zk-promises: Anonymous Moderation, Reputation, and Blocking from Anonymous Credentials with Callbacks

Maurice Shih
University of Maryland
maurices@umd.edu

Michael Rosenberg
University of Maryland
micro@cs.umd.edu

Hari Kailad
University of Maryland
harikeshkailad@gmail.com

Ian Miers
University of Maryland
imiers@umd.edu

Abstract

Anonymity is essential for free speech and expressing dissent, but platform moderators need ways to police bad actors. For anonymous clients, this may involve banning their accounts, docking their reputation, or updating their state in a 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*. This raises a challenge: how can multiple parties interact with private state stored by an anonymous client while ensuring state integrity and supporting oblivious updates?

We propose zk-promises, a framework supporting stateful anonymous credentials where the state machines are Turing-complete and support asynchronous callbacks. Client state is stored in what we call a zk-object held by the client, zero-knowledge proofs ensure the object can only be updated as programmed, and callbacks allow third party updates even for anonymous clients, e.g, for downvotes or banning. Clients scan for callbacks periodically and update their state. When clients authenticate, they anonymously assert some predicate on their state and that they have scanned recently (e.g, within the past 24 hours).

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 design, implement, and benchmark 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

Systems like Tor and Apple Private Relay make anonymous speech on the web possible, allowing the free expression of ideas when it might otherwise not be safe. But, particularly on the web, anonymous speech is a double-edged sword platforms may not wish to allow: while necessary for free expression, it can also be used to derail debate, disrupt social consensus, or distribute disinformation.

Policing participants on an anonymous platform poses a fundamental challenge: the site does not know who committed the offending act, and so cannot easily penalize their bad behavior by banning their account, limiting their ability to post, or docking their reputation score. This holds even if the penalty itself, such as banning the poster from making future posts, does not directly identify the client. In a non-anonymous system, these actions can be accomplished by a simple update to the client's state in a server-side database. In the anonymous setting, the site cannot know the client's identity or their current state.

Starting with BLAC [TAKS10], a long line of work [TAKS08, AKS12, AK12, XF14, RMM22, MC23] builds schemes for anonymous blocklisting. These provide an elegant solution to banning anonymous users that does not rely on a Trusted Third-Party (TTP) who can de-anonymize users. These schemes allow clients to make a zero-knowledge proof that they did not author posts on some blocklist or, for more advanced schemes, that the weighted sum of flags on posts they did author is below some threshold.

Existing anonymous blocklisting schemes, while innovative, suffer from efficiency and functionality limitations that fall short of what applications need. These protocols cannot support the basic features we expect for moderation on non-anonymous forums, like upvote/downvote scoring, rate-limiting low-reputation users, or implementing temporary bans and probationary periods. Moreover, the stateless nature of these systems necessitates a one-size-fits-all approach to the statements clients make about the blocklist, making very active and less active clients do the same work. This computational expense forces real-world deployments to either impose stringent limits on all clients or suspend the system until moderators can adjudicate each event, significantly hampering the system's utility and scalability.

In this paper, we design and implement a new solution,

zk-promises: an anonymous credential scheme that supports arbitrary programmatic logic and mutable client state while preserving client anonymity. The core of our construction allows asynchronous anonymous callbacks to modify state stored client-side, while assuring both integrity and anonymity. A set of callbacks can safely be associated with a sequence of client actions (e.g., posts on a forum, edits to Wikipedia, etc.) without identifying the user or linking their actions together. Third parties can then invoke these callbacks to invoke a particular function (e.g., setting the client's state to banned, decrementing their reputation, or placing them on a probationary period). Clients periodically scan for invoked callbacks and apply them to their state. When clients authenticate, clients assert some predicate on their state and that they have recently (e.g., within 24 hours) scanned for pending callbacks.

The challenge of third party updates to private state and asynchronous negative feedback. Consider a version of Wikipedia that supports anonymous edits. A user 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. The core challenge in this setting is that state updates are asynchronous—the user is not guaranteed to be online when the state update is made. In addition, depending on application, users may have ample motive and opportunity to evade an update.[1] We refer to this as the *asynchronous negative feedback problem*.

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. 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. 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.

Preventing banned users from making new accounts. If a banned user can simply make a new account, bans are not effective. This is a challenge, even without anonymity. One option is to have a single account with robust identity requirements, e..g, real-world identity verification. Anonymous credentials, including zk-promises support such an approach without linking every action to the identity. There are options that do not require real-world identity verification. Services like StackExchange [Atw09] define arbitrary server-side logic tying allowed behavior to the account's history, e.g., rate-limiting the number of posts and, separately, the number of (possibly spam) links in posted question for new accounts, or gating moderator privileges behind a sequence of achievements. 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 user.

A starting point: zk-objects as anonymous credentials. We start, somewhat surprisingly, by repurposing techniques developed for privacy-preserving cryptocurrency payments and smart contracts. Starting with Zerocash [BCG+14] and commercial derivatives like Zcash, TornadoCash, and Railgun [HBHW, PSS, rai], through to academic [KMS+16, BCG+20, XCZ+22] and commercial products on privacy-preserving smart contract systems [Wil, ale], 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 (a *double-spend* in payment systems). When these zero-knowledge proofs are instantiated with zk-SNARKs, 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 to track

[1] 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.

their tickets in object state, check if tickets are mapped to posted callbacks, and execute 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 [TAKS08] ticket approach for simple bans, but overcomes a few key challenges to offer drastically improved performance, and generalize to arbitrary Turing-complete state machines. First, to achieve our performance requirements, we must describe a fixed-size state machine that allows users to incrementally iterate through an unbounded list of open callbacks, using zk-friendly data structures, while revealing nothing to the server and preventing skipping callbacks.[2] Second, we must support multiple callbacks and callers, while ensuring confidentiality of callback arguments, the authenticity of the caller, and the integrity (i.e., non-evasion) of the callback and the object itself. Since neither the client nor the caller are trusted,[3] we must carefully design a protocol that avoids malicious inputs, fault injections, and key misbinding issues. With these resolved, we achieve a scheme which permits arbitrary callbacks on user-defined zk-objects.

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

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

1.1 Related work

The approach we take builds, conceptually, on a line of foundational work on anonymous credentials [Cha82, CL01], allowing clients to make privacy-preserving assertions about their credentials. Such schemes have been extended to be stateful [CGH09], allowing, e.g., an updateable reputation score. The challenge here, however, is the mechanism to support updates by third parties that are asynchronous. We discuss specialized protocols for this, often themselves using techniques from anonymous credentials, below. We note that the problem is distinct from a long-studied problem of revocation of anonymous credentials (e.g. [CL01, BL07]). These schemes assume the revoking authority knows either the public key or private key of a party.

