Passlog: Private Authentication Logging with Public State

Abstract. Account compromise is challenging to prevent completely, and so users need to be able to quickly detect compromise in order to minimize the damage. Authentication logging systems make it possible for a user to fetch a comprehensive list of all logins made to her accounts. However, existing systems provide authentication logging at the expense of allowing the log provider to learn sensitive information or tamper with authentication logs. To address this problem, we present Passlog, a privacy-preserving authentication logging system where one or more parties can run the log service, and any number of parties can audit the log service. In Passlog, the log record state is public, which enables any number of parties to maintain and audit the log service, but does not reveal private information. The challenge is to hide the identity of the user and web service from the log service while still allowing the log service to enforce that every authentication is correctly recorded. We design, implement, and evaluate Passlog to support both a centralized and decentralized log service. Our implementation of Passlog with an auditable log service running on one server with eight CPU cores and a client and relying party running on four cores each executes an authentication in 1025ms.

1. Introduction

Account security is a weak point in many computer systems. The question is not whether user accounts will be compromised, but rather which accounts will be compromised when, how fast the breaches will be discovered, and how much damage will be caused. For example, attackers stole LastPass data by compromising an engineer's personal computer and obtaining credentials [83], an attack that spanned 2.5 months and resulted in LastPass advising all customers to change their passwords. Accounts are challenging to secure in part because they involve humans, who have many accounts in many different contexts and very limited capacity to remember and update authentication information, yet great capacity to cause devastating consequences through a single mistake.

We argue that user authentication systems must be designed not just to *resist* compromise, but also to facilitate *detection and recovery*. Unfortunately, the best widely deployed alternatives today require us to pick one. The gold standard in resisting compromise is FIDO2/passkeys—authenticator devices that can be used with many services, employ secure hardware, and do not require delegating any security-critical credential secrets to the cloud. Yet discovering whether and where a FIDO2 authenticator has been misused is nearly impossible. Conversely, single sign-on systems such as OpenID (e.g., sign in with Google) provide comprehensive logging across many services, yet increase the attack surface by trusting cloud providers and often storing web cookies far more privileged than the user's immediate intent to access one service.

Larch [29] was the first proposal to achieve the best of both worlds—combining FIDO2 and its benefits with comprehensive, centralized logging of all accesses made to any of a user's accounts across all contexts (personal and work). While larch had the benefit of working with existing, unmodified services supporting existing authentication standards (so-called *relying parties*), it paid a price for backwards compatibility. An attacker who controls both the larch log provider and a relying party can link a user across accounts, harming privacy. One consequence is that log providers must maintain secrets, and there is no way to store log records on a widely replicated and publicly auditable ledger.

This paper answers the question: what's the best design point for user authentication if we *don't* need unmodified relying parties. We presents *Passlog*, a decentralized authentication logging system where one or more parties can run the logging service, and any number of parties can verify that the log is being maintained correctly. None of the parties that run or audit the log service learn sensitive information or user credentials—Passlog protocol messages and public state only reveal that a particular operation is taking place at a particular time. In this way, adding parties to the system only *improves* integrity and does not degrade privacy or account security.

We design, implement, and evaluate Passlog in two primary modes of deployment: (1) a centralized log service with many auditors that ensure that the log service is behaving correctly, and (2) a decentralized log service where multiple parties play the role of the log service and auditors can still verify correct behavior. With one eight-core log server and four-core clients and relying parties, an authentication takes 1025ms. With two eight-core log servers with 256GB RAM each, privately fetching a log record from a store of 10M records takes 126s (this can be performed in the background).

1.1. Contributions

We now describe the challenges and techniques in designing and building Passlog.

Challenge: Enforcing correct log structure with privacy. The log service needs to be able to ensure that clients are maintaining the structure of their log records correctly, even

if the clients are malicious. In particular, every authentication, whether made by an honest or malicious, should always be recorded, and a malicious client should not be able to tamper with other clients' log records. At the same time, for privacy, the log server should not be able to learn any information about the mapping of log records to clients. These goals are fundamentally in tension.

