ALPACA: Anonymous Blocklisting with Constant-Sized Updatable Proofs

Jiwon Kim
University of Michigan

Abhiram Kothapalli
University of California, Berkeley

Orestis Chardouvelis
Carnegie Mellon University

Riad S. Wahby
Carnegie Mellon University

Paul Grubbs
University of Michigan

***Abstract*—In recent years, online anonymity has become increasingly important but is under threat due to the challenges of moderating anonymous spaces. A promising cryptographic solution, known as *anonymous blocklisting*, allows users to post anonymously while still enabling moderation. Moderation via anonymous blocklisting roughly works by requiring that when users post a message they attach a cryptographic proof that they did not author any posts on a "blocklist".**

Existing anonymous blocklisting schemes are unfortunately still far from achieving practical performance for large blocklists. This is essentially due to all prior works requiring a user to (cryptographically) reprocess blocklist entries many times. Relatedly, prior works have relatively high verification times and proof sizes.

In this work, we introduce ALPACA, the first anonymous blocklisting system with the property that a user only needs to do a *constant* amount of work per blocklist entry. Thus, our scheme has asymptotically optimal performance. Our scheme is also the first to have verification times and proof sizes that are independent of the number of blocklist entries.

Our key technique is a new variant of *incrementally verifiable computation* (IVC), designed to ensure anonymity. Along the way, we introduce new definitions to formally establish security. On a mid-range laptop, ALPACA's proof generation time is always 6.15 seconds and proof size is 25.6KBs. On a server, the verification time is always 400ms.

1. Introduction

Anonymous spaces, in which a user does not need to divulge their identity to participate, have historically been an extremely important part of how users interact with the Internet. From the earliest Usenet forums, which allowed anonymous posting via invalid emails [1], to 4chan and other anonymous message boards [2], and modern anonymous social apps like YikYak [3], users continue to seek out spaces where they can interact and discuss controversial topics without fear of censure or retribution. Anonymous browsing tools like TOR even allow users to interact anonymously with websites not designed to allow this [4].

Despite the importance of anonymous spaces on the internet, anonymity online is increasingly under threat [1, 5–7]. This is in large part because of the difficulty of *moderating* anonymous spaces: the very freedom that anonymity provides can enable—or even magnify [8]—harassment, threats, and cyberbullying. This leads to external pressure on providers to shut down anonymous spaces: for example, the widely-used YikYak app was banned on many college campuses and ultimately shut down in part because of the prevalence of cyberbullying; it was recently revived but is again facing bans [9, 10]. It also leads to providers choosing to add friction or ban anonymous interactions with services: for example, it is difficult to edit Wikipedia over TOR—most TOR exit nodes are IP-blocked from making edits to Wikipedia. This leads to exclusion of users from vulnerable populations who actually need the ability to interact anonymously [11] and deprives the wider community of their contributions: e.g., anonymous editors on Wikipedia actually make high-quality contributions [12]. The risks of *non*-anonymous interactions to vulnerable populations are all too real: the Syrian Wikipedia developer and dissident Bassel Khartabil was arrested and ultimately executed by the Syrian government in 2015, allegedly in part because of his online activism [13].

Cryptography can help resolve the seeming tension between anonymity and moderation. A key technical tool for this is *anonymous blocklisting* (AB). In AB, users can post messages on a public bulletin board anonymously; however, the provider can add messages that violate policies to a "blocklist". To post, users must prove that they are not the author of any of the messages on the blocklist. A long line of work, starting with the seminal BLAC paper [14], has built AB schemes based on zero-knowledge proofs with various different performance profiles; the more recent SNARKBlock work [15] was the first to apply modern zkSNARKs to anonymous blocklisting. However, previous works suffer from notable drawbacks: first, their proof sizes and verifying time are not scalable to large blocklists; even the state-of-the-art SNARKBlock has costs that are asymptotically superlinear in the total number of blocks. Second, they require even honest clients to re-do work for blocks already processed.

In this work, we present ALPACA, the first anonymous blocklisting scheme with asymptotically *constant* online cost

and proof size. ALPACA also does not re-do any work; once a client processes a block, they never need to process it again. Our main contributions are (1) the ALPACA scheme itself, built as a new and nontrivial application of incrementally verifiable computation (IVC) [16], (2) a detailed provable-security analysis, including new game-based security definitions that may be of independent interest, and (3) a new variant of incremental verifiable computation that provides an *iteration-hiding* guarantee not given by prior schemes, and an extension of the Nova IVC scheme [17] that satisfies iteration-hiding. We also perform a detailed evaluation of ALPACA, which gives evidence for its practicality: for example, updating the local non-membership proof of ALPACA is up to 35× faster than SNARKBlock, the previous state-of-the-art.

1.1. Overview of Contributions

ALPACA starts with representing the blocklist as an append-only list. As in prior work, each entry in the list in ALPACA contains a token for a posted message; to post, a user must prove none of the tokens are for their messages. The key conceptual idea of our construction is that lists are recursively-defined data structures. A list is either empty, or it comprises a head item and a tail, which is itself a list. Thus, ALPACA uses a naturally recursive proof primitive—incrementally verifiable computation (IVC)—to generate non-membership proofs in two steps: (1) the token in the head of the list is not the user's, and (2) the user knows a proof that none of the tokens in the tail of the list are theirs either.

To post messages, the user proves, via a zkSNARK tailored to the IVC scheme, that they know a valid IVC proof of non-membership on the blocklist, and uses an auxiliary zkSNARK to prove well-formedness of the new message's token. This is roughly how the steady-state functionality of ALPACA works for enrolled users that are never blocked—see §5 for more details—but does not give us two important pieces of functionality a real blocklisting system needs. First, it needs to be possible to *un-block* users. Temporary blocks are extremely common in applications like Wikipedia, where blocks are an attempt at corrective action rather than punishments [18]. The second important piece of functionality is that we need to allow users to "cold-start" in the system without needing to process every historical entry on the blocklist, even ones from before they registered.

ALPACA enables unblocking by having the provider publicly post a "ban-over" signature on an entry in the list. If a user was blocked from posting because of this entry, they can use this signature to skip the token check for that entry and continue to update their non-membership proof. We use a similar idea for cold-starting: when a user obtains their credential, the identity provider (blindly) signs it along with the current head of the list; this lets them skip the update step for everything before.

We carefully adapt the basic design to allow for these extensions with sacrificing security. In §6, we prove our scheme is anonymous (i.e., proofs hide identities), blocklistable (i.e., blocks cannot be bypassed), and non-frameable (i.e., users cannot be blocked for a message they did not post). To do this, we developed novel game-based definitions of security for these properties (§3.1). Our definitions are likely to be of independent interest for future research in anonymous blocklisting and related areas.

One real-world issue our anonymity definition captures is cold-starting: that is, the fact that users may enter the system at different times. In trying for the first time to prove our scheme with this definition we made the observation that with standard IVC the scheme is not necessarily secure. This is because existing formulations of IVC requires the *number of iterations* of the function F to be revealed publicly in the proof [16, 17, 19–21]. This means that two users who join at different times can immediately be distinguished based on the number of blocks they have processed since starting.

Fixing this issue requires designing a new IVC variant with an *iteration-hiding* property. This is technically nontrivial because the iteration count is critically required by algorithms that show up in the knowledge soundness proofs for existing schemes. We resolve this by carefully embedding the iteration count inside the IVC proof in such a manner that it can be efficiently hidden before the proof is published but still accessible by procedures in the soundness proof. We believe this result is also of independent interest; it is plausible that other applications may require hiding the iteration count.

ALPACA instantiates this high-level design with our iteration-hiding extension (§4) of the Nova IVC scheme [17]. Users locally maintain their non-membership proofs as Nova IVC proofs of repeated application of a carefully-designed function F to the blocklist. To post, users "wrap" the IVC proof, which is not zero-knowledge, in an outer zkSNARK, and produce an auxiliary zkSNARK that their credential produced the new message's token.

Our local state size, online proving time, proof size, and verification time are all asymptotically constant—they do not depend at all on the size of the blocklist. However, getting practical performance required non-trivial design and engineering effort. We instantiate our function F with SNARK-friendly cryptographic primitives, such as the Poseidon hash function [22] and the Schnorr signature scheme over the Pasta curve cycle [23]. This minimizes the R1CS constraint count of F, which is critical because it minimizes the time needed to update the non-membership proof.

In section 7, we give our evaluation of ALPACA. Our function F for non-membership proof only requires 31842 R1CS constraints, and other statements proved during posting are only 16287 R1CS constraints. We also measured prover time, proof size, and verifier time taken as the blocklist grows. We empirically verified performance is independent of blocklist size. We also found the ALPACA prover time is better than SNARKBlock after 2^{20} blocks, verifier time is better after 2^{17} blocks, and proof size is always better. The time taken to update the local state in ALPACA (ie, running one more step of IVC) is only 630ms—the

equivalent operation in SNARKBlock takes over 22 seconds, giving us a roughly 35× performance improvement.

2. Overview

We now introduce a high-level overview of our anonymous blocklisting scheme with setting, threat model, security goals, and core techniques of our construction.

2.1. Setting

Our scheme considers a posting platform that allows users to post messages to a public forum (e.g. Wikipedia [24]), Reddit [25], and X [26]). There are three participants in our scheme. A *service provider* operates the service including viewing the posting requests and managing a blocklist, which we also assume is public. An *identity provider* performs registration, which in our system refers to issuing a credential for a new user. However, the identity provider does not learn the credential (§5). *Users*, after obtaining their credential, use the service by posting messages. In our system, a user's credential is not user's account information (e.g. a password). We also assume that the users access the service through an anonymous channel (e.g. Tor) or the service itself is anonymous.

We design an anonymous blocklist where a service provider can block users. Our blocklist provides two primary functionalities—blocking and unblocking. When the service provider determines a message violates its service policy, via a report, etc., it *blocks* the author of the message by adding a cryptographic token of the message to the blocklist. The service provider can also *unblock*, meaning lifting a ban of the user. We model the simplest blocklist policy where a user cannot use the service if they are blocked at least once. We do not assume any unblocking policy. A service provider can unblock based on the duration, the significance of the violation, as a batch, etc. We discuss the blocking and unblocking policies in more detail in §9.

2.2. Threat Model

We assume that Sybil resistance is ensured by the identity provider in a sense that a person cannot own more than one account, and an account cannot own more than one credential. Thus, we rule out the case where a blocked user can successfully be authorized with their other credentials. We also assume that an adversary cannot steal other credentials and impersonate other users.

We assume that the service provider gives all participants a consistent view of the blocklist and the posted messages, but may mount other attacks to either deanonymize users or frame them. By framing, we mean that the service provider prevents a specific user from posting, even though none of their previous posts violate the policy.

We assume that the identity provider cannot directly modify the blocklist. It may attempt to learn the credential of the users during the registration procedure. We also assume that the identity and the service providers don't collude (§3.1).

We also model users as malicious. Users can be blocked if they violate the service policy. Blocked users may try to maliciously post messages even though they are blocked. Users may try to deanonymize other users. Furthermore, users may also try to frame other users as blocked so they cannot access the service. We assume that users may collude with the service provider to deanonymize or frame other users (note that a user colluding with the service provider can post messages even if they are blocked).

2.3. Security Goals

As the name suggests, the highest security goal of anonymous blocklisting is *anonymity*. We also want *blocklisting* to function in a desired way—blocking the blocked users and not blocking the unblocked users. We informally introduce here the security goals of an anonymous blocklist using the terminology in [14, 27] and defer formalism to §3.1.

Blocklistability. A service provider can block users successfully. In other words, if a user is blocked, then they must not be able to access the service.

Anonymity. Anonymity encompasses two sub-properties. An adversary must not learn identity from the view of service provider (backward anonymity). An adversary also must not be able to link two posts to the same identity even though it does not know identity (unlinkability).

Non-frameability. If a user has not violated the service policy, the user must not be framed as a blocked user. In other words, if a user is not blocked by the service provider, they must not be considered as a blocked user. "Framing" differs from impersonation attacks where an adversary steals a credential and gets voluntarily blocked by posting messages it knows will be blocked.

Previous works [14, 27] also introduce misauthentication resistance: only the credentials and tokens generated by following the protocols should be verified. We do not discuss this definition as it is implied by blocklistability. [27] additionally suggests revocation auditability: a revoked user should be able to disconnect without revealing their sensitive identity information. Since revocation auditability is only applicable to pseudonym systems, we do not discuss it.

2.4. Technical Overview

We now switch our focus to performance and provide a brief overview of our core design techniques to achieve constant costs. We discuss the details of these techniques in §5.

In our scheme, each posted message comes attached with token that is associated with a hidden identity. To block the author of a particular message, the service provider adds the token to an append-only list which we refer to as the blocklist, which is logically an indexed linked list.

To post a message, the user must generate a new token associated with their hidden identity along with the proof that no token in the blocklist is associated with this identity. To compute this proof efficiently, the user leverages

incrementally verifiable computation (IVC) [16], which is a particular proof technique that supports efficiently updating a proof alongside as running computation is updated. In ALPACA, every user keeps a running proof that their hidden identity is not associated with any of the tokens on the blocklist thus far. Whenever a new block is added, they use IVC to efficiently update their proof with respect to the updated blocklist. A key property of IVC is that the running proof does not grow in size over time (i.e., is constant-sized) and the time required to update only scales with a single step of the computation (checking the latest block in our case). When a user posts a message, they use a zero-knowledge proof that they have a valid running proof of non-membership (which is not zero-knowledge by default) and that the corresponding token is constructed with their hidden identity. This posting proof is also constant-sized. This results in a significant asymptotic improvement over previous works [15], which require at least $O(\log n)$-sized proofs and proof time where n is the blocklist size.

However, in practice we would like to grant the service provider the power to remove a block. The provider cannot physically remove a block from the blocklist as this would invalidate the users' running proofs. Instead, we have the service provider generate a "ban-over" signature on a block to indicate that the block is deactivated. The corresponding user then uses this signature to bypass the block and successfully generate a non-membership proof.

Our blocklisting scheme further supports a "cold-start" model. Previous works [14, 15] require synchronization from the first block when a new user joins. A new user, however, is naturally not blocked by the past blocks before they join. From this observation, whenever a new user registers, the identity provider signs the last block of the blocklist as the new user's cold-start block. This allows the user to only generate a running non-membership proof starting from the cold-start block, saving unnecessary computation. To ensure anonymity in the cold-start setting, we construct a *iteration-hiding* variant of IVC (§4).

3. Anonymous Blocklisting

We provide the definition of an anonymous blocklisting scheme (AB), adopting the definition in [14, 15, 27]. We also formalize the security properties required for an AB.

Definition 1 (Anonymous Blocklisting System). An *anonymous blocklisting system*

$$\mathsf{AB} = \begin{pmatrix} \mathsf{Setup}, \mathsf{Reg.User}, \mathsf{Reg.Id}, \mathsf{Tok}, \\ \mathsf{Sync}, \mathsf{Auth}, \mathsf{Ver}, \mathsf{Aud}, \\ \mathsf{BInit}, \mathsf{BAdd}, \mathsf{BRem}, \mathsf{BDig} \end{pmatrix}$$

is a tuple of twelve algorithms. An AB scheme is characterized by spaces $\mathsf{PK}, \mathsf{SK}, \mathsf{VK}, \mathsf{C}, \mathsf{M}, \mathsf{T}, \mathsf{BL}, \mathsf{B}$, and two distinct proof spaces $\Pi_{\mathsf{sync}}, \Pi_{\mathsf{auth}}$. Spaces $\mathsf{PK}, \mathsf{SK}, \mathsf{VK}$ are public, secret and verifying key spaces; the space of credentials is C; the space of messages is M; the space of tokens is T; the space of a blocklist is BL; and the space of blocks is B. We informally describe the functionalities of each algorithm. We provide the formal syntax definition in Appendix A.

Setup generates the public parameters and key pairs of the participants including the identity and service provider. A new user obtains a credential $c \in \mathsf{C}$ through an interactive protocol $\langle \mathsf{Reg.User}, \mathsf{Reg.Id} \rangle$ with the identity provider.

When a user wants to post a message, they generate a token tok through Tok. The user also runs Sync to fetch the latest blocklist and generate a proof π_{sync} that none of their tokens are in the blocklist. With π_{sync} and tok, Auth produces a proof π_{auth} that the user is not blocked and tok is the token of the message that the user requests to post. Ver verifies π_{auth} and outputs 1 if π_{auth} is valid and 0 otherwise. Additionally, a user can check their blocked status via Aud.

BInit initializes the blocklist. BAdd adds a token to the blocklist whereas BRem unblocks a token in the blocklist. BDig creates a digest of the current blocklist.

3.1. Security Definitions

In this section, we formalize the security properties introduced in §2. Following high-level definitions from prior work [14, 15, 27], we formalize blocklistability, anonymity, and non-frameability with security games (Figure 1). In our games, we model identity and service provider non-collusion by omitting registration from the adversary $\mathcal{A}$'s view. We do this mostly for simplicity; we believe our schemes would remain secure even without this assumption but leave confirming this fact to future work.

