# zk-promises: Making Zero-Knowledge Objects Accept the Call for Banning and Reputation

Maurice Shih University of Maryland maurices@umd.edu

Harikesh Kailad University of Maryland harikeshkailad@gmail.com Michael Rosenberg University of Maryland micro@umd.edu

> Ian Miers University of Maryland imiers@umd.edu

#### Abstract

Privacy preserving systems often need to allow anonymity while requiring accountability. For anonymous clients, depending on application, this may mean banning/revoking their accounts, docking their reputation, or updating their state in some complex access control scheme. Frequently, these operations happen asynchronously when some violation, e.g., a forum post, is found well after the offending action occurred. Malicious clients, naturally, wish to evade this *asynchronous negative feedback*. Considering privacy-preserving analogues of modern access control and reputation schemes raises a more fundamental technical challenge with far broader applications: how do we allow multiple parties to interact with private state stored by an anonymous client while ensuring state integrity and supporting oblivious updates?

We propose zk-promises, a framework which supports Turing-complete state machines with arbitrary asynchronous callbacks. In zk-promises, client state is stored in a zk-object. Updates to the zk-object, represented as a cryptographic commitment to the new, modified object, require a zkSNARK that ensures integrity and atomicity while providing confidentiality. Clients can modify and prove their state by calling valid methods (e.g, to show they are authorized to post) and can give callbacks to third parties (e.g., to later hold them accountable). Through careful protocol design, we ensure clients who advance their state-machine are forced to ingest callbacks that are called by a third party.

zk-promises allows us to build a privacy-preserving account model. State that would normally be stored on a trusted server can be privately outsourced to the client while preserving the server's ability to update the account. To demonstrate the feasibility of our approach, we build an anonymous reputation system with better than state-of-the-art performance and features, supporting asynchronous reputation updates, banning, and reputation-dependent rate limiting to better protect against Sybil attacks.

### 1 Introduction

Balancing anonymity and accountability is a common theme in privacy-preserving systems. Depending on the application, accountability might involve "simply" banning an anonymous account, or it could require a complex update in a stateful access control system. The challenge is that accountability frequently happens asynchronously—often there is an arbitrary delay between action and consequence—and malicious clients may, naturally, try to avoid the negative consequence. For example, in a privacy-preserving version of Wikipedia, an anonymous vandal may try to evade a server forcing them to transition into a banned state when their malicious edit is later discovered by an administrator. Since the client's identity and current status are unknown, even the simple case of this problem, where the state is limited to a single "isBanned" bit, is challenging—we don't know which client's state to update.

The difficulty of updating anonymous clients to hold them accountable, however, extends beyond banning and raises a more fundamental question: how can multiple parties interact with a private state machine stored by an unknown entity without compromising the state or leaking access patterns? While solutions exist for the simple case of banning (e.g., [TAKS10, TAKS08, RMM22]) they are concretely inefficient and do not provide a solution to the more general problem. The challenge of third party updates to private state and asynchronous **negative feedback.** In our setting, some secret state is stored by an untrusted client. Our goal is to let third parties update this state while ensuring (1) state confidentiality, i.e., that the third party learns nothing about the state; (2) obliviousness in update and access, i.e., the third party cannot identify the user by linking accesses or updates; and (3) integrity, i.e., that updates conform to validation logic and access control. The core challenge in this setting is that state updates are asynchronous—the client is not guaranteed to be online when the state update is made. In addition, depending on application, users may have ample motive and opportunity to evade an update. We refer to this as the asynchronous negative feedback problem.

Returning to our Wikipedia example: a client who makes an edit must show that their account is in a valid state in order to authenticate. Various existing anonymous credential systems support this. The problem comes when we need to update this state—e.g., when moderators score the quality of a Wikipedia edit or ban the user's account—after the user's session has ended. Revocation systems for anonymous credentials [CL01, BCC04] are insufficient: we do not know the identity to revoke. And there is no direct way to update the user's state, since the user is anonymous and the location of the state is thus unknown. Finally, there is no incentive for the user to accept an indirect update, e.g., some signed statement posted on a server. This is in contrast to, e.g., an anonymous payment system, where users are financially incentivized to receive updates that increase their account balance.

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

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

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

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

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

Unfortunately, PEREA and successors [AK12, XF14], have substantial performance and functionality problems. Users are effectively stateless: a user's only private state is a fixed pseudorandom function key for generating tickets. Each authentication requires that a user regenerate a fixed k tickets from a global sliding *detection window* of size w and prove those tickets are not in a ban list.<sup>1</sup> This has two consequences:

1. Limited functionality. PEREA only supports checking if there is a ticket in the window. In subsequent work [AK12, XF14], tickets contain a single integer rating, users prove the sum of tickets in the window is below a

<sup>&</sup>lt;sup>1</sup>Even with a zero-knowledge proof which hides how many tickets the user checked something PEREA and derivatives lack—there is no way to trust the user is honestly counting how many tickets it has open.

threshold. But this does not provide the complex, programmable logic or state available in deployed moderation systems [ets].

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

A starting point: zk-objects. We start, somewhat surprisingly, by repurposing techniques developed for privacy-preserving cryptocurrency payments and smart contracts. Starting with Zerocash [BCG<sup>+</sup>14] and commercial derivatives like Zcash, TornadoCash, and Railgun [HBHW, PSS, rai], through to academic [KMS<sup>+</sup>16, BCG<sup>+</sup>20, XCZ<sup>+</sup>22] and commercial systems [Wil, ale] on privacy-preserving smart contract systems, there is a robust line of work which builds efficient protocols for privately manipulating Turing-complete state machines. Although designed for blockchains, these systems can just as easily operate in a centralized setting with a trusted server. We give a new view of this style of computation, which we refer to as the *zk-object model*.

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

**Our contribution.** We design, implement, and benchmark zk-promises, a protocol that augments the zk-object model with asynchronous oblivious callbacks. Conceptually, our approach is simple: we give zk-objects access to a public bulletin board that supports efficient membership and non-membership checks from inside the program. The bulletin board maps psuedorandom tickets to callbacks, complete with encrypted method arguments, posted by third parties. Through careful protocol design, we force programs (in the zk-object model) to track their tickets in local state, check if tickets are mapped to posted callbacks, and executed the callbacks to update their state if needed. With zk-promises, we then build an example application for programmable anonymous blocking and moderation, complete with user reputation and more complex functionality such as reputation-dependent rate limiting.

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

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

- 1. supports Turing-complete state machines and programmable logic for zk-objects with arbitrary callbacks;
- 2. adds asynchronous callbacks to the line of privacy-preserving payment and smart contract systems developed in [BCG<sup>+</sup>14, KMS<sup>+</sup>16, BCG<sup>+</sup>20];
- 3. yields an anonymous reputation and blocking system with significant performance and feature improvements over the state of the art, such as multi-dimensional reputation with support for arbitrary state-dependent logic such as probationary periods and reputation-dependent rate limiting for improved Sybil resistance; and
- 4. is concretely efficient, offering server verification times of less than 3ms and client-side authentication times of 1–10s in realistic scenarios.

### 1.1 Related work

There is a very large body of reputation and anonymous blocklisting schemes. For related work, we constrain ourselves to discussing systems that have anonymous clients (as opposed to relying on pseudonyms); do not need trusted third parties (TTPs) to anonymize, de-anonymize, or ban users; and allow asynchronous and *negative* feedback. For a broader view, we refer readers to the excellent overview by Henry and Goldberg [HG11] for blocklisting and Gurtler and Goldberg [GG21] for reputation.

 $<sup>^{2}</sup>$ Naïvely using a zkSNARK for every such statement is insufficient: the verifier learns the length of the statement (and the statement itself), exposing how many tickets the user has.

 $<sup>^3\</sup>mathrm{The}$  client may evade calls and the caller may, e.g., provide a garbage callback to deadlock the client.

We use the same definition of anonymity in the terminology of Gurtler and Goldberg's taxonomy for reputation systems, "Reputation-Usage Unlinkability" for Votee Privacy—the entity being rated remains private and unlinkable when being rated or when showing their rating. This property excludes most of the reputation systems Gurtler and Goldberg identify, including all of the secure multiparty computation (SMC) approaches. If we also exclude systems that require TTPs or only support positive feedback, we are left with the following approaches which all come from the literature on anonymous blocklisting.

A general approach for TTP-free anonymous blocking is to use pseudorandom tickets that clients provide, e.g., per Wikipedia edit, and a server-side blocklist  $\mathcal{L}$  of banned tickets (e.g. corresponding to vandalistic edits). Because clients generate pseudorandom tickets, they are anonymous and unlinkable across multiple edits without relying on TTPs. Accountability comes by enabling clients to prove their tickets are not on the blocklist  $\mathcal{L}$ . These schemes differ in how this proof is done and the protocols around it.

In token-based blocklisting schemes in the line of BLAC [TAKS10, AKS12, RMM22], each authentication requires the client prove they are not in the blocklist. They iterate over the entire blocklist, proving each ticket was not derived from their PRF key. This requires  $O(|\mathcal{L}|)$  work per client per authentication.

As an improvement, the BLAC authors [TAKS08] and subsequent work [AK12, XF14, MC23], have each client keep a distinct queue Q of tickets they used, requiring they prove  $\forall q_i \in Q, q_i \notin \mathcal{L}$ . As we described in depth earlier, this introduces some major limitations. These schemes are limited by the fact that client state, i.e., the queue, and proofs about it, must be small. In general, the state cannot be mutated except by adding or removing entries for the queue. As Ma and Chow [MC23] point out, this approach requires either severe rate limiting (as every client does worst case work) or global halting until a misbehavior is adjudicated. Ma and Chow propose an elegant mitigation, offering multiple queues, where adjudication can happen in parallel. But this is a workaround, not a full resolution.

Finally, a common shortcoming of all of the above schemes is lack of arbitrary client state and update logic. These systems support simple banning and reputation as a linear function over integer ratings in a finite-sized queue. This is insufficient to support, e.g., a probationary period for new accounts, rate limiting beyond simple counters per epoch, or any kind of complex reputation metric, rating, or weighting.

We note a separate area of work on revocation for anonymous credential systems (e.g. [CL01, BL07]). While sometimes drawing on similar techniques, such as membership and non-membership proofs, these schemes address a simpler problem by assuming the revoking authority knows either the public key or private key of a party. They do not address the problem of updating an unknown user.

# 2 Overview

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

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

Figure 1 gives pseudocode for our example AnonUserRecord. It stores both a multidimensional reputation score (e.g., containing separate scores for spam versus hateful content), state for a leaky-bucket-style rate limit on edits per day, and metadata to support callbacks. When making an edit to Wikipedia, users will prove they called ShowAuthMakeCB and provide the server with the generated callbacks. The server can later use these two callbacks to either update their reputation or ban them. Executing this method also proves to the server the following authorization checks passed:

- 1. The user's reputation, defined as the dot product of the their reputation with a public weight vector—their *projected reputation*—exceeds a specified threshold.
- 2. The user is within a rate limit for edits. To demonstrate the flexibility of zk-promises, we use a leaky bucket for the rate limit where the rate depends on the user's reputation.<sup>4</sup>
- 3. The user has recently scanned for all their open callbacks.