For a broad view of existing specialized protocols, we refer readers to the overviews of Henry and Goldberg for anonymous blocklisting [HG11] and Gurtler and Goldberg [GG21] for reputation. The anonymity property we want—that users' actions are not linked to each other or to the user—corresponds to what Gurtler and Godlberg term Reputation-Usage Unlinkability for Votee Privacy. Gurtler and Goldberg give a number of schemes that provide this property, but if we exclude schemes that require a trusted third parties (TTP) who can de-anonymize users or schemes only supporting positive feedback, the relevant related work is restricted to specific approaches in the anonymous blocklisting literature for token-based anonymous blocklisting (previously called blacklisting).

The core of anonymous blocklisting schemes is a pseudo-random token clients associate with each action (e.g., a forum post) and an anonymous credential embedding the key used to generate the tokens. In BLAC [TAKS10] and its successors [AKS12, RMM22], clients had to iterate over the entire blocklist for each authentication, proving each token could not be derived using their key. As an improvement, in PEREA [TAKS08], the author develop a new approach where clients use a pseudo-random function and a counter to enumerate only their past n tokens and show they are not blocked, avoiding the need to compute proofs about the entire blocklist. However, this method, refined in subsequent work [AK12, XF14, MC23], presents a trade-off, since clients are stateless and cannot store how many tokens they have actually made, n is a fixed value shared across all clients. As a result: (1) n must be small enough to keep authentication efficient, as client work scales linearly with n. (2) A small n necessitates either severe rate limiting on clients or frequent global halts for misbehavior adjudication..

Ma and Chow [MC23] note this tradeoff and propose a partial solution using multiple parallel adjudication queues. However, this approach, while elegant, does not fully resolve the underlying issues. Moreover, all of these schemes lack the comprehensive state management and logic required to implement the complex moderation procedures common in non-anonymous forums.

2 An example application: anonymous banning and reputation dependent rate limiting.

In this section, we introduce an example application of zk-promises. This example serves two purposes. First, it demonstrates zk-promises and its programming model for enforcing callbacks on arbitrary anonymous objects.