We provide the definition of completeness in the full version.

Blocklistability. A blocked user must not be verified successfully without negligible probability. $\mathcal{C}$ is the set of credentials registered through $\mathcal{O}_{\mathsf{Reg}}$, to which an adversary can query its chosen key k. We only allow adversaries to request with fresh keys. $\mathcal{A}$ is also given access to $\mathcal{O}_{\mathsf{BAdd}}$ and $\mathcal{O}_{\mathsf{BRem}}$, through which it can block or unblock tokens from the blocklist. The flag $\mathsf{all_blocked}$ at the end of the game ensures that all registered credentials are blocked. This is to make sure that any credential that $\mathcal{A}$ chooses from $\mathcal{C}$ is blocked. In this setting, $\mathcal{A}$ tries to output a tuple of token, message, digest, and authentication proof that Ver accepts. For an adversary $\mathcal{A}$ and an anonymous blocklisting scheme AB, we define the BL advantage of $\mathcal{A}$ against AB as

$$\mathbf{Adv}^{\mathsf{BL}}_{\mathsf{AB}}(\mathcal{A}) := \Pr[\mathsf{BL}^{\mathcal{A}}(\lambda) = 1].$$

We say AB has blocklistability if $\mathbf{Adv}^{\mathsf{BL}}_{\mathsf{AB}}(\mathcal{A}) \leq \mathsf{negl}(\lambda)$.

Anonymity. Not only do we require that a credential must not be learned from its token or proof, but we also require that two tokens or proofs must not be linked. The game AN in Figure 1 captures both properties. The challenger registers two credentials c_0 and c_1, possibly at different times. The challenger chooses a bit $b \in \{0, 1\}$ and sets the challenge credential as c_b. $\mathcal{A}$ is given a pair of c_b's token and the corresponding authentication proof, $(\mathsf{tok}', \pi'_{\mathsf{auth}})$. $\mathcal{A}$ has access to $\mathcal{O}_{\mathsf{Tok}}$, to which it can query its chosen credential

Game $\mathsf{BL}^{\mathcal{A}}(\lambda)$:
$(\mathsf{pp}, \mathsf{pk}, \mathsf{sk}, \mathsf{vk}) \leftarrow \mathsf{Setup}(1^\lambda)$,
$(\mathsf{sk}_{\mathsf{id}}, \mathsf{sk}_{\mathsf{sp}}) \leftarrow \mathsf{sk}$
$B \leftarrow \mathsf{BInit}(), \mathcal{C} \leftarrow \emptyset, \mathcal{S} \leftarrow \emptyset$
$(\mathsf{tok}', m', d, \pi'_{\mathsf{auth}}) \leftarrow \mathcal{A}^{\mathcal{O}_{\mathsf{Reg}}, \mathcal{O}_{\mathsf{BAdd}}, \mathcal{O}_{\mathsf{BRem}}}(\mathsf{pk})$
$\mathsf{all_blocked} = \bigwedge_{c \in \mathcal{C}} \mathsf{blocked}(c)$
return $\mathsf{Ver}(\mathsf{pk}, \mathsf{vk}, \mathsf{tok}', m', d, \pi'_{\mathsf{auth}}) = 1 \wedge$
$\mathsf{all_blocked} = 1$

$\mathcal{O}_{\mathsf{Reg}}(k)$:
if $\nexists c \in \mathcal{C}$ s.t. $c.k = k$:
$c \leftarrow\$ \langle \mathsf{Reg.Id}, \mathsf{Reg.User} \rangle(\mathsf{sk}_{\mathsf{id}}, k)$
$\mathcal{C} \leftarrow \mathcal{C} \cup \{c\}$
return c

$\mathcal{O}_{\mathsf{BAdd}}(c, \mathsf{tok})$:
if tok references c:
return $B \leftarrow \mathsf{BAdd}(\mathsf{tok}, B)$

$\mathcal{O}_{\mathsf{BRem}}(c, b)$:
$(\mathsf{tok}, h_i, i) \leftarrow b$
if $b \in B \wedge$ tok references c :
$\sigma_{\mathsf{ban_over}} \leftarrow \mathsf{Sig.Sign}(\mathsf{sk}_{\mathsf{sp}}, b)$
$\mathcal{S} \leftarrow \mathcal{S} \cup \{(b, \sigma_{\mathsf{ban_over}})\}$
return $(B, \mathcal{S})$

$\mathsf{blocked}(c)$:
$\exists b \in B$ s.t. for $(\mathsf{tok}, h_i, i) \leftarrow b$,
tok references $c \wedge \exists (b', \sigma') \in \mathcal{S}$ s.t. $b' = b$

Game $\mathsf{AN}^{\mathcal{A}}(\lambda)$
$(\mathsf{pp}, \mathsf{pk}, \mathsf{sk}, \mathsf{vk}) \leftarrow \mathsf{Setup}(1^\lambda), B \leftarrow \mathsf{BInit}()$
$b \leftarrow\$ \{0, 1\}, \mathsf{tok}' \leftarrow \perp$
for $i \in \{0, 1\}$:
$b_{\mathsf{start}} \leftarrow \mathcal{A}^{\mathcal{O}_{\mathsf{BAdd}}, \mathcal{O}_{\mathsf{BRem}}}(B)$
$c_i \leftarrow\$ \langle \mathsf{Reg.Id}, \mathsf{Reg.User} \rangle(b_{\mathsf{start}})$
$b' \leftarrow \mathcal{A}^{\mathcal{O}_{\mathsf{Chal}}, \mathcal{O}_{\mathsf{Tok}}, \mathcal{O}_{\mathsf{Auth}}, \mathcal{O}_{\mathsf{BAdd}}, \mathcal{O}_{\mathsf{BRem}}}(\mathsf{pk})$
return $b = b' \wedge \mathsf{tok}' \neq \perp$

$\mathcal{O}_{\mathsf{Chal}}(m)$:
if $\mathsf{tok}' = \perp$:
$\mathsf{tok}' \leftarrow\$ \mathsf{Tok}(c_b, m)$
$\pi_{\mathsf{sync}} \leftarrow\$ \mathsf{Sync}(\mathsf{pk}, B, c_b)$
$\pi'_{\mathsf{auth}} \leftarrow\$ \mathsf{Auth}(\mathsf{tok}', B, \mathsf{pk}, c_b, \pi_{\mathsf{sync}})$
return $(\mathsf{tok}', \pi'_{\mathsf{auth}})$

$\mathcal{O}_{\mathsf{Tok}}(i, m)$:
$\mathsf{tok} \leftarrow\$ \mathsf{Tok}(c_i, m)$
return tok

$\mathcal{O}_{\mathsf{Auth}}(i, \mathsf{tok})$
if $\mathsf{tok} \neq \mathsf{tok}'$:
$\pi_{\mathsf{sync}} \leftarrow\$ \mathsf{Sync}(\mathsf{pk}, B, c_i)$
return $\pi_{\mathsf{auth}} \leftarrow\$ \mathsf{Auth}(\mathsf{tok}, B, \mathsf{pk}, c_i, \pi_{\mathsf{sync}})$

$\mathcal{O}_{\mathsf{BAdd}}(\mathsf{tok})$:
if $\mathsf{tok} \neq \mathsf{tok}'$:
return $B \leftarrow \mathsf{BAdd}(\mathsf{tok}, B)$

$\mathcal{O}_{\mathsf{BRem}}(b)$:
return $B \leftarrow \mathsf{BRem}(b, B)$

Game $\mathsf{NF}^{\mathcal{A}}(\lambda)$:
$(\mathsf{pp}, \mathsf{pk}, \mathsf{sk}, \mathsf{vk}) \leftarrow \mathsf{Setup}(1^\lambda)$
$B \leftarrow \mathsf{BInit}(), \mathcal{C} \leftarrow \emptyset, \mathcal{S} \leftarrow \emptyset$
$b_{\mathsf{start}} \leftarrow B$
$c' \leftarrow\$ \langle \mathsf{Reg.Id}, \mathsf{Reg.User} \rangle(b_{\mathsf{start}})$
$\mathcal{C} \leftarrow \mathcal{C} \cup \{c'\}$
$m' \leftarrow \mathcal{A}^{\mathcal{O}_{\mathsf{Reg}}, \mathcal{O}_{\mathsf{Tok}}, \mathcal{O}_{\mathsf{Auth}}, \mathcal{O}_{\mathsf{BAdd}}, \mathcal{O}_{\mathsf{BRem}}}(\mathsf{pk})$
$\mathsf{tok}' \leftarrow\$ \mathsf{Tok}(c', m')$
$\pi'_{\mathsf{sync}} \leftarrow\$ \mathsf{Sync}(c', B, \mathsf{pk})$
$\pi'_{\mathsf{auth}} \leftarrow\$ \mathsf{Auth}(\mathsf{tok}', c', B, \mathsf{pk}, \pi'_{\mathsf{sync}})$
$d' \leftarrow \mathsf{BDig}(B)$
return $\mathsf{Ver}(\mathsf{pk}, \mathsf{vk}, \mathsf{tok}', m', d', \pi'_{\mathsf{auth}}) = 0 \wedge$
$\mathsf{blocked}(c') = 0$

$\mathcal{O}_{\mathsf{Reg}}(k)$:
if $k \neq c'.k'$:
$c \leftarrow\$ \langle \mathsf{Reg.Id}, \mathsf{Reg.User} \rangle(\mathsf{sk}_{\mathsf{id}}, k)$
$\mathcal{C} \leftarrow \mathcal{C} \cup \{c\}$
return c

$\mathcal{O}_{\mathsf{Tok}}(m)$:
$\mathsf{tok} \leftarrow\$ \mathsf{Tok}(c', m)$
return tok

$\mathcal{O}_{\mathsf{Auth}}(\mathsf{tok})$
$\pi_{\mathsf{sync}} \leftarrow\$ \mathsf{Sync}(c', B, \mathsf{pk})$
$\pi_{\mathsf{auth}} \leftarrow\$ \mathsf{Auth}(\mathsf{pk}, \mathsf{tok}, B, c', \pi_{\mathsf{sync}})$
return π_{auth}

$\mathcal{O}_{\mathsf{BAdd}}(k, \mathsf{tok})$:
if $\exists c \in \mathcal{C}$ s.t. $c.k = k \wedge$ tok references c:
$B \leftarrow \mathsf{BAdd}(\mathsf{tok}, B)$
$b \leftarrow$ last block of B
$\mathcal{S} \leftarrow \{(k, b)\}$
return B

$\mathcal{O}_{\mathsf{BRem}}(b)$:
$\mathcal{S} \leftarrow \mathcal{S} \setminus \{(k, b)\}$
return $B \leftarrow \mathsf{BRem}(b, B)$

$\mathsf{blocked}(c)$:
$\exists b \in B$ s.t. $(c.k, b) \in \mathcal{S} \wedge \mathsf{BanOver}(b) = 0$

Figure 1: Blocklistability game BL, Anonymity game AN, and Non-frameability game NF. The games are specific to our construction but it is not hard to generalize for any anonymous blocklisting schemes. We also use $c.k$ to denote the key k in a credential c.

in $\{c_0, c_1\}$ and message pairs. $\mathcal{A}$ can also query $\mathcal{O}_{\mathsf{Auth}}$ to obtain an authentication proof of chosen tokens. $\mathcal{A}$ further has access to $\mathcal{O}_{\mathsf{BAdd}}$ and $\mathcal{O}_{\mathsf{BRem}}$. Based on the obtained tokens, authentication proofs, and blocklist status, $\mathcal{A}$ tries to guess the challenge bit b. For an adversary $\mathcal{A}$ and scheme AB, we define the AN advantage of $\mathcal{A}$ against AB as

$$\mathbf{Adv}^{\mathsf{AN}}_{\mathsf{AB}}(\mathcal{A}) := \Pr[\mathsf{AN}^{\mathcal{A}}(\lambda) = 1]$$

and say AB is anonymous if $\mathbf{Adv}^{\mathsf{AN}}_{\mathsf{AB}}(\mathcal{A}) \leq 1/2 \pm \mathsf{negl}(\lambda)$.

We prevent a trivial win of $\mathcal{A}$ by the $\mathsf{tok} \neq \mathsf{tok}'$ constraint in $\mathcal{O}_{\mathsf{BAdd}}$. Without this constraint, $\mathcal{A}$ can easily determine c_b. The trick is to (a) first make both credentials not blocked, (b) block c_b by querying $\mathcal{O}_{\mathsf{BAdd}}$ with tok', and (c) query $\mathcal{O}_{\mathsf{Auth}}$ with $(0, \mathsf{tok}_0)$ and $(1, \mathsf{tok}_1)$ where tok_0 is c_0's token and tok_1 is c_1's. Since c_b is blocked, one of two queries will fail, instantly revealing c_b.

Similarly, we prevent another trivial win of $\mathcal{A}$ through the $\mathsf{tok} \neq \mathsf{tok}'$ constraint in $\mathcal{O}_{\mathsf{Auth}}$. Without this constraint, $\mathcal{A}$ can easily guess b by first making both credentials not blocked and querying both $(0, \mathsf{tok}')$ and $(1, \mathsf{tok}')$ to $\mathcal{O}_{\mathsf{Auth}}$. Since both c_0 and c_1 are not blocked, Sync will succeed for both queries. However, Auth checks if tok' references the queried credential. Thus, only one of two queries will succeed, which is the query with c_b. As the adversary cannot obtain the authentication proof for tok' by $\mathcal{O}_{\mathsf{Auth}}$, $\mathcal{O}_{\mathsf{Chal}}$ additionally returns π_{auth} for tok'.

Non-Frameability. Non-frameability requires that a non-blocked user not be framed as a blocked user, which we formalize in the game NF in Figure 1. The challenger creates a challenge credential c'. $\mathcal{A}$ is given access to $\mathcal{O}_{\mathsf{Reg}}$, to which it can query a key k chosen adversarially. $\mathcal{A}$ can also access $\mathcal{O}_{\mathsf{Tok}}$ and $\mathcal{O}_{\mathsf{Auth}}$ to obtain tokens and authentication proofs of c' on chosen messages and tokens. $\mathcal{A}$ is also given access to $\mathcal{O}_{\mathsf{BAdd}}$ and $\mathcal{O}_{\mathsf{BRem}}$, through which $\mathcal{A}$ can block or unblock tokens from the blocklist. The condition $\mathsf{blocked}(c') = 0$ at the end of the game ensures that c' is not blocked. In this setting, $\mathcal{A}$ tries to make Ver fail for the c'. For an adversary $\mathcal{A}$ and an anonymous blocklisting scheme AB, we define the NF advantage of $\mathcal{A}$ against NF as

$$\mathbf{Adv}^{\mathsf{NF}}_{\mathsf{AB}}(\mathcal{A}) := \Pr[\mathsf{NF}^{\mathcal{A}}(\lambda) = 1].$$

AB satisfies non-frameability if $\mathbf{Adv}^{\mathsf{NF}}_{\mathsf{AB}}(\mathcal{A}) \leq \mathsf{negl}(\lambda)$.

We remark that $\mathcal{A}$ in NF seems to have less power than the adversaries in the other games at first, as it does not generate inputs to Ver other than m'. However, if $\mathcal{A}$ has ability to generate Ver inputs, $\mathcal{A}$ can trivially win NF by dishonestly producing values that will always fail, such as bot values. Thus, we let the challenger compute inputs to

Ver on $\mathcal{A}$'s choice of message m'.

We clarify the purpose of a set $\mathcal{S}$ of credential key and block pairs in the game. Imagine how an adversary can try to frame c' without impersonating c'. The adversary will construct a token adversarially with another credential c. Say that this adversarial token can somehow be treated as c''s token as well. The adversary queries $\mathcal{O}_{\mathsf{BAdd}}$ with c's key k and the token. The system now blocks c, and not c' directly, but c' can be considered as a blocked credential. To capture this, we use $\mathcal{S}$ for bookkeeping which credential $\mathcal{O}_{\mathsf{BAdd}}$ *intended* to block. blocked also refers to $\mathcal{S}$ to check if c' is blocked.

Comparing our definitions to BLAC. The knowledgeable reader may wonder why we chose to build our own new definitions, instead of using the prior definitions given in the journal version of the BLAC paper [14]. Though these definitions have some interesting advantages, ultimately they are unsuitable for our purposes. BLAC's definitions are extremely general, and handle settings (such as multiple service providers with different blocklists) that we do not. They are also described with respect to a *common* set of oracles that all their definitions share, rather than having oracles for different security definitions that encode state-keeping and trivial win checking more explicitly, as our definitions do. We feel our definitional approach is simpler and better-suited to answering core conceptual questions about the security of anonymous blocklisting protocols. A more thorough provable-security analysis of anonymous blocklisting is an interesting question for future work.