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

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

### 2.1 Callbacks and their lifecycle

The core of zk-promises is a *callback*, a function which is used to modify the user record (here, **ban** and **updateRep**). Callbacks are first-class objects that

 $<sup>^4\</sup>mathrm{Edits}$  require the bucket has remaining capacity. Once a user's bucket is full, they must wait for the bucket to partially drain.

```
class AnonUserRecord:
   /Attributes are accessible via methods only
  private const instanceID
  private authorizedCaller // Pubkey of owner
  private cbMgr // For enforcing issued callbacks
     Reputation can be a multidimensional vector of ints
  private rep = (0, 0, 0) // E.g. (hate, spam, quality) // We define a leaky bucket for rate limiting
  private leakyBucket = 0
  private lastPost = currentTime() // Used for leak rate
  AnonUserRecord(wikiPk): // Constructor
instanceID = randomUID()
    this.cbMgr = CallbackManager(this)
    authorizedCaller = wikiPk // for access control.
     Shows the user is allowed to edit and sends
     callbacks to hold them accountable for that edit. As
     arguments are public, servers can dynamically change
     what weights, thresholds or cutoffs the client must
   / use.
  public ShowAuthorizedForEditandMakeCallbacks(
    curTime, repWeights, repThreshold, rateTreshold,
    cutOffTime
  ):
    this.leakyBucket -= this.calcDrain(curTime)
    // Authorization checks for edits.Checks reputation
// is sufficient, edit is within rate limit, and that
// all callbacks were handled as of cutOffTime.
    if this.rep.dotProd(repWeights) > repThreshold
     and this.leakyBucket < rateTreshold
     and this.cbMgr.lastFullScanTime >= cutOffTime:
      updateCb = this.cbMgr.makeCb(this.updateRep)
      banCb = this.cbMgr.makeCb(this.ban)
      this.leakyBucket++
      this.lastPost = curTime
      return (updateCb, banCb)
    else:
      return null // user isn't authorized
  {\bf public} \ {\rm updateRep} \left( \ {\rm caller} \ , \ \ {\rm delta} \right):
       Users cannot update their own rep
    if caller == authorizedCaller:
       this.rep += delta
  private calcDrain(currentTime):
    if |\,rep\,|\,> 10 // threshold arbitrarily set to 10 |\,leakRate\,=\,10/86400 // 10 auths per day in seconds
    else: // lower reputation gets 1 edit per day
leakRate = 1/86400 // 1 auth per day in seconds
    elapsedTime = curTime - this.lastPost
        annot drain more than current bucket capacity
    return min(leakRate * elapsedTime, this.leakyBucket)
```

Figure 1: Pseudocode for an anonymous user record with reputation, bans, and rate limiting.

```
class CallbackManager:
                                        / Ptr to AnonForumUser
  private userObj
                                      // List of callbacks
// cbList iterator
  private cbList = []
  private curCbIter \stackrel{!}{=} 0
 private scanStartTime = 0 // Current scan start
private lastFullScanTime = 0 // Start of last full scan
 CallbackManager(userObj) // Constructor
    // Bind this cbManager to object
    this.userObj = userObj
      / Initialize the iterator
    this.curCbIter = this.cbList.begin()
  // Creates a new callback. Cannot run if
// currently in the process of settling
private makeCb(func):
    // This means we're not settling, ie not in a scan
assert this.curCbIter == this.cbList.begin()
    ticket = randBytes(32)
    this.cbList.append(ticket, func)
    return ticket
  // Shows all callbacks were handled as of some time.
public showSettledUpTo(cutoffTime):
    {\bf return} this.lastFullScanTime >= cutoffTime
      Incrementally scans the list of open callbacks
     If a callback is on the bulletin board, its settled
   ^{\prime\prime} / If one is expired, its removed from the open list
  public scanIncremental(bulletin, curTime):
    if this.curCbIter == this.cbList.begin():
       // If starting a new scan, mark the start time
       this.scanStartTime = curTime
    // Take the next callback and see if its been called
     (ticket, func) = *this.curCbIter
    call = bulletin [ticket]
      /If the call is on the bulletin board, execute it
     if call not null:
       (caller, cbArgs) = call
       // Mark the callback for deletion and move to next
       this.curCbIter.deleteAndIncr()
    else:
          Move to next item in callback list
       this.curCbIter.incr()
    if this.curCbIter == this.cbList.end():
          We have a new full complete scan, update times
       this.lastFullScanTime = this.scanStartTime
       this.curCbIter = this.cbList.begin()
```

Figure 2: Pseudocode for the callback manager, responsible for creating and settling callbacks.

can be passed around, transferred over the network, and stored, e.g., alongside an edit that is queued for moderation. Conceptually, they have a source that is making the callback (in our case Wikipedia moderators) and destination (the anonymous user).

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

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

Callbacks consist of pseudorandom tickets, which must be tracked to prevent evasion of asynchronous negative feedback. In zk-promises, we build a separate zk-object, called a *callback manager*, to create, track, and handle callbacks. As shown in Figure 2, it keeps a list cbList of all created callbacks. In order to get up to date, a user repeatedly calls scanIncremental, incrementally iterating over every callback on the list to check if it has been called or expired. When one full iteration of that loop completes, lastFullScanTime is set to the start time of the current loop. This implies a polling-like model where users assert they have scanned through all open callbacks as of some time tracked by the callback manager.

In our example, callbacks are represented by a random 32-byte *ticket* tik created by the user's zk-object invoking makeCb in ShowAuthMakeCB. The server, Wikipedia in our example, later calls the callback by placing the ticket on a bulletin board along with the method arguments. For example, a moderator, after reviewing the post, could decrement the anonymous users reputation by 3 via posting (tik, -3) to the bulletin board. The callback would then be ingested when the user scans through its open callbacks and is forced to run updateRep(-3).<sup>5</sup>

### 2.2 Security properties

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

- **Confidentiality** An object's contents are only directly visible to the owner. Function callers may deduce some amount of information solely based on the call they make on the object.
- **Obliviousness** An object update does not reveal which object was updated. Nor can updates on the same object be linked to each other.

 $<sup>^5</sup> This$  must happen by the next time the user calls <code>ShowAuthMakeCB</code>, as that code requires <code>this.cbMgr.lastFullScanTime</code>  $\geq \texttt{cutOffTime}$ 

Setup(serviceProviders)	IncrTime from ${\cal A}$
1: authorizedCallers := serviceProviders	1: curTime $+= 1$
2: curTime $\coloneqq 0$	Call(tile arro) from D
3 : userObjs := {} // Default value is app-specific	Call(Lik, args) from $P_j$
4: createdCbs := {} // Callbacks created but uncalled	1: assert $P_j \in authorized Callers$
5: pendingCbs := {} // Callbacks called but pending ingestion and execution	$_2:  (\mathcal{P}_i, \mathcal{P}', meth, aux) := createdCbs[tik]$
	3: assert $P' = P_j$
ExecAndMakeCb $\begin{pmatrix} \mathcal{P}_j, \text{sid}, \text{creationMeth}, \text{cbMeth}, \\ \mathcal{P}_i \end{pmatrix}$ from $\mathcal{P}_i$	4 : delete createdCbs[tik]
aux <sub>pub</sub> , aux <sub>priv</sub> , cutorr i me	5 : pendingCbs[tik] := $(P_i, meth, args, aux, curTime)$
1: tik $\leftarrow A$	6: send ("Called", tik, args, curTime) to $\mathcal{P}_i$
2 : assert ∄createdCbs[tik] and ∄pendingCbs[tik]	
3: assert $\forall (\mathcal{P}_i, callTime) \in pendingCbs$ :	$IngestCall(tik) \text{ from } \mathcal{P}$
4: callTime > cutoffTime // All old calls have been processed	$1:  (\mathcal{P}, meth, args, aux, callTime) := pendingCbs[tik]$
$5: createdCbs[tik] := (\mathcal{P}_i, \mathcal{P}_j, cbMeth, aux_{pub})$	$2: \text{ userObjs}[\mathcal{P}_i].\text{meth}(\text{tik}, \text{args}, \text{curTime}, \text{callTime}, \text{aux})$
$6: userObjs[\mathcal{P}_i].creationMeth($	3: delete pendingCbs[tik])
7 : tik, curTime, aux <sub>pub</sub> , aux <sub>priv</sub> , cutoffTime)	
8: send ("CreatedCb", sid, tik, cbMeth, $aux_{pub}$ , cutoffTime) to $\mathcal{P}_j$	

Figure 3: A simple ideal functionality  $\mathcal{F}_{zkpr}$  for zk-promises.  $\mathcal{A}$  is the adversary, who determines the structure of tickets and passage of time. The functionality returns  $\perp$  if any table lookup fails.

- **Integrity** An object is only updated according to its programmed methods, and callbacks must be applied. Authorized entities may exclusively make changes, and only accordance to the programmed methods.
- Atomicity There is one valid version of an object at a time and it cannot be rolled back (i.e., no forking or double spending). The act of state transition consumes the prior state.

Formally, we capture these properties in an ideal functionality for a zk-callback system in Figure 3. This ideal functionality describes a generic system for manipulating objects via method calls that can produce callbacks. The callbacks are later called and, subsequently, ingested and applied to the underlying object. When callbacks are called, they cannot be ignored and must eventually be ingested.

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

### **3** Notation and cryptographic preliminaries

### 3.1 General notation

We write  $x \coloneqq z$  to denote variable assignment.  $y \coloneqq \mathsf{A}(x; r)$  denotes the execution of a probabilistic algorithm  $\mathsf{A}$  on input x, using randomness r. We write  $y \leftarrow S$ to denote sampling uniformly from a set S and  $y \leftarrow \mathsf{Alg}(x)$  to denote sampling  $r \leftarrow \{0, 1\}^*$  and assigning  $y \coloneqq \mathsf{Alg}(x; r)$ . The security parameter of our system is denoted by  $\lambda$ .

### 3.2 Cryptographic primitives

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

- **Setup**(P)  $\rightarrow$  (srs<sub>P</sub>, srs<sub>V</sub>,  $\tau$ ) Receives a description of the predicate P and returns two structured reference strings, one used for proving, and one for verifying. Setup also produces a trapdoor  $\tau$  used for simulating proofs (only used in establishing the zero-knowledge property).
- **Prove**( $\operatorname{srs}_P, x, w$ )  $\rightarrow \pi$  Computes a proof that  $(x, w) \in R$ .
- **Verify** $(srs_V, x, \pi)$  Verifies  $\pi$  with respect to x, i.e., verifies that there exists a w such that  $(x, w) \in R$ .

A NIZKPoK is *perfectly correct* if Verify succeeds on every proof that is honestly computed, with an honestly generated srs. A NIZKPoK is *perfectly zero-knowledge* if there exists a simulator  $Sim(\tau, x)$  which, given trapdoor  $\tau$  and instance x, produces proofs which are distinguishable with negligible probability from honest proofs generated with  $Prove(srs_P, x, w)$  for any witness w. Finally, a NIZKPoK has *knowledge soundness* if there exists an efficient extractor which, given access to a prover, can extract the prover's witness. For a more in-depth treatment of zero-knowledge proofs, see [Tha].

We say that a NIZKPoK scheme is a succinct noninteractive argument of knowledge (zkSNARK) if the runtime Verify is  $O(\log |P|)$ , that is, at most logarithmic in the size of the predicate P.

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

1. If pk is a valid public key, then RerandPk(pk)  $\rightarrow$  (pk', r) returns a fresh public key with randomness r such that pk' is computationally indistinguishable from a public key generated with Keygen.

2. If (sk, pk) is a valid keypair and (pk', r) is honestly generated by RerandPk(pk), then RerandSk $(sk, r) \rightarrow sk'$  returns a secret key corresponding to pk', i.e., SkToPk(sk') = pk'.

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

#### 3.3 zk-objects

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

**Object.** An object contains arbitrary state, e.g, payment account balances, reputation, ban status, account creation date, etc. In addition, an object contains a *serial number* (or *nullifier* or *nonce*)—a random string<sup>6</sup> which is revealed when the object is updated. By checking if a transaction updating an object contains a serial number that has already been revealed, we prevent stale object states from being replayed.