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

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

Second, it will provide the concrete application we evaluate in Section 6. The application features have been chosen to emphasize the flexibility of our approach and we stress that the techniques described here have broader applications beyond this example, 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. We assume the user hides their IP address and avoids other forms of fingerprinting, e.g., by using the Tor Browser. 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 *projected reputation*, defined as the dot product of their reputation with a public weight vector, exceeds a specified threshold.
2. The user is within their 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.[4]
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. In addition, the `updateRep` method is restricted to Wikipedia.

```
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 = currTime() // Used for leak rate

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

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

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

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

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

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

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 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 unwarranted modification, e.g., a user 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

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

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

  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):
    return this.lastFullScanTime >= cutoffTime

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

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

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

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

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.

We note that the callback manager, although defined as a separate object, will, for efficiency, be stored in the same commitment along side user object, and calls will be inlined.

In our example, callbacks are represented by a random 32-byte *ticket* tik created by the user's zk-object invoking `makeCb` in `ShowAuthMakeCB`. The server, Wikipedia in our example, later calls the callback by placing the ticket on a bulletin board along with the method arguments. For example, a moderator, after reviewing the post, could decrement the anonymous users reputation by 3 via posting $(\mathsf{tik}, -3)$ to the bulletin board. The callback would then be ingested when the user scans through its open callbacks and is forced to run `updateRep(moderator, -3)`.[5] We note that the reputation update portion of the callback, -3 in this example, will be encrypted.

2.2 Security properties

We first describe the security properties off the zk-object model before then discussing the specific security properties of our example applicaton 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.

Integrity An object is only updated according to its programmed methods. From this it follows callbacks must be applied. Authorized entities may exclusively make changes, and only in accordance with 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.

We capture these properties in a simplified ideal functionality for a zk-callback system, described later in Figure 4. 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. The ideal functionality guarantees that when a client asserts it has handled all pending callbacks, it must have actually done so. For ease of exposition, we limit the ideal functionality to expressing the scheme itself, and omit details in the real protocol for replay prevention and authenticated channels.

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

Security Goals for Anonymous Reputation. For the anonymous reputation system example application, we require a strong anonymity property: unlinkability. Any action by a user should not be associable with any other action, even if all servers and other users are malicious. Naturally, we need at least two honest users for any meaningful notion of anonymity and assume clients use an anonymized internet connection (e.g., Tor or Apple Private Relay). This anonymity property should hold even after users are banned (so-called *backward anonymity* [HG11]). At a high level, zk-promises meets these requirements because the only information associated with each client action is a pseudorandom tag—the zero-knowledge proof and commitments hide all other state.

We also require *non-frameability*—no one other than the service provider can ban a user or alter their state. Finally, we need basic correctness and authenticity requirements that allow legitimate clients to use the system while prohibiting clients with insufficient reputation. All of these properties are in-line with previous work.

We note two practical constraints on security. First, we assume, as does all previous work taking the token-based approach, that all clients share access to a single view of the blocklist (in our case, this is the bulletin board). This can be accomplished by, e.g., gossiping the hash. Failure to do so could allow a malicious server to launch targeted attacks by feeding a target client a special blocklist that blocks everyone else. The feasibility of this attack depends heavily on the setting.

A more significant concern is that while client posts are unconditionally anonymous from the server's perspective, this anonymity does not hold if client state is disclosed (e.g., through compromise or subpoena). The client must store non-expired callbacks and check if they have been used. Consequently, the client maintains a list identifying their actions until the callbacks expire. This limitation seems inherent to the functionality: we cannot have accountability if even the client does not know what they did in the past. Looking ahead, our construction does not maintain identifying information for any actions after callbacks have expired. With an expiration time of, e.g., 24 hours, this would allow a user to build up a history of good behavior (e.g., posts not being banned in 24 hours), without tying their account and reputation to their posting history. However, the exposure risk for posts prior to callback expiry may make anonymous reputation systems unsuitable for extremely sensitive applications.

3 Notation and cryptographic preliminaries

3.1 General notation

We write $x := z$ to denote variable assignment. $y := \mathsf{A}(x;r)$ denotes the execution of a probabilistic algorithm 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 $y := \mathsf{Alg}(x;r)$. The security parameter of our system is denoted by λ.

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:

$\mathsf{Setup}(P) \to (\mathsf{srs}_P, \mathsf{srs}_V, \tau)$ 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* τ used for simulating proofs (only used in establishing the zero-knowledge property).

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

$\mathsf{Verify}(\mathsf{srs}_V, x, \pi)$ Verifies π 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 $\mathsf{Sim}(\tau, x)$ which, given trapdoor τ and instance x, produces proofs which are perfectly indistinguishable from honest proofs generated with $\mathsf{Prove}(\mathsf{srs}_P, x, w)$ 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 [Tha23].

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 a circuit to compute 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 Σ = (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 $\mathsf{RerandPk}(\mathsf{pk}) \to (\mathsf{pk}', r)$ returns a fresh public key with randomness r such that pk' is computationally indistinguishable from a public key generated with Keygen.

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

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[6] 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 r amounts to proving knowledge of an *authentication path*—a list of the siblings of x's ancestors—whose iterated hash equals r.

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

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

