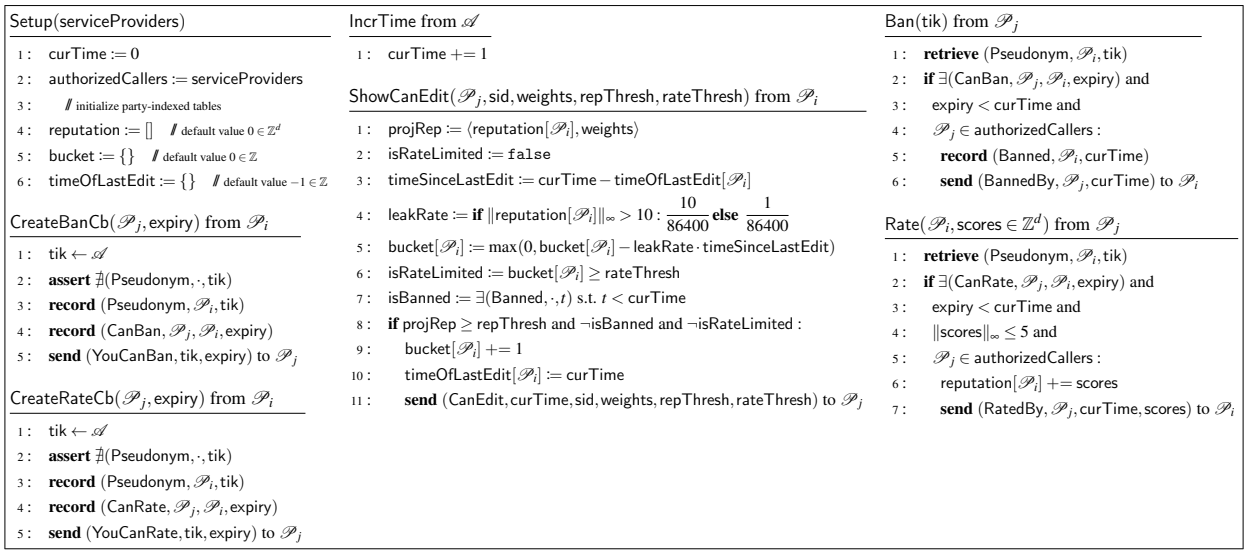


Figure 5.3: A simple ideal functionality for an anonymous blocklisting system. $\mathscr{A}$ is the adversary, who determines the structure of tickets and passage of time.

keys it will accept calls from, or by permissive mutable list, whereby public keys be part of the user record itself and unauthorized calls are rate-limited rather than rejected outright, or by any access control regime in between.

6 Case study: anonymous reputation from zk-promises

We now use our building blocks from Sections 4 and 5 to construct a reputation system from zk-promises.

We define an anonymous reputation system as an ideal functionality in Figure 5.3. The system allows the user to create callbacks for two methods: Rate and Ban. To rate a user, the service provider selects an integer in the range $[-5,5]$ and uses that as the callback public input. To ban a user, no public input is necessary, but a dummy value is encrypted to not leak which method is being called. By construction, the ideal functionality has the security properties described in Section 3.2, namely, confidentiality, obliviousness, integrity, and atomicity.

To demonstrate generality, we implement support for multi-dimensional reputation, where reputation is a vector. In our prototype, the user shows that the dot product of this vector with

a server-provided weighting vector, is above a certain threshold. We refer to this as *projected reputation*. In addition, the user shows they are not rate-limited, which is realized with a leaky-bucket. To complete an action, we require the bucket has remaining capacity. Once a user's bucket is full, they must wait for the bucket to partially drain. The drain rate is a function of the user's reputation and elapsed time.

Using the building blocks from the previous section, we build a reputation system which realizes the ideal functionality, with a reputation dimension $d = 2$. Our construction adds the variable lastPosted to obj, as described in our rate-limiting discussion. To achieve the confidentiality, obliviousness, and integrity properties of the ideal functionality, we use most of the features from the previous section: authenticated and confidential callback inputs, hidden ticket values, expiry, variable-length tokens, and private method IDs. Finally, we realize caller authorization logic by extending the aux parameter of calls to contain a second signature σ' over the rest of the call $(\mathsf{tik}, \mathsf{args}, \sigma)$. This signature is verified against a list of authorized caller public keys.

6.1 Security argument

We will now present a proof sketch that this construction, the *real world*, realizes the ideal functionality. In the next section, we benchmark the construction.

We show how each of the ideal functionality procedures are realized by the real-world scheme.

Realizing Create* and ShowCanEdit. In the real world, all of these procedures are performed at once by ExecMethodAndCreateCallback. In order to issue a ticket for the rate or ban method, ExecMethodAndCreateCallback rerandomizes $\mathsf{pk}_{\mathsf{spid}}$ and stores it as tik along with the expiry, encryption key, and method ID in obj.cbList. The user then posts the new object to $\mathsf{bb}_{\mathsf{obj}}$, and sends the service provider the transaction ID and the opening to the commitment for $(\mathsf{tik}, \mathsf{exp}, k, \mathsf{meth})$. We consider the call a success if VerifyCreate succeeds.

The CreateBanCb and CreateRateCb methods require the adversary to pick a unique tik. This is captured by the uniqueness check in VerifyCreate.

ShowCanEdit implements the leaky bucket functionality from the real world scheme verbatim.

Realizing Ban and Rate. When callbacks are called, they must be applied at that point in time. Suppose the server calls $\mathsf{Call}(\mathsf{tik}, \mathsf{args})$ on method meth at time t. We argue that, when ExecMethodAndCreateCallback succeeds, any future ExecMethodAndCreateCallback with args containing lastFullLoopTime threshold $t' > t$ must reflect the object post-callback. Note ExecMethodAndCreateCallback with lastFullLoopTime threshold t' must show that obj.lastFullLoopTime $\geq t'$. This can only occur if a ScanOne sequence that started at time at earliest t' has completed, which, in turn, only succeeds if the user has done one of the following: (1) executed $\mathsf{meth}(\mathsf{obj}, \mathsf{args}, x)$ and deleted the associated tik from cbList, (2) ignored the call and deleted tik from cbList, (3) or ignored the call and left tik in cbList. However, the latter two cases may only happen if $\mathsf{tik} \notin \mathsf{bul}_{\mathsf{cb}}$, which contradicts the fact that $\mathsf{Call}(\mathsf{tik}, \mathsf{args})$ was called at time t. Thus, $\mathsf{meth}(\mathsf{obj}, \mathsf{args}, x)$ was called, and the current object state reflects the called callback.

The other necessary property for the real world scheme is that only permitted parties can ban or rate users. Suppose party $\mathscr{P}_i$ applies a $(\mathsf{tik}, \mathsf{args})$ from $\mathsf{bb}_{\mathsf{cb}}$ during ScanOne. Firstly, the tuple could only appear in $\mathsf{bb}_{\mathsf{cb}}$ if its signature is valid, i.e., the party who ran Call knows tik's secret key. In addition, $(\mathsf{tik}, \mathsf{args})$ is only applied if tik appears in the user's cbList, which, in turn, can only happen if tik was given to $\mathscr{P}_i$ during ExecMethodAndCreateCallback. Thus, the calling party must know the secret key of the tik given at ExecMethodAndCreateCallback. This occurs only if they are the same party (or, the party opposite $\mathscr{P}_i$ during ExecMethodAndCreateCallback was corrupted). Finally, we note that the signature check of σ' in VerifyCall ensures that the caller is in authorizedCallers.

All remaining properties of the ideal functionality, e.g., reputation-conditioned rate limiting, follow immediately from our choice of Φ in the zero-knowledge proofs.

7 Experiments

In this section, we describe the implementation and evaluation of the application described in Section 6 built with zk-promises. We build two versions, one for the decentralized and one for the

	Prove (ms)				Verify (ms)
Callback Capacity:	*Unlimited*	2^{16}	2^{32}	2^{64}	-
	Signature-based	Tree-Based			Both
`Show Authorized`	337	316	372	558	2.81
`Incremental Scan`	571	535	638	972	2.81
`Show Authorized + Incremental Scan`	909	716	829	1123	2.81

Table 5.1: Table containing the runtimes for proving and verifying the zero knowledge circuits. The signature-based variants correspond to the centralized setting and the tree-based variants correspond to the decentralized setting. Incremental scan includes the cost of applying the callback. However, the verification runtimes are same across both settings.

centralized setting.

Instantiating cryptographic primitives. We use Groth16 [Gro16] as our zkSNARK and Poseidon [GKR+21] for all hash functions. For circuit-friendly encryption, we use key-prefixed Poseidon in counter mode as a stream cipher. For signatures, we implement Schnorr over the Jubjub curve [ZCa19].

Hardware. All benchmarks were performed on a desktop computer with a 2021 Intel i9-11900KB CPU with 8 physical cores and 64GiB RAM running Ubuntu 20.04 with kernel 5.15.0-69-generic.

Code. zk-promises is written in Rust, using the Arkworks [Ar21] zkSNARK crates and Rayon for parallelization where possible. The Criterion-rs crate was used for all microbenchmarks and statistics.

7.1 Methodology

In zk-promises, method calls are verified by a zero-knowledge proof realized as a circuit representing the logic in the call (e.g., that the account is in good standing to make an edit, that a generated callback was stored for future scanning, or that all pending callbacks were scanned for). We note that the logic for scanning for a callback also includes the logic for ingesting the callback and, to hide which callback was made, includes logic for both calls.

Recall, zk-promises can be built in both a decentralized and centralized setting, depending on how the bulletin board is instantiated. In the decentralized setting, we build a Merkle tree over all

callback entries in the bulletin board and require proof to show membership in that tree. In the centralized setting, the bulletin board is operated by a single server that is trusted for integrity but not confidentiality or obliviousness. This allows us to replace Merkle tree membership checks with simple signature checks.