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

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

**Updating an object.** To update an object commitment obj, a user must submit a new object commitment obj' and a zero-knowledge proof  $\pi$  of the conjunction of the following statements:

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

where  $\Phi$  is a predicate which determines validity of an update (e.g., a method mutating the object) and the public inputs to the ZKP are the serial number sn, the (committed) object obj', and (the root of) the Merkle tree T.

 $<sup>^6 \</sup>rm Whether$  this is directly used as the nonce or materialized through a PRF and some key material depends on implementation.

### 4 Construction of zk-objects with callbacks

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

#### 4.1 Basic system

zk-promises is, at its core, a zk-object system, and thus carries with it all the same requirements (fresh serial numbers, persistent states in a bulletin board, zero-knowledge proofs over user predicates, etc.). In order to not trivially deanonymous the users, we assume that all communication done by the user is through anonymous channels. In this section we augment zk-objects to support callbacks. In doing so, we ensure that all these base requirements are still met.

**Data structures.** There are two globally accessible data structures in zkpromises.  $bb_{obj}$  stores every committed object obj on the bulletin board.  $bb_{cb}$ stores every callback posted to the bulletin board. These structures permit efficient lookup of items inside zero-knowledge proofs. In addition,  $bb_{cb}$  must support efficient non-membership proofs. Non-membership requires the bulletin board operator to commit to the complement of  $bb_{cb}$ . For signature-backed bulletin boards, this requires a *bulletin board rollover*—a full recomputation of the complement set—whenever  $bb_{cb}$  changes.

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

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

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

**Methods.** zk-promises permits deployers to define multiple *methods*, functions which mutate the object, these methods can both create callbacks and be called by them. For example, as we saw in Section 2, it may be desirable for one callback creation method to enforce rate limiting on itself, while another creation method does not.

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

- $\operatorname{Setup}(\Phi) \to \operatorname{pp}$  Takes a representation of zk-promises application-specific predicates and produces public parameters for our zero-knowledge proof scheme. This may be a trusted setup procedure as required by, e.g., the Groth16 zkSNARK [Gro16].
- **ExecMethodAndCreateCallback**(pp, obj,  $pk_{spid}$ , meth, x)  $\rightarrow$  (obj',  $\pi$ , cbData, aux) Invoked by a user with object obj for service provider with public key  $pk_{spid}$ , this executes the specified method and creates a callback ticket tik for the service provider. Additional public input is given in x. The function may modify obj. The resulting object obj', its zero-knowledge proof  $\pi$ , and any additional execution metadata cbData are sent to  $bb_{obj}$ . The same, plus some auxiliary data aux are sent to the service provider. In the most basic system, cbData contains tik and the object's serial number, and aux is empty.
- VerifyCreate(pp,  $sk_{spid}$ ,  $obj', \pi$ , cbData, aux)  $\rightarrow \{0, 1\}$  Invoked by the service provider spid with secret key  $sk_{spid}$ , this verifies  $\pi$  with respect to obj' and cbData, and verifies that these values appear in  $bb_{obj}$ . This also optionally takes an auxiliary input aux.
- $\label{eq:call} \mbox{Call}(pp,tik,args) \mbox{ Invoked by a service provider, calls the callback represented by tik, with callback arguments args. These values are sent to <math display="inline">bb_{cb}.$
- **VerifyCall**(pp, tik, args, aux)  $\rightarrow \{0, 1\}$  Invoked by the manager of bb<sub>cb</sub>, this verifies that the call (tik, args) (with optional caller-provided auxiliary input aux) is well-formed. The notion of well-formedness is application-defined.
- **ScanOne**(pp, obj, x)  $\rightarrow$  (obj',  $\pi$ , cbData) Invoked by a user with object obj, this takes one step in the callback list, checks on bb<sub>cb</sub> if the callback has been called, and, if so, executes it. Additional public input is given in x. The order of iteration through the list is application-specific. The resulting object obj', its zero-knowledge proof  $\pi$ , and the execution's metadata cbData are sent to bb<sub>obj</sub>.
- **VerifyMethodExec**(pp, obj',  $\pi$ , cbData)  $\rightarrow \{0, 1\}$  Invoked by the manager of bb<sub>obj</sub>, this verifies  $\pi$  with respect to obj' and cbData. This is used to verify bulletin board submissions from ExecMethodAndCreateCallback and ScanOne. On verification success, (obj',  $\pi$ , cbData) is posted to bb<sub>obj</sub>.

We now describe each algorithm in detail.

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

We describe the update more formally. Let  $\mathsf{pk}_{\mathsf{spid}}$  represent the public key of the service provider the user intends to give the callback to, and let x represent

any additional public values, including the method  $\mathsf{meth}'$  the user wishes to create a callback for. The user performs the following updates to its objects.

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

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

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

*zk-object bookkeeping:* 

The old object exists.  $obj \in bb_{obj}$ The serial number is revealed. obj.sn = snThe entry has been appended.  $obj'.cbList = obj.cbList||entry^7$ 

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

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

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

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

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

- 1. verifies that  $(obj', \pi, cbData)$  appears on  $bb_{obj}$ , i.e., that the callback was created;
- 2. verifies the proof  $\pi$  with respect to inputs cbData and obj'; and
- 3. extracts tik from cbData or aux, and verifies that it has never received tik in the past

 $<sup>^{7}</sup>$ We omit details for now on how precisely the callback list works. For now, it may be treated as a fixed-size list, where the callback manager retains a running index of unfilled slots. We extend this later in this section.

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

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

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

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

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

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

- 1. Set  $obj' \coloneqq meth(obj, args, x)$
- 2. Delete tik from obj'.cbList
- 3. Set obj'.sn to be a fresh serial number

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

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

The old object exists.  $obj \in bb_{obj}$ 

The serial number is revealed. obj.sn = sn

entry *is the current entry.* entry = obj.cbList.nextUnscanned()

Callback was applied if called. If (tik, args)  $\in$  bb<sub>cb</sub> then  $\Phi_{meth}(obj, obj', entry, args, x) = 1$  and obj'.cbList = obj.cbList.tail()

No-op if not called. If tik  $\notin bb_{cb}$ , then obj' = obj

As in ExecMethodAndCreateCallback, the user sends (obj',  $\pi$ , cbData) to the object bulletin bb<sub>obj</sub>. We note that the last two statements are the reason we require efficient zero-knowledge proofs of membership and non-membership in bb<sub>cb</sub>.

In practice, ScanOne may be batched. A user may process 100 callbacks at a time, and produce a single zero-knowledge proof that all 100 were applied correctly.

VerifyMethodExec(pp, obj',  $\pi$ , cbData)  $\rightarrow$  {0,1}. The object bulletin board must handle zk-object updates from ExecMethodAndCreateCallback and ScanOne.

To do so, it does the same proof verification as VerifyCreate. Specifically, given  $(\pi, cbData, obj')$  it verifies the zero-knowledge proof  $\pi$  with respect to its inputs cbData and obj'. In addition, as with any zk-object update, it must ensure that the serial number is not repeated. That is,  $cbData.sn \notin S$ , where S is the set of all serial numbers observed by the bulletin board operator. If all checks succeed, the operator posts  $(obj', \pi, cbData)$  to  $bb_{obj}$ .

VerifyCall(pp, tik, args, aux)  $\rightarrow$  {0, 1}. The operator of the callback bulletin board must verify incoming calls (tik, args) before placing them on the bulletin board. In our basic construction, aux is empty, and there is nothing that requires verification. Later in this section, we will let aux be a digital signature, and condition acceptance on the signature's successful verification.

### 4.2 Unlinking Create and Call

Currently, a passive observer of the (possibly public) bulletin board sees tik, meaning it can correlate executions of ExecMethodAndCreateCallback and Call. While this does not identify the user, it may, e.g., leak what post was moderated.

To fix this, we slightly modify the zero-knowledge proof and public input of ExecMethodAndCreateCallback. The user replaces entry in cbData with a commitment  $com_{entry} \coloneqq Com(entry; s)$  for some randomness s, and opens the commitment in the proof. In addition, the auxiliary data sent to the service provider is now aux := (entry, s). In VerifyCreate, the service provider checks that  $com_{entry} = Com(entry; s)$ .

The Call algorithm is unchanged. Since tik is not sent in the clear at any time before Call, there is no longer any event to link it to.

### 4.3 Expiry

Currently, there is no limit on the amount of time that can pass between creation and calling of a callback. This gives service providers the power to rate old and irrelevant posts for any reason. In addition, it requires the user to store all callbacks indefinitely until they are called. Over time, this makes a full scan computationally infeasible.

We solve both of these problems by associating an *expiry* with every callback. With expirable callbacks, users only have to store the tickets that have not expired, and calls are limited to the expiry period, e.g., at most one day after creation. We detail the changes to the base system that this feature entails.

ExecMethodAndCreateCallback now takes an expiry exp as an argument, and includes exp in entry.

In VerifyCall, the bulletin board operator does as before, but additionally stores the time t that the call was received. So each element of  $bb_{cb}$  is now of the form (tik, args, t).

ScanOne now takes as part of its public input x the current time c. When deciding whether to apply entry =  $(tik, meth, exp) \in obj.cbList$ , they do as follows:

1. If tik was posted, i.e.,  $(tik, args, t) \in bb_{cb}$ :

- (a) If  $t < \exp$ , the callback was called in time. The user applies it: obj' := meth(obj, args, x) and deletes this entry from cbList.
- (b) If  $t \ge \exp$ , the callback was called after expiry. The user ignores it obj' := obj and deletes this entry from cbList.
- 2. If tik was not posted, i.e., tik  $\notin bb_{cb}$ :
  - (a) If  $c \ge \exp$ , the callback has already expired. The user ignores it obj' := obj and deletes this entry from cbList.
  - (b) If  $c < \exp$ , the callback is unposted and unexpired. The user ignores it  $obj' \coloneqq obj$  and leaves this entry cbList.

The user's zero-knowledge proof in ScanOne is also updated to reflect the above logic.

#### 4.4 Authenticity and confidentiality for callback inputs

Our construction so far still has the following limitations: (1) callback arguments are sent in the clear in Call, meaning a passive adversary can, e.g., learn correlations between activity and moderation decisions; and (2) function arguments are not authenticated in any way, meaning that a malicious bulletin board provider or an active adversary can modify the callback arguments in a Call payload.

We would like to give service providers the ability to include encrypted, non-malleable arguments in their call. This must satisfy a few constraints simultaneously:

- 1. If a service provider posts a ticket and callback arguments, the arguments cannot be malleable.
- 2. If the arguments are malformed, the user must be able to reject the callback during ScanOne.
- 3. If the arguments are well-formed, the user cannot reject the callback during ScanOne.

We now describe how to add authenticity and confidentiality in a way that meets these goals.

Authenticity: tickets are signature pubkeys. To bind tik to args, we interpret tik as a public key for a signature scheme. Surprisingly, this requires few modifications to our protocol overall, and no modifications to our zero-knowledge proofs.

Let  $\Sigma$  represent an EUF-CMA-secure pubkey-rerandomizable signature scheme. We assume a public-key infrastructure for all service providers. That is, every service provider has an associated keypair ( $pk_{spid}, sk_{spid}$ ), and a user, given spid, can discover  $pk_{spid}$ .

In ExecMethodAndCreateCallback, rather than selecting a random tik, the user computes  $(tik, r) \leftarrow \Sigma$ .RerandPk $(pk_{spid})$  and places r in aux for the service provider.