4. Iteration-Hiding IVC

Recall that in the non-membership proof synchronization procedure, ALPACA uses incrementally verifiable computation (IVC), which supports efficiently updating a proof of correct execution alongside a running computation. In particular, IVC enables a user to compute a proof π_n attesting that a prescribed function F applied n times to some initial public input z_0 results in some public output z_n. This is done incrementally: Given a running proof π_i of i steps, a user can compute an updated running proof π_{i+1} of $i+1$ steps (which does not grow in size) in $O(|F|)$ time. In each step i the function F may additionally take a secret input ω_i. In our particular setting, F publicly passes forward a running digest of the blocklist so far and privately checks the users credential against the block at that particular step.

To support *iteration-hiding* we would like to modify IVC to prove a statement of the form "F applied *some* number of times to an initial input z_0 results in the output z_i". However, we must still ensure that the malicious prover cannot feasibly construct a satisfying IVC proof with an embedded iteration count i that is overwhelmingly large to the point where the soundness proof fails. To handle this, the IVC statement is modified to contain an upper-bound for the total number of iterations (i.e., "F applied less than N times to an initial input z_0 results in the output z_i"), which in practice can be set to a comfortably large number.

We begin by defining the syntax for iteration-hiding IVC, which enables a prover to incrementally prove that a function F applied less than N times to an initial input z_0 results in the output z_n. We define the corresponding security properties in Appendix B.

Definition 2 (Iteration-Hiding IVC). An incrementally verifiable computation (IVC) scheme is defined by a tuple of PPT algorithms $(\mathcal{G}, \mathcal{P}, \mathcal{V})$, denoted the generator, prover and verifier respectively, and a deterministic algorithm $\mathcal{K}$ denoting the encoder with the following syntax.

- $\mathcal{G} : \{0,1\}^\lambda \times \mathbb{N} \to \mathsf{PP}$. Given a security parameter λ, and a size parameter ℓ outputs public parameters pp.
- $\mathcal{K} : \mathsf{PP} \times \mathsf{F} \to \mathsf{PK} \times \mathsf{VK}$. Given public parameters pp and a polynomial-time computable function $F : \mathsf{Z} \times \mathsf{W} \to \mathsf{Z}$, outputs a prover key pk and verifier key vk.
- $\mathcal{P} : \mathsf{PK} \times (\mathsf{Z} \times \mathsf{Z} \times \mathbb{N}) \times (\Pi_i \times \mathsf{W}) \to \Pi_{i+1}$: Given a prover key pk an initial input z_0, claimed output z_i, an iteration upper bound N, an IVC proof of π_i of i iterations of F, and non-deterministic input ω_i, outputs a new IVC proof π_{i+1} of $i+1$ iterations of F.
- $\mathcal{V} : \mathsf{VK} \times (\mathsf{Z}, \mathsf{Z}, \mathbb{N}) \times \Pi_i \to \{0,1\}$: Given a verifier's key vk for a specific function F, an initial input z_0, claimed output z_i, an IVC proof π_i, outputs 1(accept) or 0 (reject).

Our starting point for iteration-hiding IVC is the Nova IVC scheme [17], which works by having the prover prove an augmented function F' in each step, which in addition to running the original step function F also runs additional "glue code" that in step $i+1$ compresses a claim u_i of step i into a running claim U_i of all $i-1$ prior steps using an IVC-compatible *folding scheme* (§B.1). Then, checking the resulting claim U_{i+1} implies checking i steps of execution. Alongside, the prover folds the corresponding witnesses w_i and W_i into an updated witness W_{i+1} for U_{i+1}. To complete the recursion cycle, the prover produces a new claim u_{i+1} and a corresponding witness w_{i+1} for the latest step of F'. In the final step n Nova hides the witnesses w_n and W_n using a zkSNARK which is sufficient to hide all the intermediate private inputs.

In the Nova IVC scheme the function F' additionally increments and forwards the iteration count in each step as part of its public output to be checked by the final verifier. In our case, we hide this iteration count by relegating it to the corresponding witness w_i. To do so, we have F' instead take as input a *hiding commitment* of the iteration count, and check it against an iteration count passed alongside as a private input. If this check passes, F' then outputs a new hiding commitment of the incremented counter which enforces that the correct iteration count is provided in the next step. In this way, the iteration count is hidden by the final zkSNARK. Moreover, to ensure that the iteration counter remains bounded, we have F' pass the bound N through each step of execution and check that $i \leq N$.

Construction 1 (Iteration-Hiding IVC). Let NIFS be an IVC-compatible non-interactive folding scheme for relation pair $(\mathcal{R}_1, \mathcal{R}_2)$. Let Com be a hiding commitment scheme. As guaranteed by the IVC-compatibility property, let $\mathsf{enc}_{\mathsf{inst}}$

be the invertible function which maps the inputs and outputs of F to an instance that can be checked in $\mathcal{R}_2$. Consider a polynomial-time function F. We construct an IVC scheme by first defining an augmented function F' where all input arguments are taken as secret inputs in Figure 2.

$F'(\mathsf{vk_{fs}}, \mathsf{U}_i, \mathsf{u}_i, (z_0, z_i, N), (\omega_i, i, r_i, r_{i+1}), \pi)$:
1: Compute the next output $z_{i+1} \leftarrow F(z_i, \omega_i)$.
2: **if** $i = 0$:
3: Check that $z_0 = z_i$ to ensure that the statement holds in the base case.
4: **else**:
5: Check that $i \leq N$.
6: Parse u_i as (C, u'_i), a commitment to the witness and the remainder.
7: Check that u'_i references U_i in the output of the prior iteration:

$$\mathsf{u}'_i \stackrel{?}{=} \mathsf{enc}_{\mathsf{inst}}(\mathsf{hash}(\mathsf{vk_{fs}}, z_0, z_i, \mathsf{U}_i, N), \mathsf{Com}(i, r_i)).$$

8: Let $\mathsf{U}_{i+1} \leftarrow \mathsf{NIFS}.\mathcal{V}(\mathsf{vk_{fs}}, \mathsf{U}_i, \mathsf{u}_i, \pi)$.
9: Output $\mathsf{x} \leftarrow (\mathsf{hash}(\mathsf{vk_{fs}}, z_0, z_{i+1}, \mathsf{U}_{i+1}, N), \mathsf{Com}(i+1, r_{i+1}))$.

Figure 2: Augmented IVC step function F'.

Theorem 1. *Construction 1 is iteration-hiding IVC.*

Proof sketch. Our proof closely follows that of Kothapalli et al. [17]. To prove knowledge-soundness we must show that for every malicious prover $\mathcal{P}^*$ that successfully outputs a valid IVC proof Π_n with probability ϵ there exists an extractor $\mathcal{E}$ that can output the intermediate witnesses $(\omega_0, \ldots, \omega_{n-1})$ with the same probability. We show inductively that $\mathcal{E}$ can construct an expected polynomial-time extractor $\mathcal{E}_i$ that outputs that outputs witnesses $(\omega_i, \ldots, \omega_{n-1})$ for all $i \in \{0, \ldots, n-1\}$. At a high level, to construct an extractor $\mathcal{E}_{i-1}$, we first assume the existence of $\mathcal{E}_i$ that satisfies the inductive hypothesis. We then use $\mathcal{E}_i(\mathsf{pp})$ to construct an adversary for the underlying folding scheme (which we denote as $\widetilde{\mathcal{P}}_{i-1}$). This in turn guarantees an extractor for the non-interactive folding scheme, which we denote as $\widetilde{\mathcal{E}}_{i-1}$. We then use $\widetilde{\mathcal{E}}_{i-1}$ to construct $\mathcal{E}_{i-1}$ that satisfies the inductive hypothesis. By the construction of F' we have that n is referenced in the witness portion of Π_n. Then by the bound check in F' we must have that $n \leq N$ ensuring that the extractor runtime is bounded by the predetermined upper-bound. □

5. Anonymous Blocklisting with Constant-sized Updatable Proofs

In this section, we present a new anonymous blocklisting scheme that gives constant (online) prover time, proof size and verifier time. We first introduce *updatable synchronization* in §5.1. Then, we provide a full construction with pseudocode in §5.2 that utilizes the updatable synchronization.

5.1. Updatable Synchronization

Recall that the synchronization protocol takes the blocklist as input and scans all entries of the blocklist to output a non-membership proof. On the other hand, we define an *updatable synchronization protocol*, which scans only the updated entries and updates the previous proof.

Definition 3 (Updatable Synchronization). *Updatable synchronization protocol* is a polynomial time algorithm

$$\mathsf{USync} : \mathsf{PK}^3 \times \mathsf{B} \times \mathsf{C} \times \Pi_{\mathsf{usync}} \to \Pi_{\mathsf{usync}},$$

such that for $B \in \mathsf{BL}$, given public keys $\mathsf{pk} \in \mathsf{PK}^3$, $b \in \mathsf{B}$, $c \in \mathsf{C}$, and a proof $\pi_{\mathsf{usync}} \in \Pi_{\mathsf{usync}}$ that proves no token in $B \subseteq B'$ references c, USync outputs a proof $\pi'_{\mathsf{usync}} \in \Pi_{\mathsf{usync}}$ that no token in $B' = \{B, b\}$ references c. USync is required to satisfy the following in relation to Sync:

- For all credentials $c \in \mathsf{C}$ and a blocklist $B' = \{B, b\}$, where $B, B' \in \mathsf{BL}$, $b \in \mathsf{B}$, and $\mathsf{pk} \in \mathsf{PK}^3$:

$$\mathsf{USync}(\mathsf{pk}, b, c, \mathsf{Sync}(\mathsf{pk}, B, c)) = \mathsf{Sync}(\mathsf{pk}, B', c)$$

- For all credentials $c \in \mathsf{C}$, a single-entry blocklist $B = \{b\} \in \mathsf{BL}$ and $\mathsf{pk} \in \mathsf{PK}^3$, there exists a proof $\pi_\perp$ such that:

$$\mathsf{USync}(\mathsf{pk}, b, c, \pi_\perp) = \mathsf{Sync}(\mathsf{pk}, B, c)$$

5.2. Full Construction

Construction 2 (ALPACA). An anonymous blocklisting scheme with constant-sized updatable proofs is a tuple of twelve algorithms

$$\left(\begin{array}{c}\mathsf{Setup}, \mathsf{Reg.User}, \mathsf{Reg.Id}, \mathsf{Tok}, \mathsf{USync}, \mathsf{Auth} \\ \mathsf{Ver}, \mathsf{Aud}, \mathsf{BInit}, \mathsf{BAdd}, \mathsf{BRem}, \mathsf{BDig}\end{array}\right)$$

with function F and relations $\mathcal{R}_{\mathsf{IVC}}$ and $\mathcal{R}_{\mathsf{post}}$ (Figure 3).

- $\mathcal{R}_{\mathsf{IVC}} := \Big\{(\mathsf{pp}, F, (N, z_0, z_i), \pi_{\mathsf{ivc}\,i}) : \mathsf{vk} \leftarrow \mathcal{K}(\mathsf{pp}, F) \wedge \mathcal{V}(\mathsf{vk}, (N, z_0, z_i), \pi_{\mathsf{ivc}\,i}) = 1\Big\}$
  where N is the upper bound of i, $z_0 = (\mathsf{com}_\perp, h_\perp, \mathsf{pk_{id}}, \mathsf{pk_{sp}})$, $z_i = (\mathsf{com}^i_k, h_j, \mathsf{pk_{id}}, \mathsf{pk_{sp}})$.
- $\mathcal{R}_{\mathsf{post}} := \Big\{(\mathsf{pk_{id}}, \mathsf{tok}, \mathsf{com}^{\ell+1}_k; c, r_{\ell+1}) : \mathsf{tag} = \mathsf{PRF}_k(r_{\mathsf{nonce}}, \mathsf{mh}) \wedge \mathsf{Sig}.\mathcal{V}(\mathsf{pk_{id}}, \sigma_{\mathsf{start}}, (\mathsf{com}_k, b_{\mathsf{start}})) \wedge \mathsf{open}(\mathsf{com}_k, r) = k \wedge \mathsf{open}(\mathsf{com}^{\ell+1}_k, r_{\ell+1}) = k\Big\}$
  where $(\mathsf{tag}, r_{\mathsf{nonce}}, \mathsf{mh}) \leftarrow \mathsf{tok}$, $(k, r, \mathsf{com}_k, b_{\mathsf{start}}, \sigma_{\mathsf{start}}) \leftarrow c$.

Figure 3: Relations $\mathcal{R}_{\mathsf{IVC}}$ and $\mathcal{R}_{\mathsf{post}}$. $\mathcal{R}_{\mathsf{IVC}}$ describes that a user's previous posts are not blocked. $\mathcal{R}_{\mathsf{post}}$ describes that a new post request is well-formed. In $\mathcal{R}_{\mathsf{IVC}}$, h_j in z_i is $h_{\mathsf{start}+i}$ when attested in USync and $h_{\mathsf{start}+i-1}$ when attested in Auth.

$\mathsf{Setup}(1^\lambda)$:
1: $\mathsf{sk_{sp}}, \mathsf{pk_{sp}} \leftarrow\!\$\ \mathsf{Sig}.\mathcal{G}(1^\lambda)$
2: $\mathsf{sk_{id}}, \mathsf{pk_{id}} \leftarrow\!\$\ \mathsf{Sig}.\mathcal{G}(1^\lambda)$
3: $\mathsf{pp_{nova}} \leftarrow\!\$\ \mathsf{zkSNARK_{IVC}}.\mathcal{G}(1^\lambda)$
4: $\mathsf{pk_{nova}}, \mathsf{vk_{nova}} \leftarrow\!\$\ \mathsf{zkSNARK_{IVC}}.\mathcal{K}(\mathsf{pp_{nova}}, F)$
5: $\mathsf{pp_{post}} \leftarrow\!\$\ \mathsf{zkSNARK}.\mathcal{G}(1^\lambda)$
6: $\mathsf{pk_{post}}, \mathsf{vk_{post}} \leftarrow\!\$\ \mathsf{zkSNARK}.\mathcal{K}(\mathsf{pp_{post}}, \mathcal{R}_{\mathsf{post}})$
7: $\mathsf{pp} \leftarrow (\mathsf{pp_{post}}, \mathsf{pp_{nova}})$, $\mathsf{pk} \leftarrow (\mathsf{pk_{id}}, \mathsf{pk_{sp}}, (\mathsf{pk_{post}}, \mathsf{pk_{nova}}))$, $\mathsf{sk} \leftarrow (\mathsf{sk_{id}}, \mathsf{sk_{sp}})$, $\mathsf{vk} \leftarrow (\mathsf{vk_{post}}, \mathsf{vk_{nova}})$
8: **return** $(\mathsf{pp}, \mathsf{pk}, \mathsf{sk}, \mathsf{vk})$

Figure 4: Setup Protocol.

Setup (Figure 4). Setup generates public parameters and key pairs for four entities. It generates public key and

secret key pairs for the service provider and the identity provider. Construction 2 instantiates two zkSNARK systems, $\mathsf{zkSNARK}_{\mathsf{IVC}}$ for $\mathcal{R}_{\mathsf{IVC}}$ and zkSNARK for $\mathcal{R}_{\mathsf{post}}$. Thus Setup additionally generates proving key and verifying key pairs used in $\mathsf{zkSNARK}_{\mathsf{IVC}}$ and zkSNARK.

$\underline{\mathsf{Reg.User1}(1^\lambda)}$:
1: $k \leftarrow\$ \mathcal{K}$, $r \leftarrow\$ \{0,1\}^\lambda$
2: $\mathsf{com}_k := \mathsf{Com}(k, r)$
3: **return** (k, r, com_k)

$\underline{\mathsf{Reg.User2}(k, r, \mathsf{com}_k, b_{\mathsf{start}}, \sigma_{\mathsf{start}})}$:
1: $c \leftarrow (k, r, \mathsf{com}_k, b_{\mathsf{start}}, \sigma_{\mathsf{start}})$
2: **return** c

$\underline{\mathsf{Reg.Id}(\mathsf{sk}_{\mathsf{id}}, \mathsf{com}_k, B)}$:
1: $b_{\mathsf{start}} \leftarrow$ the last block of B
2: $\sigma_{\mathsf{start}} \leftarrow \mathsf{Sig.Sign}_{\mathsf{sk}_{\mathsf{id}}}(\mathsf{com}_k, b_{\mathsf{start}})$
3: **return** $(b_{\mathsf{start}}, \sigma_{\mathsf{start}})$

Figure 5: Interactive Registration Protocol.

Registration (Figure 5). The registration protocol is an interactive protocol between a new user and the identity provider. In Reg.User1, the user first samples its credential key k and commits to the key with randomness r, i.e. $\mathsf{com}_k \leftarrow \mathsf{Com}(k, r)$. Given com_k, the identity provider sets the last block of the blocklist as the user's cold-start block b_{start}. It approves the user's registration through its signature σ_{start} on com_k and b_{start}. As an output, the user obtains their credential $c = (k, r, \mathsf{com}_k, b_{\mathsf{start}}, \sigma_{\mathsf{start}})$.