For both settings, we create and benchmark circuits to prove methods for authorization, callback scanning, and a combined single circuit of those two statements.

While the centralized setting offers improved efficiency, it requires on additional step by the server. Periodically, the server must *rebuild* the non-membership data structure, updating the signed timestamp on each value to prevent use of an expired but still signed non-membership entry.[11] We measure the runtime of this process as well.

Finally, we also implement *batched scan*—a circuit which settles more than one callback at a time. This allows the client to condense scans of multiple callbacks into a single proof for the server to verify. As with *scan incremental* this includes (now repeated $|\texttt{batch}|$ times) the logic for executing both callbacks.

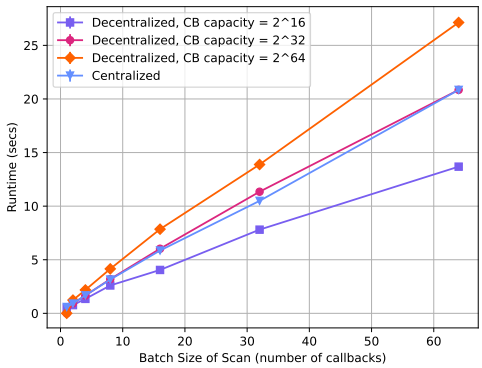
7.2 Results

Over all benchmarks, the maximum observed relative standard error of the median was 1.2%.

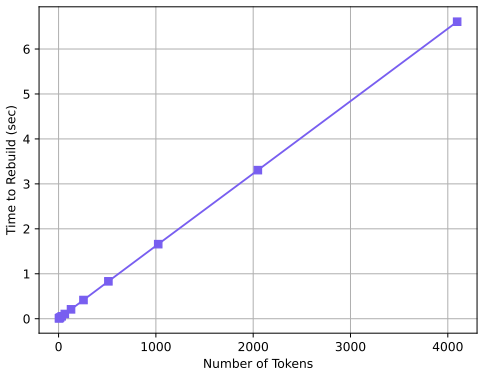
Table 5.1 shows the Groth16 proof and verify computation time for the three zero-knowledge proof circuits. One circuit shows the client is authorized, another scans for a single callback (*scan incremental*), and one combines both into a single operation. Section 12 gives a breakdown of circuit sizes. As expected, prover runtime (in the tree-backed setting) increases linearly as the global allowed number of callbacks (and therefore the cost of the Merkle tree membership check) increases. The signature-based setting, which replaces the membership check with a fixed single signature verification, takes constant time.

While scan incremental only checks for a single callback, a complete scan must handle all open

[11] This is not necessary in the decentralized setting, since proofs are made with respect to an up-to-date root of a sparse Merkle tree, which has efficient membership and non-membership proof algorithms.



(5.4a) Client's proving runtime to batch settle callbacks as the size of the batch increases.



(5.4b) Server's runtime to *rebuild* the (signature-based) callback bulletin board.

callbacks, requiring multiple invocations. We constructed circuits for batched scans to handle a fixed number of pending callbacks. Benchmarking results can be found in Figure 5.4a. As expected, a batched scan is more efficient compared to the same number of single-scans. For example, for the tree height of 32, computation savings range from 20–40% when doing batched settles of between 2 and 32.

The verification times are constant, at 2.8ms for all circuits as seen in Table 5.1. Batching Groth16 proof verification gives a throughput of up to 1,194 verifications per second, a 4× throughput improvement over individual verifications.

Finally, we plot the rebuilding time for signature-backed callback bulletin boards in Figure 5.4b. As expected, the runtime grows linearly with the number of callbacks called by the server.

Communication costs for zk-promises are very low, with constant proof sizes and small public inputs. Concretely, Show Authorized uses 608B and 568Bfor signature-based and tree-based settings, respectively. Similarly, Incremental Scan and Batched Scan use 396B and 268B, respectively, and Show Authorized + Incremental Scan use 748B and 708B, respectively.

7.3 Discussion

We now discuss the feasibility of our prototype reputation system in a real-world deployment.

Choice of configuration

The performance of any deployment of zk-promises is determined by four configurable parameters.

- *(maximum) broadcast latency* —how frequently the bulletin board rebuilds and announces updates,
- *expiry*—how long before callbacks expire,
- *(maximum) scan interval*—how frequently the client must complete a full scan (i.e, the cutoff that `lastFullScanTime` must be newer than), and
- *batch size*—how many callbacks clients scan for at a time.

Let us also define the average *authentication rate* of a user as $\mathsf{avgAuth}_r$. During each scan, the client must process the callbacks that did not expire in the previous scan, and all added callbacks since that previous scan. The total number of callbacks that must be checked is:

$$(\mathsf{avgAuth}_r \cdot \mathsf{expiry}) + (\mathsf{avgAuth}_r \cdot \mathsf{scanInterval}).$$

If the batch size is equal to this, the server will only have to verify a single (batched) scan for the average client. Increasing the batch size this way is a tradeoff as each client will now have to do the work proportional to the average number of callbacks a user generates.

If we combine batched scans with `Show Authorized` into a single circuit, then the server only has the overhead of one zkSNARK verification per authentication for clients whose number of outstanding callbacks is at most the batch size. Each client, in contrast, needs to compute less than 30 seconds of work per authentication, even in very large settings, as shown in figure Figure 5.4a.

Decreasing the broadcast latency will increase the liveness of zk-promises, as the smaller this value is, the faster callbacks are announced. The broadcast latency should be set to the smallest value a server can handle and still keep up with computation.

Expiry should be set so that most malicious client actions can be caught before the callback

expires. Setting scan interval is a tradeoff in how synchronized the clients are with bulletin board and how much work (scanning) the client must do.

Example scenario: Wikipedia

We now examine a real-world scenario: moderating Wikipedia edits. Wikipedia (across all languages) has an edit rate of 18 edits per second (average rate across April 2023–March 2024 [Wikb]). We know from the above experiments that a single lightweight server can support 357 zk-promises verifications per second (or 1,428, using batched verification). Thus, the increased load on the server per authentication is easily handled, even if all edits are anonymous.

The peak month of edits in that time frame (January) for English Wikipedia [Wikb] [12] had an average editor make around 239 edits a month, and 90% of edits are reverted within 5hrs [JNV]. Thus we set expiry to 1 day. If we set the scan interval to 24hrs then the batch size should be set to $\frac{239}{31} \cdot (1+1) = 16$, so that the server only has to verify a single proof for the average user. This means the computation required to create the proof for batched scan is less then 10sec, as shown in Figure 5.4a. We feel client side proving times are reasonable, as proofs can be completed in the background while the client makes an edit.

We note that our numbers are an overestimation of the performance requirements, as most edits are (non-anonymous) automated editors [PCL+07]. So the actual rate of anonymous edits for anonymous users is lower.

8 Conclusion

In this paper, we define zk-promises, which adds callbacks to the zk-object model. While we have used this to demonstrate the feasibility of an anonymous reputation system, the potential applications are much broader. A long line of work on proof-carrying data [CT10, BCCT12] have explored zero-knowledge incremental computation. Zerocash [BCG+14] introduced the idea of replay- and

[12] Additional data on edits such as reversion rate are only available for select languages, so we switch to statistics for only English Wikipedia.

forking-prevention for simple objects (limited to payments). Hawk [KMS+16] generalized the class of functions, but each object was isolated and could not interact with other object types. Zexe [BCG+20] offered inter-object communication between oblivious objects of different types and, additionally, used recursive proofs to hide which object is called. But up until now, using this programming model in many real applications has been challenging.

To explain, we borrow a distinction from the cryptocurrency literature. Smart contract systems, such as Ethereum, typically operate in the *account model*, where there is a single authoritative state for an account and methods can be called on it. In contrast, in [BCG+14, KMS+16, BCG+20, XCZ+22], state is split into multiple locations. This is sometimes referred to as the *UTXO model*. The UTXO model is generally regarded as impractical to work with, and brings with it challenges similar to the asynchronous negative feedback problem we articulate in Section 1.

zk-promises yields a practical account model for privacy-preserving computation, as shown by our example application. Users can store their state in a single account that gets asynchronously updated by other calls. In the zk-promises model, users are responsible for sequencing updates to their object requested by others, but they cannot drop particular update requests. We believe this is useful for many applications, ranging from anonymous access control to privacy-preserving smart contract systems.

A second consequence of this model is the possibility of having zk-objects themselves make callbacks. Currently, in our prototype applications, the caller is identified by a signature, but this can easily tied to a TEE, a public smart contract, or even another zk-object. This would allow an account model where objects can, programmatically, be controlled by another entity in full or in part even if that entity is not trusted to know the state of the object.

9 Appendix: Extensions

While zk-promises already offers a lot of flexibility in callback creation and execution, we identify some potential future routes of extension.

Service provider privacy. zk-promises explicitly does not aim to provide privacy for the party calling the callback. Specifically, zk-promises permits the user to know which service provider called a callback (by simply looking at which ticket was used), as well as the contents of the callback arguments.

These notions of privacy are distinct and are resolved in distinct ways. Caller privacy can be achieved by following the blocklisting mechanism of SnarkBlock and BLAC [RMM22, TAKS10]. We replace token issuance with an interactive protocol between the user and the service provider, whereby the user computes an oblivious pseudorandom function (OPRF) over a nonce provided by the service provider, where the key k is some user-specific key. This $(\mathsf{tok}, \mathsf{nonce})$ pair is used during redemption. To sweep, the user must traverse the entire bulletin until it finds a pair such that $\mathsf{PRF}_k(\mathsf{nonce}) = \mathsf{tok}$, and then sweep the attached payload. Because this traversal is inherently sequential over (a portion of) the bulletin board, this method can only reasonably be deployed over limited time spans, and not globally.