In VerifyCreate, the service provider computes  $\mathsf{sk}'_{\mathsf{spid}} \coloneqq \Sigma$ .RerandSk $(\mathsf{sk}_{\mathsf{spid}}, r)$ and checks that tik is well-formed, i.e., tik =  $\Sigma$ .SkToPk $(\mathsf{sk}'_{\mathsf{spid}})$ .

In Call, instead of sending (tik, args), the service provider sends (tik, args,  $\sigma$ ), where  $\sigma = \Sigma$ .Sign<sub>sk'snid</sub> (args). This binds the arguments to the ticket.

In VerifyCall, instead of posting (tik, args) unconditionally, it now receives  $aux = \sigma$  and posts the call iff  $\Sigma$ . Verify<sub>tik</sub> $(y, \sigma)$  is true.

Confidentiality: encrypted method augments and in-circuit decryption. To encrypt args, it suffices to have the caller simply compute and post  $\operatorname{args} := \operatorname{Enc}_k(\operatorname{plaintextArgs})$ , where k is an encryption key and Enc is a circuit-friendly CPA-secure encryption algorithm (CCA-security is not necessary, since args is now signed).

We must be careful about how k is determined, though. If we do not properly bind a ticket's associated symmetric encryption key to the ticket itself, a user could intentionally use the wrong key to decrypt arguments to the wrong value. On the other hand, if we require the client to prove the decrypted payload is well-formed, a malicious caller can deadlock a client by giving it an invalid ciphertext.

We thus include a separate encryption key with every entry in the callback list. This forces clients to use the correct key. Because callbacks are authenticated by the ticket holder, clients cannot tamper with the ciphertext. Further, because each ticket is bound to its encryption key, callers cannot use malformed ciphertext to deadlock the client. They can give a ciphertext with an ill-formed decryption, but this will merely result in a failed callback.

ExecMethodAndCreateCallback now generates a fresh encryption key k, and includes k in entry.

In Call, rather than letting args be plaintext, it uses its knowledge of k to compute  $\operatorname{args} = \operatorname{Enc}_k(\operatorname{plaintextArgs})$ .

In ScanOne, the user simply decrypts the payload in-circuit. That is, rather than using args directly, it uses  $Dec_k(args)$ , where k comes from the current entry in obj.cbList.

#### 4.5 Variable-length callback lists

The core of our extension to the zk-object model requires us to track issued callbacks via cbList. We must represent cbList in a way that gives efficient circuit operations for append, remove, and iteration. Naïve approaches do not work here: a large fixed-size array means a full scan requires traversing even the empty slots in the array. And a sparse Merkle tree has less-than-ideal overhead, requiring log n hashes where n is the number of elements of the list, for a total of  $O(n \log n)$  work for iteration. This leaves us, seemingly, needing to resort to expensive mechanisms for constant-overhead zk-memory [FKL<sup>+</sup>21].

We observe that we do not need to support O(1) random-access remove operations on cbList to scan and ingest callbacks. Since users must ensure all called callbacks are ingested, they must traverse the full list at some point. Thus, it suffices to support O(1) removal *amortized*, while traversing a list in order, and O(1) append for new callbacks. We define the representation of a list  $\ell$  as

$$h_{\ell} \coloneqq H(H(H(H(\ell_1), \ell_2), \ell_3) \dots \ell_n)$$

where H is a collision-resistant hash function. This supports O(1) appending, since  $\ell' \coloneqq \ell || x$  implies  $h_{\ell'} = H(h_{\ell}, x)$ , but it does not support efficient removal.

To support O(1) removal when incrementally iterating through the list, we encode list traversal as a state machine in our user object, keeping track of the previous cbList as well as the new version that will be the result of a complete scan. Removal consists of *not* adding the value to the new list. We note we can readily make this incremental logic handle multiple entries in the list at a single time and thus create a batch proof. The complete zero-knowledge relation for variable-length cbList is provided in Appendix B.

#### 4.6 Private and public method identifiers

So far we have not specified whether the method being called in ExecMethodAndCreateCallback or ScanOne is hidden from passive adversaries or not. We describe how to implement either choice, and discuss tradeoffs.

If method IDs are private, then  $\Phi$  is a single predicate that is used for every operation, and takes in meth as an argument. While this provides method privacy, it means that a user performing a ScanOne operation must prove a circuit whose size is the sum of all the method sizes.

Public method IDs can be used to mitigate this. That is, we may let each  $\Phi_{meth}$  be a separate circuit, and have every ExecMethodAndCreateCallback and ScanOne also post meth in the clear to the bulletin board.

Finally, we note is possible to trade off method anonymity with performance by grouping different methods together into the same circuit.

### 5 Security argument

Our construction of zk-promises realizes the ideal functionality  $\mathcal{F}_{zkpr}$  in Figure 3 against computationally bounded adversaries with static corruption of parties. The security of zk-promises rests on the assumption that our commitment scheme is binding and hiding; our signature scheme is pubkey-rerandomizable and EUF-CMA-secure; our encryption scheme is IND-CPA-secure; and our NIZKPoK scheme is knowledge-sound, perfectly correct, and statistically zero-knowledge.

For space reasons, we defer a proof sketch to Appendix A.

# 6 Programming in the zk-object model

Given the basic construction, we now describe some applications and alternative configurations.

Authorization logic. First, because the zk-object model supports Turingcomplete state machines with arbitrary state, we can implement a variety of applications. zk-objects readily represent accounts with stateful access control policies spanning almost any authorization logic. As hinted at in Figure 2, we can program rate limits, where the rate depends on the client's current state. These give much more flexibility than existing techniques which enforce a fixed limit on a counter that resets, e.g., daily, (see, e.g.,  $[CHK^+06]$ ) and applies equally to both new and well-established accounts. Similarly, we can support logic for temporary bans, or a probation policy where an additional infraction, while in a probation state, results in a ban. Finally, because we can build Turing-complete state machines, state itself is unbounded. We could make a zk-object store a client's entire history of actions and defining arbitrary logic on top of that.

We are also not limited to purely anonymous access. zk-objects can allow both anonymous and pseudonymous authentication for different pseudonyms across various domains,<sup>8</sup> where an infraction on one website affects the account backing all of them.

**Non-anonymous callbacks, revocation, and privileges.** Because the callback management mechanism is itself a zk-object, we can also realize non-anonymous callbacks, where the object is anonymous, but callbacks are placed to a fixed identifier.<sup>9</sup> Given accounts issued to a known identity, e.g., an employee, this would allow for updates or revocation (e.g, in the case of promotion or employment termination).

**Callback ordering.** Similarly, we can change the semantics of callbacks themselves. We can force callbacks to be evaluated in the order they are issued. Or in the order they expire. The later would reduce the clients need to scan for open callbacks, at the cost of blocking settlement of subsequent calls. We could, similarly, imagine different tiers of calls with different priorities and scanning interval requirements.

**Callbacks from TEEs, public smart contracts, other zk-objects**. Currently we restrict the source of a callback to a public key, where the bulletin boards a signature over the callback and its arguments under the key. However, other options are possible. For example, the signing key could be controlled by a TEE, guaranteeing that callbacks could only be made by programs under certain conditions. Similarly, we could restrict callbacks to be called by a specific program run by the bulletin board (e.g., a smart contract instance on a blockchain). Finally, we could restrict callbacks to those coming from other zk-objects. Here we would require the bulletin board to, instead of a signature, verify a proof that the call came from the execution of a method on another zk-object. In this case, we could imagine calls are either bound to a class of objects or to a specific instance ID. This would require some form of integrity labeling for calls.

Any reputation system must have a mechanism to prevent users from rating themselves. We note this can be achieved via a strict allow list, whereby  $\Phi$  is hard-coded with a set of signing public keys it will accept calls from, or by

<sup>&</sup>lt;sup>8</sup>Pseudonyms could be stored in the object directly via a mapping from domain to pseudonym, or derived via a PRF evaluation.

 $<sup>^9 \</sup>rm Naively,$  such callbacks would be single-use. To allow multiple calls, we could, e.g., postfix the callback with a counter and force the client to retrieve the next counter value.

	Prove (ms)			Verify (ms)	
Callback Capacity:	Unlimited	$2^{16}$	$2^{32}$	$2^{64}$	-
	Signature-based	Tree-Based		Both	
Show Authorized	337	316	372	558	2.81
Incremental Scan	571	535	638	972	2.81
Show Authorized + Incremental Scan	909	716	829	1123	2.81

Table 1: Runtimes for proving and verifying the zero-knowledge circuits. The signature-based variants correspond to the centralized setting and the tree-based variants correspond to the decentralized setting. Incremental scan includes the cost of applying the callback. However, the verification runtimes are same across both settings.

permissive mutable list, whereby public keys be part of the user record itself and unauthorized calls are rate-limited rather than rejected outright, or by any access control regime in between.

# 7 Experiments

In this section, we describe the implementation and evaluation of the application described in Section 2 built with zk-promises. We build two versions, one for the decentralized and one for the centralized setting.

**Instantiating cryptographic primitives.** We use Groth16 [Gro16] as our zkSNARK and Poseidon [GKR<sup>+</sup>21] for all hash functions. For circuit-friendly encryption, we use key-prefixed Poseidon in counter mode as a stream cipher. For signatures, we implement Schnorr over the Jubjub curve [ZCa19].

**Hardware.** All benchmarks were performed on a desktop computer with a 2021 Intel i9-11900KB CPU with 8 physical cores and 64GiB RAM running Ubuntu 20.04 with kernel 5.15.0-69-generic.

**Code.** zk-promises is written in Rust, using the Arkworks [Ar22] zkSNARK crates and Rayon for parallelization where possible. The Criterion-rs crate was used for all microbenchmarks and statistics.

#### 7.1 Methodology

In zk-promises, method calls are verified by a zero-knowledge proof realized as a circuit representing the logic in the call (e.g., that the account is in good standing to make an edit, that a generated callback was stored for future scanning, or that all pending callbacks were scanned for). We note that the logic for scanning for a callback also includes the logic for ingesting the callback and, to hide which callback was made, includes logic for both calls.

Recall, zk-promises can be built in both a decentralized and centralized setting, depending on how the bulletin board is instantiated. In the decentralized setting, we build a Merkle tree over all callback entries in the bulletin board and require proof to show membership in that tree. In the centralized setting, the bulletin board is operated by a single server that is trusted for integrity but not confidentiality or obliviousness. This allows us to replace Merkle tree membership checks with simple signature checks.

For both settings, we create and benchmark circuits to prove methods for authorization, callback scanning, and a combined single circuit of those two statements.

While the centralized setting offers improved efficiency, it requires on additional step by the server. Periodically, the server must *rebuild* the nonmembership data structure, updating the signed timestamp on each value to prevent use of an expired but still signed non-membership entry.<sup>10</sup> We measure the runtime of this process as well.

Our second construction uses Merkle trees for verifying membership and non-membership. While a Merkleized key-value store could suffice, verifying (non-) membership for this tree in a fixed-size circuit is costly. Each operation incurs worst-case costs; for a naive sparse Merkle tree, it involves hashing a Merkle path of depth d, where d is the bit length of the hashed keys (in our case, 256 bits). Instead, we implement two Merkle trees, one storing tuples of (key, value) and one storing intervals of unused keys. Here, the circuit need only verify a path of length log  $|\mathbf{bb}_{cb}|$ . Updating the trees consists of at most three insertions outside the circuit: one to add a callback, and two to split the unused interval in two.

Finally, we also implement *batched scan*—a circuit which settles more than one callback at a time. This allows the client to condense scans of multiple callbacks into a single proof for the server to verify. As with *scan incremental* this includes (now repeated |batch| times) the logic for executing both callbacks.