Technique: Private verifiable sequences. To resolve this tension, we introduce a new primitive, private verifiable sequences. Private verifiable sequences allow different clients to read values from and write values to different sequences in a public key-value store. For privacy, the public state hides which values belong to which clients. To guarantee that client sequences are maintained correctly, we also ensure that clients are only able to append to their own sequences. We describe our construction, which leverages zero-knowledge proofs and authenticated data structures, in Section 3.

Challenge: Verifying encrypted log contents. We not only need to make sure that the log structure is maintained correctly; we also need to ensure that log records correctly encrypt each authentication. A relying party should only authenticate a client if it is convinced that the log service has received a log record that encrypts some authentication-specific string under the user's original encryption key. This property should hold even if the client is malicious and constructing its authentication request to try to circumvent logging. At the same time, we need to ensure that the relying party cannot link a user across different accounts even if it colludes with the log service.

Technique: Zero-knowledge proofs of well-formed records. To address this problem, we show how clients can generate zero-knowledge proofs that convince relying parties that their log records are well-formed. The log records need to be well-formed relative to per-user state. To ensure that this per-user state cannot be used to link a user across accounts, we show how a client can send relying parties a commitment to this state at registration time and then prove that subsequent authentications are valid relative to this commitment.

Challenge: Detecting log service misbehavior. One of our requirements for Passlog is that it should be publicly auditable—if the log service misbehaves, there should be public evidence that could cause users to lose trust in the log service. We need to ensure that other parties can detect if (1) the log service participates in an authentication without storing the corresponding log record, or (2) the log service tampers with existing authentication records.

Technique: An accountable log design. We address both types of misbehavior through their own mechanism. To catch a log service that tries to not record an authentication, we show how relying parties can store a small amount of data that allow clients can retrieve to verify that their log records are being maintained correctly. To detect a log service that tries to tamper with existing log records, we show how to leverage techniques from authenticated data structures while providing strong user privacy.

Challenge: Privately retrieving log records. While our private verifiable sequences primitive hides the mapping of log entries to users, read access patterns have the potential to reveal that two log entries belong to the same user. For example, if the log provider receives multiple back-to-back requests for log records, it could infer that a user is fetching her log contents and link all of the requested log records to the same user. This information in turn could be used to link a user across accounts.

Technique: Authenticated private information retrieval for sparse Merkle trees. To prevent this leakage, we show how to fetch log records, along with membership and non-membership proofs, with strong privacy and integrity (the problem of authenticated private information retrieval [24]). We need to store authentication records in a sparse Merkle tree to allow auditors to detect log misbehavior. However, because the client does not know the layout of the tree, fetching these membership and non-membership proofs would naively require d rounds of communication, where d is the tree depth (256 in our implementation). Instead, we show how clients can use an incremental distributed point function [12] to directly encode the Merkle path they are fetching—this solution only requires one round of communication.

Limitations. Passlog requires relying parties to update their software. Section 8 describes how Passlog can be implemented via a new signature type in the FIDO2 specification, thereby leveraging widespread support of FIDO2 in browsers and facilitating incremental adoption. Also, while Passlog protocol messages hide user identities, the log provider could use other information (e.g., a client's IP address) to link a user across accounts. A client could hide its IP address via an anonymizing proxy like Tor [34] or iCloud private relay [75].

2. Design overview

A Passlog deployment involves the following four types of parties: users, relying parties, the log service, and auditors.

Users. In our setting, millions of users authenticate to different web services, or relying parties, regularly. Users rely on multiple client devices (e.g., a phone and laptop) that store constant-size secret state. Devices can synchronize secret state via existing browser profile synchronization mechanisms. Similarly, devices share information on audits performed on users' authentication history, such as last audit time.

Relying parties. Relying parties are online services that manage client accounts (e.g., Bank of America or Amazon). Users register with a relying party to create an account and then later authenticate to the account.