Redeemer data privacy is a more difficult problem, since the user controls their user record, and is tasked with updating it. One way of achieving this, would be for a user to cede control over their record to a committee of mutually distrusting parties. Then, to sweep a token, the parties would engage in a multiparty computation (MPC) protocol to update the record. Another option is to permit the user record to be updated by a single party using fully homomorphic encryption, and then only rely on MPC to generate zkSNARK for the bulletin.

Lazy callback settlement. Currently, client prover time during ScanOne is linear in the size of cbList, since it must check if each callback has been called or has expired. This cost can be reduced if we loosen the time bounds on when callbacks must apply.

Let cbList be as before, but now sorted by expiry time. This can be achieved by in-circuit sorting with a more complex data structure, or by setting a fixed expiry period for all present and future callbacks. To call ScanOne, the user will now simply iterate through cbList until it finds an expiry time that is in the future.

In this regime, a user only needs to do work linear in the number of expired callbacks. This

comes at the cost of immediacy of calls—any Call can take up to the expiry time until it is applied to the user.

Retention period. In order to permit users to be offline for arbitrary periods of time, zk-promises requires that bulletin boards keep all state forever. If we loosen this, permitting users to be offline for, e.g., at most one year, then we may permit bulletin board operators to delete all items older than one year.

This extension only requires one extra assertion in ScanOne: that lastFullLoopTime is greater than the current time minus the retention period. This way, a user who doesn't fully settle at least once within the retention period is permanently locked out, and cannot make progress by claiming their callbacks (which may or may not have been called) are not in $\mathsf{bb}_{\mathsf{cb}}$.

Rate limiting redemptions. Currently, redemption is unconditional—the bulletin accepts any payload with a valid signature. There is no need to stop here, though. If, for example, we wish for redeemers to also have credentials, we can add a NIZK to the redemption logic to ensure that. Similarly, if we wish to add rate limiting to token redemptions, we can add that as well, using standard anonymous credential rate limiting methods [RWGM23].

Additional provenance checking of calls. In a decentralized setting, it may be desirable for a smart contract, rather than a specific service provider, invoke Call for some callbacks. To handle this, it suffices to add public auxiliary input to the callback bulletin board. Specifically, rather than storing $(\mathsf{tik}, \mathsf{args}, t)$ in $\mathsf{bb}_{\mathsf{cb}}$, the bulletin board manager stores $(\mathsf{tik}, \mathsf{args}, t, \mathsf{addr})$, where addr is the source address of the Call. In Ethereum, for example, it is not necessary to trust any third party that addr is correct. Similarly, call payloads can be accompanied by an attestation from a trusted execution environment (TEE) that the values were computed correctly.

The changes to ExecMethodAndCreateCallback and ScanOne are similar to the allowlist method in the previous subsection. The user may predetermine the smart contract addresses it is willing to receive calls from, and check in the ScanOne ZKP that this holds.

10 Appendix: Formal description of zk-promises

In this section we formally describe the entirety of zk-promises, once all the features in Section 4 are added. We write the zero-knowledge relations for ExecMethodAndCreateCallback and ScanOne in Figures 5.5 and 5.6, respectively.

$\mathsf{Setup}(\Phi) \to \mathsf{pp}$**.** This performs Groth16 CRS generation for the relations in Figures 5.5 and 5.6. The output pp contains proving and verifying keys.

ExecMethodAndCreateCallback$(\mathsf{pp}, \mathsf{obj}, \mathsf{pk}_{\mathsf{spid}}, \mathsf{meth}', x) \to (\mathsf{obj}', \pi, \mathsf{cbData}, \mathsf{aux})$**.** Let obj be the user object, let $\mathsf{pk}_{\mathsf{spid}}$ be the verifying key for service provider ID spid for an EUF-CMA-secure signature scheme Σ, and let s be the commitment randomness used in that objects commitment in $\mathsf{bb}_{\mathsf{obj}}$. Let meth' be the method the user wishes to create a callback for. Let $x = (t, \mathsf{curTime})$, where t is the minimum value for lastFullLoopTime that the service provider will accept and curTime be the current global time.

The ExecMethodAndCreateCallback algorithm proceeds as follows:

1. Compute a new ticket as a rerandomized verifying key $(\mathsf{tik}, r) \leftarrow \Sigma.\mathsf{RerandPk}(\mathsf{pk}_{\mathsf{spid}})$
2. Pick expiry expiry and a fresh encryption key k and build the callback list entry $\mathsf{entry} := (\mathsf{tik}, \mathsf{exp}, k, \mathsf{meth}')$. Commit to the entry $\mathsf{com}_{\mathsf{entry}} := \mathsf{Com}(\mathsf{tik}, \mathsf{exp}, k, \mathsf{meth}'; s_{\mathsf{entry}})$ where s_{entry} is fresh randomness.
3. Clones the zk-object obj to a new one obj'. Append tik to their private callback list, compute $\mathsf{obj}'.h_{\mathsf{cb}} := H(\mathsf{obj}.h_{\mathsf{cb}}, \mathsf{entry})$, $\mathsf{obj}'.h_{\mathsf{cb}}^{(\mathrm{old})} := \mathsf{obj}'.h_{\mathsf{cb}}$, $\mathsf{obj}'.\mathsf{lastFullLoopTime} := \mathsf{curTime}$, and pick a fresh serial number $\mathsf{obj}'.\mathsf{sn}$. Finally commit to the new object, $\mathsf{com}'_{\mathsf{obj}} := \mathsf{Com}(\mathsf{obj}'; s'_{\mathsf{obj}})$ where s'_{obj} is fresh randomness.
4. Let $\mathsf{cbData} = (\mathsf{com}_{\mathsf{entry}}, \mathsf{obj.sn})$ and let $\mathsf{aux} = (s_{\mathsf{entry}}, \mathsf{entry})$
5. The user computes a zero-knowledge proof π of $R_{\mathsf{create}}^{\Phi_{\mathsf{create}}}$.
6. The user sends $(\pi, \mathsf{com}'_{\mathsf{obj}}, \mathsf{cbData})$ to $\mathsf{bb}_{\mathsf{obj}}$

7. The user sends $(\pi, \mathsf{com}'_{\mathsf{obj}}, \mathsf{cbData}, \mathsf{aux})$ to the service provider

VerifyCreate$(\mathsf{pp}, \mathsf{sk}_{\mathsf{spid}}, \mathsf{obj}', \pi, \mathsf{cbData}, \mathsf{aux})$**.** Let $\mathsf{sk}_{\mathsf{spid}}$ be the service provider's signing key. The service provider performs the following:

1. Verify obj' appears on $\mathsf{bb}_{\mathsf{obj}}$, i.e., that the callback was created;
2. Verify the proof π with respect to inputs cbData, curTime, and obj'; and
3. Unpack aux and verify $\mathsf{cbData.com}_{\mathsf{entry}} = \mathsf{Com}(\mathsf{tik}, \mathsf{exp}, k, \mathsf{meth}'; s_{\mathsf{entry}})$
4. Verify that tik has never been used before in a callback initiated by this service provider

ScanOne$(\mathsf{pp}, \mathsf{obj}, x) \to (\mathsf{obj}', \pi, \mathsf{cbData})$**.** Let obj be the user object, let x be the current time curTime, and let s_{obj} be the commitment randomness to obj. The user does as follows:

1. Clones the obj to a new one obj' and pick a fresh serial number obj'.sn. If $h_{\mathsf{cb}}^{(\mathrm{old})} = h_{\mathsf{cb}}$ (i.e., the beginning of the scanning process), update obj'.loopStartTime := curTime.
2. Let $\mathsf{entry} = (\mathsf{tik}, \mathsf{exp}, k, \mathsf{meth})$ represent the current entry in obj.cbList. If $(\mathsf{tik}, \mathsf{args}, t) \in \mathsf{bb}_{\mathsf{cb}}$ for some args and $t <$ exp, absorb the callback: $\mathsf{obj}' := \mathsf{meth}(\mathsf{obj}, \mathsf{curTime}, m, x)$, where $m = \mathsf{Dec}_k(\mathsf{args})$. If $\mathsf{tik} \notin \mathsf{bb}_{\mathsf{cb}}$ and exp is in the future, then leave entry in obj'.cbList. In any other case, delete entry from obj'.cbList. Finally, update $h'^{(\mathrm{new})}_{\mathsf{cb}} = H(h_{\mathsf{cb}}^{(\mathrm{new})}, \mathsf{entry})$ if the entry was left in, and $h'^{(\mathrm{new})}_{\mathsf{cb}} = h_{\mathsf{cb}}^{(\mathrm{new})}$ otherwise.
3. If $h'^{(\mathrm{old})}_{\mathsf{cb}} = h_{\mathsf{cb}}$ this is the last step of a scanning process. Update obj'.lastFullLoopTime := obj.loopStartTime.
4. Compute a zero-knowledge proof π of R_{settle}.
5. Let $\mathsf{cbData} = (\mathsf{com}_{\mathsf{entry}}, \mathsf{obj.sn})$
6. The user sends $(\pi, \mathsf{com}'_{\mathsf{obj}}, \mathsf{cbData})$ to $\mathsf{bb}_{\mathsf{obj}}$

$$
\left(
\begin{array}{l}
\mathsf{com}'_{\mathsf{obj}}, \mathsf{com}_{\mathsf{entry}}, \mathsf{sn}, t, \mathsf{curTime}, \mathsf{bb}_{\mathsf{obj}}; \\
\mathsf{com}_{\mathsf{obj}}, s_{\mathsf{obj}}, s'_{\mathsf{obj}}, \\
s_{\mathsf{entry}}, \mathsf{tik}, \mathsf{exp}, k, \mathsf{meth}'
\end{array}
\right) :
$$

$$
\begin{array}{l}
\mathsf{com}_{\mathsf{obj}} \in \mathsf{bb}_{\mathsf{obj}} \\
\mathsf{com}_{\mathsf{obj}} = \mathsf{Com}(\mathsf{obj}; s_{\mathsf{obj}}) \\
\mathsf{com}'_{\mathsf{obj}} = \mathsf{Com}(\mathsf{obj}'; s'_{\mathsf{obj}}) \\
\mathsf{com}_{\mathsf{entry}} = \mathsf{Com}(\mathsf{tik} \| \mathsf{exp} \| k \| \mathsf{meth}'; s_{\mathsf{entry}}) \\
\mathsf{obj}.\mathsf{lastFullLoopTime} \geq t \\
\mathsf{obj}'.\mathsf{lastCreated} = \mathsf{curTime} \\
\mathsf{obj}.\mathsf{sn} = \mathsf{sn} \\
\Phi_{\mathsf{create}}(\mathsf{obj}, \mathsf{obj}', \mathsf{meth}', \mathsf{curTime}) = 1 \\
\mathsf{obj}'.h_{\mathsf{cb}} = H(\mathsf{obj}.h_{\mathsf{cb}}, (\mathsf{tik}, \mathsf{exp}, k, \mathsf{meth}')) \\
\mathsf{obj}.h_{\mathsf{cb}}^{(\mathrm{old})} = \mathsf{obj}.h_{\mathsf{cb}}
\end{array}
$$