#### 7.2 Results

Over all benchmarks, the maximum observed relative standard error of the median was 1.2%.

Table 1 shows the Groth16 proof and verify computation time for the three zero-knowledge proof circuits. One circuit shows the client is authorized, another scans for a single callback (*scan incremental*), and one combines both into a single operation. Appendix D gives a breakdown of circuit sizes. As expected, prover runtime (in the tree-backed setting) increases linearly as the global allowed number of callbacks (and therefore the cost of the Merkle tree membership check) increases. The signature-based setting, which replaces the membership check with a fixed single signature verification, takes constant time.

While Incremental Scan only checks for a single callback, a complete scan must handle all open callbacks, requiring multiple invocations. We constructed

 $<sup>^{10}</sup>$ This is not necessary in the decentralized setting, since proofs are made with respect to an up-to-date root of a sparse Merkle tree, which has efficient membership and non-membership proof algorithms.



Figure 4: Benchmarks for handling multiple callbacks.

circuits for batched scans to handle a fixed number of pending callbacks. Benchmarking results can be found in Figure 4a. As expected, a batched scan is more efficient compared to the same number of single-scans. For example, for the tree height of 32, computation savings range from 20–40% when doing batched settles of between 2 and 32.

The verification times are constant, at 2.8ms for all circuits, as seen in Table 1. Batching Groth16 proof verification gives a throughput of up to 1,194 verifications per second, a  $4 \times$  throughput improvement over individual verifications.

Finally, we plot the rebuilding time for signature-backed callback bulletin boards in Figure 4b. As expected, the runtime grows linearly with the number of callbacks called by the server.

Communication costs for zk-promises are very low, with constant proof sizes and small public inputs. Concretely, Show Authorized uses 608B and 568B for signature-based and tree-based settings, respectively. Similarly, Incremental Scan and batched scans use 396B and 268B, respectively in the Merkle-tree backed setting, and Show Authorized + Incremental Scan use 748B and 708B, respectively.

#### 7.3 Discussion

We now discuss the feasibility of our prototype reputation system in a real-world deployment.

#### 7.3.1 Choice of configuration

The performance of any deployment of zk-promises is determined by four configurable parameters:

- (maximum) broadcast latency —how frequently the bulletin board rebuilds and announces updates,
- *expiry*—how long before callbacks expire,
- (maximum) scan interval—how frequently the client must complete a full scan (i.e, the cutoff that lastFullScanTime must be newer than), and
- batch size—how many callbacks clients scan for at a time.

Let us also define the average *authentication rate* of a user as **avgAuth**. During each scan, the client must process the callbacks that did not expire in the previous scan, and all added callbacks since that previous scan. The total number of callbacks that must be checked is:

 $(avgAuth \cdot expiry) + (avgAuth \cdot scanInterval).$ 

If the batch size is equal to this, the server will only have to verify a single (batched) scan for the average client. Increasing the batch size this way is a tradeoff as each client will now have to do the work proportional to the average number of callbacks a user generates.

If we combine batched scans with Show Authorized into a single circuit, then the server only has the overhead of one zkSNARK verification per authentication for clients whose number of outstanding callbacks is at most the batch size. Each client, in contrast, needs to compute less than 30 seconds of work per authentication, even in very large settings, as shown in Figure 4a.

Decreasing the broadcast latency will increase the liveness of zk-promises, as the smaller this value is, the faster callbacks are announced. The broadcast latency should be set to the smallest value a server can handle and still keep up with computation.

Expiry should be set so that most malicious client actions can be caught before the callback expires. Setting scan interval is a tradeoff in how synchronized the clients are with bulletin board and how much work (scanning) the client must do.

#### 7.3.2 Example scenario: Wikipedia

We now examine a real-world scenario: moderating Wikipedia edits. Wikipedia (across all languages) has an edit rate of 18 edits per second (average rate across April 2023–March 2024 [Wik]). We know from the above experiments that a single lightweight server can support 357 zk-promises verifications per second (or 1,428, using batched verification). Thus, the increased load on the server per authentication is easily handled, even if all edits are anonymous.

The peak month of edits in that time frame (January) for English Wikipedia [Wik]<sup>11</sup> had an average editor make around 239 edits a month, and 90% of edits are reverted within 5hrs [JNV]. Thus we set expiry to 1 day. If we set the scan

 $<sup>^{11}</sup>$ Additional data on edits such as reversion rate are only available for select languages, so we switch to statistics for only English Wikipedia.

interval to 24hrs then the batch size should be set to  $\frac{239}{31} \cdot (1+1) = 16$ , so that the server only has to verify a single proof for the average user. This means the computation required to create the proof for batched scan is less then 10sec, as shown in Figure 4a. We feel client side proving times are reasonable, as proofs can be completed in the background while the client makes an edit.

We note that our numbers are an overestimation of the performance requirements, as most edits are (non-anonymous) automated editors [PCL $^+07$ ]. So the actual rate of anonymous edits for anonymous users is lower.

### 8 Conclusion

In this paper, we define zk-promises, which adds callbacks to the zk-object model. While we have used this to demonstrate the feasibility of an anonymous reputation system, the potential applications are much broader. A long line of work on proof-carrying data [CT10, BCCT12] have explored zero-knowledge incremental computation. Zerocash [BCG<sup>+</sup>14] introduced the idea of replay- and forking-prevention for simple objects (limited to payments). Hawk [KMS<sup>+</sup>16] generalized the class of functions, but each object was isolated and could not interact with other object types. Zexe [BCG<sup>+</sup>20] offered inter-object communication between oblivious objects of different types and, additionally, used recursive proofs to hide which object is called. But up until now, using this programming model in many real applications has been challenging.

To explain, we borrow a distinction from the cryptocurrency literature. Smart contract systems, such as Ethereum, typically operate in the *account model*, where there is a single authoritative state for an account and methods can be called on it. In contrast, in [BCG<sup>+</sup>14, KMS<sup>+</sup>16, BCG<sup>+</sup>20, XCZ<sup>+</sup>22], state is split into multiple locations. This is sometimes referred to as the *UTXO model*. The UTXO model is generally regarded as impractical to work with, and brings with it challenges similar to the asynchronous negative feedback problem we articulate in Section 1.

zk-promises yields a practical account model for privacy-preserving computation, as shown by our example application. Users can store their state in a single account that gets asynchronously updated by other calls. In the zk-promises model, users are responsible for sequencing updates to their object requested by others, but they cannot drop particular update requests. We believe this is useful for many applications, ranging from anonymous access control to privacy-preserving smart contract systems.

A second consequence of this model is the possibility of having zk-objects themselves make callbacks. Currently, in our prototype applications, the caller is identified by a signature, but this can easily be tied to a TEE, a public smart contract, or even another zk-object. This would allow an account model where objects can, programmatically, be controlled by another entity in full or in part even if that entity is not trusted to know the state of the object.

### 9 Extensions and Future Work

We now detail some extensions and directions for future work.

#### 9.1 Additional Functionality

While zk-promises already offers a lot of flexibility in callback creation and execution, we identify some potential future routes of extension.

**Service provider privacy.** zk-promises explicitly does not aim to provide privacy for the party calling the callback. Specifically, zk-promises permits the user to know which service provider called a callback (by simply looking at which ticket was used), as well as the contents of the callback arguments.

These notions of privacy are distinct and are resolved in distinct ways. Caller privacy can be achieved by following the blocklisting mechanism of SnarkBlock and BLAC [RMM22, TAKS10]. We replace token issuance with an interactive protocol between the user and the service provider, whereby the user computes an oblivious pseudorandom function (OPRF) over a nonce provided by the service provider, where the key k is some user-specific key. This (tok, nonce) pair is used during redemption. To sweep, the user must traverse the entire bulletin until it finds a pair such that  $PRF_k(nonce) = tok$ , and then sweep the attached payload. Because this traversal is inherently sequential over (a portion of) the bulletin board, this method can only reasonably be deployed over limited time spans, and not globally.

Redeemer data privacy is a more difficult problem, since the user controls their user record, and is tasked with updating it. One way of achieving this, would be for a user to cede control over their record to a committee of mutually distrusting parties. Then, to sweep a token, the parties would engage in a multiparty computation (MPC) protocol to update the record. Another option is to permit the user record to be updated by a single party using fully homomorphic encryption, and then only rely on MPC to generate zkSNARK for the bulletin.

Lazy callback settlement. Currently, client prover time during ScanOne is linear in the size of cbList, since it must check if each callback has been called or has expired. This cost can be reduced if we loosen the time bounds on when callbacks must apply.

Let cbList be as before, but now sorted by expiry time. This can be achieved by in-circuit sorting with a more complex data structure, or by setting a fixed expiry period for all present and future callbacks. To call ScanOne, the user will now simply iterate through cbList until it finds an expiry time that is in the future.

In this regime, a user only needs to do work linear in the number of expired callbacks. This comes at the cost of immediacy of calls—any Call can take up to the expiry time until it is applied to the user.

**Retention period.** In order to permit users to be offline for arbitrary periods of time, zk-promises requires that bulletin boards keep all state forever. If we

loosen this, permitting users to be offline for, e.g., at most one year, then we may permit bulletin board operators to delete all items older than one year.

This extension only requires one extra assertion in ScanOne: that lastFullScanTime is greater than the current time minus the retention period. This way, a user who doesn't fully settle at least once within the retention period is permanently locked out, and cannot make progress by claiming their callbacks (which may or may not have been called) are not in bb<sub>cb</sub>.

**Rate limiting redemptions.** Currently, redemption is unconditional—the bulletin accepts any payload with a valid signature. There is no need to stop here, though. If, for example, we wish for redeemers to also have credentials, we can add a NIZK to the redemption logic to ensure that. Similarly, if we wish to add rate limiting to token redemptions, we can add that as well, using standard anonymous credential rate limiting methods [CHK<sup>+</sup>06] which are easy to realize using existing anonymous credential schemes using zkSNARKs [RWGM23].

Additional provenance checking of calls. In a decentralized setting, it may be desirable for a smart contract, rather than a specific service provider, invoke Call for some callbacks. To handle this, it suffices to add public auxiliary input to the callback bulletin board. Specifically, rather than storing (tik, args, t) in bb<sub>cb</sub>, the bulletin board manager stores (tik, args, t, addr), where addr is the source address of the Call. In Ethereum, for example, it is not necessary to trust any third party that addr is correct. Similarly, call payloads can be accompanied by an attestation from a trusted execution environment (TEE) that the values were computed correctly.

The changes to ExecMethodAndCreateCallback and ScanOne are similar to the allowlist method in the previous subsection. The user may predetermine the smart contract addresses it is willing to receive calls from, and check in the ScanOne ZKP that this holds.

### 9.2 Instantiating zk-promises with Different Cryptographic Primitives

The approaches in zk-promises are generic to the cryptographic building blocks. This raises the possibility of building new systems based on the approach in zkpromises but using different primitives, or co-designing new primitives together for improved efficiency.

**Zero-knowledge Proof Systems.** While Groth16 was selected in this work, zk-promises is compatible with any zero-knowledge proof system. Alternate systems can be used for additional properties such as trustless setup ([BBHR18, BBB<sup>+</sup>18]) or with universal setup (e.g., [MBKM19, GWC19, CHM<sup>+</sup>20]).

Since cryptographic primitives are used in a black-box manner, creating a post quantum version of zk-promises is possible. For example, Groth16 can be replaced with [AHIV17, BBHR19].

Membership Proofs and Accumulators. Callbacks in zk-promises depend on an efficient membership and non-membership checks. While we implemented this with merkle-trees, other approaches are possible like RSA accumulators [Bd94, LLX07, BBF19], polynomial commitments [KZG10], Verkle trees [Kus], and Reckle trees [PSG<sup>+</sup>24] offer tradeoffs for this such as witness size, witness creation and verification costs, and batching. In particular, batching of membership proofs can reduce the amount of computation required for the client to sweep their open callbacks.