where Φ 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.

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 6. 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 de-anonymize 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 zk-promises. $\mathsf{bb}_{\mathsf{obj}}$ stores every committed object obj on the bulletin board. $\mathsf{bb}_{\mathsf{cb}}$ stores every callback posted to the bulletin board. These structures permit efficient lookup of items inside zero-knowledge proofs. In addition, $\mathsf{bb}_{\mathsf{cb}}$ must support efficient non-membership proofs. Non-membership requires the bulletin board operator to commit to the complement of $\mathsf{bb}_{\mathsf{cb}}$. For signature-backed bulletin boards, this requires a *bulletin board rollover*—a full recomputation of the complement set—whenever $\mathsf{bb}_{\mathsf{cb}}$ 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 append-only log growing as block height grows.

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

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

Methods. zk-promises permits deployers to define multiple *methods*, functions which mutate the object, these methods can both create callbacks and be called by them. For example, as we saw in Section 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 zk-promises. In our zero-knowledge proofs, we will use $\Phi_{\mathsf{meth}}(\mathsf{obj}, \mathsf{obj}', \ldots)$ to represent a valid transition from obj to obj' using method meth.

Algorithms. zk-promises consists of the following algorithms:

$\mathsf{Setup}(\Phi) \to \mathsf{pp}$ Takes a representation of zk-promises application-specific predicates and produces public

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

parameters for our zero-knowledge proof scheme. This may be a trusted setup procedure as required by, e.g., the Groth16 zkSNARK [Gro16].

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

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

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

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

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

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

We now describe each algorithm in detail.

$\mathsf{ExecMethodAndCreateCallback}(\mathsf{pp}, \mathsf{obj}, \mathsf{pk}_{\mathsf{spid}}, \mathsf{meth}, x) \to (\mathsf{obj}',\pi,\mathsf{cbData},\mathsf{aux})$**.** This algorithm performs two important functions: (1) it updates obj according to the given method; and (2) it creates a new callback entry 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 meth' the user wishes to create a callback for. The user performs the following updates to its objects.

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

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

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

zk-object bookkeeping:

- *The old object exists.* $\mathsf{obj} \in \mathsf{bb}_{\mathsf{obj}}$
- *The serial number is revealed.* $\mathsf{obj}.\mathsf{sn} = \mathsf{sn}$
- *The entry has been appended.* $\mathsf{obj}'.\mathsf{cbList} = \mathsf{obj}.\mathsf{cbList} \| \mathsf{entry}$[7]

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

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

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

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

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

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

[7] 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 cbData and 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 $\mathsf{bb}_{\mathsf{cb}}$. We will later extend Call to sign and encrypt args.

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

We describe the update more formally. Let x represent some additional public data and let 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., $(\mathsf{entry.tik}, \mathsf{args}) \in \mathsf{bb}_{\mathsf{cb}}$, then the user performs the following updates to its object:

1. Set $\mathsf{obj}' := \mathsf{meth}(\mathsf{obj},\mathsf{args},x)$
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 $\mathsf{obj}' := \mathsf{obj}$ and skips (1) and (2).

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

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

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

entry *is the current entry.* $\mathsf{entry} = \mathsf{obj.cbList.nextUnscanned}()$

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

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

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

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

VerifyMethodExec$(\mathsf{pp}, \mathsf{obj}', \pi, \mathsf{cbData}) \to \{0, 1\}$. The object bulletin board must handle zk-object updates from ExecMethodAndCreateCallback and ScanOne. To do so, it does the same proof verification as VerifyCreate. Specifically, given $(\pi,\mathsf{cbData},\mathsf{obj}')$ it verifies the zero-knowledge proof π with respect to its inputs cbData and obj'. In addition, as with any zk-object update, it must ensure that the serial number is not repeated. That is, cbData's obj.sn $\notin S$, where S is the set of all serial numbers observed by the bulletin board operator. If all checks succeed, the operator posts $(\mathsf{obj}',\pi,\mathsf{cbData})$ to $\mathsf{bb}_{\mathsf{obj}}$.

VerifyCall$(\mathsf{pp},\mathsf{tik},\mathsf{args},\mathsf{aux}) \to \{0,1\}$. The operator of the callback bulletin board must verify incoming calls (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 $\mathsf{com}_{\mathsf{entry}} := \mathsf{Com}(\mathsf{entry};s)$ for some randomness s, and opens the commitment in the proof. In addition, the auxiliary data sent to the service provider is now aux := (entry,s). In VerifyCreate, the service provider checks that $\mathsf{com}_{\mathsf{entry}} = \mathsf{Com}(\mathsf{entry};s)$.

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

4.3 Callback Expiry

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

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

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

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

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

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

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

4.4 Authenticity and confidentiality for callback inputs

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

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

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

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

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

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

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

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

In Call, instead of sending $(\mathsf{tik},\mathsf{args})$, the service provider sends $(\mathsf{tik},\mathsf{args},\sigma)$, where $\sigma = \Sigma.\mathsf{Sign}_{\mathsf{sk}'_{\mathsf{spid}}}(\mathsf{args})$. This binds the arguments to the ticket.

In VerifyCall, instead of posting $(\mathsf{tik},\mathsf{args})$ unconditionally, it now receives $\mathsf{aux} = \sigma$ and posts the call iff $\Sigma.\mathsf{Verify}_{\mathsf{tik}}(y,\sigma)$ is true.

Confidentiality: encrypted method arguments and in-circuit decryption. To encrypt args, the caller computes $\mathsf{args} := \mathsf{Enc}_k(\mathsf{plaintextArgs})$, where k is an encryption key and Enc is a circuit-friendly CCA-secure authenticated symmetric encryption algorithm. Two challenges arise. First, the scheme must be key-committing to prevent users from intentionally using the wrong key to decrypt arguments incorrectly. We can achieve this generically by simply requiring a proof of decryption under the correct key. However, this leads to our second problem: requiring clients to prove decryption correctness could allow malicious callers to deadlock clients with invalid ciphertexts.