Figure 5.5: The $R_{\mathsf{create}}^{\Phi_{\mathsf{create}}}$ relation

VerifyMethodExec$(\mathsf{pp}, \mathsf{obj}', \pi, \mathsf{cbData})$**.** In VerifyMethodExec$_{\mathsf{create}}$, the maintainer of the bulletin board $\mathsf{bb}_{\mathsf{obj}}$ receives $(\pi, \mathsf{com}'_{\mathsf{obj}}, \mathsf{sn}, \mathsf{cbData})$. It first checks sn has not appeared in its set of observed serial numbers. Next, it verifies π with respect to cbData and its own curTime and $\mathsf{bb}_{\mathsf{obj}}$ representative. On success, it adds $(\pi, \mathsf{com}'_{\mathsf{obj}}, \mathsf{cbData})$ to $\mathsf{bb}_{\mathsf{obj}}$.

In VerifyMethodExec$_{\mathsf{settle}}$, the maintainer receives the same payload. The behavior is identical to above, with the only change being that it also uses a $\mathsf{bb}_{\mathsf{cb}}$ representative as public input for π verification.

VerifyCall$(\mathsf{pp}, \mathsf{tik}, \mathsf{args}, \mathsf{aux})$**.** The maintainer of the callback bulletin board $\mathsf{bb}_{\mathsf{cb}}$ receives $(\mathsf{tik}, \mathsf{args}, \mathsf{aux} = \sigma)$. It interprets tik as a signature public key pk, and then checks $\Sigma.\mathsf{Verify}_{\mathsf{pk}}(\mathsf{args}, \sigma)$. On success, it posts $(\mathsf{tik}, \mathsf{args}, \mathsf{curTime})$ to $\mathsf{bb}_{\mathsf{cb}}$.

11 Appendix: Membership and non-membership

zk-promises is phrased in terms of bulletin boards and data structures that admit efficiently checkable membership/non-membership arguments. We identify and implement two such variants.

Merkle trees. We may represent our object and used callback sets $\mathsf{bb}_{\mathsf{obj}}$ and $\mathsf{bb}_{\mathsf{cb}}$ as Merkle trees,

$$
\left(\begin{array}{l}
\mathsf{com}_{\mathsf{obj}}, \mathsf{com}'_{\mathsf{obj}}, \mathsf{com}_{\mathsf{entry}}, \mathsf{nul}, t, x, \mathsf{curTime}, \\
\mathsf{bb}_{\mathsf{obj}}, \mathsf{bb}_{\mathsf{cb}}; \\
s_{\mathsf{obj}}, s'_{\mathsf{obj}}, s_{\mathsf{entry}}, \\
\mathsf{wasCalled}, \mathsf{tik}, \mathsf{exp}, k, \mathsf{meth}, \mathsf{timeCalled}, \mathsf{args}, m
\end{array}\right) :
$$

$$
\begin{aligned}
&\mathsf{com}_{\mathsf{obj}} \in \mathsf{bb}_{\mathsf{obj}} \\
&\mathsf{com}_{\mathsf{obj}} = \mathsf{Com}(\mathsf{obj}; s_{\mathsf{obj}}) \\
&\mathsf{com}'_{\mathsf{obj}} = \mathsf{Com}(\mathsf{obj}'; s'_{\mathsf{obj}}) \\
&\mathsf{obj.sn} = \mathsf{sn} \\
&\mathsf{wasCalled} = \mathsf{tik} \in \mathsf{bb}_{\mathsf{cb}} \\
&(\mathsf{timeCalled}, \mathsf{args}) = \textbf{if } \mathsf{wasCalled} : \mathsf{bb}_{\mathsf{cb}}[\mathsf{tik}] \textbf{ else } (\bot, \bot) \\
&m = \mathsf{Dec}_k(\mathsf{args}) \\
&\mathsf{scanning} := \mathsf{obj}.h^{(\mathrm{old})}_{\mathsf{cb}} = \mathsf{obj}.h_{\mathsf{cb}} \\
&\mathsf{scanning}' := \mathsf{obj}'.h^{(\mathrm{old})}_{\mathsf{cb}} = \mathsf{obj}'.h_{\mathsf{cb}} \\
&\mathsf{entry} := (\mathsf{tik}, \mathsf{exp}, k, \mathsf{meth}) \\
&\mathsf{deleteEntry} := \mathsf{curTime} < \mathsf{exp} \text{ and } \mathsf{tik} \notin \mathsf{bb}_{\mathsf{cb}} \\
&\mathsf{absorbEntry} := \mathsf{timeCalled} < \mathsf{exp} \text{ and } \mathsf{tik} \in \mathsf{bb}_{\mathsf{cb}} \\
&\neg\mathsf{absorbEntry} \vee \Phi_{\mathsf{ingest}}(\mathsf{obj}, \mathsf{obj}', \mathsf{meth}, m, \mathsf{curTime}) \\
&\mathsf{obj}'.h^{(\mathrm{old})}_{\mathsf{cb}} = H(\mathsf{obj}.h^{(\mathrm{old})}_{\mathsf{cb}}, \mathsf{entry}) \\
&\mathsf{obj}'.h^{(\mathrm{new})}_{\mathsf{cb}} = \textbf{if } \mathsf{deleteEntry} : \mathsf{obj}.h^{(\mathrm{new})}_{\mathsf{cb}} \\
&\quad \textbf{else } H(\mathsf{obj}.h^{(\mathrm{new})}_{\mathsf{cb}}, \mathsf{entry}) \\
&\mathsf{obj}'.\mathsf{loopStartTime} = \textbf{if } \neg\mathsf{scanning} : \mathsf{curTime} \\
&\quad \textbf{else } \mathsf{obj.loopStartTime} \\
&\mathsf{obj}'.\mathsf{lastFullLoopTime} = \textbf{if } \neg\mathsf{scanning}' : \mathsf{obj.loopStartTime} \\
&\quad \textbf{else } \mathsf{obj.lastSwept} \\
&\mathsf{obj}'.h^{(\mathrm{old})}_{\mathsf{cb}} = \textbf{if } \neg\mathsf{scanning}' : \emptyset \textbf{ else } \mathsf{obj}.h^{(\mathrm{old})}_{\mathsf{cb}} \\
&\mathsf{obj}'.h^{(\mathrm{new})}_{\mathsf{cb}} = \textbf{if } \neg\mathsf{scanning}' : \emptyset \textbf{ else } \mathsf{obj}.h^{(\mathrm{new})}_{\mathsf{cb}} \\
&\mathsf{obj}'.h_{\mathsf{cb}} = \textbf{if } \neg\mathsf{scanning}' : \mathsf{obj}.h^{(\mathrm{new})}_{\mathsf{cb}} \textbf{ else } \mathsf{obj}.h_{\mathsf{cb}}
\end{aligned}
$$

Figure 5.6: The $R^{\Phi_{\mathsf{ingest}}}_{\mathsf{ingest}}$ relation

where the leaves are the elements of the set. To prove of $x \in S$, it suffices to prove knowledge of an authentication path from a leaf with value x to the root of the tree. In these proofs, the root is public input. To keep a proof of membership up to date, it suffices to download a *frontier* of the append-only tree. This permits a communication cost tradeoff of logarithmic to linear, depending on privacy requirements [RWGM23].

To allow non-membership proofs for $\mathsf{bb}_{\mathsf{cb}}$, we simply prove membership in the *complement* of $\mathsf{bb}_{\mathsf{cb}}$, i.e., $C = \{0,1\}^{256} \setminus \{\mathsf{bb}_{\mathsf{cb}}.\mathsf{tik}_i\}_i$. Specifically, we partition C into semi-open ranges of integers $[a,b) \subseteq \{0,1\}^{256}$, and define a Merkle tree T whose leaves are those ranges. Then to prove non-membership in $\mathsf{bb}_{\mathsf{cb}}$, it suffices to show knowledge of a tik, a, b such that $a \leq \mathsf{tik} < b$ and $[a,b)$ is in T. We note that the structure of T is liable to change every time $\mathsf{bb}_{\mathsf{cb}}$ is modified, so proofs of membership are not necessarily updatable using the frontier method.

Signatures. In a centralized bulletin board setting, it is also possible to represent set membership using signatures. The manager of the bulletin boards maintains two signature keypairs $(\mathsf{pk}_{\mathsf{obj}}, \mathsf{sk}_{\mathsf{obj}})$ and $(\mathsf{pk}_{\mathsf{cb}}, \mathsf{sk}_{\mathsf{cb}})$. Every time a value is posted to a bulletin board, the manager signs the value and returns the signature. To prove membership of x in a bulletin board with public key pk, it suffices to prove knowledge of a signature σ such that $\mathsf{Verify}_{\mathsf{pk}}(x)$ is true. Compared to Merkle trees, membership signatures have the benefit of not requiring updating—a valid σ will always be valid regardless of how the corresponding set changes.