$\underline{\mathsf{USync}(\mathsf{pk}, b_j, b_{j-1}, \sigma^j_{\mathsf{ban_over}}; c, (\pi_{\mathsf{usync}_i}, \mathsf{st}_i))}$:
1: $(\mathsf{pk}_{\mathsf{id}}, \mathsf{pk}_{\mathsf{sp}}, (\mathsf{pk}_{\mathsf{post}}, \mathsf{pk}_{\mathsf{nova}})) \leftarrow \mathsf{pk}$, $(\mathsf{pk}_{\mathsf{IVC}}, \mathsf{pk}_{\mathsf{zkSNARK}_{\mathsf{IVC}}}) \leftarrow \mathsf{pk}_{\mathsf{nova}}$
2: $(k, r, \mathsf{com}_k, b_{\mathsf{start}}, \sigma_{\mathsf{start}}) \leftarrow c$
3: $h_0 \leftarrow h_\perp, \mathsf{com}^0_k \leftarrow \mathsf{com}_\perp$
4: **if** $b_j = b_{\mathsf{start}}$:
5: $\quad \pi_{\mathsf{ivc}\,i} \leftarrow \pi_\perp$, $h_j \leftarrow h_0$, $b_{j-1} \leftarrow b_\perp, r_i \leftarrow r_\perp, i \leftarrow 0$
6: **else**:
7: $\quad (\pi_{\mathsf{ivc}\,i}, \mathsf{com}^i_k, h_j, \mathsf{pk}_{\mathsf{id}}, \mathsf{pk}_{\mathsf{sp}}) \leftarrow \pi_{\mathsf{usync}_i}, (i, r_i) \leftarrow \mathsf{st}_i$
8: $r_{i+1} \leftarrow\$ \{0,1\}^\lambda$
9: $(\mathsf{com}^{i+1}_k, h_{j+1}, \mathsf{pk}_{\mathsf{id}}, \mathsf{pk}_{\mathsf{sp}}) \leftarrow F(\mathsf{com}^i_k, h_j, \mathsf{pk}_{\mathsf{id}}, \mathsf{pk}_{\mathsf{sp}};$
$\quad c, \sigma^j_{\mathsf{ban_over}}, b_j, b_{j-1}, r_i, r_{i+1})$
10: $z_0 \leftarrow (\mathsf{com}^0_k, h_0, \mathsf{pk}_{\mathsf{id}}, \mathsf{pk}_{\mathsf{sp}})$, $z_{i+1} \leftarrow (\mathsf{com}^{i+1}_k, h_{j+1}, \mathsf{pk}_{\mathsf{id}}, \mathsf{pk}_{\mathsf{sp}})$
11: $\pi_{\mathsf{ivc}\,i+1} \leftarrow \mathsf{IVC}.\mathcal{P}(\mathsf{pk}_{\mathsf{IVC}}, (N, z_0, z_{i+1}), (c, \sigma^j_{\mathsf{ban_over}}, b_j, b_{j-1}, r_i, r_{i+1}),$
$\quad \pi_{\mathsf{ivc}\,i})$
12: $\pi_{\mathsf{usync}_{i+1}} \leftarrow (\pi_{\mathsf{ivc}\,i+1}, \mathsf{com}^{i+1}_k, h_{j+1}, \mathsf{pk}_{\mathsf{id}}, \mathsf{pk}_{\mathsf{sp}}), \mathsf{st}_{i+1} \leftarrow (i+1, r_{i+1})$
13: **return** $(\pi_{\mathsf{usync}_{i+1}}, \mathsf{st}_{i+1})$

Figure 6: Updatable Synchronization Protocol. We use j to denote $\mathsf{start} + i$. The pseudocode of function F is in Figure 12, which we discuss in detail later in this section.

Updatable Synchronization (Figure 6). The goal of USync is to generate a proof that no token of a user's credential c is on the blocklist. USync first takes advantage of the cold-start model. Since the user is clearly not blocked before the cold-start block b_{start}, USync checks blocks after b_{start}.

We also take advantage of the append-only structure of our blocklist. Since only update is addition of blocks, USync runs offline whenever a new block is added. And for each USync operation, instead of scanning all blocks in the blocklist, it only sees the recently added block and updates the given proof. When the proof is produced by USync instead of Sync, we denote it as π_{usync}. For instance, given π_{usync_i} that c is not blocked by $[b_{\mathsf{start}}, b_{\mathsf{start}+i-1}]$, USync outputs $\pi_{\mathsf{usync}_{i+1}}$ that c is not blocked by $[b_{\mathsf{start}}, b_{\mathsf{start}+i}]$.

USync uses IVC under the hood to update the proof. USync first runs the IVC function F that checks if the new block $b_j = b_{\mathsf{start}+i}$ references c. If not, F successfully outputs an incremental step result $z_{i+1} = (\mathsf{com}^{i+1}_k, h_{j+1}, \mathsf{pk}_{\mathsf{id}}, \mathsf{pk}_{\mathsf{sp}})$. USync then runs the IVC prover to obtain $\pi_{\mathsf{ivc}\,i+1}$ that proves the correctness of current iteration of F. In other words, if $\pi_{\mathsf{ivc}\,i+1}$ is valid, it means that the user is not blocked from b_{start} to $b_{\mathsf{start}+i}$. Finally, USync constructs

$$\begin{aligned}\pi_{\mathsf{usync}_{i+1}} &= (\pi_{\mathsf{ivc}\,i+1}, z_{i+1}) \\ &= (\pi_{\mathsf{ivc}\,i+1}, \mathsf{com}^{i+1}_k, h_{j+1}, \mathsf{pk}_{\mathsf{id}}, \mathsf{pk}_{\mathsf{sp}})\end{aligned}$$

and outputs $\pi_{\mathsf{usync}_{i+1}}$ with the status information st_{i+1}.

$\underline{\mathsf{Tok}(m, c)}$:
1: $(k, r, \mathsf{com}_k, b_{\mathsf{start}}, \sigma_{\mathsf{start}}) \leftarrow c$
2: $r_{\mathsf{nonce}} \leftarrow\$ \{0,1\}^\lambda$
3: $\mathsf{mh} = \mathsf{Hash}(m)$
4: $\mathsf{tag} = \mathsf{PRF}_k(r_{\mathsf{nonce}}, \mathsf{mh})$
5: **return** $\mathsf{tok} \leftarrow (\mathsf{tag}, r_{\mathsf{nonce}}, \mathsf{mh})$

Figure 7: Token Extraction Protocol

Token Extraction (Figure 7). When a user wants to post a new message, they first generate a token through Tok. Tok uses a pseudorandom function PRF with the user's key k from their credential. A token $\mathsf{tok} = (\mathsf{tag}, r_{\mathsf{nonce}}, \mathsf{mh})$ consists of the hash of the message mh, a randomness r_{nonce}, and $\mathsf{tag} = \mathsf{PRF}_k(r_{\mathsf{nonce}}, \mathsf{mh})$.

$\underline{\mathsf{Auth}(\mathsf{pk}, \mathsf{tok}; c, \pi_{\mathsf{usync}_\ell}, \mathsf{st}_\ell)}$:
1: $(\pi_{\mathsf{ivc}\,\ell}, \mathsf{com}^\ell_k, h_{n+1}, \mathsf{pk}_{\mathsf{id}}, \mathsf{pk}_{\mathsf{sp}}), \leftarrow \pi_{\mathsf{usync}_\ell}, (\ell, r_\ell) \leftarrow \mathsf{st}_\ell$
2: $(k, r, \mathsf{com}_k, b_{\mathsf{start}}, \sigma_{\mathsf{start}}) \leftarrow c$
3: $h_0 \leftarrow h_\perp, \mathsf{com}^0_k \leftarrow \mathsf{com}_\perp$
4: $(\mathsf{pk}_{\mathsf{id}}, \mathsf{pk}_{\mathsf{sp}}, (\mathsf{pk}_{\mathsf{post}}, \mathsf{pk}_{\mathsf{nova}})) \leftarrow \mathsf{pk}$
5: $(\pi_{\mathsf{usync}_{\ell+1}}, \mathsf{st}_{\ell+1}) \leftarrow\$ \mathsf{ReRand}(\mathsf{pk}, c, \pi_{\mathsf{usync}_\ell}, \mathsf{st}_\ell)$
6: $(\pi_{\mathsf{ivc}\,\ell+1}, \mathsf{com}^{\ell+1}_k, h_{n+1}, \mathsf{pk}_{\mathsf{id}}, \mathsf{pk}_{\mathsf{sp}}) \leftarrow \pi_{\mathsf{usync}_{\ell+1}}, (\ell+1, r_{\ell+1}) \leftarrow \mathsf{st}_{\ell+1}$
7: $z_0 \leftarrow (\mathsf{com}^0_k, h_0, \mathsf{pk}_{\mathsf{id}}, \mathsf{pk}_{\mathsf{sp}})$, $z_{\ell+1} \leftarrow (\mathsf{com}^{\ell+1}_k, h_{n+1}, \mathsf{pk}_{\mathsf{id}}, \mathsf{pk}_{\mathsf{sp}})$
8: $\pi_{\mathsf{nova}} \leftarrow \mathsf{zkSNARK}_{\mathsf{IVC}}.\mathcal{P}(\mathsf{pk}_{\mathsf{nova}}, (N, z_0, z_{\ell+1}), \pi_{\mathsf{ivc}\,\ell+1})$
9: $\pi_{\mathsf{post}} \leftarrow \mathsf{zkSNARK}.\mathcal{P}(\mathsf{pk}_{\mathsf{post}}, (\mathsf{pk}_{\mathsf{id}}, \mathsf{tok}, \mathsf{com}^{\ell+1}_k; c, r_{\ell+1}))$
10: return $\pi_{\mathsf{auth}} \leftarrow (\pi_{\mathsf{nova}}, \pi_{\mathsf{post}}, \mathsf{com}^0_k, h_0, \mathsf{com}^{\ell+1}_k, h_{n+1}, N)$

$\underline{\mathsf{ReRand}(\mathsf{pk}, c, \pi_{\mathsf{usync}_\ell}, \mathsf{st}_\ell)}$:
1: $(\mathsf{pk}_{\mathsf{id}}, \mathsf{pk}_{\mathsf{sp}}, (\mathsf{pk}_{\mathsf{post}}, \mathsf{pk}_{\mathsf{nova}})) \leftarrow \mathsf{pk}$, $(\mathsf{pk}_{\mathsf{IVC}}, \mathsf{pk}_{\mathsf{zkSNARK}_{\mathsf{IVC}}}) \leftarrow \mathsf{pk}_{\mathsf{nova}}$
2: $(k, r, \mathsf{com}_k, b_{\mathsf{start}}, \sigma_{\mathsf{start}}) \leftarrow c$
3: $h_0 \leftarrow h_\perp, \mathsf{com}^0_k \leftarrow \mathsf{com}_\perp$
4: $(\pi_{\mathsf{ivc}\,\ell}, \mathsf{com}^\ell_k, h_{n+1}, \mathsf{pk}_{\mathsf{id}}, \mathsf{pk}_{\mathsf{sp}}) \leftarrow \pi_{\mathsf{usync}_\ell}, (\ell, r_\ell) \leftarrow \mathsf{st}_\ell$
5: $r_{\ell+1} \leftarrow\$ \{0,1\}^\lambda$
6: $(\mathsf{com}^{\ell+1}_k, h_{n+1}, \mathsf{pk}_{\mathsf{id}}, \mathsf{pk}_{\mathsf{sp}}) \leftarrow F(\mathsf{com}^\ell_k, h_{n+1}, \mathsf{pk}_{\mathsf{id}}, \mathsf{pk}_{\mathsf{sp}};$
$\quad c, \sigma_\perp, b_\perp, b_\perp, r_\ell, r_{\ell+1})$
7: $z_0 \leftarrow (\mathsf{com}^0_k, h_0, \mathsf{pk}_{\mathsf{id}}, \mathsf{pk}_{\mathsf{sp}})$, $z_{\ell+1} \leftarrow (\mathsf{com}^{\ell+1}_k, h_{n+1}, \mathsf{pk}_{\mathsf{id}}, \mathsf{pk}_{\mathsf{sp}})$
8: $\pi_{\mathsf{ivc}\,\ell+1} \leftarrow \mathsf{IVC}.\mathcal{P}(\mathsf{pk}_{\mathsf{IVC}}, (N, z_0, z_{\ell+1}), (c, \sigma_\perp, b_\perp, b_\perp, r_\ell, r_{\ell+1}), \pi_{\mathsf{ivc}\,\ell})$
9: $\pi_{\mathsf{usync}_{\ell+1}} \leftarrow (\pi_{\mathsf{ivc}\,\ell+1}, \mathsf{com}^{\ell+1}_k, h_{n+1}, \mathsf{pk}_{\mathsf{id}}, \mathsf{pk}_{\mathsf{sp}}), \mathsf{st}_{\ell+1} \leftarrow (\ell+1, r_{\ell+1})$
10: **return** $(\pi_{\mathsf{usync}_{\ell+1}}, \mathsf{st}_{\ell+1})$

Figure 8: Authorization Protocol. The sub-procedure ReRand re-randomizes $\pi_{\mathsf{usync}_\ell}$ to $\pi_{\mathsf{usync}_{\ell+1}}$ so that each π_{auth} from every Auth request contains re-randomized $\mathsf{com}^{\ell+1}_k$.

Authorization (Figure 8). After obtaining a token, a user runs Auth to obtain a zero-knowledge proof π_{auth} that they are "eligible" to post. By "eligible", (a) no block in the

blocklist references the user's credential c, and (b) the new post request is valid. Each statement is incorporated in the relations $\mathcal{R}_{\mathsf{IVC}}$ and $\mathcal{R}_{\mathsf{post}}$. Auth generates two zero-knowledge proofs π_{nova} and π_{post} for $\mathcal{R}_{\mathsf{IVC}}$ and $\mathcal{R}_{\mathsf{post}}$ respectively with c as a witness.

Notice that USync has been updating the proof for (a) until the last block of the blocklist. Let the final proof be $\pi_{\mathsf{usync}_\ell}$ where ℓ is the total number of USync iterations. Auth has three main steps: re-randomizing $\pi_{\mathsf{usync}_\ell}$ through a sub-procedure ReRand, generating zero-knowledge proofs π_{nova} and π_{post}, and constructing π_{auth}.

The reason Auth re-randomizes $\pi_{\mathsf{usync}_\ell}$ is to ensure unlinkability of π_{auth}. Auth's output π_{auth} contains the public input and output of F, which would be z_0 and z_ℓ without re-randomization. Recall that z_ℓ contains com_k^ℓ. All π_{auth} from the same c then would contain the same com_k^ℓ. An adversary would easily be able to link different π_{auth} by observing com_k^ℓ, leading to break anonymity.

This is why we run one more IVC in Auth. We resolve the issue by re-randomizing the commitment for every Auth request through an IVC iteration. This time, function F does not check if the given block references c as no more blocks are left to check (this is why the input b_j is set as $b_\perp$ for F). F only re-randomizes commitment com_k^ℓ to $\mathsf{com}_k^{\ell+1}$ and outputs $z_{\ell+1} = (\mathsf{com}_k^{\ell+1}, h_{n+1}, \mathsf{pk}_{\mathsf{id}}, \mathsf{pk}_{\mathsf{sp}})$. It has the same hash h_{n+1} as that in z_ℓ.

With the re-randomized updatable synchronization proof $\pi_{\mathsf{usync}_{\ell+1}} = (\pi_{\mathsf{ivc}_{\ell+1}}, z_{\ell+1})$, Auth runs the $\mathsf{zkSNARK}_{\mathsf{IVC}}$ prover. It compresses $\pi_{\mathsf{usync}_{\ell+1}}$ to a zero-knowledge proof π_{nova}, i.e., it knows $\pi_{\mathsf{ivc}_{\ell+1}}$ such that $z_{\ell+1} = F^{\ell+1}(z_0)$. The zkSNARK prover generates another zero-knowledge proof π_{post} that $(\mathsf{pk}_{\mathsf{id}}, \mathsf{tok}, \mathsf{com}_k^{\ell+1}; c, r_{\ell+1}) \in \mathcal{R}_{\mathsf{post}}$, i.e. tok is generated using c's key, c is honestly formed, and the final commitment $\mathsf{com}_k^{\ell+1}$ commits to the same key as that of c.