To address these issues, we include a separate encryption key with each callback list entry and mandate its use for decryption via the circuit. This ensures clients use the correct key, while ticket holder authentication prevents ciphertext tampering. If decryption fails, the circuit allows clients to skip the callback, preventing deadlocks from malformed ciphertexts and resulting only in a failed callback. Here, the circuit must be carefully crafted to ignore malformed ciphertexts. CCA security is insufficient, since the worst case attacker is the callback maker who deliberately encrypts an invalid message into a valid ciphertext.

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 $\mathsf{args} = \mathsf{Enc}_k(\mathsf{plaintextArgs})$.

In ScanOne, the user simply decrypts the payload in-circuit. That is, rather than using args directly, it uses $\mathsf{Dec}_k(\mathsf{args})$, where k comes from the current entry in obj.cbList. If decryption fails, it skips the callback.

4.5 Private and public method identifiers

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

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

Public method IDs can be used to mitigate this. That is, we may let each Φ_{meth} be a separate circuit, and have every

ExecMethodAndCreateCallback and ScanOne also post meth in the clear to the bulletin board.

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

5 Implementation

We now detail key design decisions when building an implementation of our system. We note that these are for our system as a generic framework for anonymous callbacks, and not for the specific proof of concept application for anonymous reputation and banning we evaluate in the next section.

5.1 Supporting centralized and decentralized settings

We have, up until now, defined zk-promises generically in terms of bulletin boards, suggesting it can be instantiated both in a centralized setting and in a decentralized one run by a transparency log or a blockchain. We now detail how each setting is accomplished and discuss optimizations.

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. Both settings raise interesting design considerations.

Efficient callback lookups. We require a mechanism to verifiably lookup callbacks and either: (1) Confirm their absence from the bulletin board, or (2) Retrieve the associated callback arguments if present. To do this, one might use a conventional verifiable key-value store using a sparse Merkle tree, with keys as paths and values as leaves. However, since we are limited to a fixed size circuit for the ZK proof, this incurs worst-case costs for verifiable reads. This requires d hashes where d is the bit length of the hashed keys (in our case, 256 bits).

In the setting of zk-promises, however, we can do better, since the trees themselves are public and there structure can be verified outside the circuit. We implement two Merkle trees, one storing tuples of (key, value) at the leaves and one storing intervals of unused keys. A lookup consists of either retrieving a tuple and checking it contains the appropriate key, or retrieving an unused interval and checking the key falls in the interval. Here, the circuit need only verify a path of length $\log|\mathsf{bb}_{\mathsf{cb}}|$. 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. In addition, we efficiently verify membership or non-membership by only checking a single valid path in one of the two Merkle trees, instead of computing both and conditionally selecting the valid one. This is similar to the approach taken in Verdict [TKPS21]. Verdict merges the membership and non-membership trees, instead adding a pointer-like hash indicated the next leaf. Compared to our approach, this saves the cost, outside of the circuit, of maintaining two data structures and offers some other properties, at the cost of more implementation complexity and the marginal cost of an additional hash check in the circuit.

Optimizations for the centralized setting. In the centralized setting, we have a trusted party who can maintain the bulletin board. As such, we can replace Merkle membership checks, using $\log|\mathsf{bb}_{\mathsf{obj}}|$ hashes, with a single signature verification.

5.2 Efficiently tracking callbacks

A key challenge in zk-promises is efficiently managing the cbList, which tracks issued callbacks. We require a data structure that supports efficient append, remove, and iteration operations within the arithmetic circuit used for the ZK proof. 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 we do not want to resort to expensive mechanisms for *persistent* zero-knowledge random access memory (e.g, [OWWB20]), that would be necessary both to support random access and persistence of the now modified callback list across scans.

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

$$h_\ell := H(H(H(H(\ell_1),\ell_2),\ell_3)...\ell_n),$$

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

To support amortized $O(1)$ 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.

5.3 Pruning and expiring old state

In the current construction, $\mathsf{bb}_{\mathsf{obj}}$ and the list of revealed serial numbers grow indefinitely. We can address both of these by defining an expiry period for zk-object commitments older than t (e.g., one year). If the user has not updated their state in that time, they are locked out. The set of witnessed serial numbers can similarly be pruned with the same cutoff, since any

serial number revealed at time t must correspond to an object that is older than t, has been pruned, and cannot be replayed.

Pruning state in the centralized setting. Recall that, in the centralized setting, checking membership in the ledger is accomplished simply by checking a signature from the server on the entry. Pruning, in this case, requires we expire the signature, not just remove data. To do so, we include an expiry time in the signed value, and modify the circuit to prove all signatures are valid and unexpired. This additional comparison against the current time removes the need of the server regularly rotate signing keys and re-sign all valid entries.

5.4 Software

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

zk-promises code. zk-promises is written in Rust [8], using the Arkworks [Ar22] zkSNARK crates and Rayon for parallelization where possible.

6 Evaluation

In this section, we evaluate a prototype of the application described in Section 2 built with the implementation of zk-promises described in Section 5 . We benchmark circuits for two versions of the prototype application, one for the decentralized setting and one for the centralized setting of zk-promises.

Our prototype application consists of a client and server written in Rust that communicate via RPC calls. The client holds a zk-object corresponding to its user account record (see Figure 1). This record includes the user's reputation, state for a reputation dependent leaky-bucket rate-limit on the users ability to post, and ban status. The client submits its zk-object updates to the server. When the client wishes to post anonymously, they submit an update to their object that both demonstrates their current authorization to post, expends part of their rate limit, and provides callbacks that allow the server to later either ban them or modify their reputation.

The server, implemented using Actix Web [Act], hosts the bulletin boards and can process requests such as user registration, client operations on the zk-object, and callbacks for moderation. The server does not implement a full forum or wiki, rather it models the component necessary for access control and applying moderation decisions (i.e., bans and upvotes/downvotes) In this setting, we benchmark both constructions, i.e., the Merkle tree construction where no parties are trusted for integrity, and the signature bases approach where the parties(s) holding a signing key are trusted for integrity but not anonymity.

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.

6.1 Experiments

Our measurements are done in the client-server setting, where clients send requests to a server to authenticate. We measure the median time it takes a client to complete various operations and the median time it takes the server to process them. Over all benchmarks, the maximum observed relative standard error of the median was 1.2%.