Signature non-membership proofs work similarly. As above, we partition the complement set into ranges and prove membership in that signed set. Since the $\mathsf{bb}_{\mathsf{cb}}$ complement set *shrinks* over time, a valid proof of non-membership at time t should not necessarily be valid at time $t+1$, since the value might have become a member of $\mathsf{bb}_{\mathsf{cb}}$ in the meantime. To handle this, we must invalidate every old non-membership signature. We can do this by adding an epoch to every value and making the verification equation $\mathsf{Verify}_{\mathsf{pk}}(x)$, or by picking a new signing key every epoch and publicizing it through the bulletin board. In either case, the bulletin board manager must re-sign the entire complement every epoch.

Circuit	Single Server	De-centralized		
Epoch Capacity:	*Unlimited*	2^{16}	2^{32}	2^{64}
Number of Constraints				
`ShowAuthMakeCB`	28,603	25,352	29,720	38,456
`ShowAuthMakeCB1Sweep`	60,694	55,971	64707	82,179

Table 5.2: The number on constraints in `ShowAuthMakeCB` and `ShowAuthMakeCB1Sweep` circuits.

Batch Size	Single Server	De-centralized		
Epoch Capacity:	*Unlimited*	2^{16}	2^{32}	2^{64}
Number of Constraints				
1	60,694	55,971	64,707	82,179
2	105,872	100,743	113,847	140,055
4	194,210	187,209	209,049	252,729
8	370,886	360,141	399,453	478,077
16	724,238	706,005	780,261	928,773
32	1,430,942	1,397,733	1,541,877	1,830,165
64	2,844,350	2,781,189	3,065,109	3,632,949

Table 5.3: The number on constraints in batched scan circuits.

12 Appendix: Number of constraints

Table 5.2 lists the number on constraints in `ShowAuthMakeCB` and `ShowAuthMakeCB1Sweep` circuits in the centralized and decentralized settings. In the `ShowAuthMakeCB` circuit, the user shows that the record is in good standing and creates a callback. The `ShowAuthMakeCB1Sweep` circuit also does this but also scans for a single callback.

Table 5.3 lists the number on constraints in `BatchedScan` circuits in the centralized and decentralized settings with various batch sizes.

Bibliography

[ACL12] ACLU Blog - Jay Stanley. A Creeping Private-Sector "Checkpoint Society"—and a Small Step to Protect Your Privacy, 2012. URL: https://www.aclu.org/blog/national-security/creeping-private-sector-checkpoint-society-and-small-step-protect-your.

[AHIV17] Scott Ames, Carmit Hazay, Yuval Ishai, and Muthuramakrishnan Venkitasubramaniam. Ligero: Lightweight sublinear arguments without a trusted setup. In Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *ACM CCS 2017*, pages 2087–2104. ACM Press, October / November 2017. doi:10.1145/3133956.3134104.

[AK12] Man Ho Au and Apu Kapadia. PERM: practical reputation-based blacklisting without TTPS. In Ting Yu, George Danezis, and Virgil D. Gligor, editors, *ACM CCS 2012*, pages 929–940. ACM Press, October 2012. doi:10.1145/2382196.2382294.

[AKS12] Man Ho Au, Apu Kapadia, and Willy Susilo. BLACR: TTP-free blacklistable anonymous credentials with reputation. In *NDSS 2012*. The Internet Society, February 2012.

[ale] How Aleo Works. URL: https://www.aleo.org/how-aleo-works/.

[Ame06] American Civil Liberties Union. Librarians Speak Out for First Time After Being Gagged by Patriot Act, 2006. URL: https://www.aclu.org/press-releases/librarians-speak-out-first-time-after-being-gagged-patriot-act.

[App21] Apple. iCloud Private Relay Overview, 2021. URL: https://www.apple.com/icloud/docs/iCloud_Private_Relay_Overview_Dec2021.pdf.

[Ar21] Arkworks-rs. *Arkworks Ecosystem Homepage*, 2021. URL: https://arkworks.rs/.

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

[Atw09] Jeff Atwood. New question / answer rate limits, Feb 2009. URL: https://stackoverflow.blog/2009/02/21/new-question-answer-rate-limits/.

[BBB+18] Benedikt Bünz, Jonathan Bootle, Dan Boneh, Andrew Poelstra, Pieter Wuille, and Greg Maxwell. Bulletproofs: Short proofs for confidential transactions and more. In IEEE S&P 2018 [IEE18], pages 315–334. doi:10.1109/SP.2018.00020.

[BBF19] Dan Boneh, Benedikt Bünz, and Ben Fisch. Batching techniques for accumulators with applications to IOPs and stateless blockchains. In Alexandra Boldyreva and Daniele Micciancio, editors, *CRYPTO 2019, Part I*, volume 11692 of *LNCS*, pages 561–586. Springer, Heidelberg, August 2019. doi:10.1007/978-3-030-26948-7_20.

[BBHR18] Eli Ben-Sasson, Iddo Bentov, Yinon Horesh, and Michael Riabzev. Scalable, transparent, and post-quantum secure computational integrity. Cryptology ePrint Archive, Report 2018/046, 2018. https://eprint.iacr.org/2018/046.

[BBHR19] Eli Ben-Sasson, Iddo Bentov, Yinon Horesh, and Michael Riabzev. Scalable zero knowledge with no trusted setup. In Alexandra Boldyreva and Daniele Micciancio, editors, *CRYPTO 2019, Part III*, volume 11694 of *LNCS*, pages 701–732. Springer, Heidelberg, August 2019. doi:10.1007/978-3-030-26954-8_23.

[BBS04] Dan Boneh, Xavier Boyen, and Hovav Shacham. Short group signatures. In Franklin [Fra04], pages 41–55. doi:10.1007/978-3-540-28628-8_3.

[BCC04] Ernest F. Brickell, Jan Camenisch, and Liqun Chen. Direct anonymous attestation. In Vijayalakshmi Atluri, Birgit Pfitzmann, and Patrick McDaniel, editors, *ACM CCS 2004*, pages 132–145. ACM Press, October 2004. doi:10.1145/1030083.1030103.

[BCCT12] Nir Bitansky, Ran Canetti, Alessandro Chiesa, and Eran Tromer. Recursive composition and bootstrapping for SNARKs and proof-carrying data. Cryptology ePrint Archive, Report 2012/095, 2012. https://eprint.iacr.org/2012/095.

[BCD+17] Foteini Baldimtsi, Jan Camenisch, Maria Dubovitskaya, Anna Lysyanskaya, Leonid Reyzin, Kai Samelin, and Sophia Yakoubov. Accumulators with applications to anonymity-preserving revocation. Cryptology ePrint Archive, Report 2017/043, 2017. https://eprint.iacr.org/2017/043.

[BCFK19] Daniel Benarroch, Matteo Campanelli, Dario Fiore, and Dimitris Kolonelos. Zero-knowledge proofs for set membership: Efficient, succinct, modular. Cryptology ePrint Archive, Report 2019/1255, 2019. https://eprint.iacr.org/2019/1255.

[BCG+14] Eli Ben-Sasson, Alessandro Chiesa, Christina Garman, Matthew Green, Ian Miers, Eran Tromer, and Madars Virza. Zerocash: Decentralized anonymous payments from bitcoin. In *2014 IEEE Symposium on Security and Privacy*, pages 459–474. IEEE Computer Society Press, May 2014. doi:10.1109/SP.2014.36.

[BCG+15] Eli Ben-Sasson, Alessandro Chiesa, Matthew Green, Eran Tromer, and Madars Virza. Secure sampling of public parameters for succinct zero knowledge proofs. In *2015 IEEE Symposium on Security and Privacy*, pages 287–304. IEEE Computer Society Press, May 2015. doi:10.1109/SP.2015.25.

[BCG+20] Sean Bowe, Alessandro Chiesa, Matthew Green, Ian Miers, Pratyush Mishra, and Howard Wu. ZEXE: Enabling decentralized private computation. In *2020 IEEE Symposium on Security and Privacy*, pages 947–964. IEEE Computer Society Press, May 2020. doi:10.1109/SP40000.2020.00050.

[BCKL08] Mira Belenkiy, Melissa Chase, Markulf Kohlweiss, and Anna Lysyanskaya. P-signatures and noninteractive anonymous credentials. In Ran Canetti, editor, *TCC 2008*, volume 4948 of *LNCS*, pages 356–374. Springer, Heidelberg, March 2008. doi:10.1007/978-3-540-78524-8_20.

[BCL+21] Benedikt Bünz, Alessandro Chiesa, William Lin, Pratyush Mishra, and Nicholas Spooner. Proof-carrying data without succinct arguments. In Malkin and Peikert [MP21], pages 681–710. doi:10.1007/978-3-030-84242-0_24.

[BCMS20] Benedikt Bünz, Alessandro Chiesa, Pratyush Mishra, and Nicholas Spooner. Proof-carrying data from accumulation schemes. Cryptology ePrint Archive, Report 2020/499, 2020. https://eprint.iacr.org/2020/499.

[BDFG21] Dan Boneh, Justin Drake, Ben Fisch, and Ariel Gabizon. Halo infinite: Proof-carrying data from additive polynomial commitments. In Malkin and Peikert [MP21], pages 649–680. doi:10.1007/978-3-030-84242-0_23.

[BG13] Stephanie Bayer and Jens Groth. Zero-knowledge argument for polynomial evaluation with application to blacklists. In Thomas Johansson and Phong Q. Nguyen, editors, *EUROCRYPT 2013*, volume 7881 of *LNCS*, pages 646–663. Springer, Heidelberg, May 2013. doi:10.1007/978-3-642-38348-9_38.

[BG21] Michael Bailey and Rachel Greenstadt, editors. *USENIX Security 2021*. USENIX Association, August 2021.