Log service. The log service is responsible for maintaining encrypted authentication records. The log service both participates in authentications and manages the store of records (the log). At a high level, the client queries the log service for each authentication, and if the authentication request is well-formed, the log service records the authentication.

The log services make the log of records public for anyone to audit. To ensure that users can access their accounts, the log service should be highly available, and to ensure that

old records on the log are not being deleted, the log service should provide strong integrity guarantees. Notably though, the log service learns no private information via protocol messages, such as which account a user is authenticating to or which accounts belong to a particular user.

In Section 5, we discuss two settings for instantiating the log service:

A centralized log service. The client interacts with a single log provider that manages and periodically publishes the log state, which consists of encrypted records. Auditors can check that this log state is maintained correctly (similar to existing transparency logs [19], [44], [51], [53], [57], [59]). In this scenario, the log provider would use standard replication techniques [49], [64] for strong availability guarantees.

A decentralized log service. The client interacts with multiple nodes that together run a Byzantine fault-tolerant consensus protocol [17], [50] to agree on the log state. As long as some fraction of the nodes are honest and online, the system provides strong availability and integrity guarantees.

Auditors. Auditors are users or organizations that monitor the log state and check that updates to the log state are correct and consistent (e.g., the log service is not deleting old records). If an auditor detects misbehavior, it can report evidence of this misbehavior. This evidence could cause users to stop trusting the log provider. Auditors can also choose to help clients fetch log records.

2.1. Protocol flow

Background. We use authenticated encryption and a commitment scheme. A commitment cm to a value $x \in \{0,1\}^*$ with opening $r \in \{0,1\}^{256}$ is hiding and binding—a computationally bounded adversary cannot learn x from cm without the opening r (hiding) or, given r, find another opening such that cm opens to $x \neq x'$ (binding). We can commit to x with opening r by simply computing $\mathsf{Commit}(x, r)$ as $H(x||r)$ where H is a cryptographic hash function.

We describe the high-level flow for Passlog below and in Figure 1 (see Section 4 for the full protocol). Note that, unlike single sign-on systems (e.g., "Sign-in with Google") or larch [29], Passlog does not require clients to enroll with the log service—the log service maintains no per-user state.

Step 1: Setup. The client starts by sampling a secret key and fetching the log service's public key. (We implement this "logical" secret key with multiple keys, but describe it as a single key for simplicity here.)

Step 2: Register with a relying party. When the client first creates an account with a relying party, it sends the relying party state for verifying future authentications. This state includes (1) the log service's public key and (2) a commitment to the client's secret key. The relying party stores this information to verify subsequent authentications. As in prior work [29], we only provide strong security and logging guarantees for a particular relying party if the client registers honestly.

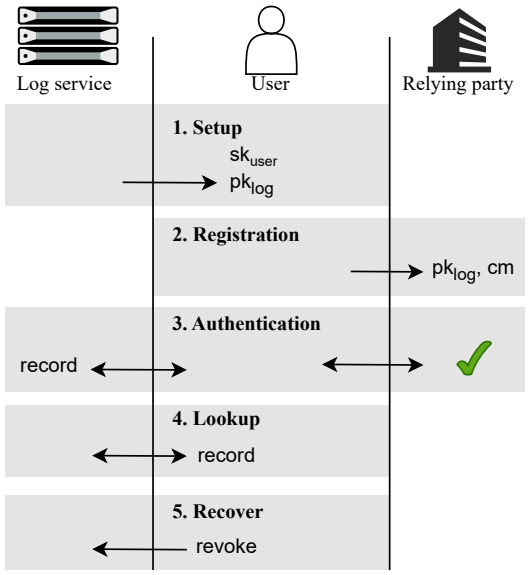

Figure 1: We outline the five user operations in Passlog. During setup, the client samples a secret key and fetches a public key from the log service. To register with a relying party, the client sends a commitment to its secret key, along with the log's public key, to the relying party. At authentication time, the client runs a protocol with the log service and the relying party where, at the end of the protocol, the log service obtains an encrypted record, and the relying party is convinced that the authentication is valid. The client can then later retrieve authentication records from the log service. If the client detects any suspicious activity, it can send a revocation request to the log service.