Finally, Auth outputs π_{auth} such that

$$\pi_{\mathsf{auth}} = (\pi_{\mathsf{nova}}, \pi_{\mathsf{post}}, \mathsf{com}_k^0, h_0, \mathsf{com}_k^{\ell+1}, h_{n+1}, N).$$

$\mathsf{Ver}(\mathsf{pk}, \mathsf{vk}, \mathsf{tok}, m, d, \pi_{\mathsf{auth}})$:
1: $(\pi_{\mathsf{nova}}, \pi_{\mathsf{post}}, \mathsf{com}_k^0, h_0, \mathsf{com}_k^{\ell+1}, h_{n+1}, N) \leftarrow \pi_{\mathsf{auth}}$
2: $(\mathsf{tag}, r_{\mathsf{nonce}}, \mathsf{mh}) \leftarrow \mathsf{tok}$
3: $(\mathsf{pk}_{\mathsf{id}}, \mathsf{pk}_{\mathsf{sp}}, (\mathsf{pk}_{\mathsf{post}}, \mathsf{pk}_{\mathsf{nova}})) \leftarrow \mathsf{pk}$, $(\mathsf{vk}_{\mathsf{post}}, \mathsf{vk}_{\mathsf{nova}}) \leftarrow \mathsf{vk}$
4: $z_0 \leftarrow (\mathsf{com}_k^0, h_0, \mathsf{pk}_{\mathsf{id}}, \mathsf{pk}_{\mathsf{sp}})$, $z_{\ell+1} \leftarrow (\mathsf{com}_k^{\ell+1}, h_{n+1}, \mathsf{pk}_{\mathsf{id}}, \mathsf{pk}_{\mathsf{sp}})$
5: **return** $d = h_{n+1} \wedge \mathsf{mh} = \mathsf{Hash}(m) \wedge$
$\mathsf{zkSNARK}_{\mathsf{IVC}}.\mathcal{V}(\mathsf{vk}_{\mathsf{nova}}, (N, z_0, z_{\ell+1}), \pi_{\mathsf{nova}}) = 1 \wedge$
$\mathsf{zkSNARK}.\mathcal{V}(\mathsf{vk}_{\mathsf{post}}, (\mathsf{pk}_{\mathsf{id}}, \mathsf{tok}, \mathsf{com}_k^{\ell+1}), \pi_{\mathsf{post}}) = 1$

Figure 9: Verification Protocol.

Verification (Figure 9). The verification protocol either accepts or rejects the user's request to post a new message. Specifically, since π_{auth} contains two zero-knowledge proofs π_{nova} and π_{post}, Ver runs each zkSNARK verifier $\mathsf{zkSNARK}_{\mathsf{IVC}}.\mathcal{V}$ and $\mathsf{zkSNARK}.\mathcal{V}$ for each proof. Additionally, it checks the consistency between m and tok. Ver also checks if the digest d is the same as h_{n+1} from π_{auth} to verify π_{auth} has been honestly produced using the same blocklist that Ver refers to.

$\mathsf{Aud}(B, c)$:
1: $(k, r, \mathsf{com}_k, b_{\mathsf{start}}, \sigma_{\mathsf{start}}) \leftarrow c$
2: $(b_1, \cdots, b_n) \leftarrow B$
3: **for** $b_i \in [b_{\mathsf{start}}, b_n]$:
4: $(\mathsf{tag}, r_{\mathsf{nonce}}, \mathsf{mh}, h_i, i) \leftarrow b_i$
5: if $\mathsf{tag} = \mathsf{PRF}_k(r_{\mathsf{nonce}}, \mathsf{mh})$:
6: **return** 0
7: **return** 1

Figure 10: Audit Protocol.

Audit (Figure 10). When a user wants to check if they are blocked, they use Aud. Given a blocklist B and a user's credential c, Aud iterates from the cold-start block to the last block and returns 0 if there exists a block with c's token.

$\mathsf{BInit}()$:
1: $B \leftarrow \emptyset$, $h_0 \leftarrow h_\perp$
2: $b_0 = (\mathsf{tok}_\perp, h_0, 0)$
3: **return** $B \leftarrow \{B, b_0\}$

$\mathsf{BDig}(B)$:
1: $b_n \leftarrow$ last element of B
2: **return** $d = \mathsf{Hash}(b_n)$

$\mathsf{BAdd}(\mathsf{tok}, B)$:
1: $b_n \leftarrow$ last element of B
2: $h_{n+1} = \mathsf{Hash}(b_n)$
3: $b_{n+1} \leftarrow (\mathsf{tok}, h_{n+1}, n+1)$
4: **return** $B \leftarrow \{B, b_{n+1}\}$

$\mathsf{BRem}(B, b)$:
1: $\sigma_{\mathsf{ban_over}} \leftarrow \mathsf{Sig.Sign}(\mathsf{sk}_{\mathsf{sp}}, b)$
2: **return** $B \leftarrow \{B, \sigma_{\mathsf{ban_over}}\}$

Figure 11: Blocklist Protocols. Although we abuse the notation and describe adding a ban-over signature to the blocklist, the actual construction makes use of a distinct set for ban-over signatures.

Blocklist Protocols (Figure 11). Blocklist protocols contain four sub-protocols:

- Blocklist Initialization Protocol: When a service provider starts its service, BInit creates an empty blocklist B. Also, it adds a trivial tuple $b_0 = (\mathsf{tok}_\perp, h_\perp, 0)$ to B.
- Blocklist Addition Protocol: When the service provider decides to block a message m, BAdd adds the corresponding token tok of m to the B. Let the size of B be n. The service provider first generates a hash of the last block, $h_{n+1} = \mathsf{Hash}(b_n)$. Then it appends a new block $b_{n+1} = (\mathsf{tok}, h_{n+1}, n+1)$ to B.
- Blocklist Removal Protocol: When the service provider wants to unblock a block b in the blocklist, they produce a signature $\sigma_{\mathsf{ban_over}}$ on b through BRem to indicate that ban on b has been lifted.
- Blocklist Digest Protocol: Given a n-length blocklist B, BDig extracts the digest d by hashing the last block b_n.

IVC Function F (Figure 12). We finally give a detailed construction of function F. IVC is involved in USync and Auth. USync uses IVC in order to synchronize π_{usync_i} with the latest blocklist and to prove that the user is not blocked. Auth uses IVC in order to re-randomize the given proof $\pi_{\mathsf{usync}_\ell}$ to $\pi_{\mathsf{usync}_{\ell+1}}$.

We describe our approach to design function F for these purposes step by step—from F for a general list with PRF input and outputs to F for our specific construction.

1. F for a general list: Let us take a step back and consider a list where each entry consists of a PRF input and output pair and the hash of the previous entry. We want to

$F(\mathsf{com}_k^i, h_j, \mathsf{pk}_{\mathsf{id}}, \mathsf{pk}_{\mathsf{sp}}; c, \sigma^j_{\mathsf{ban_over}}, b_j, b_{j-1}, r_i, r_{i+1})$:

1: $\mathsf{verify_inputs}(\mathsf{com}_k^i, h_j, \mathsf{pk}_{\mathsf{id}}, \mathsf{pk}_{\mathsf{sp}}, c, \sigma^j_{\mathsf{ban_over}}, b_j, b_{j-1}, r_i)$
2: **if** $b_j = b_\perp$:
3: $\quad \mathsf{com}_k^{i+1} \leftarrow \mathsf{Com}(k, r_{i+1})$
4: $\quad$ **return** $(\mathsf{com}_k^{i+1}, h_j, \mathsf{pk}_{\mathsf{id}}, \mathsf{pk}_{\mathsf{sp}})$
5: **else**:
6: $\quad \mathsf{com}_k^{i+1} \leftarrow \mathsf{Com}(k, r_{i+1})$
7: $\quad h_{j+1} \leftarrow \mathsf{Hash}(b_j)$
8: $\quad$ **return** $(\mathsf{com}_k^{i+1}, h_{j+1}, \mathsf{pk}_{\mathsf{id}}, \mathsf{pk}_{\mathsf{sp}})$

$\mathsf{verify_inputs}(\mathsf{com}_k^i, h_j, \mathsf{pk}_{\mathsf{id}}, \mathsf{pk}_{\mathsf{sp}}, c, \sigma^j_{\mathsf{ban_over}}, b_j, b_{j-1}, r_i)$:

1: $(k, r, \mathsf{com}_k, b_{\mathsf{start}}, \sigma_{\mathsf{start}}) \leftarrow c$
2: $(\mathsf{tag}_j, r^j_{\mathsf{nonce}}, \mathsf{mh}_j, h_j, j) \leftarrow b_j$
3: **if** $b_j = b_\perp$:
4: $\quad \mathsf{verify\ open}(\mathsf{com}_k^i, r_i) = k$
5: **else if** $b_j = b_{\mathsf{start}}$:
6: $\quad \mathsf{verify\ open}(\mathsf{com}_k, r) = k$
7: $\quad \mathsf{verify\ Sig}.\mathcal{V}(\mathsf{pk}_{\mathsf{id}}, \sigma_{\mathsf{start}}, (\mathsf{com}_k, b_{\mathsf{start}})) = 1$
8: **else**:
9: $\quad \mathsf{verify\ open}(\mathsf{com}_k, r) = k$
10: $\quad \mathsf{verify\ Sig}.\mathcal{V}(\mathsf{pk}_{\mathsf{id}}, \sigma_{\mathsf{start}}, (\mathsf{com}_k, b_{\mathsf{start}})) = 1$
11: $\quad \mathsf{verify\ open}(\mathsf{com}_k^i, r_i) = k$
12: $\quad \mathsf{verify}\ h_j = \mathsf{Hash}(b_{j-1})$
13: $\quad \mathsf{verify}\ \mathsf{tag}_j \neq \mathsf{PRF}_k(r^j_{\mathsf{nonce}}, \mathsf{mh}^j) \vee \mathsf{Sig}.\mathcal{V}(\mathsf{pk}_{\mathsf{sp}}, \sigma^j_{\mathsf{ban_over}}, b_j) = 1$

Figure 12: IVC function F. We use j as $\mathsf{start}+i$. Given ith result and witnesses, F verifies the inputs and computes $(i+1)$th result.

construct F such that it ensures a key k is not used for any entry's pair. An intuitive approach is to iterate the list from the first entry to the last entry and check if the output is the PRF result of the input with k.

On top of this, F needs to ensure that the same key k is used for all entries and that the same list is used as what the verifier sees. We employ a cryptographic commitment and hash for this. After checking the PRF relation, F commits to k, say $\mathsf{com}_k^i \leftarrow \mathsf{Com}(k, r_i)$ for ith entry, and hashes to the current entry, say $h_i \leftarrow \mathsf{Hash}(\mathsf{entry})$. F then outputs com_k^i and h_i. During the next iteration, F verifies com_k^i and h_i.

2. Extend F for an anonymous blocklist: We now apply this intuition to our blocklisting setting. Each entry of a blocklist is a block b. Since a service provider can deactivate the PRF relation by providing a ban-over signature, F checks the following for each block: $\mathsf{tag} \neq \mathsf{PRF}_k(r_{\mathsf{nonce}}, \mathsf{mh}) \vee \mathsf{Sig}.\mathcal{V}(\mathsf{pk}_{\mathsf{sp}}, \sigma^i_{\mathsf{ban_over}}, b_i) = 1$ as well as checking com_k^i and h_i. F in our construction needs to perform additional checks as it needs to ensure that k is the user's key approved by the identity provider. As a result, F also verifies $c = (k, r, \mathsf{com}_k, b_{\mathsf{start}}, \sigma_{\mathsf{start}})$—the validity of σ_{start} and opening of com_k.

3. Extend F for the cold-start model: Since our construction further supports a cold-start model, F does not need to iterate from the first block. F can start from the user's cold-start block b_{start}. Accordingly, for iteration i, the block we check is not b_i but $b_j = b_{\mathsf{start}+i}$.

Taken all together, F takes $z_i = (\mathsf{com}_k^i, h_j, \mathsf{pk}_{\mathsf{id}}, \mathsf{pk}_{\mathsf{sp}})$ as public inputs. com_k^i and h_j are the commitments and hashes from the previous iteration. We need $\mathsf{pk}_{\mathsf{id}}$ and $\mathsf{pk}_{\mathsf{sp}}$ for the verification of σ_{start} and $\sigma_{\mathsf{ban_over}}$. F also takes witnesses

$$\omega_i = (c, \sigma^j_{\mathsf{ban_over}}, b_j, b_{j-1}, r_i, r_{i+1})$$

where r_i is for the opening of com_k^i and r_{i+1} is for generating com_k^{i+1}. F proceeds with the checks and outputs

$$z_{i+1} = (\mathsf{com}_k^{i+1}, h_{j+1}, \mathsf{pk}_{\mathsf{id}}, \mathsf{pk}_{\mathsf{sp}}).$$

4. Extend F for the re-randomization case: F is also invoked for a re-randomization purpose in Auth. As mentioned above, this is because without re-randomization, com_k^ℓ in π_{auth} can be used to link different π_{auth}.

We observe that we can address this issue by running only a part of function F. We invoke F to only compute the re-randomization of commitment $\mathsf{com}_k^{\ell+1}$ while passing the other values as inputs $(h_{n+1}, \mathsf{pk}_{\mathsf{id}}, \mathsf{pk}_{\mathsf{sp}})$. Thus, F only checks com_k^ℓ and outputs $z_{\ell+1} = (\mathsf{com}_k^{\ell+1}, h_{n+1}, \mathsf{pk}_{\mathsf{id}}, \mathsf{pk}_{\mathsf{sp}})$.

6. Security Analysis

6.1. Blocklistability

Theorem 2. *For any PPT adversary $\mathcal{A}$ and the blocklistability game* BL *in Figure 1,* $\boldsymbol{Adv}^{\mathsf{BL}}_{ALPACA}(\mathcal{A}) \leq \mathsf{negl}(\lambda)$.

Proof sketch. Our proof first proceeds with two game hops. The first game hop additionally extracts the witness used for π_{post} from the extractor of zkSNARK at the end of the game and checks $\mathcal{A}$'s output and the witness satisfy $\mathcal{R}_{\mathsf{post}}$. Similarly, the second game hop additionally extracts the witness for π_{nova} from the extractor of $\mathsf{zkSNARK}_{\mathsf{IVC}}$ and checks $\mathcal{A}$'s output and the witness satisfy $\mathcal{R}_{\mathsf{IVC}}$. We are now in a game where we can extract the credential c that $\mathcal{A}$ uses to output values that Ver verifies. Our strategy is to show that both Ver and the relations $\mathcal{R}_{\mathsf{post}}$ and $\mathcal{R}_{\mathsf{IVC}}$ cannot be satisfied with $\mathcal{A}$'s chosen c. We consider several attempts $\mathcal{A}$ can make to choose c and show that each attempt will fail under our security assumptions.

First, the most straightforward attempt is using a credential $c = (k, r, \mathsf{com}_k, b_{\mathsf{start}}, \sigma_{\mathsf{start}}) \in \mathcal{C}$, the set of credentials registered through $\mathcal{O}_{\mathsf{Reg}}$. All credentials in $\mathcal{C}$ are blocked due to all_blocked condition. Recall that $\mathcal{A}$ needs to pass all checks in every iteration of function F to satisfy $\mathcal{R}_{\mathsf{IVC}}$. The only way for $\mathcal{A}$ to pass is by forging the service provider's ban-over signature(s) so that it can bypass the blocked condition. However, this probability is negligible, due to the strong unfogeability of our signature scheme.

Now consider another attempt where $\mathcal{A}$ comes up with an adversarial credential $c^* = (k^*, r^*, \mathsf{com}_k^*, b^*_{\mathsf{start}}, \sigma^*_{\mathsf{start}}) \notin \mathcal{C}$ on its own to avoid all_blocked condition. Function F for $\mathcal{R}_{\mathsf{IVC}}$ and $\mathcal{R}_{\mathsf{post}}$ verify the cold-start signature $\sigma^*_{\mathsf{start}}$. $\mathcal{A}$ fails again for the same reason as it cannot forge an accepted $\sigma^*_{\mathsf{start}}$ without the identity provider's secret key.

We then consider the case where $\mathcal{A}$ combines credentials c and c^* and constructs another adversarial credential, e.g. $c' = (k^*, r^*, \mathsf{com}_k, b_{\mathsf{start}}, \sigma_{\mathsf{start}})$, to make the cold-start signature be verified as well as avoiding all_blocked

condition. The message of σ_{start}, $(\mathsf{com}_k, b_{\mathsf{start}})$ must also be from $c \in \mathcal{C}$ to make σ_{start} be accepted. $\mathcal{A}$, however, cannot win even with c' because function F for $\mathcal{R}_{\mathsf{IVC}}$ and $\mathcal{R}_{\mathsf{post}}$ evaluate com_k. Since (k^*, r^*) is not the pair that com_k commits to, the opening of com_k fails without negligible probability due to the binding property of our commitment scheme. Therefore, $\mathcal{A}$ must use all elements from c, which immediately falls back to the first case.