Client. Table 1 shows the (multi-threaded) client run time to complete one of three operations with their account, and the corresponding time for the server to verify the operation. `Show Authorized` is a simple assertion the client is authorized and has scanned through all pending callbacks, it corresponds to the logic in `ShowAuthorizedForEditAndMakeCallbacks` in Figure 1. `Incremental Scan` advances the client scan by one callback, corresponding to the similarly named method in Figure 2. Since, in many cases, the client may have only a single callback and need to show they are authorized, we also give a combined procedure that performs both operations, saving some overhead. As expected, client runtime (in the decentralized setting) increases linearly with 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 construct a batched scan method for clients to handle a fixed number of pending callbacks. Benchmarks can be found in Figure 3. As expected, a batched scan is more efficient compared to the same number of single-scan calls. For example, for the tree height of 32, computation savings range from 20–40% when doing batched scans from 2 to 32 callbacks. Incrementing the batch size increases the client computation by about 242ms, 343ms, and 422ms for tree heights of 2^{16}, 2^{32}, and 2^{64}, respectively.

Server. In our prototype implementation , we instantiate the bulletin board as a single server where clients can gossip the hash of the data to ensure a consistent view of the bulletin board.

The server-side verification times for each client request are constant, regardless of client operation, at 2.8ms, because the only difference in operation is the circuit, and proof verification times are constant for all circuit sizes. To test throughput of the server, we sent client requests at a rate of 1 per millisecond. It took on average 3.73ms to process a request with a standard deviation of 0.02ms. This gives of us a throughput of 268

[8] Code repository: https://github.com/moshih/zk-promises

	Client Computation (ms)							Server Computation (ms)
Callback Capacity:	2^{16}	2^{18}	2^{23}	2^{26}	2^{32}	2^{64}	*Unlimited*	-
	Tree-Based						Signature-based	(Identical for both settings)
`Show Authorized`	316	325	339	347	372	558	337	2.81
`Incremental Scan`	535	563	585	585	638	972	571	2.81
`Show Authorized + Incremental Scan`	716	741	772	791	829	1123	909	2.81

Table 1: Runtimes for client and server computation. 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. Because the proofs produced by a client are constant size, server side verification times are the same for both applications

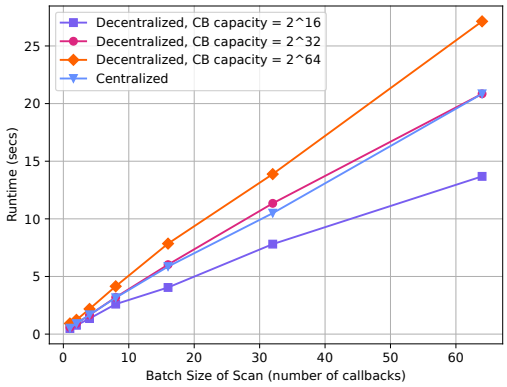

Figure 3: Client's proving runtime to batch settle callbacks as the size of the batch increases.

client requests per second. We attribute the additional 1ms of processing time, relative to the raw verification, from the overhead of the server handling the HTTP request.

We note that applying techniques for batching Groth16 proof verification would give a substantial performance improvement. In particular, in a standalone microbenchmark, we saw a 4× throughput improvement over individual verifications. However, we did not integrate this into our server, as it requires additional implementation of a queue, logic for when to trigger a batch verification, and handling asynchronous authorizations for, e.g., a forum post.

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.

We detail standalone measurements for the costs of maintaining the bulletin board in terms of size and data structure costs, in Appendix E.

6.2 Discussion and case studies

Performance of zk-promises depends both on how we configure the system and on the actual demands of the scenario, i.e., how many posts a user makes and how many result in moderator action. To determine feasibility, we need data from real forums, not just for active users and posts, but for the frequency of moderation events. Unfortunately, this data, particularly for moderation events, is not readily available in many cases. We find two case study datasets. First, the number of edits and reversions to Wikipedia (with reversions being the moderation event), and, separately, data from the Stack Exchange platform, specifically the top 3 by number of users: Stack Overflow, Super User and Ask Ubuntu.

Wikipedia. In the Wikipedia scenario, we assume editors are anonymous and are penalized via callbacks when their edits are reverted.

Wikipedia (across all languages) has an edit rate of 18 edits per second (average rate across April 2023–March 2024 [Wik]), well below the throughput our single server can handle for authentication. Data on edit-reversions (our moderation event) is only available for select languages. For the largest, English Wikipedia, the average editor makes around 239 edits a month, and 90% of edit reversals are done within 5 hours of the edit [JNV], so we can safely have callbacks expire after 24 hours. In this scenario, we think it's reasonable to require clients to scan for callbacks at least once every 24 hours, and we can let client accounts expire after 1 year of inactivity.

Using the peak month statistic, we expect $239 \cdot 12 = 2{,}868$ edits, which, with 39,711 active users, gives us a total of 113,891,148 callbacks. $\mathsf{bb}_{\mathsf{cb}}$, $\mathsf{bb}_{\mathsf{obj}}$, and the list of revealed serial numbers would need a capacity of at least 2^{26}. For this configuration, each authentication takes 347ms of client computation and each pending edit made by a client adds a cost of at most 585ms per day to scan, with that cost decreasing if we batch scans. Increasing batch size, however, imposes a cost on inactive clients, who must pay that scan cost even if they did not make an edit.

We believe these numbers are reasonable, especially since client computation can be done in parallel to drafting the edit,

and the drafting time will take substantially longer (perhaps upwards of 30 seconds). We note that our numbers are an overestimation of the system requirements, as most edits are (non-anonymous) editing bots [PCL+07].

Stack Exchange. Stack Exchange is one of the largest question-and-answer website with over 26 million users registered on their most popular sub site, StackOverflow [sta]. In each of these sites, users may post questions, answer them, and earn achievements. Unfortunately, publicly available data for Stack Exchange is more limited than data about Wikipedia. Data we retrieved from the statistics API is shown below. We do not have moderation events or active users. Instead, the API gives us total users, and the number of questions, answers, and badges per minute, which we can use as a stand-in for moderation events.

Site:	StackOverflow	SuperUser	AskUbuntu
Answers/min	6.23	0.29	0.28
Badges/min	4.24	0.09	0.07
Questions /min	2.86	0.06	0.06

Similarly to the Wikipedia case study, we set cutoff time to be one year , scan time to be one day, and expiry to be one day. If we assume all three events generate callbacks, then the capacities needed for Stack Overflow, Super User and Ask Ubuntu are 2^{23}, 2^{18}, and 2^{18}, respectively. Authentication takes 339ms or less (325ms for Super User and Ask Ubuntu) of client computation and each pending edit made by a client adds a cost of at most 585ms (563ms for Super User and Ask Ubuntu) per day to scan per callback, with that cost decreasing substantially with batching. Since the server load is less than that of Wikipedia, again, a single lightweight server can handle all client interactions. Both from the client and server side, zk-promises can practically be deployed to support Stack Exchange traffic.

7 Conclusion

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

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

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

A second consequence of this model is the possibility of having zk-objects themselves make callbacks. Currently, in our prototype applications, the caller is identified by a signature, but this can easily 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.

Acknowledgements We would like to thank Hal Triedman for his insights into the Wikipedia moderation, Christina Garman for useful comments on an earlier draft, and the anonymous reviewers for their helpful discussion of encrypted callback mechanics and deployment considerations.

Ethics and OpenScience

Ethics