Step 3: Authenticate to a relying party. In order to authenticate to a relying party, the client must convince the relying party that it knows the secrets it committed to at registration and that the authentication has been correctly logged. At a high level, authentication proceeds in two steps:

1) The client generates an encrypted log record and uses our *verifiable private sequences* primitive to add it to its sequence of records stored at the log service. If the log service successfully stores the encrypted log record, it signs the record, along with a relying-party-chosen challenge.
2) The client proves to the relying party that the signed log record is "well-formed" relative to the relying party's identity and the secret key that the client committed to at registration time.

The relying party can optionally store the encrypted log record to make it possible to detect log service misbehavior.

Step 4: Look up authentication records. The user's client can periodically request log records by querying the log service and an auditor. These lookups can take place in the background to detect suspicious authentications. For example, the user's laptop can periodically query the log and check for discrepancies between the returned log records and authentications made on a user's registered devices. Users can also manually inspect the log contents.

Step 5: Recover after compromise. If a user learns of one or more suspicious authentications while reviewing log records, she can lock down all of her accounts by notifying the log service. Locking down the accounts prevents an attacker from accessing more of the user's accounts. Notably, a user does not need to remember all of the accounts that she has registered with Passlog to freeze them all at once. The user can then use an offline recovery mechanism to gain access to the frozen accounts (see Section 8). While this is useful for the accounts that the attacker did not access before the freeze, the attacker could have changed the credentials for the accounts accessed before the freeze, essentially locking the user out. For these accounts, the user would need to work with the relying party directly to determine the best course of action to recover from compromise.

2.2. Security and privacy properties

We describe the Passlog security and privacy properties.

Security against a malicious log. A malicious log service cannot authenticate to a user's account. In this way, users' authentication credentials do not leave the client.

Privacy against a malicious log. A malicious log service does not learn any information about a user's authentication history, beyond timing, via protocol messages. In particular, an attacker that controls the log service cannot learn which relying parties users are authenticating to, which records correspond to the same user or relying party, or even if a user is authenticating with the log service for the first time.

Unlinkability against a colluding log and relying parties. An attacker that controls the log service and an arbitrary number of relying parties cannot link a user across accounts using protocol messages. For example, if Google runs a log service, it cannot use protocol messages to link one user across two distinct Google accounts.

Audit integrity against a malicious log. If a log service is honest and then compromised at time t, the client will be able to correctly retrieve log records corresponding to authentications before time t. This property follows from the availability and integrity of the append-only log state.

Publicly verifiable proofs of log misbehavior. Users and relying parties can together retroactively prove that the log service misbehaved at authentication time. More specifically, they can prove that a log service signed a log record without correctly logging it. Similarly, auditors can prove that the log service tampered with the log state. The ability to prove log misbehavior raises the bar for an attacker that compromises the log: if the log server is caught misbehaving, users will likely stop trusting the log service and switch to another.

Non-goals. Passlog provides unlinkability against an attacker that controls the log service and relying parties only at the protocol level—a client's IP address could still link a client across accounts. A client could use an anonymizing proxy like Tor [34] or iCloud private relay [75] to prevent the log service from learning its IP address.

The timing, number, and type of requests could also reveal private information. A relying party can also learn the identity of the log service associated with an account. This information can help an attacker narrow down the set of users potentially associated with an account, and so it is helpful for a user to use a log service with many users.

In a deployment with a centralized log service, if the log service denies service, then users will not be able to access their accounts. Standard replication techniques [49], [64] can help protect against unintentional crashes. In a deployment with a decentralized log service, the availability guarantees match those of the underlying Byzantine fault-tolerant consensus protocol (see Section 5.4).

As in prior work [29], Passlog ensures that authentications are logged if the attacker compromises either a client device or the log service. If the attacker gains access to the relying party credentials, then we do not guarantee that every authentication is correctly logged.