Techniques for batched verifications also exist, such as Zebra [RPX<sup>+</sup>22] for blind signatures and SnarkPack [GMN22] that has logarithmic scaling for batches.

The use of more efficient hash functions in circuits than Poseidon will have concrete improvements for client computation. We could even imagine codesigning a zero-knowledge proof system and membership construction with better efficiency.

**zero-knowledge Virtual Machines.** While this work is done with custom zero-knowledge circuits, zero-knowledge virtual machines (zkVMs) can be utilized instead. Multiple works such as Lasso, Jolt, RISC Zero, and TinyRAM [STW23, AST24, RIS, BSCG<sup>+</sup>20] can be used to convert programs from existing languages such as x86 and RISC-V into zero-knowledge proofs.

# References

- [AHIV17] Scott Ames, Carmit Hazay, Yuval Ishai, and Muthuramakrishnan Venkitasubramaniam. Ligero: Lightweight sublinear arguments without a trusted setup. In Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, ACM CCS 2017, pages 2087–2104. ACM Press, October / November 2017. doi: 10.1145/3133956.3134104.
- [AK12] Man Ho Au and Apu Kapadia. PERM: practical reputation-based blacklisting without TTPS. In Ting Yu, George Danezis, and Virgil D. Gligor, editors, ACM CCS 2012, pages 929–940. ACM Press, October 2012. doi:10.1145/2382196.2382294.
- [AKS12] Man Ho Au, Apu Kapadia, and Willy Susilo. BLACR: TTP-free blacklistable anonymous credentials with reputation. In *NDSS 2012*. The Internet Society, February 2012.
- [ale] How Aleo Works. URL: https://www.aleo.org/ how-aleo-works/.
- [Ar22] Arkworks-rs. Arkworks Ecosystem Homepage, 2022. URL: https: //arkworks.rs/.
- [AST24] Arasu Arun, Srinath Setty, and Justin Thaler. Jolt: Snarks for virtual machines via lookups. In Annual International Conference on the Theory and Applications of Cryptographic Techniques, pages 3–33. Springer, 2024.

- [Atw09] Jeff Atwood. New question / answer rate limits, Feb 2009. URL: https://stackoverflow.blog/2009/02/21/ new-question-answer-rate-limits/.
- [BBB<sup>+</sup>18] Benedikt Bünz, Jonathan Bootle, Dan Boneh, Andrew Poelstra, Pieter Wuille, and Greg Maxwell. Bulletproofs: Short proofs for confidential transactions and more. In 2018 IEEE Symposium on Security and Privacy, pages 315–334. IEEE Computer Society Press, May 2018. doi:10.1109/SP.2018.00020.
- [BBF19] Dan Boneh, Benedikt Bünz, and Ben Fisch. Batching techniques for accumulators with applications to iops and stateless blockchains. In Advances in Cryptology-CRYPTO 2019: 39th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 18–22, 2019, Proceedings, Part I 39, pages 561–586. Springer, 2019.
- [BBHR18] Eli Ben-Sasson, Iddo Bentov, Yinon Horesh, and Michael Riabzev. Fast reed-solomon interactive oracle proofs of proximity. In Ioannis Chatzigiannakis, Christos Kaklamanis, Dániel Marx, and Donald Sannella, editors, *ICALP 2018*, volume 107 of *LIPIcs*, pages 14:1– 14:17. Schloss Dagstuhl, July 2018. doi:10.4230/LIPIcs.ICALP. 2018.14.
- [BBHR19] Eli Ben-Sasson, Iddo Bentov, Yinon Horesh, and Michael Riabzev. Scalable zero knowledge with no trusted setup. In Alexandra Boldyreva and Daniele Micciancio, editors, CRYPTO 2019, Part III, volume 11694 of LNCS, pages 701–732. Springer, Heidelberg, August 2019. doi:10.1007/978-3-030-26954-8\_23.
- [BCC04] Ernie Brickell, Jan Camenisch, and Liqun Chen. Direct anonymous attestation. In Proceedings of the 11th ACM Conference on Computer and Communications Security, CCS '04, page 132–145, New York, NY, USA, 2004. Association for Computing Machinery. doi:10.1145/1030083.1030103.
- [BCCT12] Nir Bitansky, Ran Canetti, Alessandro Chiesa, and Eran Tromer. Recursive composition and bootstrapping for SNARKs and proofcarrying data. Cryptology ePrint Archive, Report 2012/095, 2012. https://eprint.iacr.org/2012/095.
- [BCG<sup>+</sup>14] Eli Ben-Sasson, Alessandro Chiesa, Christina Garman, Matthew Green, Ian Miers, Eran Tromer, and Madars Virza. Zerocash: Decentralized anonymous payments from bitcoin. In 2014 IEEE Symposium on Security and Privacy, pages 459–474. IEEE Computer Society Press, May 2014. doi:10.1109/SP.2014.36.
- [BCG<sup>+</sup>20] Sean Bowe, Alessandro Chiesa, Matthew Green, Ian Miers, Pratyush Mishra, and Howard Wu. ZEXE: Enabling decentralized private computation. In 2020 IEEE Symposium on Security and Privacy,

pages 947–964. IEEE Computer Society Press, May 2020. doi: 10.1109/SP40000.2020.00050.

- [Bd94] Josh Cohen Benaloh and Michael de Mare. One-way accumulators: A decentralized alternative to digital sinatures (extended abstract). In Tor Helleseth, editor, *EUROCRYPT'93*, volume 765 of *LNCS*, pages 274–285. Springer, Heidelberg, May 1994. doi:10.1007/ 3-540-48285-7\_24.
- [BL07] Ernie Brickell and Jiangtao Li. Enhanced privacy ID: A direct anonymous attestation scheme with enhanced revocation capabilities. Cryptology ePrint Archive, Report 2007/194, 2007. https://eprint.iacr.org/2007/194.
- [BSCG<sup>+</sup>20] Eli Ben-Sasson, Alessandro Chiesa, Daniel Genkin, Eran Tromer, Madars Virza, Ohad Barta, Lior Greenblatt, Shaul Kfir, Michael Riabzev, Gil Timnat, and Arnon Yogev. Tinyram architecture specification v2.000. 2020.
- [CHK<sup>+</sup>06] Jan Camenisch, Susan Hohenberger, Markulf Kohlweiss, Anna Lysyanskaya, and Mira Meyerovich. How to win the clonewars: Efficient periodic n-times anonymous authentication. In Ari Juels, Rebecca N. Wright, and Sabrina De Capitani di Vimercati, editors, ACM CCS 2006, pages 201–210. ACM Press, October / November 2006. doi:10.1145/1180405.1180431.
- [CHM<sup>+</sup>20] Alessandro Chiesa, Yuncong Hu, Mary Maller, Pratyush Mishra, Psi Vesely, and Nicholas P. Ward. Marlin: Preprocessing zkSNARKs with universal and updatable SRS. In Anne Canteaut and Yuval Ishai, editors, *EUROCRYPT 2020, Part I*, volume 12105 of *LNCS*, pages 738–768. Springer, Heidelberg, May 2020. doi:10.1007/ 978-3-030-45721-1\_26.
- [CL01] Jan Camenisch and Anna Lysyanskaya. An efficient system for non-transferable anonymous credentials with optional anonymity revocation. In Birgit Pfitzmann, editor, *EUROCRYPT 2001*, volume 2045 of *LNCS*, pages 93–118. Springer, Heidelberg, May 2001. doi: 10.1007/3-540-44987-6\_7.
- [CT10] Alessandro Chiesa and Eran Tromer. Proof-carrying data and hearsay arguments from signature cards. In Andrew Chi-Chih Yao, editor, *ICS 2010*, pages 310–331. Tsinghua University Press, January 2010.
- [ets] Etsy Engineering Threat Modeling for Marketing Campaigns. Section: Consumer Product Develophttps://www.etsy.com/codeascraft/ ment. URL: threat-modeling-for-marketing-campaigns?utm\_source= OpenGraph&utm\_medium=PageTools&utm\_campaign=Share.

- [FKL<sup>+</sup>21] Nicholas Franzese, Jonathan Katz, Steve Lu, Rafail Ostrovsky, Xiao Wang, and Chenkai Weng. Constant-overhead zero-knowledge for RAM programs. In Giovanni Vigna and Elaine Shi, editors, ACM CCS 2021, pages 178–191. ACM Press, November 2021. doi: 10.1145/3460120.3484800.
- [GG21] Stan Gurtler and Ian Goldberg. SoK: Privacy-preserving reputation systems. PoPETs, 2021(1):107–127, January 2021. doi:10.2478/ popets-2021-0007.
- [GKR<sup>+</sup>21] Lorenzo Grassi, Dmitry Khovratovich, Christian Rechberger, Arnab Roy, and Markus Schofnegger. Poseidon: A new hash function for zero-knowledge proof systems. In Michael Bailey and Rachel Greenstadt, editors, USENIX Security 2021, pages 519–535. USENIX Association, August 2021.
- [GMN22] Nicolas Gailly, Mary Maller, and Anca Nitulescu. SnarkPack: Practical SNARK aggregation. In Ittay Eyal and Juan A. Garay, editors, FC 2022, volume 13411 of LNCS, pages 203–229. Springer, Heidelberg, May 2022. doi:10.1007/978-3-031-18283-9\_10.
- [Gro16] Jens Groth. On the size of pairing-based non-interactive arguments. In Marc Fischlin and Jean-Sébastien Coron, editors, EUROCRYPT 2016, Part II, volume 9666 of LNCS, pages 305–326. Springer, Heidelberg, May 2016. doi:10.1007/ 978-3-662-49896-5\_11.
- [GWC19] Ariel Gabizon, Zachary J. Williamson, and Oana Ciobotaru. PLONK: Permutations over lagrange-bases for oecumenical noninteractive arguments of knowledge. Cryptology ePrint Archive, Paper 2019/953, 2019. https://eprint.iacr.org/2019/953. URL: https://eprint.iacr.org/2019/953.
- [HBHW] Daira Emma Hopwood, Sean Bowe, Taylor Hornby, and Nathan Wilcox. Zcash Protocol Specification, Version 2023.4.0 [NU5]. URL: https://zips.z.cash/protocol/protocol.pdf.
- [HG11] Ryan Henry and Ian Goldberg. Formalizing anonymous blacklisting systems. In 2011 IEEE Symposium on Security and Privacy, pages 81–95. IEEE Computer Society Press, May 2011. doi:10.1109/SP. 2011.13.
- [JNV] URL: https://nbviewer.org/github/wikimedia-research/ automoderator-measurement/blob/main/baselines/T348860\_ median\_time\_to\_revert.ipynb#Time-to-Revert-Percentiles.
- [KMS<sup>+</sup>16] Ahmed E. Kosba, Andrew Miller, Elaine Shi, Zikai Wen, and Charalampos Papamanthou. Hawk: The blockchain model of cryptography and privacy-preserving smart contracts. In 2016 IEEE Sym-

posium on Security and Privacy, pages 839–858. IEEE Computer Society Press, May 2016. doi:10.1109/SP.2016.55.