Anonymous speech raises a number of questions, both on when to allow it and how to ensure anonymity protections, when offered, are sound. Moderation tools, similarly, raise questions for what policies are appropriate. Our goal here is simply to provide more options.

Open Science

In keeping with the USENIX open science policy, the artifacts (codebase) is available with the stable URL here: https://zenodo.org/records/14728077.

References

[Act] Actix web. URL: https://actix.rs/.

[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/.

[Atw09] Jeff Atwood. New question / answer rate limits, Feb 2009. URL: https://stackoverflow.blog/2009/02/21/new-question-answer-rate-limits/.

[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 proof-carrying data. Cryptology ePrint Archive, Report 2012/095, 2012. https://eprint.iacr.org/2012/095.

[BCG+14] Eli Ben-Sasson, Alessandro Chiesa, Christina Garman, Matthew Green, Ian Miers, Eran Tromer, and Madars Virza. Zerocash: Decentralized anonymous payments from bitcoin. In *2014 IEEE Symposium on Security and Privacy*, pages 459–474. IEEE Computer Society Press, May 2014. doi:10.1109/SP.2014.36.

[BCG+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.

[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.

[CGH09] Scott E. Coull, Matthew Green, and Susan Hohenberger. Controlling access to an oblivious database using stateful anonymous credentials. In Stanislaw Jarecki and Gene Tsudik, editors, *PKC 2009*, volume 5443 of *LNCS*, pages 501–520. Springer, Heidelberg, March 2009. doi:10.1007/978-3-642-00468-1_28.

[Cha82] David Chaum. Blind signatures for untraceable payments. In David Chaum, Ronald L. Rivest, and Alan T. Sherman, editors, *CRYPTO'82*, pages 199–203. Plenum Press, New York, USA, 1982.

[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.

[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+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.

[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.

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

[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.

[OWWB20] Alex Ozdemir, Riad S. Wahby, Barry Whitehat, and Dan Boneh. Scaling verifiable computation using efficient set accumulators. In Srdjan Capkun and Franziska Roesner, editors, *USENIX Security 2020*, pages 2075–2092. USENIX Association, August 2020.

[PCL+07] Reid Priedhorsky, Jilin Chen, Shyong (Tony) K Lam, Katherine Panciera, Loren Terveen, and John Riedl. Creating, destroying, and restoring value in wikipedia. In *Proceedings of the 2007 ACM international conference on supporting group work*, pages 259–268, 2007.

[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.

[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.

[RWGM23] Michael Rosenberg, Jacob White, Christina Garman, and Ian Miers. zk-creds: Flexible anonymous credentials from zk-snarks and existing identity infrastructure. In *2023 IEEE Symposium on Security and Privacy (SP)*, pages 790–808, 2023. doi:10.1109/SP46215.2023.10179430.

[sta] URL: https://stackexchange.com/sites#users.

[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.

[Tha23] Justin Thaler. Proofs, Arguments, and Zero-Knowledge. 2023. URL: https://people.cs.georgetown.edu/jthaler/ProofsArgsAndZK.pdf.

[TKPS21] Ioanna Tzialla, Abhiram Kothapalli, Bryan Parno, and Srinath Setty. Transparency dictionaries with succinct proofs of correct operation. Cryptology ePrint Archive, Report 2021/1263, 2021. https://eprint.iacr.org/2021/1263.

[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+22] Alex Luoyuan Xiong, Binyi Chen, Zhenfei Zhang, Benedikt Bünz, Ben Fisch, Fernando Krell, and Philippe Camacho. VERI-ZEXE: Decentralized private computation with universal setup. Cryptology ePrint Archive, Report 2022/802, 2022. https://eprint.iacr.org/2022/802.

[XF14] Li Xi and Dengguo Feng. Farb: Fast anonymous reputation-based blacklisting without ttps. *Proceedings of the 13th Workshop on Privacy in the Electronic Society*, 2014.

URL: https://api.semanticscholar.org/CorpusID:11490543.

[ZCa19] ZCash. What is jubjub?, 2019. URL: https://web.archive.org/web/20221104080552/https://z.cash/technology/jubjub/.

A Security definition and proof sketch

We will now present a proof sketch that our construction realizes the ideal functionality $\mathcal{F}_{\mathsf{zkpr}}$ in Figure 4 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. We assume all parties, in the real protocol, operate over anonymous connections (e.g., Tor) We also assume all parties have access to the same view of the bulletin board.

The simulator Sim first runs Setup and gets the necessary trapdoors for the simulation and extraction of zero-knowledge proofs in Setup. Sim maintains a table mapping objects in its simulated real-world protocol to their ideal counterparts in $\mathcal{F}_{\mathsf{zkpr}}$ 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 $\mathsf{bb}_{\mathsf{cb}}$, 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 $(\mathsf{tik}, \mathsf{args})$ from $\mathsf{bb}_{\mathsf{cb}}$ during ScanOne. Firstly, the tuple could only appear in $\mathsf{bb}_{\mathsf{cb}}$ if its signature is valid, i.e., the party who ran Call knows tik's secret key. In addition, $(\mathsf{tik}, \mathsf{args})$ is only applied if tik appears in the user's cbList, which, in turn, can only happen if tik was given to $\mathcal{P}_i$ during ExecMethodAndCreateCallback. Thus, the calling party must know the secret key of the tik given at ExecMethodAndCreateCallback. This occurs only if they are the same party (or, the party opposite $\mathcal{P}_i$ during ExecMethodAndCreateCallback was corrupted). Finally, we note that the signature check of σ' 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 ExecMethodAndCreateCallback**.** In the real world, callbacks are created by ExecMethodAndCreateCallback. In order to issue a ticket, ExecMethodAndCreateCallback rerandomizes $\mathsf{pk}_{\mathsf{spid}}$ and stores it as tik along with the encryption key and method ID in obj.cbList. The user then posts the new object to $\mathsf{bb}_{\mathsf{obj}}$, 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 $\mathsf{Call}(\mathsf{tik}, \mathsf{args})$ 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 $\mathsf{meth}(\mathsf{obj}, \mathsf{args}, x)$ and deleted the associated tik from cbList, (2) ignored the call and deleted tik from cbList, (3) or ignored the call and left tik in cbList. However, the latter two cases may only happen if $\mathsf{tik} \notin \mathsf{bul}_{\mathsf{cb}}$, which contradicts the fact that $\mathsf{Call}(\mathsf{tik}, \mathsf{args})$ was called at time t. Thus, $\mathsf{meth}(\mathsf{obj}, \mathsf{args}, x)$ was called, and the current object state reflects the called callback.

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.

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

$\mathsf{ExecMethodAndCreateCallback}(\mathsf{pp}, \mathsf{obj}, \mathsf{pk}_{\mathsf{spid}}, \mathsf{meth}', x) \to (\mathsf{obj}', \pi, \mathsf{cbData}, \mathsf{aux})$**.** Let obj be the user object, let $\mathsf{pk}_{\mathsf{spid}}$ be the verifying key for service provider ID spid for an

Setup(serviceProviders)

1: authorizedCallers := serviceProviders
2: curTime := 0
3: userObjs := {} // Default value is app-specific
4: createdCbs := {} // Callbacks created but uncalled
5: pendingCbs := {} // Callbacks called but pending ingestion and execution

ExecAndMakeCb($\mathcal{P}_j$,creationMeth,cbMeth, $\mathsf{aux}_{\mathsf{pub}}$,$\mathsf{aux}_{\mathsf{priv}}$,cutoffTime) from $\mathcal{P}_i$

1: **assert** ∄createdCbs[tik] and ∄pendingCbs[tik]
2: **assert** $\forall(\mathcal{P}_i$,callTime) ∈ pendingCbs:
3: callTime > cutoffTime // All old calls have been processed
4: tik $\leftarrow\$ \{0,1\}^\lambda$
5: createdCbs[tik] := $(\mathcal{P}_i,\mathcal{P}_j$,cbMeth,$\mathsf{aux}_{\mathsf{pub}}$)
6: userObjs[$\mathcal{P}_i$].creationMeth(
7: tik,curTime,$\mathsf{aux}_{\mathsf{pub}}$,$\mathsf{aux}_{\mathsf{priv}}$,cutoffTime)
8: **send** ("CreatedCb",$\mathsf{aux}_{\mathsf{pub}}$) to $\mathcal{A}$
9: **send** ("CreatedCb",tik,cbMeth,$\mathsf{aux}_{\mathsf{pub}}$,cutoffTime) to $\mathcal{P}_j$

IncrTime from $\mathcal{A}$

1: curTime += 1

Call(tik,args) from $\mathcal{P}_j$

1: **assert** $\mathcal{P}_j$ ∈ authorizedCallers
2: $(\mathcal{P}_i,\mathcal{P}'$,meth,aux) := createdCbs[tik]
3: **assert** $\mathcal{P}' = \mathcal{P}_j$