As discussed in Section 2.1, an attacker that compromises the client may authenticate to a relying party and change the credentials. In this case, the authentication will be logged correctly, but subsequent authentications with the new credentials will not be recorded.

3. Verifiable private sequences

To build Passlog, we construct a new primitive: verifiable private sequences (VPS). At a high level, our VPS scheme allows a client to write values in a sequence to a public object store and then fetch the values in the sequence from this state. For simplicity, we will describe this primitive in the setting where clients submit requests and a single server manages this public state, although we can also use this primitive in the setting where the role of the server is split across multiple parties. The challenging part in constructing verifiable private sequences is that we need to support reads and writes from many clients, but the public state should *hide the mapping of values to sequences*.

In our VPS scheme, each sequence is associated with a secret key, and each value is located at an address. This address is pseudorandom and deterministic: the client's sequence secret key is necessary to compute sequence addresses. The client can then issue read and write requests to different addresses derived from the secret key. VPS ensures that write requests are "well-formed"—element $i-1$ in the sequence is inserted before element i, and a client must know the sequence secret key in order to write to that sequence.

3.1. Syntax

A VPS scheme is a tuple of efficient algorithms, parameterized by security parameter λ and value space $\mathcal{V}$.

Initialize the state.

- $\mathsf{VPS.Init}(1^\lambda) \to \mathsf{st}$: The server initializes the state.

Commit to the state.

- $\mathsf{VPS.GetCm}(\mathsf{st}) \to \mathsf{cm}_{\mathsf{st}}$: Given the current state, the server outputs a commitment to the state $\mathsf{cm}_{\mathsf{st}}$. The client can use this commitment for subsequent operations.

Create a sequence.

- $\mathsf{VPS.KeyGen}(1^\lambda) \to k$: The client samples a sequence secret key k.

Get an address.

- $\mathsf{VPS.GetAddr}(k, i) \to \mathsf{addr}$: Given a sequence secret key k and index $i \in \mathbb{N}$, the client outputs the corresponding address addr.

Read from a sequence.

- $\mathsf{VPS.Read}(\mathsf{st}, \mathsf{addr}) \to (v, \pi_{\mathsf{inc}})$: Given a client-provided address addr and current state st, the server returns the associated value $v \in \mathcal{V}$ and a proof of inclusion π_{inc}. If addr is not present in st, the server returns $\perp$ and a proof π_{inc} that addr is not present.
- $\mathsf{VPS.VerifyRead}(\mathsf{cm_{st}}, \mathsf{addr}, v, \pi_{\mathsf{inc}}) \to \{0, 1\}$: The client uses the inclusion proof π_{inc} to verify that (addr, v) is present in the state committed to by $\mathsf{cm_{st}}$. If $v = \perp$, then the client can use π_{inc} to verify that (addr, v) is not present in the corresponding state. The client outputs "1" if verification succeeds and "0" otherwise.

Append to a sequence.

- $\mathsf{VPS.ProveAppend}(\mathsf{cm_{st}}, k, i, \mathsf{addr}, \pi_{\mathsf{inc}}) \to \pi_{\mathsf{appd}}$: The client generates a proof that addr can be correctly inserted in the sequence with key k at location i. Generating this proof requires a proof of inclusion π_{inc} of the address for location $i-1$ with key k relative to the state commitment $\mathsf{cm_{st}}$. If $i = 0$ (this is the first insertion), then $\pi_{\mathsf{inc}} = \perp$.
- $\mathsf{VPS.Append}(\mathsf{st_{old}}, \mathsf{addr}, v, \pi_{\mathsf{appd}}) \to \mathsf{st_{new}}$: The server checks if the proof π_{appd} certifies that the address-value pair (addr, v) can be correctly appended to the current state $\mathsf{st_{old}}$. If the check passes, the server adds (addr, v) to the state to generate $\mathsf{st_{new}}$. Otherwise, it returns the unmodified $\mathsf{st_{old}}$.

3.2. Properties

We informally introduce the security and privacy properties of verifiable private sequences here. We formalize the definitions for our construction in Appendix 1.