- [Kus] John Kuszmaul. Verkle trees.
- [KZG10] Aniket Kate, Gregory M. Zaverucha, and Ian Goldberg. Constantsize commitments to polynomials and their applications. In Masayuki Abe, editor, ASIACRYPT 2010, volume 6477 of LNCS, pages 177–194. Springer, Heidelberg, December 2010. doi:10.1007/ 978-3-642-17373-8\_11.
- [LLX07] Jiangtao Li, Ninghui Li, and Rui Xue. Universal accumulators with efficient nonmembership proofs. In Jonathan Katz and Moti Yung, editors, ACNS 07, volume 4521 of LNCS, pages 253–269. Springer, Heidelberg, June 2007. doi:10.1007/978-3-540-72738-5\_17.
- [MBKM19] Mary Maller, Sean Bowe, Markulf Kohlweiss, and Sarah Meiklejohn. Sonic: Zero-knowledge SNARKs from linear-size universal and updatable structured reference strings. In Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz, editors, ACM CCS 2019, pages 2111–2128. ACM Press, November 2019. doi:10.1145/3319535.3339817.
- [MC23] Jack P. K. Ma and Sherman S. M. Chow. SMART Credentials in the Multi-queue of Slackness (or Secure Management of Anonymous Reputation Traits without Global Halting). In 2023 IEEE 8th European Symposium on Security and Privacy (EuroS&P), pages 896–912, Delft, Netherlands, July 2023. IEEE. URL: https://ieeexplore.ieee.org/document/ 10190534/, doi:10.1109/EuroSP57164.2023.00057.
- [PCL<sup>+</sup>07] Reid Priedhorsky, Jilin Chen, Shyong (Tony) K Lam, Katherine Panciera, Loren Terveen, and John Riedl. Creating, destroying, and restoring value in wikipedia. In *Proceedings of the 2007 ACM international conference on supporting group work*, pages 259–268, 2007.
- [PSG<sup>+</sup>24] Charalampos Papamanthou, Shravan Srinivasan, Nicolas Gailly, Ismael Hishon-Rezaizadeh, Andrus Salumets, and Stjepan Golemac. Reckle trees: Updatable merkle batch proofs with applications. Cryptology ePrint Archive, 2024.
- [PSS] Alexey Pertsev, Roman Semenov, and Roman Storm. Tornado Cash Privacy Solution Version 1.4.
- [rai] RAILGUN project whitepaper. URL: https://assets.railgun. org/docs/whitepaper/Railgun%20Project%20Whitepaper%20-% 20July%202021.pdf.

- [RIS] URL: https://www.risczero.com/.
- [RMM22] Michael Rosenberg, Mary Maller, and Ian Miers. SNARKBlock: Federated anonymous blocklisting from hidden common input aggregate proofs. In 2022 IEEE Symposium on Security and Privacy, pages 948–965. IEEE Computer Society Press, May 2022. doi:10.1109/SP46214.2022.9833656.
- [RPX<sup>+</sup>22] Deevashwer Rathee, Guru Vamsi Policharla, Tiancheng Xie, Ryan Cottone, and Dawn Song. ZEBRA: Anonymous credentials with practical on-chain verification and applications to KYC in DeFi. Cryptology ePrint Archive, Report 2022/1286, 2022. https:// eprint.iacr.org/2022/1286.
- [RWGM23] Michael Rosenberg, Jacob White, Christina Garman, and Ian Miers. zk-creds: Flexible anonymous credentials from zksnarks and existing identity infrastructure. In 2023 IEEE Symposium on Security and Privacy (SP), pages 790–808, 2023. doi:10.1109/SP46215.2023. 10179430.
- [STW23] Srinath Setty, Justin Thaler, and RS Wahby. Lasso: Unlocking the lookup singularity. Technical report, Cryptology ePrint Archive, Report 2023/1216, 2023.
- [TAKS08] Patrick P. Tsang, Man Ho Au, Apu Kapadia, and Sean W. Smith. PEREA: towards practical TTP-free revocation in anonymous authentication. In Peng Ning, Paul F. Syverson, and Somesh Jha, editors, ACM CCS 2008, pages 333–344. ACM Press, October 2008. doi:10.1145/1455770.1455813.
- [TAKS10] Patrick P. Tsang, Man Ho Au, Apu Kapadia, and Sean W. Smith. Blac: Revoking repeatedly misbehaving anonymous users without relying on ttps. ACM Trans. Inf. Syst. Secur., 13(4), dec 2010. doi:10.1145/1880022.1880033.
- [Tha] Justin Thaler. Proofs, Arguments, and Zero-Knowledge. URL: https://people.cs.georgetown.edu/jthaler/ ProofsArgsAndZK.pdf.
- [Wik] URL: https://stats.wikimedia.org/#/en.wikipedia.org.
- [Wil] Dr. Zachary J. Williamson. The AZTEC protocol. URL: https://github.com/AztecProtocol/aztec-v1/blob/master/ AZTEC.pdf.
- [XCZ<sup>+</sup>22] Alex Luoyuan Xiong, Binyi Chen, Zhenfei Zhang, Benedikt Bünz, Ben Fisch, Fernando Krell, and Philippe Camacho. VERI-ZEXE: Decentralized private computation with universal setup. Cryptology ePrint Archive, Report 2022/802, 2022. https://eprint.iacr. org/2022/802.

- [XF14] Li Xi and Dengguo Feng. Farb: Fast anonymous reputationbased blacklisting without ttps. Proceedings of the 13th Workshop on Privacy in the Electronic Society, 2014. URL: https: //api.semanticscholar.org/CorpusID:11490543.
- [ZCa19] ZCash. What is jubjub?, 2019. URL: https://web.archive.org/ web/20221104080552/https://z.cash/technology/jubjub/.

# A Proof sketch

We will now present a proof sketch that our construction realizes the ideal functionality  $\mathcal{F}_{zkpr}$  in Figure 3 against computationally bounded adversaries with static corruption of parties. We do so by briefly describe the simulator Sim for our real protocol in the ideal world and arguing for its correctness assuming the underlying cryptographic primitives are correct.

The simulator Sim first runs Setup and gets the necessary trapdoors for the simulation and extraction of zero-knowledge proofs in Setup, Simmaintains a table mapping objects in its simulated real-world protocol to their ideal counterparts in  $\mathcal{F}_{zkpr}$  and, when necessary, updates the mapping when it needs to create a real-world object for an ideal-world one (or vice versa).

Because all messages between parties are accompanied by zero-knowledge proofs or ciphertexts, the simulator can extract on all adversarial generated messages, look up the corresponding ideal-world objects in its table, and proxy the requests to the ideal functionality. Similarly, for any honest interactions in the ideal functionality, the simulator can model the adversary's view of the real-world protocol by simulating the zero-knowledge proofs with respect to random commitments, serial numbers, and ciphertexts. In the case of executed callbacks, we cannot directly extract arguments (as there is no zero-knowledge proof associated with a call.) However, callbacks to a simulated party are decryptable with the keys the simulator holds.

Simulating Call and IngestCall. In the real world, Call is performed by adding a ticket, the arguments, and the current time to  $bb_{cb}$ , which is accessed later in ScanOne. This directly corresponds to our pendingCbs table, whose new entry is also sent to the callback creator.

To simulate, we must also ensure that the real and ideal functionalities behave equivalently with who can call callbacks. Suppose party  $\mathcal{P}_i$  applies a (tik, args) from bb<sub>cb</sub> during ScanOne. Firstly, the tuple could only appear in bb<sub>cb</sub> if its signature is valid, i.e., the party who ran Call knows tik's secret key. In addition, (tik, args) is only applied if tik appears in the user's cbList, which, in turn, can only happen if tik was given to  $\mathcal{P}_i$  during ExecMethodAndCreate-Callback. Thus, the calling party must know the secret key of the tik given at ExecMethodAndCreateCallback. This occurs only if they are the same party (or, the party opposite  $\mathcal{P}_i$  during ExecMethodAndCreateCallback was corrupted). Finally, we note that the signature check of  $\sigma'$  in VerifyCall ensures that the caller is in authorizedCallers. For simulating IngestCall, we note that, in the real world, ScanOne makes progress in its list of outstanding callbacks, and deletes it from the list. This is precisely what IngestCall performs.

**Simulating ExecAndMakeCb.** In the real world, callbacks are created by ExecMethodAndCreateCallback. In order to issue a ticket, ExecMethodAndCreateCallback rerandomizes  $pk_{spid}$  and stores it as tik along with the encryption key and method ID in obj.cbList. The user then posts the new object to  $bb_{obj}$ , and sends the service provider the transaction ID and the opening to the commitment for (tik, k, meth). We consider the call a success if VerifyCreate succeeds.

The ideal-world user can only run ExecAndMakeCb when all callbacks called before cutoffTime have been ingested. This directly corresponds with the lastFullScanTime variable in the real world.

Suppose the real-world server calls Call(tik, args) on method meth at time t. We argue that any future ExecMethodAndCreateCallback with lastFullScanTime > t must reflect the object post-callback. Note ExecMethodAndCreateCallback with lastFullScanTime threshold t' must show that obj.lastFullScanTime  $\geq t'$ . This can only occur if a ScanOne sequence that started at time at earliest t' has completed, which, in turn, only succeeds if the user has done one of the following: (1) executed meth(obj, args, x) and deleted the associated tik from cbList, (2) ignored the call and deleted tik from cbList, (3) or ignored the call and left tik in cbList. However, the latter two cases may only happen if tik  $\notin$  bul<sub>cb</sub>, which contradicts the fact that Call(tik, args) was called at time t. Thus, meth(obj, args, x) was called, and the current object state reflects the called callback.

# **B** Formal description of zk-promises

In this section we formally describe the entirety of zk-promises, once all the features in Section 4 are added. We write the zero-knowledge relations for ExecMethodAndCreateCallback and ScanOne in Figures 5 and 6, respectively. In these relations, when a new object obj' is created from obj, a fresh serial number is chosen. If this serial number has been revealed before, then the use will be unable to use this state as it will be treated as a stale state.

**Setup**( $\Phi$ )  $\rightarrow$  **pp.** This performs Groth16 CRS generation for the relations in Figures 5 and 6. The output **pp** contains proving and verifying keys.

**ExecMethodAndCreateCallback(pp, obj, pk**<sub>spid</sub>, meth', x)  $\rightarrow$  (obj',  $\pi$ , cbData, aux). Let obj be the user object, let pk<sub>spid</sub> be the verifying key for service provider ID spid for an EUF-CMA-secure signature scheme  $\Sigma$ , and let s be the commitment randomness used in that objects commitment in bb<sub>obj</sub>. Let meth' be the method the user wishes to create a callback for. Let x = (t, curTime), where t is the minimum value for lastFullScanTime that the service provider will accept and curTime be the current global time.

The  ${\sf ExecMethodAndCreateCallback}$  algorithm proceeds as follows:

1. Compute a new ticket as a rerandomized verifying key  $(\mathsf{tik}, r) \leftarrow \Sigma.\mathsf{RerandPk}(\mathsf{pk}_{\mathsf{spid}})$ 

- 2. Pick expiry expiry and a fresh encryption key k and build the callback list entry entry := (tik, exp, k, meth'). Commit to the entry  $com_{entry} := Com(tik, exp, k, meth'; s_{entry})$  where  $s_{entry}$  is fresh randomness.
- 3. Clones the zk-object obj to a new one obj'. Append tik to their private callback list, compute obj'. $h_{cb} \coloneqq H(obj.h_{cb}, entry)$ ,  $obj'.h_{cb}^{(old)} \coloneqq obj'.h_{cb}$ ,  $obj'.lastFullScanTime \coloneqq curTime$ , and pick a fresh serial number obj'.sn. Finally commit to the new object,  $com'_{obj} \coloneqq Com(obj'; s'_{obj})$  where  $s'_{obj}$  is fresh randomness.
- 4. Let  $cbData = (com_{entry}, obj.sn)$  and let  $aux = (s_{entry}, entry)$
- 5. The user computes a zero-knowledge proof  $\pi$  of  $R_{\text{create}}^{\Phi_{\text{create}}}$ .
- 6. The user sends  $(\pi, \mathsf{com}'_{\mathsf{obi}}, \mathsf{cbData})$  to  $\mathsf{bb}_{\mathsf{obj}}$
- 7. The user sends  $(\pi, \mathsf{com}'_{\mathsf{obj}}, \mathsf{cbData}, \mathsf{aux})$  to the service provider

The information that the user sends to the service provider does not deanonymize the user as it contains no multi-user or long term identifier. It contains a zero-knowledge proof,  $\pi$ , the cryptographic commitment com'<sub>obj</sub>, and cbData, and aux. cbData contains a cryptographic commitment of the (single-use) entry and a (single-use) fresh serial number. aux contains  $s_{entry}$ , the (single-use) fresh randomness and the callback list entry, entry.