[BGH19] Sean Bowe, Jack Grigg, and Daira Hopwood. Halo: Recursive proof composition without a trusted setup. Cryptology ePrint Archive, Report 2019/1021, 2019. https://eprint.iacr.org/2019/1021.

[BGM17] Sean Bowe, Ariel Gabizon, and Ian Miers. Scalable multi-party computation for zk-SNARK parameters in the random beacon model. Cryptology ePrint Archive, Report 2017/1050, 2017. https://eprint.iacr.org/2017/1050.

[BKSV20] Karim Baghery, Markulf Kohlweiss, Janno Siim, and Mikhail Volkhov. Another look at extraction and randomization of groth's zk-SNARK. Cryptology ePrint Archive, Report 2020/811, 2020. https://eprint.iacr.org/2020/811.

[BKSV21] Karim Baghery, Markulf Kohlweiss, Janno Siim, and Mikhail Volkhov. Another look at extraction and randomization of groth's zk-snark. In *Financial Cryptography and Data Security*, 2021.

[BL09] Ernie Brickell and Jiangtao Li. Enhanced privacy id from bilinear pairing. Cryptology ePrint Archive, Report 2009/095, 2009. https://ia.cr/2009/095.

[BL13] Foteini Baldimtsi and Anna Lysyanskaya. Anonymous credentials light. In Ahmad-Reza Sadeghi, Virgil D. Gligor, and Moti Yung, editors, *ACM CCS 2013*, pages 1087–1098. ACM Press, November 2013. doi:10.1145/2508859.2516687.

[BMM+19] Benedikt Bünz, Mary Maller, Pratyush Mishra, Nirvan Tyagi, and Psi Vesely. Proofs for inner pairing products and applications. Cryptology ePrint Archive, Report 2019/1177, 2019. URL: https://eprint.iacr.org/2019/1177.

[BMM+21] Benedikt Bünz, Mary Maller, Pratyush Mishra, Nirvan Tyagi, and Psi Vesely. Proofs for inner pairing products and applications. In Mehdi Tibouchi and Huaxiong Wang, editors, *ASIACRYPT 2021, Part III*, volume 13092 of *LNCS*, pages 65–97. Springer, Heidelberg, December 2021. doi:10.1007/978-3-030-92078-4_3.

[BMW03] Mihir Bellare, Daniele Micciancio, and Bogdan Warinschi. Foundations of group signatures: Formal definitions, simplified requirements, and a construction based on general assumptions. In Eli Biham, editor, *EUROCRYPT 2003*, volume 2656 of *LNCS*, pages 614–629. Springer, Heidelberg, May 2003. doi:10.1007/3-540-39200-9_38.

[Bow17a] Sean Bowe. *Bellman: zk-SNARKs in Rust*, 2017. URL: https://electriccoin.co/blog/bellman-zksnarks-in-rust/.

[Bow17b] Sean Bowe. *BLS12-381: New zk-SNARK Elliptic Curve Construction*, 2017. URL: https://electriccoin.co/blog/new-snark-curve/.

[But] Vitalik Buterin. An Incomplete Guide to Rollups. URL: https://vitalik.ca/general/2021/01/05/rollup.html.

[CCDW20] Weikeng Chen, Alessandro Chiesa, Emma Dauterman, and Nicholas P. Ward. Reducing participation costs via incremental verification for ledger systems. Cryptology ePrint Archive, Report 2020/1522, 2020. https://ia.cr/2020/1522.

[CDHK15] Jan Camenisch, Maria Dubovitskaya, Kristiyan Haralambiev, and Markulf Kohlweiss. Composable and modular anonymous credentials: Definitions and practical constructions. In Tetsu Iwata and Jung Hee Cheon, editors, *ASIACRYPT 2015, Part II*, volume 9453 of *LNCS*, pages 262–288. Springer, Heidelberg, November / December 2015. doi:10.1007/978-3-662-48800-3_11.

[CDL16] Jan Camenisch, Manu Drijvers, and Anja Lehmann. Anonymous attestation using the strong Diffie Hellman assumption revisited. Cryptology ePrint Archive, Report 2016/663, 2016. https://eprint.iacr.org/2016/663.

[CG08] Jan Camenisch and Thomas Groß. Efficient attributes for anonymous credentials. In Ning et al. [NSJ08], pages 345–356. doi:10.1145/1455770.1455814.

[Cha81] David L. Chaum. Untraceable electronic mail, return addresses, and digital pseudonyms. *Commun. ACM*, 24(2):84–90, feb 1981. doi:10.1145/358549.358563.

[Cha82] David Chaum. Blind signatures for untraceable payments. In David Chaum, Ronald L. Rivest, and Alan T. Sherman, editors, *CRYPTO'82*, pages 199–203. Plenum Press, New York, USA, 1982.

[Cha85] David Chaum. Security without identification: Transaction systems to make big brother obsolete. *Communications of the ACM*, 1985.

[CHK+06] Jan Camenisch, Susan Hohenberger, Markulf Kohlweiss, Anna Lysyanskaya, and Mira Meyerovich. How to win the clonewars: Efficient periodic n-times anonymous authentication. In Ari Juels, Rebecca N. Wright, and Sabrina De Capitani di Vimercati, editors, *ACM CCS 2006*, pages 201–210. ACM Press, October / November 2006. doi:10.1145/1180405.1180431.

[CHL05] Jan Camenisch, Susan Hohenberger, and Anna Lysyanskaya. Compact e-cash. In Ronald Cramer, editor, *EUROCRYPT 2005*, volume 3494 of *LNCS*, pages 302–321. Springer, Heidelberg, May 2005. doi:10.1007/11426639_18.

[CHM+20] Alessandro Chiesa, Yuncong Hu, Mary Maller, Pratyush Mishra, Psi Vesely, and Nicholas P. Ward. Marlin: Preprocessing zkSNARKs with universal and updatable SRS. In Anne Canteaut and Yuval Ishai, editors, *EUROCRYPT 2020, Part I*, volume 12105 of *LNCS*, pages 738–768. Springer, Heidelberg, May 2020. doi:10.1007/978-3-030-45721-1_26.

[CKWK19] Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz, editors. *ACM CCS 2019*. ACM Press, November 2019.

[CL01] Jan Camenisch and Anna Lysyanskaya. An efficient system for non-transferable anonymous credentials with optional anonymity revocation. In Birgit Pfitzmann, editor, *EUROCRYPT 2001*, volume 2045 of *LNCS*, pages 93–118. Springer, Heidelberg, May 2001. doi:10.1007/3-540-44987-6_7.

[CL02] Jan Camenisch and Anna Lysyanskaya. Dynamic accumulators and application to efficient revocation of anonymous credentials. In Moti Yung, editor, *CRYPTO 2002*, volume 2442 of *LNCS*, pages 61–76. Springer, Heidelberg, August 2002. doi:10.1007/3-540-45708-9_5.

[CL03] Jan Camenisch and Anna Lysyanskaya. A signature scheme with efficient protocols. In Stelvio Cimato, Clemente Galdi, and Giuseppe Persiano, editors, *SCN 02*, volume 2576 of *LNCS*, pages 268–289. Springer, Heidelberg, September 2003. doi:10.1007/3-540-36413-7_20.

[CL04] Jan Camenisch and Anna Lysyanskaya. Signature schemes and anonymous credentials from bilinear maps. In Franklin [Fra04], pages 56–72. doi:10.1007/978-3-540-28628-8_4.

[CPZ20] Melissa Chase, Trevor Perrin, and Greg Zaverucha. The Signal private group system and anonymous credentials supporting efficient verifiable encryption. In Jay Ligatti, Xinming Ou, Jonathan Katz, and Giovanni Vigna, editors, *ACM CCS 2020*, pages 1445–1459. ACM Press, November 2020. doi:10.1145/3372297.3417887.

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

[CS03] Jan Camenisch and Victor Shoup. Practical verifiable encryption and decryption of discrete logarithms. In Dan Boneh, editor, *CRYPTO 2003*, volume 2729 of *LNCS*, pages 126–144. Springer, Heidelberg, August 2003. doi:10.1007/978-3-540-45146-4_8.

[CSF+08] D. Cooper, S. Santesson, S. Farrell, S. Boeyen, R. Housley, and W. Polk. *RFC 5280 - Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile*. Internet Activities Board, May 2008. doi:10.17487/RFC5280.

[CT10] Alessandro Chiesa and Eran Tromer. Proof-carrying data and hearsay arguments from signature cards. In Andrew Chi-Chih Yao, editor, *ICS 2010*, pages 310–331. Tsinghua University Press, January 2010.

[Cv91] David Chaum and Eugène van Heyst. Group signatures. In Donald W. Davies, editor, *EUROCRYPT'91*, volume 547 of *LNCS*, pages 257–265. Springer, Heidelberg, April 1991. doi:10.1007/3-540-46416-6_22.

[CV02] Jan Camenisch and Els Van Herreweghen. Design and implementation of the idemix anonymous credential system. In Vijayalakshmi Atluri, editor, *ACM CCS 2002*, pages 21–30. ACM Press, November 2002. doi:10.1145/586110.586114.

[DFKP16] Antoine Delignat-Lavaud, Cédric Fournet, Markulf Kohlweiss, and Bryan Parno. Cinderella: Turning shabby X.509 certificates into elegant anonymous credentials with the magic of verifiable computation. In IEEE S&P 2016 [IEE16], pages 235–254. doi:10.1109/SP.2016.22.

[DGS+18] Alex Davidson, Ian Goldberg, Nick Sullivan, George Tankersley, and Filippo Valsorda. Privacy pass: Bypassing internet challenges anonymously. *PoPETs*, 2018(3):164–180, July 2018. doi:10.1515/popets-2018-0026.