Completeness. Completeness requires that reads reflect writes. More precisely, consider any sequence of well-formed append and read operations. (Here, well-formed simply means that element $i-1$ is inserted in the sequence before element i.) For any such sequence, if the client writes a value v to a sequence at location i, then subsequently reading from that sequence at location i returns v.

Privacy. Privacy ensures that append requests (and the resulting public state) hide the mapping of append requests to sequences.

Append soundness. Append soundness guarantees that the server can check that client append requests are "well-formed". Client appends must:

- Append to sequences where the client knows the corresponding secret key.
- Append to sequences in order (i.e., for $i > 0$, the client must append element $i-1$ before element i).

Read soundness. Read soundness requires that for a given commitment to the state $\mathsf{cm_{st}}$ and address addr, an attacker cannot produce two values v, v' with inclusion proofs $\pi_{\mathsf{inc}}, \pi'_{\mathsf{inc}}$ where $v \neq v'$ that both verify.

Limitation: Series of reads leak access patterns. Each value is associated with a random-looking but deterministic address addr, and so the server learns when two requests are for the same address (the access pattern). This leakage could help a malicious server use append and read requests to infer which addresses map to the same sequence. We discuss how to hide read addresses using private information retrieval [21] in Section 5.2.

3.3. Building blocks

Our construction uses a pseudorandom function where, for security parameter λ, $\mathsf{PRF} : \{0,1\}^\lambda \times \mathbb{N} \to \{0,1\}^\lambda$. It also uses zero-knowledge proofs and authenticated data structures.

Zero-knowledge proofs of knowledge. Zero-knowledge arguments of knowledge [38] allow a prover to prove that it knows some witness w such that for some circuit $\mathcal{C}$ and instance x, $\mathcal{C}(x, w) = 1$ without revealing w to the verifier. We consider non-interactive zero-knowledge proofs of knowledge in the random-oracle model [9], [11], [31]. For simplicity, throughout the paper we will refer to these as "zero-knowledge proofs." A zero-knowledge proof system ZKPoK for circuit $\mathcal{C}$, public input x, and witness w is defined as:

- $\mathsf{ZKPoK.Prove}(\mathcal{C}, x, w) \to \pi$: Output a proof π.
- $\mathsf{ZKPoK.Verify}(\mathcal{C}, x, \pi) \to \{0, 1\}$: Output 1 if π verifies with respect to x, and 0 otherwise.

We rely on the correctness, soundness, zero-knowledge, and proof of zero-knowledge properties of ZKPoK in our VPS construction. Our implementation uses a PLONK-style proof system [36], and so requires a trusted setup (the parameters generated from the setup are implicit arguments to the prove and verify algorithms). In a real deployment, these parameters could be generated via a multi-party computation [58].

Authenticated data structures. Authenticated data structures [5], [66], [77], [78], [79], [80] make it possible to prove that a key-value store is being maintained according to some invariants. More specifically, a server holding a key-value store can generate and publish a concise commitment to its state. Authenticated data structures support short proofs—for our VPS scheme, we need support for inclusion and exclusion [44], [51], [53], [59]. Clients can check these proofs relative to the commitment. An authenticated data structure ADS is defined by the following tuple of algorithms with respect to value space $\mathcal{V}$.

- $\mathsf{ADS.Init}() \to \mathsf{st}$: Output an empty key-value store.
- $\mathsf{ADS.Commit}(\mathsf{st}) \to \mathsf{cm_{st}}$: Output a state commitment.
- $\mathsf{ADS.Append}(\mathsf{st_{old}}, \mathsf{addr}, v) \to \mathsf{st_{new}}$. Given old state $\mathsf{st_{old}}$, address $\mathsf{addr} \in \{0,1\}^*$, and value $v \in \mathcal{D}$, output a new state $\mathsf{st_{new}} = \mathsf{st_{old}} \cup \{(\mathsf{addr}, v)\}$.