VerifyCreate(pp,  $sk_{spid}$ , obj',  $\pi$ , cbData, aux). Let  $sk_{spid}$  be the service provider's signing key. The service provider performs the following:

- 1. Verify obj' appears on bb<sub>obi</sub>, i.e., that the callback was created;
- 2. Verify the proof  $\pi$  with respect to inputs cbData, curTime, and obj'; and
- 3. Unpack aux and verify cbData.com<sub>entry</sub> = Com(tik, exp, k, meth'; s<sub>entry</sub>)
- 4. Verify that tik has never been used before in a callback initiated by this service provider

**ScanOne(pp, obj**, x)  $\rightarrow$  (**obj**',  $\pi$ , **cbData**). Let **obj** be the user object, let x be the current time curTime, and let  $s_{obj}$  be the commitment randomness to **obj**. The user does as follows:

- 1. Clones the obj to a new one obj' and pick a fresh serial number obj'.sn. If  $h_{cb}^{(old)} = h_{cb}$  (i.e., the beginning of the scanning process), update obj'.loopStartTime := curTime.
- 2. Let entry = (tik, exp, k, meth) represent the current entry in obj.cbList. If (tik, args, t)  $\in$  bb<sub>cb</sub> for some args and  $t < \exp$ , absorb the callback: obj' := meth(obj, curTime, m, x), where  $m = \text{Dec}_k(\text{args})$ . If tik  $\notin$  bb<sub>cb</sub> and exp is in the future, then leave entry in obj'.cbList. In any other case, delete entry from obj'.cbList. Finally, update  $h'_{cb}^{(\text{new})} = H(h_{cb}^{(\text{new})}, \text{entry})$  if the entry was left in, and  $h'_{cb}^{(\text{new})} = h_{cb}^{(\text{new})}$  otherwise.



Figure 5: The  $R_{\text{create}}^{\Phi_{\text{create}}}$  relation

- 3. If  $h'_{cb}^{(old)} = h_{cb}$  this is the last step of a scanning process. Update obj'.lastFullScanTime := obj.loopStartTime.
- 4. Compute a zero-knowledge proof  $\pi$  of  $R_{\mathsf{settle}}$ .
- 5. Let  $cbData = (com_{entry}, obj.sn)$
- 6. The user sends  $(\pi, \operatorname{com}'_{obi}, \operatorname{cbData})$  to  $\mathsf{bb}_{obj}$

VerifyMethodExec(pp, obj',  $\pi$ , cbData). In VerifyMethodExec<sub>create</sub>, the maintainer of the bulletin board bb<sub>obj</sub> receives ( $\pi$ , com'<sub>obj</sub>, sn, cbData). It first checks sn has not appeared in its set of observed serial numbers. Next, it verifies  $\pi$  with respect to cbData and its own curTime and bb<sub>obj</sub> representative. On success, it adds ( $\pi$ , com'<sub>obj</sub>, cbData) to bb<sub>obj</sub>.

In VerifyMethodExec<sub>settle</sub>, the maintainer receives the same payload. The behavior is identical to above, with the only change being that it also uses a  $bb_{cb}$  representative as public input for  $\pi$  verification.

VerifyCall(pp, tik, args, aux). The maintainer of the callback bulletin board  $bb_{cb}$  receives (tik, args,  $aux = \sigma$ ). It interprets tik as a signature public key pk, and then checks  $\Sigma$ .Verify<sub>pk</sub>(args,  $\sigma$ ). On success, it posts (tik, args, curTime) to  $bb_{cb}$ .

# C Membership and non-membership

zk-promises is phrased in terms of bulletin boards and data structures that admit efficiently checkable membership/non-membership arguments. We identify and implement two such variants.

 $com_{obj}, com'_{obj}, com_{entry}, nul, t, x, curTime,$  $bb_{obj}, bb_{cb};\\$  $s_{obj}, s'_{obj}, s_{entry},$ wasCalled, tik, exp, k, meth, timeCalled, args, m $\mathsf{com}_{\mathsf{obj}} \in \mathsf{bb}_{\mathsf{obj}}$  $\mathsf{com}_{\mathsf{obj}} = \mathsf{Com}(\mathsf{obj}; s_{\mathsf{obj}})$  $\operatorname{com}_{\operatorname{obj}}' = \operatorname{Com}(\operatorname{obj}'; s'_{\operatorname{obj}})$ obj.sn = sn $\mathsf{wasCalled} = \mathsf{tik} \in \mathsf{bb}_{\mathsf{cb}}$  $(timeCalled, args) = if wasCalled : bb_{cb}[tik] else (\bot, \bot)$  $m = \mathsf{Dec}_k(\mathsf{args})$ scanning := obj. $h_{cb}^{(old)}$  = obj. $h_{cb}$ scanning := obj'. $h_{cb}^{(old)}$  = obj'. $h_{cb}$ entry := (tik, exp, k, meth)  $deleteEntry := curTime < exp and tik \notin bb_{cb}$  $absorbEntry := timeCalled < exp and tik \in bb_{cb}$  $\neg \mathsf{absorbEntry} \lor \Phi_{\mathsf{ingest}}(\mathsf{obj},\mathsf{obj}',\mathsf{meth},m,\mathsf{curTime})$  $\begin{array}{l} \text{obj}'.h_{cb}^{(\text{old})} = H(\text{obj}.h_{cb}^{(\text{old})}, \text{entry}) \\ \text{obj}'.h_{cb}^{(\text{new})} = \mathbf{if} \text{ deleteEntry}: \text{obj}.h_{cb}^{(\text{new})} \end{array}$ else  $H(obj.h_{cb}^{(new)}, entry)$ obj'.loopStartTime = if  $\neg$ scanning : curTime else obj.loopStartTime  $obj'.lastFullScanTime = if \neg scanning' : obj.loopStartTime$ else obj.lastSwept  $\operatorname{obj}'.h_{\operatorname{cb}}^{(\operatorname{old})} = \operatorname{if} \neg \operatorname{scanning}' : \emptyset \operatorname{else obj}.h_{\operatorname{cb}}^{(\operatorname{old})}$  $\operatorname{obj}'.h_{\operatorname{cb}}^{(\operatorname{new})} = \operatorname{if} \neg \operatorname{scanning}' : \emptyset \operatorname{else obj}.h_{\operatorname{cb}}^{(\operatorname{new})}$  $obj'.h_{cb} = if \neg scanning' : obj.h_{cb}^{(new)} else obj.h_{cb}$ 

Figure 6: The  $R_{\text{ingest}}^{\Phi_{\text{ingest}}}$  relation

Merkle trees. We may represent our object and used callback sets  $bb_{obj}$  and  $bb_{cb}$  as Merkle trees, where the leaves are the elements of the set. To prove of  $x \in S$ , it suffices to prove knowledge of an authentication path from a leaf with value x to the root of the tree. In these proofs, the root is public input. To keep a proof of membership up to date, it suffices to download a *frontier* of the append-only tree. This permits a communication cost tradeoff of logarithmic to linear, depending on privacy requirements [RWGM23].

To allow non-membership proofs for  $bb_{cb}$ , we simply prove membership in the *complement* of  $bb_{cb}$ , i.e.,  $C = \{0, 1\}^{256} \setminus \{bb_{cb}.tik_i\}_i$ . Specifically, we partition C into semi-open ranges of integers  $[a, b) \subseteq \{0, 1\}^{256}$ , and define a Merkle tree T whose leaves are those ranges. Then to prove non-membership in  $bb_{cb}$ , it suffices to show knowledge of a tik, a, b such that  $a \leq tik < b$  and [a, b) is in T. We note that the structure of T is liable to change every time  $bb_{cb}$  is modified, so proofs of membership are not necessarily updatable using the frontier method.

**Signatures.** In a centralized bulletin board setting, it is also possible to represent set membership using signatures. The manager of the bulletin boards maintains two signature keypairs  $(\mathsf{pk}_{\mathsf{obj}},\mathsf{sk}_{\mathsf{obj}})$  and  $(\mathsf{pk}_{\mathsf{cb}},\mathsf{sk}_{\mathsf{cb}})$ . Every time a value is posted to a bulletin board, the manager signs the value and returns the signature. To prove membership of x in a bulletin board with public key  $\mathsf{pk}$ , it suffices to prove knowledge of a signature  $\sigma$  such that  $\mathsf{Verify}_{\mathsf{pk}}(x)$  is true. Compared to Merkle trees, membership signatures have the benefit of not requiring updating—a valid  $\sigma$  will always be valid regardless of how the corresponding set changes.

Signature non-membership proofs work similarly. As above, we partition the complement set into ranges and prove membership in that signed set. Since the  $bb_{cb}$  complement set *shrinks* over time, a valid proof of non-membership at time t should not necessarily be valid at time t+1, since the value might have become a member of  $bb_{cb}$  in the meantime. To handle this, we must invalidate every old non-membership signature. We can do this by adding an epoch to every value and making the verification equation  $Verify_{pk}(x)$ , or by picking a new signing key every epoch and publicizing it through the bulletin board. In either case, the bulletin board manager must re-sign the entire complement every epoch.

# **D** Number of Constraints

Table 2 lists the number on constraints in ShowAuthMakeCB and ShowAuthMakeCB1Sweep circuits in the centralized and decentralized settings. In the ShowAuthMakeCB circuit, the user shows that the record is in good standing and creates a callback. The ShowAuthMakeCB1Sweep circuit also does this but also scans for a single callback.

Table 3 lists the number on constraints in BatchedScan circuits in the centralized and decentralized settings with various batch sizes.

Circuit	Single Server	De-centralized		zed		
Epoch Capacity:	Unlimited	$2^{16}$	$2^{32}$	$2^{64}$		
Number of Constraints						
ShowAuthMakeCB	28,603	$25,\!352$	29,720	$38,\!456$		
ShowAuthMakeCB1Sweep	60,694	$55,\!971$	64707	$82,\!179$		

Table 2: The number on constraints in ShowAuthMakeCB and ShowAuthMakeCB1Sweep circuits.

Batch Size	Single Server	De-centralized			
Epoch Capacity:	Unlimited	$2^{16}$	$2^{32}$	$2^{64}$	
Number of Constraints					
1	60,694	55,971	64,707	82,179	
2	$105,\!872$	100,743	$113,\!847$	$140,\!055$	
4	$194,\!210$	187,209	209,049	252,729	
8	$370,\!886$	360,141	$399,\!453$	$478,\!077$	
16	724,238	706,005	780,261	928,773	
32	$1,\!430,\!942$	$1,\!397,\!733$	$1,\!541,\!877$	$1,\!830,\!165$	
64	$2,\!844,\!350$	2,781,189	$3,\!065,\!109$	$3,\!632,\!949$	

Table 3: The number on constraints in batched scan circuits.