[DMMK18] Debajyoti Das, Sebastian Meiser, Esfandiar Mohammadi, and Aniket Kate. Anonymity trilemma: Strong anonymity, low bandwidth overhead, low latency - choose two. In IEEE S&P 2018 [IEE18], pages 108–126. doi:10.1109/SP.2018.00011.

[e-E] e-Estonia. ID-card. URL: https://e-estonia.com/solutions/e-identity/id-card/.

[EJN] Steve Ellis, Ari Juels, and Sergey Nazarov. Chainlink: A decentralized oracle network. URL: https://research.chain.link/whitepaper-v1.pdf.

[Eth] Ethereum Development Docs. Optimistic rollups. URL: https://ethereum.org/en/developers/docs/scaling/optimistic-rollups/.

[ets] Etsy Engineering | Threat Modeling for Marketing Campaigns. Section: Consumer Product Development. URL: https://www.etsy.com/codeascraft/threat-modeling-for-marketing-campaigns?utm_source=OpenGraph&utm_medium=PageTools&utm_campaign=Share.

[Fac] Facebook. Help Center - What names are allowed on Facebook? URL: https://www.facebook.com/help/112146705538576.

[FCC20] FCC. Internet access services: Status as of december 31, 2018. https://docs.fcc.gov/public/attachments/DOC-366980A1.pdf, 2020.

[Fed] Federal Office for Information Security. German eID. URL: https://www.bsi.bund.de/EN/Topics/ElectrIDDocuments/German-eID/german-eID.html.

[FKL18] Georg Fuchsbauer, Eike Kiltz, and Julian Loss. The algebraic group model and its applications. In Hovav Shacham and Alexandra Boldyreva, editors, *CRYPTO 2018, Part II*, volume 10992 of *LNCS*, pages 33–62. Springer, Heidelberg, August 2018. doi:10.1007/978-3-319-96881-0_2.

[FKL+21] Nicholas Franzese, Jonathan Katz, Steve Lu, Rafail Ostrovsky, Xiao Wang, and Chenkai Weng. Constant-overhead zero-knowledge for RAM programs. In Giovanni Vigna and Elaine Shi, editors, *ACM CCS 2021*, pages 178–191. ACM Press, November 2021. doi:10.1145/3460120.3484800.

[Fra04] Matthew Franklin, editor. *CRYPTO 2004*, volume 3152 of *LNCS*. Springer, Heidelberg, August 2004.

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

[Fuc17] Georg Fuchsbauer. Subversion-zero-knowledge SNARKs. Cryptology ePrint Archive, Report 2017/587, 2017. https://eprint.iacr.org/2017/587.

[Fuc18] Georg Fuchsbauer. Subversion-zero-knowledge SNARKs. In Michel Abdalla and Ricardo Dahab, editors, *PKC 2018, Part I*, volume 10769 of *LNCS*, pages 315–347. Springer, Heidelberg, March 2018. doi:10.1007/978-3-319-76578-5_11.

[GGM14] Christina Garman, Matthew Green, and Ian Miers. Decentralized anonymous credentials. In *NDSS 2014*. The Internet Society, February 2014.

[GKK+19] Lorenzo Grassi, Daniel Kales, Dmitry Khovratovich, Arnab Roy, Christian Rechberger, and Markus Schofnegger. Starkad and Poseidon: New hash functions for zero knowledge proof systems. Cryptology ePrint Archive, Report 2019/458, 2019. https://eprint.iacr.org/2019/458.

[GKR+21] Lorenzo Grassi, Dmitry Khovratovich, Christian Rechberger, Arnab Roy, and Markus Schofnegger. Poseidon: A new hash function for zero-knowledge proof systems. In Bailey and Greenstadt [BG21], pages 519–535.

[GMN22] Nicolas Gailly, Mary Maller, and Anca Nitulescu. Snarkpack: Practical snark aggregation. In Ittay Eyal and Juan Garay, editors, *Financial Cryptography and Data Security*, pages 203–229, Cham, 2022. Springer International Publishing.

[Gooa] Google. Access age-restricted content & features - Google Account Help. URL: https://support.google.com/accounts/answer/10071085?p=age-verify.

[Goob] Google. Watch age-restricted videos. URL: https://support.google.com/youtube/answer/10070779.

[Gro16] Jens Groth. On the size of pairing-based non-interactive arguments. In Marc Fischlin and Jean-Sébastien Coron, editors, *EUROCRYPT 2016, Part II*, volume 9666 of *LNCS*, pages 305–326. Springer, Heidelberg, May 2016. doi:10.1007/978-3-662-49896-5_11.

[GWC19] Ariel Gabizon, Zachary J. Williamson, and Oana Ciobotaru. PLONK: Permutations over lagrange-bases for oecumenical noninteractive arguments of knowledge. Cryptology ePrint Archive, Report 2019/953, 2019. https://eprint.iacr.org/2019/953.

[HBHW21] Daira Hopwood, Sean Bowe, Taylor Hornby, and Nathan Wilcox. ZCash protocol specification version 2021.2.11. Technical report, The ZCash Foundation, 2021.

[HG11] Ryan Henry and Ian Goldberg. Formalizing anonymous blacklisting systems. In *2011 IEEE Symposium on Security and Privacy*, pages 81–95. IEEE Computer Society Press, May 2011. doi:10.1109/SP.2011.13.

[HKR19] Max Hoffmann, Michael Klooß, and Andy Rupp. Efficient zero-knowledge arguments in the discrete log setting, revisited. In Cavallaro et al. [CKWK19], pages 2093–2110. doi:10.1145/3319535.3354251.

[ICA] ICAO. Machine Readable Travel Documents. Part 10: Logical Data Structure (LDS) for Storage of Biometrics and Other Data in the Contactless Integrated Circuit (IC). Technical report, International Civil Aviation Organization (ICAO). URL: https://www.icao.int/publications/Documents/9303_p10_cons_en.pdf.

[IDN] IDNYC. About IDNYC. URL: https://www1.nyc.gov/site/idnyc/about/about.page.

[IEE16] *2016 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, May 2016.

[IEE18] *2018 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, May 2018.

[IEE22] *2022 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, May 2022.

[JKTS07] Peter C. Johnson, Apu Kapadia, Patrick P. Tsang, and Sean W. Smith. Nymble: Anonymous IP-address blocking. In Nikita Borisov and Philippe Golle, editors, *PET 2007*, volume 4776 of *LNCS*, pages 113–133. Springer, Heidelberg, June 2007. doi:10.1007/978-3-540-75551-7_8.

[JNV] URL: https://nbviewer.org/github/wikimedia-research/automoderator-measurement/blob/main/baselines/T348860_median_time_to_revert.ipynb#Time-to-Revert-Percentiles.

[KG21] Chelsea Komlo and Ian Goldberg. Frost: Flexible round-optimized schnorr threshold signatures. In *Selected Areas in Cryptography*, 2021.

[KMS+16] Ahmed E. Kosba, Andrew Miller, Elaine Shi, Zikai Wen, and Charalampos Papamanthou. Hawk: The blockchain model of cryptography and privacy-preserving smart contracts. In IEEE S&P 2016 [IEE16], pages 839–858. doi:10.1109/SP.2016.55.

[KMSV21] Markulf Kohlweiss, Mary Maller, Janno Siim, and Mikhail Volkhov. Snarky ceremonies. Cryptology ePrint Archive, Report 2021/219, 2021. https://eprint.iacr.org/2021/219.

[KST21] Abhiram Kothapalli, Srinath T. V. Setty, and Ioanna Tzialla. Nova: Recursive zero-knowledge arguments from folding schemes. Cryptology ePrint Archive, Report 2021/370, 2021. URL: https://eprint.iacr.org/2021/370.

[Kus18] John Kuszmaul. Verkle trees. Technical report, MIT, 2018.

[KZG10] Aniket Kate, Gregory M. Zaverucha, and Ian Goldberg. Constant-size commitments to polynomials and their applications. In Masayuki Abe, editor, *ASIACRYPT 2010*, volume 6477 of *LNCS*, pages 177–194. Springer, Heidelberg, December 2010. doi:10.1007/978-3-642-17373-8_11.

[KZM+16] Ahmed E. Kosba, Zhichao Zhao, Andrew K. Miller, Yi Qian, T-H. Hubert Chan, Charalampos Papamanthou, and Elaine Shi. C∅c∅: A framework for building composable zero-knowledge proofs. 2016. URL: https://eprint.iacr.org/2015/1093.

[Leo] Leo | Zero-Knowledge Programming Language. URL: https://leo-lang.org/.

[LH10] Zi Lin and Nicholas Hopper. Jack: Scalable accumulator-based nymble system. pages 53–62, 01 2010. doi:10.1145/1866919.1866927.

[Lit] Litviniuk, Anna. Supportfemale 2 SVG. CC Attribution License https://creativecommons.org/licenses/by/3.0/.

[LLX07] Jiangtao Li, Ninghui Li, and Rui Xue. Universal accumulators with efficient nonmembership proofs. In Jonathan Katz and Moti Yung, editors, *ACNS 07*, volume 4521 of *LNCS*, pages 253–269. Springer, Heidelberg, June 2007. doi:10.1007/978-3-540-72738-5_17.

[Lyn] Ben Lynn. PBC Library - Pairing-Based Cryptography - About. URL: https://crypto.stanford.edu/pbc/.

[MBKM19] Mary Maller, Sean Bowe, Markulf Kohlweiss, and Sarah Meiklejohn. Sonic: Zero-knowledge SNARKs from linear-size universal and updatable structured reference strings. In Cavallaro et al. [CKWK19], pages 2111–2128. doi:10.1145/3319535.3339817.

[MC23] Jack P. K. Ma and Sherman S. M. Chow. SMART Credentials in the Multi-queue of Slackness (or Secure Management of Anonymous Reputation Traits without Global Halting). In *2023 IEEE 8th European Symposium on Security and Privacy (EuroS&P)*, pages 896–912, Delft, Netherlands, July 2023. IEEE. URL: https://ieeexplore.ieee.org/document/10190534/, doi:10.1109/EuroSP57164.2023.00057.