In the full proof, we additionally show that $\mathcal{A}$ must use the same credential for all USync steps. We omit the proof as the proof is similar to the case analyses above. □

6.2. Anonymity

Anonymity encompasses unlinkability of tokens and proofs and not revealing credentials to an adversary. The adversary $\mathcal{A}$ wins the anonymity game AN (Figure 1) if it can guess the challenge credential c_b successfully through learning c_b or linking tokens and authorization proofs π_{auth}. In the anonymity game for our scheme, we give $\mathcal{A}$ the power to choose b_{start} for the credentials c_0 and c_1. Thereby we guarantee that c_0 and c_1 can be registered at different times.

Theorem 3. *For any PPT adversary $\mathcal{A}$ and the Anonymity Game* AN *in Figure 1,* $\boldsymbol{Adv}^{\mathsf{AN}}_{ALPACA}(\mathcal{A}) \leq \frac{1}{2} \pm \mathsf{negl}(\lambda)$.

Proof sketch. Our approach is to prove anonymity of the authorization proofs and tokens. We first show anonymity of $\pi_{\mathsf{auth}} = (\pi_{\mathsf{nova}}, \pi_{\mathsf{post}}, \mathsf{com}_k^0, h_0, \mathsf{com}_k^{\ell+1}, h_{n+1}, N)$ by analyzing each element. Notice that the credential c used to generate π_{auth} is not blocked (ensured by π_{nova}). Since h_0 and h_{n+1} are hashes of blocks and c is not blocked by them, they do not imply anything about c.

Furthermore, N is a global upper-bound of the number of iterations same for all π_{auth} independent of c. This is why we use iteration-hiding IVC—observe that IVC will reveal the number of iterations, which can be used to reduce the anonymity set.

The zero-knowledge proofs π_{nova} and π_{post} do not reveal the witness c. The hiding property of com_k^0 and $\mathsf{com}_k^{\ell+1}$ hides c's key k. Also, com_k^0 is the same bot value (Figure 8) for all π_{auth} and $\mathsf{com}_k^{\ell+1}$ is re-randomized, thus fresh for every π_{auth}. As a result, π_{auth} are unlinkable.

In addition to the anonymity of proofs, we prove anonymity of tokens, $\mathsf{tok} = (\mathsf{tag}, r_{\mathsf{nonce}}, \mathsf{mh})$. By pseudo-randomness of PRF, tok is indistinguishable from a random token. Thus, an adversary cannot learn the key k that tag is computed with through observing tok.

We further discuss unlinkability of tokens. Conceptually, if $\mathcal{A}$ can successfully link tokens, it means that it can distinguish PRF output sets with one key from those with another key. Our strategy is to first proceed with a series of game hops to make a reduction between AN and the PRF key indistinguishability game, say KD, work.

We need the game hops considering the reduction flow. $\mathcal{A}'$ of KD will act as the challenger of AN, meaning it creates challenge credentials. However, these credentials contain the keys for PRF, which $\mathcal{A}'$ must not know. We can address this issue through game hops using the zero-knowledge property of π_{nova} and π_{post}. The first two game hops show that AN can be modified to simulate provers of $\mathsf{zkSNARK}_{\mathsf{IVC}}$ and zkSNARK without the witness c. Another game hop further removes USync invocations with c in the oracles. Without $(\pi_{\mathsf{usync}_{\ell+1}}, \mathsf{st}_{\ell+1})$ from USync, the hopped game uses a simulated commitment $\mathsf{com}_k^{\ell+1}$. We show that the distributions of the inputs to simulators still remain identical by showing that the distributions of the real and simulated commitments are identical, by hiding.

We are now in a game where the challenger does not have dependency on credentials and we can build a reduction. We show that if $\mathcal{A}$ wins AN, then $\mathcal{A}'$ wins KD. However, this probability is negligible due to the key indistinguishability assumption of PRF. □

6.3. Non-Frameability

Theorem 4. *For any PPT adversary $\mathcal{A}$ and the non-frameability game* NF *in Figure 1,* $\boldsymbol{Adv}^{\mathsf{NF}}_{ALPACA}(\mathcal{A}) \leq \mathsf{negl}(\lambda)$.

Proof sketch. As we discussed earlier, an adversary can attempt to frame the challenge credential c' by adding an adversarially constructed token to the blocklist, so that the c' appears to be blocked even though the service provider has never added c''s tokens.

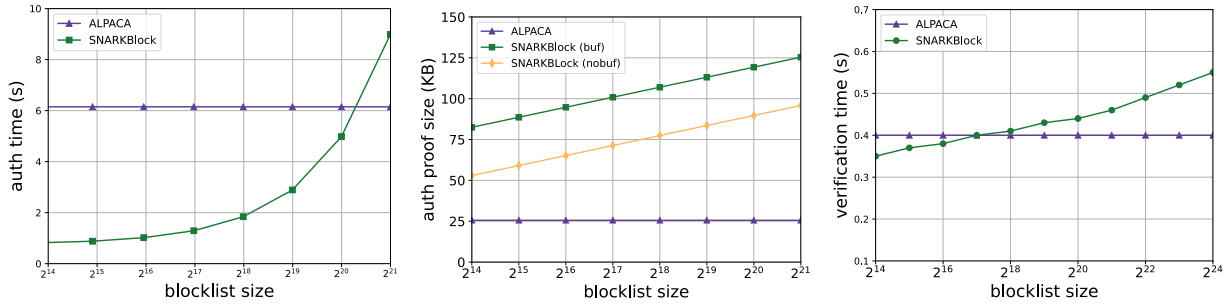
We observe that this is the only attempt $\mathcal{A}$ can make. This is because the inputs $(\mathsf{tok}', d', \pi'_{\mathsf{auth}})$ to Ver are honestly generated by the challenger with c' (except m'). Recall that the zkSNARK and $\mathsf{zkSNARK}_{\mathsf{IVC}}$ verifier will verify π_{post} and π_{nova} in π'_{auth}. Let us exclude the checks in $\mathcal{R}_{\mathsf{post}}$ and F for $\mathcal{R}_{\mathsf{IVC}}$ on tok', d', and π'_{auth} since they will trivially pass. Then only one statement in F remains that checks any block in the blocklist references c'. Thus, $\mathcal{A}$ can only make this check fail by manipulating the blocklist.

We describe the attack in more detail. $\mathcal{A}$ generates an adversarial token $\mathsf{tok} = (\mathsf{tag}, r_{\mathsf{nonce}}, \mathsf{mh})$ with a credential c that has key k, i.e. $\mathsf{tag} = \mathsf{PRF}_k(r_{\mathsf{nonce}}, \mathsf{mh})$. $\mathcal{A}$ then adds tok to the blocklist through $\mathcal{O}_{\mathsf{BAdd}}$ and never queries $\mathcal{O}_{\mathsf{BRem}}$ with the block b that contains tok. Now, say that for the block b and key k' of the challenge credential c', the equation $\mathsf{tag} = \mathsf{PRF}_{k'}(r_{\mathsf{nonce}}, \mathsf{mh})$ also holds. Ver for c' will fail since the verifier will regard c' as blocked by b.

We show that $\mathcal{A}$'s attempt fails without negligible probability, due to the collision resistance of our PRF scheme. If $\mathcal{A}$ wins with this strategy, then there exists a collision in PRF since $\mathsf{PRF}_k(r_{\mathsf{nonce}}, \mathsf{mh}) = \mathsf{PRF}_{k'}(r_{\mathsf{nonce}}, \mathsf{mh})$. We build a reduction between NF and the PRF collision resistance game, say CR, where we show that if $\mathcal{A}$ wins NF, then the adversary of CR can use the collision to win CR. □

7. Implementation and Evaluation

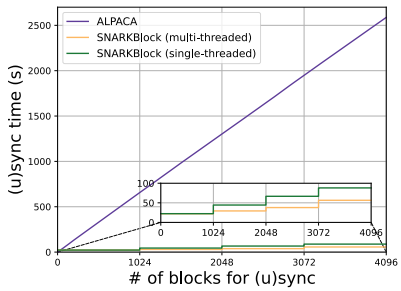
We implement ALPACA in Rust. Our implementation is roughly 5,000 lines of code and available at https://github.com/jiwonkimpark/alpaca. ALPACA works over the Pasta elliptic curves [23]. We



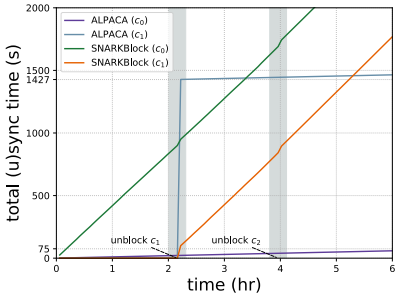
(a) Authorization time (b) Size of authorization proof π_{auth} (c) Verification time

Event	Scheme	Time (s)
BAdd	SNARKBlock	22.16
	ALPACA	**0.63**
BRem (blocked c)	SNARKBlock	22.16·ch
	ALPACA	**0.63**·n'
BRem (other c)	SNARKBlock	22.16
	ALPACA	**0**

(d) Sync time per event



(e) Sync time for multiple block additions



(f) Workload experiment (discussion in §7)

Figure 13: Comparison of costs between ALPACA and SNARKBlock [15]. The chunk size of SNARKBlock is set to 1024, as in [15]. Authorization time, proof size, and verification time of ALPACA remain the same regardless of the increase in the blocklist size. ch and n' means the number of chunks and blocks that synchronization should run. For all graphs, lower y-axis values are better.

implemented Poseidon hash [22] and instantiated our hash function, commitment scheme, PRF, and the Schnorr signature scheme with Poseidon hash. We also instantiated $\mathsf{zkSNARK}_{\mathsf{IVC}}$ and IVC scheme with Nova [17] and zkSNARK with Spartan [28]. This includes representing function F and $\mathcal{R}_{\mathsf{post}}$ as circuits using the Bellpepper gadget [29] and the CirC compiler [30] respectively and modifying the underlying curve of Spartan implementation to Pasta curve. The size of the circuit for IVC function F is 31842 constraints, and for $\mathcal{R}_{\mathsf{post}}$ is 16287 constraints.

To deploy our system in practice, we will need to use a slightly modified variant of Nova that provides iteration hiding for the synchronization protocol (§ 4). Our implementation does not yet support iteration hiding, but we estimate these modifications will add negligible overhead to the step circuit size consisting of two circuit-friendly hashes (~ 256 constraints each) and an inequality constraint (~ 64 constraints [31]). This totals to ~ 576 additional constraints, which, when compared to the current size of the circuit for relation $\mathcal{R}_{\mathsf{IVC}}$ with the existing recursion overhead (~ 10000 constraints [17]), would amount to roughly 1.4% increase in cost compared to our measured costs for authorization time, synchronization time and verification time. These costs are not represented in our graphs, but the qualitative takeaways of our experiments would not change.

We now provide an empirical analysis of ALPACA's efficiency that answers the following questions:

- How do the authorization proof size, authorization time, and verification time vary with blocklist size?
- How do the updatable synchronization runtime and proof size change with respect to the blocklist size?
- What do these results imply in a real-world scenario?

To apply the real-world cases, we experiment in two environments. We measure server costs (i.e., verification time) with an AMD EPYC 9754 processor with 128 physical cores and 530GB memory running Ubuntu Server 24.04.1 LTS, and the client costs (i.e., proving times) with a 2.3 GHz Quad-Core Intel Core i5 processor and 16GB memory running MacOS Sonoma 14.3.1.

Our baseline is the prior state-of-the-art system, SNARKBlock [15]. (We do not compare with BLAC because it is less efficient than SNARKBlock on all figures of merit.) We set the chunk size, the number of blocks in a chunk, of SNARKBlock as $1024 = 2^{10}$, following the SNARKBlock authors' evaluation. We evaluate SNARKBlock's non-buffered and buffered modes: non-buffered mode always uses 1024-sized chunks, whereas buffered mode sets the chunk size for the last chunk to a smaller value (e.g., 16) to reduce latency.

Authorization time. Figure 13(a) compares ALPACA and SNARKBlock on Auth time. For SNARKBlock this is the time required to aggregate chunk proofs.

The authorization time of ALPACA is ~ 6.15 seconds regardless of the blocklist size. In contrast, SNARKBlock's

cost grows linearly in the blocklist size. For size $n = 2^{21}$, the authorization time of ALPACA is $\sim 1.5\times$ faster than both buffered and non-buffered SNARKBlock modes.

Authorization proof size. As shown in 13(b), the size of ALPACA's authentication proof π_{auth} is also independent of blocklist size, measuring $\sim$ 25.6KB per authorization request. On the other hand, SNARKBlock's proof grows logarithmically with blocklist size in both buffered and non-buffered modes, and is concretely much larger than ALPACA's even for small blocklists.

Our achievement stems from the fact that IVC in USync updates the previous proof to a proof of the same size. The size of π_{usync} is $\sim$2MB regardless of the blocklist size. Since Auth always compresses this to a zero-knowledge proof, π_{auth} size maintains constant.

Verification time. Figure 13(c) shows verification time, which is crucial because it affects the service provider's throughput and users' perceived latency when interacting with the service. For ALPACA, verification time is $\sim$ 400ms independent of the size of the blocklist. For small blocklists, SNARKBlock's verification time is similar to or slightly better than ALPACA's; for blocklists larger than 2^{17}, however, ALPACA gives better performance. At 2^{21}, ALPACA is $\sim 1.4\times$ faster than SNARKBlock.

Synchronization time. Figures 13(d), 13(e), and 13(f) compare synchronization performance. Figure 13(d) depicts the cost of synchronizing one new event: for ALPACA, each USync invocation costs $\sim$ 630ms. In contrast, the cost to generate a new chunk proof of size 1024 in SNARKBlock is 22.16s. This illustrates the importance of buffered mode in SNARKBlock: it lets the client avoid generating a costly, full-size chunk for a small number of newly added blocks.

Figure 13(e) shows the cost of synchronizing many events at once. ALPACA's runtime is linear in the number of new blocks n, i.e., $\sim (630 \cdot n)$ms. SNARKBlock's costs are lower because the cost of a chunk proof is amortized over 1024 events. In addition, SNARKBlock can generate multiple chunk proofs in parallel.

An important difference between ALPACA and SNARKBlock is behavior when a previous block is removed. For ALPACA, users not subject to the block take no action (the blocked user, once unblocked, simply catches up to the current head of the block list). As shown in Figure 13(d), however, in SNARKBlock's case all users—including users who were not blocked—must re-generate at least the chunk proof for the chunk that contained the removed block.

Real-world workload. We also provide an end-to-end test result demonstrating how ALPACA behaves in a real-world scenario where blocks are dynamically added and removed. Although the frequency and amount vary from day to day, we observe that 300 to 400 posts are blocked a day in average in Wikipedia [24]. We simulate this in our experiment as follows. We start from a blocklist with size $n = 2^{14}$. SNARKBlock's chunk size is 1024, meaning that it has 16 chunks; new blocks are added to a 17th chunk. For this experiment we focus on three users, c_0, c_1, and c_2, where

- c_0 has never been blocked.
- c_1 was blocked by 14000th block b_{14000} (in 14th chunk).
- c_2 was blocked by 16000th block b_{16000} (in 15th chunk).

After initializing the blocklist as above, we add a total of 100 blocks periodically over 6 hours. Over that time, the blocks of c_1 and c_2 expire.

Figure 13(f) shows the accumulated synchronization cost for c_0 and c_1 over this time span. Observe that when c_1's block b_{14000} expires, both c_0 and c_1 would need to synchronize if they were using SNARKBlock. Specifically, c_0 runs the chunk proof for the 14th chunk that contains b_{14000} as well as current 17th chunk. c_1 runs four chunk proofs from 14th to 17th chunk (or two chunks if they already have proofs for 15th and 16th chunk).

In contrast, c_0 in ALPACA does not need to synchronize in response to a block's expiration, because their view of blocklist does not change. After being unblocked, c_1 runs USync from b_{14000} to the latest block.

When c_2's block b_{16000} expires, both c_0 and c_1 would once again need to run synchronization for two chunks (the 15th and 17th). In contrast, neither c_0 nor c_1 would run USync under ALPACA in response to c_2's block expiration.

In sum, when a block expires *all* users in SNARKBlock need to recompute proofs, whereas in ALPACA only the newly unblocked user runs computation, i.e., to catch up to the current state of the blocklist. As expected, every user runs exactly one USync proof for each entry in the blocklist; the effect of banning a user is to delay their running USync until their block expires.

8. Related Work

An anonymous blocklisting scheme was first introduced by Tsang et al [14]. Their work BLAC proposes adding "tickets" instead of users' credentials to the blocklist and removing them when the service provider unblocks. A $ticket = (\mathsf{tag}, \mathsf{nonce})$ contains a tag and a random nonce. $\mathsf{tag} = h(\mathsf{nonce}, sid)^x$, where h is a cryptographic hash function, sid is the service provider's ID, and x is a piece of a user's identity. For authorization, users generate a proof that proves their non-membership of the blocklist using inequalities of discrete logarithms, i.e. $\forall ticket, \mathsf{tag} \neq h(\mathsf{nonce}, sid)^x$. Although their work laid the foundations of the blocklist structure and security requirements, BLAC is highly impractical as the proof size and verification time are $O(n)$ where n is the blocklist size. This also makes the service susceptible to denial of service.