4: **delete** createdCbs[tik]
5: pendingCbs[tik] := $(\mathcal{P}_i$,meth,args,aux,curTime)
6: **send** ("Called",tik,curTime) to $\mathcal{A}$
7: **send** ("Called",tik,args,curTime) to $\mathcal{P}_i$

IngestCall(tik) from $\mathcal{P}$

1: $(\mathcal{P}$,meth,args,aux,callTime) := pendingCbs[tik]
2: userObjs[$\mathcal{P}_i$].meth(tik,args,curTime,callTime,aux
3: **delete** pendingCbs[tik]
4: **send** "Ingested" to $\mathcal{A}$

Figure 4: A simple ideal functionality $\mathcal{F}_{\mathsf{zkpr}}$ 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.

EUF-CMA-secure signature scheme Σ, and let s be the commitment randomness used in that objects commitment in $\mathsf{bb}_{\mathsf{obj}}$. Let meth' be the method the user wishes to create a callback for. Let $x = (t,\mathsf{curTime})$, where t is the minimum value for lastFullScanTime that the service provider will accept and curTime be the current global time.

The ExecMethodAndCreateCallback algorithm proceeds as follows:

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

The information that the user sends to the service provider does not de-anonymize the user as it contains no multi-user or long term identifier. It contains a zero-knowledge proof, π, the cryptographic commitment $\mathsf{com}'_{\mathsf{obj}}$, 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,$\mathsf{sk}_{\mathsf{spid}}$,$\mathsf{obj}'$,$\pi$,cbData,aux). Let $\mathsf{sk}_{\mathsf{spid}}$ be the service provider's signing key. The service provider performs the following:

1. Verify obj' appears on $\mathsf{bb}_{\mathsf{obj}}$, i.e., that the callback was created;
2. Verify the proof π with respect to inputs cbData, curTime, and obj'; and
3. Unpack aux and verify $\mathsf{cbData.com}_{\mathsf{entry}} = \mathsf{Com}(\mathsf{tik},\mathsf{exp},k,\mathsf{meth}';s_{\mathsf{entry}})$
4. Verify that tik has never been used before in a callback initiated by this service provider

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

1. Clones the obj to a new one obj' and pick a fresh serial number $\mathsf{obj}'.\mathsf{sn}$. If $h_{\mathsf{cb}}^{(\mathrm{old})} = h_{\mathsf{cb}}$ (i.e., the beginning of the scanning process), update $\mathsf{obj}'.\mathsf{loopStartTime} := \mathsf{curTime}$.
2. Let entry $=$ (tik, exp, k, meth) represent the current entry in obj.cbList. If (tik, args, t) $\in \mathsf{bb}_{\mathsf{cb}}$ for some args and $t <$ exp, absorb the callback: $\mathsf{obj}' := \mathsf{meth}(\mathsf{obj},\mathsf{curTime},m,x)$, where $m = \mathsf{Dec}_k(\mathsf{args})$. If $\mathsf{tik} \notin \mathsf{bb}_{\mathsf{cb}}$ and exp is in the future, then leave entry in $\mathsf{obj}'.\mathsf{cbList}$. In any other case, delete entry from $\mathsf{obj}'.\mathsf{cbList}$. Finally, update $h'^{(\mathrm{new})}_{\mathsf{cb}} = H(h^{(\mathrm{new})}_{\mathsf{cb}},\mathsf{entry})$ if the entry was left in, and $h'^{(\mathrm{new})}_{\mathsf{cb}} = h^{(\mathrm{new})}_{\mathsf{cb}}$ otherwise.
3. If $h'^{(\mathrm{old})}_{\mathsf{cb}} = h_{\mathsf{cb}}$ this is the last step of a scanning process. Update $\mathsf{obj}'.\mathsf{lastFullScanTime} := \mathsf{obj}.\mathsf{loopStartTime}$.
4. Compute a zero-knowledge proof π of R_{settle}.
5. Let $\mathsf{cbData} = (\mathsf{com}_{\mathsf{entry}},\mathsf{obj.sn})$
6. The user sends $(\pi,\mathsf{com}'_{\mathsf{obj}},\mathsf{cbData})$ to $\mathsf{bb}_{\mathsf{obj}}$

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

In $\mathsf{VerifyMethodExec}_{\mathsf{settle}}$, the maintainer receives the same payload. The behavior is identical to above, with the only change being that it also uses a $\mathsf{bb}_{\mathsf{cb}}$ representative as public input for π verification.

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

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.

Merkle trees. We may represent our object and used callback sets $\mathsf{bb}_{\mathsf{obj}}$ and $\mathsf{bb}_{\mathsf{cb}}$ 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

$$\left(\begin{array}{l}\mathsf{com}'_{\mathsf{obj}},\mathsf{com}_{\mathsf{entry}},\mathsf{sn},t,\mathsf{curTime},\mathsf{bb}_{\mathsf{obj}};\\ \mathsf{com}_{\mathsf{obj}},s_{\mathsf{obj}},s'_{\mathsf{obj}},\\ s_{\mathsf{entry}},\mathsf{tik},\mathsf{exp},k,\mathsf{meth}'\end{array}\right):$$
$$\mathsf{com}_{\mathsf{obj}} \in \mathsf{bb}_{\mathsf{obj}}$$
$$\mathsf{com}_{\mathsf{obj}} = \mathsf{Com}(\mathsf{obj};s_{\mathsf{obj}})$$
$$\mathsf{com}'_{\mathsf{obj}} = \mathsf{Com}(\mathsf{obj}';s'_{\mathsf{obj}})$$
$$\mathsf{com}_{\mathsf{entry}} = \mathsf{Com}(\mathsf{tik}\|\mathsf{exp}\|k\|\mathsf{meth}';s_{\mathsf{entry}})$$
$$\mathsf{obj.lastFullScanTime} \geq t$$
$$\mathsf{obj}'.\mathsf{lastCreated} = \mathsf{curTime}$$
$$\mathsf{obj.sn} = \mathsf{sn}$$
$$\Phi_{\mathsf{create}}(\mathsf{obj},\mathsf{obj}',\mathsf{meth}',\mathsf{curTime}) = 1$$
$$\mathsf{obj}'.h_{\mathsf{cb}} = H(\mathsf{obj}.h_{\mathsf{cb}},(\mathsf{tik},\mathsf{exp},k,\mathsf{meth}'))$$
$$\mathsf{obj}.h^{(\mathrm{old})}_{\mathsf{cb}} = \mathsf{obj}.h_{\mathsf{cb}}$$

Figure 5: The $R^{\Phi_{\mathsf{create}}}_{\mathsf{create}}$ relation

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

Figure 6: The $R^{\Phi_{\mathsf{ingest}}}_{\mathsf{ingest}}$ relation

Circuit	Single Server	Distributed		
Epoch Capacity:	*Unlimited*	2^{16}	2^{32}	2^{64}
Number of Constraints				
`ShowAuthMakeCB`	27,503	24,252	28,620	37,356
`ScanInc`	55,435	50,712	59,448	76,920

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

suffices to download a *frontier* of the append-only tree. This permits a communication cost tradeoff of logarithmic to linear, depending on privacy requirements [RWGM23].

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

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

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

D Number of Constraints

Table 2 lists the number on constraints in `ShowAuthMakeCB` and `ScanInc` 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 `ScanInc` circuit scans for a single callback.

Batch Size	Single Server	Distributed		
Epoch Capacity:	*Unlimited*	2^{16}	2^{32}	2^{64}
Number of Constraints				
1	55,435	50,712	59,448	76,920
2	96,448	91,319	104,423	130,631
4	176,464	169,463	191,303	234,983
8	336,496	325,751	365,063	443,687
16	656,560	638,327	712,583	861,095
32	1,296,688	1,263,479	1,407,623	1,695,911
64	2,576,944	2,513,783	2,797,703	3,365,543

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

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

E Bulletin Board Benchmarks

The bulletin board has two segments, one for credentials and another for callbacks. Each credential entry (leaf) is 32 bytes, and each callback entry is $32(i+1)$, where i is the number of callback arguments (assuming each argument is field element).

After downloading both lists (and generating the complement callbacks if needed), users generate Merkle trees on them. The root can be used to verify all users have the same shared view. For our setup, the runtime to insert an entry (leaf) into an incremental Merkle tree with a fixed max depth of 16 and 32 were 0.77ms.

In the decentralized variant, users save the Merkle tree paths relevant to their zk-object for membership/non-membership proofs and may discard the full tree. In the centralized version, we utilize signatures for compact (non-)membership proofs, allowing us to skip more expensive hashing in the circuit. For membership, this requires downloading a 64 byte signature per credential or callback. For non-membership checks for callbacks, each empty range in the complement set also has a 64 byte signature. To prove non-membership, a user proves that their callback is within one of the signed ranges. The size of this set is upper bounded by $1+|\#\text{callbacks}|$.

Concretely, the credential segment of a decentralized variant of the bulletin board with 2^{16} credential entries would be 2.09 MB. The callback segment with the same number of entries (with three arguments) would be 8.39 MB. Adding signatures would add a total of 12.58MB for credentials, callbacks, and the callback complement set.

We note that PIR can be implemented for a client to only receive relevant leaf entries and their proofs to reduce communication cost. However, periodic regeneration/updates is needed as the bulletin board refreshes every epoch.