[MP21] Tal Malkin and Chris Peikert, editors. *CRYPTO 2021, Part I*, volume 12825 of *LNCS*, Virtual Event, August 2021. Springer, Heidelberg.

[MTBI+22] Jose L. Muñoz-Tapia, Marta Belles, Miguel Isabel, Albert Rubio, and Jordi Baylina. CIRCOM: A Robust and Scalable Language for Building Complex Zero-Knowledge Circuits. 2022. URL: https://www.techrxiv.org/articles/preprint/CIRCOM_A_Robust_and_Scalable_Language_for_Building_Complex_Zero-Knowledge_Circuits/19374986.

[New] New York State Senate. Find your senator. URL: https://www.nysenate.gov/registration/nojs/form/start/find-my-senator.

[Noi] Introducing Noir | Noir Documentation. URL: https://noir-lang.org/.

[NSJ08] Peng Ning, Paul F. Syverson, and Somesh Jha, editors. *ACM CCS 2008*. ACM Press, October 2008.

[NYC] NYC Planning. Political and Administrative Districts — Download and Metadata. URL: https://www1.nyc.gov/site/planning/data-maps/open-data/districts-download-metadata.page.

[OBW22] Alex Ozdemir, Fraser Brown, and Riad S. Wahby. CirC: Compiler infrastructure for proof systems, software verification, and more. In IEEE S&P 2022 [IEE22], pages 2248–2266. doi:10.1109/SP46214.2022.9833782.

[PCL+07] Reid Priedhorsky, Jilin Chen, Shyong (Tony) K Lam, Katherine Panciera, Loren Terveen, and John Riedl. Creating, destroying, and restoring value in wikipedia. In *Proceedings of the 2007 ACM international conference on supporting group work*, pages 259–268, 2007.

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

[PSS] Alexey Pertsev, Roman Semenov, and Roman Storm. Tornado Cash Privacy Solution Version 1.4.

[Rad21] Radiolab. *The Ceremony*, 2021. URL: https://www.wnycstudios.org/podcasts/radiolab/articles/ceremony.

[rai] RAILGUN project whitepaper. URL: https://assets.railgun.org/docs/whitepaper/Railgun%20Project%20Whitepaper%20-%20July%202021.pdf.

[Rea] ReadID. Which countries have ePassports? URL: https://www.readid.com/blog/countries-epassports.

[Red20] Reddit Staff. Reddit in 2020. (https://old.reddit.com/r/blog/comments/k967mm/reddit_in_2020/, 2020.

[RMM22] Michael Rosenberg, Mary Maller, and Ian Miers. SNARKBlock: Federated anonymous blocklisting from hidden common input aggregate proofs. In IEEE S&P 2022 [IEE22], pages 948–965. Full version at https://ia.cr/2021/1577. doi:10.1109/SP46214.2022.9833656.

[RPX+22] Deevashwer Rathee, Guru Vamsi Policharla, Tiancheng Xie, Ryan Cottone, and Dawn Song. Zebra: Anonymous credentials with practical on-chain verification and applications to kyc in defi. Cryptology ePrint Archive, Paper 2022/1286, 2022. URL: https://eprint.iacr.org/2022/1286.

[RWGM23] Michael Rosenberg, Jacob D. White, Christina Garman, and Ian Miers. zk-creds: Flexible anonymous credentials from zkSNARKs and existing identity infrastructure. In *2023 IEEE Symposium on Security and Privacy*, pages 790–808. IEEE Computer Society Press, May 2023. Full version at https://ia.cr/2022/878. doi:10.1109/SP46215.2023.10179430.

[SAB+19] Alberto Sonnino, Mustafa Al-Bassam, Shehar Bano, Sarah Meiklejohn, and George Danezis. Coconut: Threshold issuance selective disclosure credentials with applications to distributed ledgers. In *NDSS 2019*. The Internet Society, February 2019.

[Sal19] Sara Salinas. Facebook pays the reviewers filtering porn and murder a tiny fraction of its median salary, explosive report says. https://www.cnbc.com/2019/02/25/facebook-pays-content-moderators-a-fraction-of-median-salary-report.html, Feb 2019.

[San22] Olivier Sanders. EPID with efficient proof of non-revocation. In Yuji Suga, Kouichi Sakurai, Xuhua Ding, and Kazue Sako, editors, *ASIACCS 22*, pages 1126–1138. ACM Press, May / June 2022. doi:10.1145/3488932.3517392.

[SPG21] Michael A. Specter, Sunoo Park, and Matthew Green. KeyForge: Non-attributable email from forward-forgeable signatures. In Bailey and Greenstadt [BG21], pages 1755–1773.

[SRM24] Maurice Shih, Michael Rosenberg, and Ian Miers. zk-promises: Making zero-knowledge objects accept the call, and applications to anonymous reputation and access control, 2024. Submitted to CCS 2024.

[TAKS08] Patrick P. Tsang, Man Ho Au, Apu Kapadia, and Sean W. Smith. PEREA: towards practical TTP-free revocation in anonymous authentication. In Ning et al. [NSJ08], pages 333–344. doi:10.1145/1455770.1455813.

[TAKS10] Patrick P. Tsang, Man Ho Au, Apu Kapadia, and Sean W. Smith. BLAC: Revoking Repeatedly Misbehaving Anonymous Users without Relying on TTPs. *ACM Transactions on Information and System Security*, 13(4):1–33, December 2010. URL: https://dl.acm.org/doi/10.1145/1880022.1880033, doi:10.1145/1880022.1880033.

[TFS04] Isamu Teranishi, Jun Furukawa, and Kazue Sako. k-Times anonymous authentication (extended abstract). In Pil Joong Lee, editor, *ASIACRYPT 2004*, volume 3329 of *LNCS*, pages 308–322. Springer, Heidelberg, December 2004. doi:10.1007/978-3-540-30539-2_22.

[TKCS11] P. P. Tsang, A. Kapadia, C. Cornelius, and S. W. Smith. Nymble: Blocking misbehaving users in anonymizing networks. *IEEE Transactions on Dependable and Secure Computing*, 8(2):256–269, 2011. doi:10.1109/TDSC.2009.38.

[TLS] TLSNotary. Prove an HTTPS page was in your browser. URL: https://tlsnotary.org/.

[Tora] Tor Blog. "One cell is enough to break Tor's anonymity". URL: https://blog.torproject.org/one-cell-enough-break-tors-anonymity/.

[Torb] Tornado Cash. Core deposit circuit. URL: https://docs.tornado.cash/tornado-cash-classic/circuits/core-deposit-circuit.

[U.S] U.S. Department of State, Bureau of Consular Affairs. Passport Services: Processing Times. URL: https://travel.state.gov/content/travel/en/passports/how-apply/processing-times.html.

[VB20] Giuseppe Vitto and Alex Biryukov. Dynamic universal accumulator with batch update over bilinear groups. Cryptology ePrint Archive, Report 2020/777, 2020. https://eprint.iacr.org/2020/777.

[Veh] Florida Highway Safety and Motor Vehicles. 2D barcode reader calibration sheet — 2016 AAMVA standard. URL: https://www.flhsmv.gov/pdf/newdl/fl_2dbarcodecalibration.pdf.

[vSSY+22] Stephan van Schaik, Alex Seto, Thomas Yurek, Adam Batori, Bader AlBassam, Christina Garman, Daniel Genkin, Andrew Miller, Eyal Ronen, and Yuval Yarom. SoK: SGX.Fail: How stuff get eXposed. 2022.

[W3C] W3C Recommendation. Verifiable Credentials Data Model v1.1. URL: https://www.w3.org/TR/2022/REC-vc-data-model-20220303/.

[Wika] https://stats.wikimedia.org/#/en.wikipedia.org/contributing/edits/normal|bar|2020-11-04~2021-11-24|~total|monthly.

[Wikb] URL: https://stats.wikimedia.org/#/en.wikipedia.org.

[Wil] Dr. Zachary J. Williamson. The AZTEC protocol. URL: https://github.com/AztecProtocol/aztec-v1/blob/master/AZTEC.pdf.

[WZC+18] Howard Wu, Wenting Zheng, Alessandro Chiesa, Raluca Ada Popa, and Ion Stoica. DIZK: A distributed zero knowledge proof system. In William Enck and Adrienne Porter Felt, editors, *USENIX Security 2018*, pages 675–692. USENIX Association, August 2018.

[XCZ+22] Alex Luoyuan Xiong, Binyi Chen, Zhenfei Zhang, Benedikt Bünz, Ben Fisch, Fernando Krell, and Philippe Camacho. VERI-ZEXE: Decentralized private computation with universal setup. Cryptology ePrint Archive, Report 2022/802, 2022. https://eprint.iacr.org/2022/802.

[XF14] Li Xi and Dengguo Feng. Farb: Fast anonymous reputation-based blacklisting without ttps. *Proceedings of the 13th Workshop on Privacy in the Electronic Society*, 2014. URL: https://api.semanticscholar.org/CorpusID:11490543.

[ZBK+22] Arantxa Zapico, Vitalik Buterin, Dmitry Khovratovich, Mary Maller, Anca Nitulescu, and Mark Simkin. Caulk: Lookup arguments in sublinear time. Cryptology ePrint Archive, Paper 2022/621, 2022. URL: https://eprint.iacr.org/2022/621.

[ZCa19] ZCash. What is jubjub?, 2019. URL: https://web.archive.org/web/20221104080552/https://z.cash/technology/jubjub/.

[ZMM+20] Fan Zhang, Deepak Maram, Harjasleen Malvai, Steven Goldfeder, and Ari Juels. Deco: Liberating web data using decentralized oracles for tls. In *ACM CCS 2020*, 2020.

[Zok] Introduction - ZoKrates. URL: https://zokrates.github.io/.