Henry and Goldberg [27] broadly survey the literature and define algorithms and security for anonymous blocklisting schemes. The authors informally introduce misauthentication resistance, backward anonymity, unlinkability, revocability, revocation auditability, and non-frameability that an anonymous blocklisting scheme can consider.

Rosenberg et al. reduce the proof size and verifier time to $O(\log n)$ in their work SNARKBlock [15]. SNARKBlock adopts a slightly different design of a user's identity and token, which is equivalent to a ticket in BLAC.

The user's identity is a PRF key k that is signed by an identity provider, and a token $(\mathsf{tag}, \mathsf{nonce})$ is generated by $\mathsf{tag} = \mathsf{PRF}_k(\mathsf{nonce})$. Same as BLAC, the service provider blocks users by adding their token $(\mathsf{tag}, \mathsf{nonce})$ to the blocklist and drop the ban by removing the token from the blocklist. SNARKBlock's approach to improve efficiency is to divide a blocklist into same-sized disjoint chunks. For this, the authors introduce a new zero-knowledge proof HICIAP that can aggregate n Groth16 "chunk proofs" into a $O(\log n)$-sized proof. SNARKBlock also defines an offline synchronization algorithm that produces chunk proofs that prove a user's non-membership in each chunk. Then, the authorization protocol leverages HICIAP and aggregates all chunk proofs into a single zero-knowledge authentication proof π. Since the commitments to witnesses included in π can be used to link different authentication proofs, HICIAP first re-randomizes the chunk proofs to ensure unlinkability. Despite the performance enhancement and its security, the practicality of SNARKBlock remains unclear since the prover time, proof size, and verification latency increase as the blocklist elongates. As the authors pointed out, another drawback is that a new user needs to run sync from the beginning, even for blocklist entries made before they joined.

Shih et al. present zk-promises [32], an anonymous scheme that provides various content moderation functionalities including blocking, reputation, and rate limits. The key idea is to use callback functions to update a user's private state; the user registers callbacks for each functionality per message, the service provider invokes a callback, and the user ingests it, which will update the user's state. After ingestion, the user posts a commitment to the updated state and a proof of correct callback ingestion to a bulletin board. This ensures that the callback was ingested against the latest user state. However, it makes their communication overhead substantial; every user has to communicate with the bulletin board for each callback ingestion. We discuss below how our approach could be used to build a more performant version of zk-promises.

9. Discussion and Future Work

In this section we briefly discuss potential directions for future work. First, we can extend ALPACA to support various blocking and unblocking policies chosen by the service provider. For instance, BLAC [14] supposes a d-strike-out blocking policy. We can support this by changing the non-membership check condition in function F; the function F receives a counter as an input, checks if the counter is less than d, and increases the counter if the current block is referencing the user. Even more complex blocking and unblocking policies are possible because of the flexibility of our IVC-based design.

Second, while ALPACA already represents a significant improvement in practicality compared to prior systems, other technical tools may offer further improvements. As a simple example: use of non-uniform incrementally verifiable computation, as in SuperNova [33], would allow users to execute batched USync proofs, which would likely result in significant cost reduction.

Third, ALPACA's techniques might be useful in anonymity applications beyond blocklisting, which can be viewed as a special case of a more generalized class of distributed, anonymous stream-processing workloads: the service provider issues a stream of commands (in ALPACA's case, "prove you are not the user who posted message Z"), and the participants in the system execute it. The same techniques could be used to efficiently implement essentially any streaming algorithm where users' local state is bounded. As a simple example, the command might instead be "decrement reputation by one point if you are the user who posted message Y". More generally, ALPACA might be used to implement the object-and-callback model described by Shih et al. [32], likely with significantly better efficiency: because of ALPACA's use of recursive proofs, users could entirely avoid posting updated state commitments. Instead, similar to this work, a proof would establish inductively that all callbacks had been properly applied, without reference to any publicly posted commitment.

ALPACA might also be useful in entirely different settings, e.g., sender-anonymous messaging, in which users communicate point-to-point without revealing their identity to anyone other than the recipient. A large body of prior work has detailed the potential for abuse and spam in this setting [34, 35]. The recent Orca [36] protocol allows for *per-user* blocklisting; a complementary primitive might be *global* blocklisting for bad actors, either built into the messaging service or implemented as an opt-in overlay. In principle, ALPACA's setting implies sender-anonymous messaging assuming that communicating parties can exchange keys out-of-band; while this approach is likely much less efficient than existing sender-anonymous systems, it suggests that an approach like ALPACA's might be adapted.

Acknowledgments

The authors wish to thanks Michael Rosenberg and the anonymous reviewers for helpful comments. This research was supported by a generous gift from Meta, NSF grant #2236784 and by DARPA under Agreement No. HR00112020022. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the United States Government or DARPA. Abhiram Kothapalli was supported in part by the AFOSR Award FA9550-24-1-0156 and research grants from the Bakar Fund, J. P. Morgan Faculty Research Award, Anaxi Labs, Protocol Labs, Bosch, Supra Inc., Sui Foundation, and the Stellar Development Foundation.

References

[1] K. Rigby, "Anonymity on the internet must be protected," https://groups.csail.mit.edu/mac/classes/6.805/student-papers/fall95-papers/rigby-anonymity.html.

[2] "4chan," https://www.4chan.org/.

[3] "Yikyak: find your herd," https://yikyak.com/.

[4] P. Syverson, R. Dingledine, and N. Mathewson, "Tor: The secondgeneration onion router," in *Usenix Security*. USENIX Association Berkeley, CA, 2004, pp. 303–320.

[5] "S.b. 287: Online pornography viewing age requirements," https://le.utah.gov/~2023/bills/sbillenr/SB0287.pdf.

[6] "S.1291 - 118th congress (2023-2024): Protecting kids on social media act," https://www.congress.gov/bill/118th-congress/senate-bill/1291/text.

[7] R. Johnson, https://x.com/SenRonJohnson/status/1354218776670203905.

[8] J. Suler, "The online disinhibition effect," *Cyberpsychology & behavior*, vol. 7, no. 3, pp. 321–326, 2004.

[9] "Dean's list: UNC system president wants to ban yik yak, other apps from campus wi-fi," https://www.newsobserver.com/news/local/education/article286128491.html, 2024.

[10] "The rise and fall of yik yak, the anonymous messaging app," https://www.nytimes.com/2017/05/27/style/yik-yak-bullying-mary-washington.html, 2017.

[11] A. Forte, N. Andalibi, and R. Greenstadt, "Privacy, anonymity, and perceived risk in open collaboration: A study of tor users and wikipedians," in *Proceedings of the 2017 ACM Conference on Computer Supported Cooperative Work and Social Computing*, 2017, pp. 1800–1811.

[12] C. Tran, K. Champion, A. Forte, B. M. Hill, and R. Greenstadt, "Are anonymity-seekers just like everybody else? an analysis of contributions to wikipedia from tor," in *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2020, pp. 186–202.

[13] "Bassel khartabil," https://en.wikipedia.org/wiki/Bassel_Khartabil, 2024.

[14] P. P. Tsang, M. H. Au, A. Kapadia, and S. W. Smith, "Blac: Revoking repeatedly misbehaving anonymous users without relying on ttps," *ACM Transactions on Information and System Security (TISSEC)*, vol. 13, no. 4, pp. 1–33, 2010.

[15] M. Rosenberg, M. Maller, and I. Miers, "Snarkblock: Federated anonymous blocklisting from hidden common input aggregate proofs," in *2022 IEEE Symposium on Security and Privacy (SP)*, May 2022, extended version: ePrint 2021/1577.

[16] P. Valiant, "Incrementally verifiable computation or proofs of knowledge imply time/space efficiency," in *Theory of Cryptography: Fifth Theory of Cryptography Conference, TCC 2008, New York, USA, March 19-21, 2008. Proceedings 5*. Springer, 2008, pp. 1–18.

[17] A. Kothapalli, S. Setty, and I. Tzialla, "Nova: Recursive zero-knowledge arguments from folding schemes," in *Annual International Cryptology Conference*. Springer, 2022, pp. 359–388.

[18] "Wikipedia:blocking policy," https://en.wikipedia.org/wiki/Wikipedia:Blocking_policy, 2024.

[19] A. Kothapalli and S. Setty, "Hypernova: Recursive arguments for customizable constraint systems," in *Annual International Cryptology Conference*. Springer, 2024, pp. 345–379.

[20] B. Bünz, A. Chiesa, P. Mishra, and N. Spooner, "Proof-carrying data from accumulation schemes," *Cryptology ePrint Archive*, 2020.

[21] B. Bünz, A. Chiesa, W. Lin, P. Mishra, and N. Spooner, "Proof-carrying data without succinct arguments," in *Advances in Cryptology–CRYPTO 2021: 41st Annual International Cryptology Conference, CRYPTO 2021, Virtual Event, August 16–20, 2021, Proceedings, Part I 41*. Springer, 2021, pp. 681–710.

[22] L. Grassi, D. Khovratovich, C. Rechberger, A. Roy, and M. Schofnegger, "Poseidon: A new hash function for {Zero-Knowledge} proof systems," in *30th USENIX Security Symposium (USENIX Security 21)*, 2021, pp. 519–535.

[23] D. Hopwood, "The pasta curves for halo 2 and beyond," https://electriccoin.co/blog/the-pasta-curves-for-halo-2-and-beyond/, 2020.

[24] "Wikipedia," https://en.wikipedia.org/wiki/Main_Page.

[25] "reddit," https://www.reddit.com/.

[26] "X," https://x.com/?lang=en.

[27] R. Henry and I. Goldberg, "Formalizing anonymous blacklisting systems," in *2011 IEEE Symposium on Security and Privacy*. IEEE, 2011, pp. 81–95.

[28] S. Setty, "Spartan: Efficient and general-purpose zksnarks without trusted setup," in *Annual International Cryptology Conference*. Springer, 2020, pp. 704–737.

[29] "Bellpepper," https://github.com/argumentcomputer/bellpepper.

[30] A. Ozdemir, F. Brown, and R. S. Wahby, "Circ: Compiler infrastructure for proof systems, software verification, and more," in *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2022, pp. 2248–2266.

[31] S. Setty, V. Vu, N. Panpalia, B. Braun, A. J. Blumberg, and M. Walfish, "Taking {Proof-Based} verified computation a few steps closer to practicality," in *21st USENIX Security Symposium (USENIX Security 12)*, 2012, pp. 253–268.

[32] M. Shih, M. Rosenberg, H. Kailad, and I. Miers, "zk-Promises: Making zero-knowledge objects accept the call for banning and reputation," Cryptology ePrint

Archive, Paper 2024/1260, 2024. [Online]. Available: https://eprint.iacr.org/2024/1260

[33] A. Kothapalli and S. Setty, "Supernova: Proving universal machine executions without universal circuits," *Cryptology ePrint Archive*, 2022.

[34] Y. Dodis, P. Grubbs, T. Ristenpart, and J. Woodage, "Fast message franking: From invisible salamanders to encryptment," in *Advances in Cryptology–CRYPTO 2018: 38th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 19–23, 2018, Proceedings, Part I 38*. Springer, 2018, pp. 155–186.

[35] P. Grubbs, J. Lu, and T. Ristenpart, "Message franking via committing authenticated encryption," in *Advances in Cryptology–CRYPTO 2017: 37th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 20–24, 2017, Proceedings, Part III 37*. Springer, 2017, pp. 66–97.

[36] N. Tyagi, J. Len, I. Miers, and T. Ristenpart, "Orca: Blocklisting in {Sender-Anonymous} messaging," in *31st USENIX Security Symposium (USENIX Security 22)*, 2022, pp. 2299–2316.

[37] A. Fiat and A. Shamir, "How to prove yourself: Practical solutions to identification and signature problems," 1986, pp. 186–194.

Appendix A. Formal Syntax of Anonymous Blocklisting

In this section, we provide the formal syntactic definitions of each protocol of anonymous blocklisting scheme.

- $\mathsf{Setup} : \{0,1\}^\lambda \to \mathsf{PP} \times \mathsf{PK}^3 \times \mathsf{SK}^2 \times \mathsf{VK}^2$. Given a security parameter, generate public parameters for proving synchronization, public and secret key pairs for id and sp, and proving and verifying key pairs for proving synchronization.
- $\langle \mathsf{Reg.Id}, \mathsf{Reg.User} \rangle : \mathsf{PK} \times \mathsf{SK} \to \mathsf{C}$. Given a public key and an identity provider secret key, the user and identity provider, interactively generate a new user's credential $c \in \mathsf{C}$.
- $\mathsf{Tok} : \mathsf{M} \times \mathsf{C} \to \mathsf{T}$. Given a message $m \in \mathsf{M}$ and a credential $c \in \mathsf{C}$, output a token $\mathsf{tok} \in \mathsf{T}$ of m.
- $\mathsf{Sync} : \mathsf{PK}^4 \times \mathsf{BL} \times \mathsf{C} \to \Pi_{\mathsf{sync}}$. Given public keys, a blocklist $B \in \mathsf{BL}$, and a credential $c \in \mathsf{C}$, output $\pi_{\mathsf{sync}} \in \Pi_{\mathsf{sync}}$ that proves no token in B references c.
- $\mathsf{Auth} : \mathsf{PK}^4 \times \mathsf{T} \times \mathsf{BL} \times \mathsf{C} \times \Pi_{\mathsf{sync}} \to \Pi_{\mathsf{auth}}$. Given public keys, a token $\mathsf{tok} \in \mathsf{T}$, a blocklist $B \in \mathsf{BL}$, a credential $c \in \mathsf{C}$, and a synchronization proof $\pi_{\mathsf{sync}} \in \Pi_{\mathsf{sync}}$, output an authorization proof $\pi_{\mathsf{auth}} \in \Pi_{\mathsf{auth}}$ that tok references c and that no token in B references c.
- $\mathsf{Ver} : \mathsf{PK}^4 \times \mathsf{VK}^2 \times \mathsf{T} \times \mathsf{M} \times \{0,1\}^\lambda \times \Pi_{\mathsf{auth}} \to \{0,1\}$. Given public keys, verifying keys, a token $\mathsf{tok} \in \mathsf{T}$, a message $m \in \mathsf{M}$, a blocklist digest d in $\{0,1\}^\lambda$, and an authorization proof $\pi_{\mathsf{auth}} \in \Pi_{\mathsf{auth}}$, output 1 if π_{auth} is valid and 0 otherwise.
- $\mathsf{Aud} : \mathsf{C} \times \mathsf{BL} \to \{0,1\}$. Given a credential $c \in \mathsf{C}$ and a blocklist $B \in \mathsf{BL}$, output 1 if no token in B references c.
- $\mathsf{BInit} : () \to \mathsf{BL}$. Initialize and output a blocklist $B \in \mathsf{BL}$.
- $\mathsf{BAdd} : \mathsf{T} \times \mathsf{BL} \to \mathsf{BL}$. Given a token $\mathsf{tok} \in \mathsf{T}$ and a blocklist $B \in \mathsf{BL}$, output a blocklist that blocks tok.
- $\mathsf{BRem} : \mathsf{BL} \times \mathsf{B} \to \mathsf{BL}$. Given a block $b \in \mathsf{B}$ and a blocklist $B \in \mathsf{BL}$ such that $b \in B$, output a blocklist B that does not ban b anymore.
- $\mathsf{BDig}: \mathsf{BL} \to \{0,1\}^\lambda$. Given a blocklist $B \in \mathsf{BL}$, output a blocklist digest $d \in \{0,1\}^\lambda$.

Appendix B. Incrementally Verifiable Computation

We begin by formally defining the security properties of incrementally verifiable computation.

Definition 4 (Iteration-Hiding IVC). An IVC scheme $(\mathcal{G}, \mathcal{K}, \mathcal{P}, \mathcal{V})$ following the syntax of Definition 2 satisfies the following requirements.

1) Perfect Completeness: For any PPT adversary $\mathcal{A}$, given $\mathsf{pp} \leftarrow \mathcal{G}(1^\lambda, \ell)$, $F, (z_0, z_i, N), (\Pi_i, \omega_i) \leftarrow \mathcal{A}(\mathsf{pp})$, $z_{i+1} \leftarrow F(z_i, \omega_i)$, $(\mathsf{pk}, \mathsf{vk}) \leftarrow \mathcal{K}(\mathsf{pp}, F)$, $\mathcal{V}(\mathsf{vk}, (z_0, z_i, N), \Pi_i) = 1$, and $\Pi_{i+1} \leftarrow \mathcal{P}(\mathsf{pk}, (z_0, z_i, N), (\Pi_i, \omega_i))$ we have that
$$\mathcal{V}(\mathsf{vk}, (z_0, z_{i+1}, N), \Pi_{i+1}) = 1$$
where F is a polynomial-time computable function.
2) Knowledge Soundness: Consider constant $N \in \mathbb{N}$. For all expected polynomial-time adversaries $\mathcal{P}^*$ there exists an expected polynomial-time extractor $\mathcal{E}$ such that for all randomness r given $\mathsf{pp} \leftarrow \mathcal{G}(1^\lambda, \ell)$, $(F, (z_0, z), \Pi) \leftarrow \mathcal{P}^*(\mathsf{pp}, \mathsf{r})$, $(\mathsf{pk}, \mathsf{vk}) \leftarrow \mathcal{K}(\mathsf{pp}, F)$, $\mathcal{V}(\mathsf{vk}, (z_0, z, N), \Pi) = 1$, and $(\omega_0, \ldots, \omega_{n-1}) \leftarrow \mathcal{E}(\mathsf{pp}, \mathsf{r})$ we have that the following probability is negligibly close to 1.
$$\Pr\left[\begin{array}{l} n \le N, \\ z_n = z \text{ where} \\ z_{i+1} \leftarrow F(z_i, \omega_i) \\ \forall i \in \{0, \ldots, n-1\} \end{array}\right] = 1$$
3) Succinctness: The size of an IVC proof Π is independent of the number of iterations n.

B.1. IVC-Compatible Folding Schemes

We now recall *folding schemes*, which are a key building block for constructing modern IVC schemes [17, 19–21]. Our particular use case of IVC requires folding schemes that satisfy additional desiderata such as NP-completeness of the relation being folded. Below we also reproduce the IVC-compatibility requirement as formalized by Kothapalli and Setty [19].

Definition 5 (Folding Scheme [17, 19]). Consider relations $\mathcal{R}_1$ and $\mathcal{R}_2$ over public parameters, structure, instance, and witness tuples, A *folding scheme* for $(\mathcal{R}_1, \mathcal{R}_2)$ is defined by PPT algorithms $(\mathcal{G}, \mathcal{P}, \mathcal{V})$ and deterministic $\mathcal{K}$ denoted the

generator, prover, verifier and encoder respectively with the following interface:

- $\mathcal{G}(1^\lambda, N) \to \mathsf{pp}$: on input security parameter λ and size bounds N, samples public parameters pp.
- $\mathcal{K}(\mathsf{pp}, \mathsf{s}) \to (\mathsf{pk}, \mathsf{vk})$: on input pp, and structure s among the instances to be folded, outputs a prover key pk and a verifier key vk.
- $\mathcal{P}(\mathsf{pk}, (\mathsf{U}, \mathsf{W}), (\mathsf{u}, \mathsf{w})) \to (\mathsf{U}', \mathsf{W}')$: on input an instance U in $\mathcal{R}_1$ and instance u in $\mathcal{R}_2$ and corresponding witness vectors W and w outputs a folded instance-witness pair $(\mathsf{U}', \mathsf{W}')$ in $\mathcal{R}_1$ with structure s_1.
- $\mathcal{V}(\mathsf{vk}, (\mathsf{U}, \mathsf{u})) \to \mathsf{U}'$: on input a vector of instances U and a vector of instances u outputs a new instance U'.

Let $\langle \mathcal{P}, \mathcal{V} \rangle$ denote the interaction between $\mathcal{P}$ and $\mathcal{V}$. We treat $\langle \mathcal{P}, \mathcal{V} \rangle$ as a function that takes as input $((\mathsf{pk}, \mathsf{vk}), (\mathsf{U}, \mathsf{W}), (\mathsf{u}, \mathsf{w}))$ and runs the interaction on prover input $(\mathsf{pk}, (\mathsf{U}, \mathsf{W}), (\mathsf{u}, \mathsf{w}))$ and verifier input $(\mathsf{vk}, (\mathsf{U}, \mathsf{u}))$. At the end of interaction $\langle \mathcal{P}, \mathcal{V} \rangle$ outputs $(\mathsf{U}', \mathsf{W}')$ where U' is the verifier's output folded instance, and W' is the prover's output folded witness.

A folding scheme for $(\mathcal{R}_1, \mathcal{R}_2)$ satisfies the following requirements.

1) Perfect Completeness: For all PPT adversaries $\mathcal{A}$, given $\mathsf{pp} \leftarrow \mathcal{G}(1^\lambda, N)$, $(\mathsf{s}, (\mathsf{U}, \mathsf{u}), (\mathsf{W}, \mathsf{w})) \leftarrow \mathcal{A}(\mathsf{pp})$, $(\mathsf{pp}, \mathsf{s}, \mathsf{U}, \mathsf{W}) \in \mathcal{R}_1$, $(\mathsf{pp}, \mathsf{s}, \mathsf{u}, \mathsf{w}) \in \mathcal{R}_2$, $(\mathsf{pk}, \mathsf{vk}) \leftarrow \mathcal{K}(\mathsf{pp}, \mathsf{s})$, and $(\mathsf{U}', \mathsf{W}') \leftarrow \langle \mathcal{P}, \mathcal{V} \rangle((\mathsf{pk}, \mathsf{vk}), (\mathsf{U}, \mathsf{u}), (\mathsf{W}, \mathsf{w}))$ we have that $(\mathsf{pp}, \mathsf{s}, \mathsf{U}', \mathsf{W}') \in \mathcal{R}_1$.
2) Knowledge Soundness: For any expected polynomial-time adversaries $\mathcal{A}$ and $\mathcal{P}^*$ suppose that given $\mathsf{pp} \leftarrow \mathcal{G}(1^\lambda, N)$, $(\mathsf{s}, (\mathsf{U}, \mathsf{u}), \mathsf{st}) \leftarrow \mathcal{A}(\mathsf{pp}, \mathsf{r})$, $(\mathsf{pk}, \mathsf{vk}) \leftarrow \mathcal{K}(\mathsf{pp}, \mathsf{s})$, and $\mathsf{U}', \mathsf{W}' \leftarrow \langle \mathcal{P}^*, \mathcal{V} \rangle((\mathsf{pk}, \mathsf{vk}), (\mathsf{U}, \mathsf{u}), \mathsf{st})$ that $(\mathsf{pp}, \mathsf{s}, \mathsf{U}', \mathsf{W}') \in \mathcal{R}_1$ with probability ϵ. Then, there is an expected polynomial-time extractor $\mathcal{E}$ such that for $\mathsf{pp} \leftarrow \mathcal{G}(1^\lambda, N)$, $(\mathsf{s}, (\mathsf{U}, \mathsf{u}), \mathsf{st}) \leftarrow \mathcal{A}(\mathsf{pp}, \mathsf{r})$, and $(\mathsf{W}, \mathsf{w}) \leftarrow \mathcal{E}(\mathsf{pp}, \mathsf{r})$ we have that $(\mathsf{pp}, \mathsf{s}, \mathsf{U}, \mathsf{W}) \in \mathcal{R}_1$, and $(\mathsf{pp}, \mathsf{s}, \mathsf{u}, \mathsf{w}) \in \mathcal{R}_2$ with probability ϵ.

Definition 6 (Succinctness). A folding scheme is succinct if the communication complexity and verifier time complexity is at most poly-logarithmic in the size of the structures and witnesses.

Definition 7 (Non-interactivity). A folding scheme $(\mathcal{G}, \mathcal{K}, \mathcal{P}, \mathcal{V})$ is *non-interactive* if the interaction between $\mathcal{P}$ and $\mathcal{V}$ consists of a single message from $\mathcal{P}$ to $\mathcal{V}$. This single message is denoted as $\mathcal{P}$'s output and as $\mathcal{V}$'s input.

Definition 8 (Committed relation). Consider a relation $\mathcal{R}$ over structure, instance, witness tuples where witnesses are in some space W. Consider a commitment scheme $\mathsf{com} = (\mathcal{G}, \mathsf{Commit})$ over message space W. Then a public parameter, structure, instance, witness tuple $(\mathsf{pp}_{\mathsf{com}}, \mathsf{s}, (C, u), (w, r))$ is in $\mathcal{R}(\mathsf{com})$ if $(\mathsf{s}, u, w) \in \mathcal{R}$, and $C = \mathsf{Commit}(\mathsf{pp}_{\mathsf{com}}, w, r)$. We say relation $\mathcal{R}$ is the underlying relation for committed relation $\mathcal{R}(\mathsf{com})$.

Definition 9 (IVC-compatible folding scheme). Consider a relation $\mathcal{R}_1$, and a committed relation $\mathcal{R}_2$ over an underlying relation $\mathcal{R}_2'$. A succinct, non-interactive folding scheme $(\mathcal{G}, \mathcal{K}, \mathcal{P}, \mathcal{V})$ with deterministic $\mathcal{V}$ for $(\mathcal{R}_1, \mathcal{R}_2)$ is IVC-compatible if it satisfies the following properties.

1) NP-completeness: There exists a deterministic polynomial-time efficiently invertible function enc such that for any arithmetic circuit F, input x, non-deterministic input w, and output y, for structure-instance-witness tuple $(\mathsf{s}, \mathsf{u}, \mathsf{w}) \leftarrow \mathsf{enc}(F, (x, y), w)$ we have that $(\mathsf{s}, \mathsf{u}, \mathsf{w}) \in \mathcal{R}_2'$ if and only if $F(x, w) = y$.
2) Partial functions: There exists deterministic, efficiently-invertible polynomial-time functions $\mathsf{enc}_{\mathsf{str}}$ and $\mathsf{enc}_{\mathsf{inst}}$ such that for any arithmetic circuit F, input x, non-deterministic input w, and output y, for $\mathcal{R}_1'$ and $\mathcal{R}_2'$ structures $(\mathsf{s}_1, \mathsf{s}_2) \leftarrow \mathsf{enc}_{\mathsf{str}}(F)$ and $\mathcal{R}_2'$ instance $\mathsf{u} \leftarrow \mathsf{enc}_{\mathsf{inst}}((x, y))$ we have that $(\mathsf{s}_2, \mathsf{u}, \mathsf{w}) = \mathsf{enc}(F, (x, y), w)$ for some $\mathcal{R}_2'$ witness w.
3) Monotonicity: For arithmetic circuits F and G, given $|F| \leq |G|$ we have that $|\mathsf{enc}_{\mathsf{str}}(F)| \leq |\mathsf{enc}_{\mathsf{str}}(G)|$. The term $|F|$ denotes the total number of gates in F and the term $|\mathsf{enc}_{\mathsf{str}}(F)|$ denotes the total number of constraints in $\mathsf{enc}_{\mathsf{str}}(F)$.
4) Default instances: There exists $(\mathsf{u}_\perp, \mathsf{w}_\perp)$ such that for any public parameters pp and structure s, we have that $(\mathsf{pp}, \mathsf{s}, \mathsf{u}_\perp, \mathsf{w}_\perp) \in \mathcal{R}_1$.

Lemma 1 (IVC-Compatible Folding Scheme). *[19, Construction 4] made non-interactive via the Fiat-Shamir transformation [37] is an IVC-compatible folding scheme.*

B.2. Iteration-Hiding IVC

Given IVC-compatible folding schemes we now provide a formal construction of our IVC scheme.

Construction 3 (Iteration-Hiding IVC). Consider a relation $\mathcal{R}_1$ and a committed relation $\mathcal{R}_2$ for a commitment scheme $(\mathsf{Commit}, \mathsf{Gen})$. Let NIFS be an IVC-compatible non-interactive folding scheme for $(\mathcal{R}_1, \mathcal{R}_2)$. Let $(\mathsf{u}_\perp, \mathsf{w}_\perp)$ be a trivially satisfiable instance-witness pair for $\mathcal{R}_1$. As guaranteed by the IVC-compatibility property, let enc denote an invertible function that encodes a claim regarding the correct execution of a function as a corresponding instance-witness pair in $\mathcal{R}_2$. Let $\mathsf{enc}_{\mathsf{str}}$ and $\mathsf{enc}_{\mathsf{inst}}$ be the partial invertible functions which map the function description and inputs/outputs respectively. Let F' denote the augmented polynomial-time function as described in Construction 1. We define the IVC scheme $(\mathcal{G}, \mathcal{K}, \mathcal{P}, \mathcal{V})$ in Figures 14, 15, 16, and 17 respectively.

$\mathcal{G}(1^\lambda, \ell)$:
1: Output $\mathsf{pp} \leftarrow \mathsf{NIFS}.\mathcal{G}(1^\lambda, \ell)$.

Figure 14: IVC generator algorithm $\mathcal{G}$.

$\underline{\mathcal{K}(\mathsf{pp}, F) \to (\mathsf{pk}, \mathsf{vk}):}$

1: Compute $\mathsf{s} \leftarrow \mathsf{enc}_{\mathsf{str}}(F')$.
2: Compute $(\mathsf{pk}_{\mathsf{fs}}, \mathsf{vk}_{\mathsf{fs}}) \leftarrow \mathsf{NIFS}.\mathcal{K}(\mathsf{pp}, \mathsf{s})$.
3: Compute and output the prover and verifier keys $\mathsf{vk} \leftarrow (\mathsf{pp}, \mathsf{vk}_{\mathsf{fs}}, \mathsf{s})$ and $\mathsf{pk} \leftarrow (F, \mathsf{pk}_{\mathsf{fs}}, \mathsf{vk})$.

Figure 15: IVC encoder algorithm $\mathcal{K}$.

$\underline{\mathcal{P}(\mathsf{pk}, (z_0, z_i, N), (\pi_i, \omega_i)) \to \pi_{i+1}:}$

1: **if** $\pi_i = \perp$:
2: Let $((\mathsf{U}_i, \mathsf{W}_i), (\mathsf{u}_i, \mathsf{w}_i)) \leftarrow ((\mathsf{u}_\perp, \mathsf{w}_\perp), (\mathsf{u}_\perp, \mathsf{w}_\perp))$.
3: Let $i \leftarrow 0$ and $r_i \xleftarrow{\$} \mathbb{F}$.
4: **else**:
5: Let $((\mathsf{U}_i, \mathsf{W}_i), (\mathsf{u}_i, \mathsf{w}_i)) \leftarrow \pi_i$ and parse (i, r_i) from $\mathsf{enc}^{-1}(\mathsf{w}_i)$.
6: Compute

$$(\mathsf{U}_{i+1}, \mathsf{W}_{i+1}, \pi) \leftarrow \mathsf{NIFS}.\mathcal{P}(\mathsf{pk}, (\mathsf{U}_i, \mathsf{W}_i), (\mathsf{u}_i, \mathsf{w}_i)).$$

7: Sample $r_{i+1} \xleftarrow{\$} \mathbb{F}$ and compute the output

$$\mathsf{x} \leftarrow F'(\mathsf{vk}_{\mathsf{fs}}, \mathsf{U}_i, \mathsf{u}_i, (z_0, z_i, N), (\omega_i, i, r_i, r_{i+1}), \pi).$$

8: Compute an instance-witness pair encoding the valid execution of F':

$$(_, \mathsf{u}'_{i+1}, \mathsf{w}_{i+1}) \leftarrow \mathsf{enc}(F', (\perp, \mathsf{x}), (\mathsf{vk}_{\mathsf{fs}}, \mathsf{U}_i, \mathsf{u}_i, (z_0, z_i, N), (\omega_i, i, r_i, r_{i+1}), \pi)).$$

9: Compute the committed instance: $\mathsf{u}_{i+1} \leftarrow (\mathsf{Commit}(\mathsf{pp}, \mathsf{w}_{i+1}), \mathsf{u}'_{i+1})$.
10: Output $\pi_{i+1} \leftarrow ((\mathsf{U}_{i+1}, \mathsf{W}_{i+1}), (\mathsf{u}_{i+1}, \mathsf{w}_{i+1}))$

Figure 16: IVC prover algorithm $\mathcal{P}$.

$\underline{\mathcal{V}(\mathsf{vk}, (z_0, z_i, N), \pi_i) \to \{0, 1\}:}$

1: **if** $\pi_i = \perp$:
2: Output 1 if $z_i = z_0$ and 0 otherwise.
3: Parse π_i as $((\mathsf{U}_i, \mathsf{W}_i), (\mathsf{u}_i, \mathsf{w}_i))$.
4: Parse u_i as (C, u'_i). Check that

$$\mathsf{hash}(\mathsf{vk}_{\mathsf{fs}}, z_0, z_i, \mathsf{U}_i, N), _ = \mathsf{enc}^{-1}_{\mathsf{inst}}(\mathsf{u}'_i).$$

5: Check that $(\mathsf{pp}, \mathsf{s}, \mathsf{U}_i, \mathsf{W}_i) \in \mathcal{R}_1$ and $(\mathsf{pp}, \mathsf{s}, \mathsf{u}_i, \mathsf{w}_i) \in \mathcal{R}_2$.

Figure 17: IVC verifier algorithm $\mathcal{V}$.