- $\mathsf{ADS.Read}(\mathsf{st}, \mathsf{addr}) \rightarrow (v, \pi_{\mathsf{inc}})$: If $(\mathsf{addr}, v) \in \mathsf{st}$, output v and a proof of inclusion π_{inc}. Otherwise, output $\perp$ for v and a proof of non-inclusion.
- $\mathsf{ADS.Verify}(\mathsf{cm_{st}}, \mathsf{addr}, v, \pi_{\mathsf{inc}}) \rightarrow \{0,1\}$: Output 1 if π_{inc} certifies that (addr, v) is an entry in the state represented by $\mathsf{cm_{st}}$, and 0 otherwise (the argument v is optional).

We rely on the correctness and soundness properties of ADS in constructing VPS. In Section 5, we describe our log design that supports the properties we need for Passlog, which include inclusion and exclusion soundness.

3.4. Construction

At a high level, the public state consists of an authenticated data structure storing address-value pairs. The address for location i of the sequence with secret key k is $\mathsf{addr}_i \leftarrow \mathsf{PRF}(k, i)$. This way, the addresses look random without the secret key and do not reveal anything about the relationship to other sequences. To ensure that clients are only appending to the end of sequences where they know the corresponding secret key, the client generates a zero-knowledge proof that its append request is well-formed. To insert item i at address $\mathsf{PRF}(k, i)$, the client proves that it knows an inclusion proof for address $\mathsf{PRF}(k, i-1)$. We present our construction in Figure 2.

Our VPS construction. We instantiate our VPS construction with a security parameter λ, zero-knowledge proof system ZKPoK, authenticated data structure ADS, and pseudorandom function PRF.
Let $\mathcal{C}$ be the circuit that takes as input a public instance $(\mathsf{addr}_i, \mathsf{cm_{st}})$ and private witness $(k, i, \pi_{\mathsf{inc}})$ and outputs 1 if the below conditions are true and 0 otherwise:

- $\mathsf{addr}_i = \mathsf{PRF}(k, i)$
- If $i > 0$:
  - $\mathsf{addr}_{i-1} = \mathsf{PRF}(k, i-1)$
  - $\mathsf{ADS.Verify}(\mathsf{cm_{st}}, \mathsf{addr}_{i-1}, \cdot, \pi_{\mathsf{inc}}) = 1$

$\mathsf{VPS.Init}(k) \rightarrow \mathsf{st}$
- Output $\mathsf{st} \leftarrow \mathsf{ADS.Init}()$.

$\mathsf{VPS.KeyGen}() \rightarrow k$
- Output $k \xleftarrow{R} \{0,1\}^{\lambda}$.

$\mathsf{VPS.GetCm}(\mathsf{st}) \rightarrow \mathsf{cm_{st}}$
- Output $\mathsf{cm_{st}} \leftarrow \mathsf{ADS.Commit}(\mathsf{st})$.

$\mathsf{VPS.GetAddr}(k, i) \rightarrow \mathsf{addr}$
- Output $\mathsf{addr} \leftarrow \mathsf{PRF}(k, i)$.

$\mathsf{VPS.Read}(\mathsf{st}, \mathsf{addr}) \rightarrow (v, \pi_{\mathsf{inc}})$
- Output $(v, \pi_{\mathsf{inc}}) \leftarrow \mathsf{ADS.Read}(\mathsf{st}, \mathsf{addr})$.

$\mathsf{VPS.VerifyRead}(\mathsf{cm_{st}}, \mathsf{addr}, v, \pi_{\mathsf{inc}})$
- Output $\{0,1\} \leftarrow \mathsf{ADS.Verify}(\mathsf{cm_{st}}, \mathsf{addr}, v, \pi_{\mathsf{inc}})$.

$\mathsf{VPS.ProveAppend}(\mathsf{cm_{st}}, k, i, \mathsf{addr}, \pi_{\mathsf{inc}}) \rightarrow \pi_{\mathsf{appd}}$
- Output $\pi_{\mathsf{appd}} \leftarrow \mathsf{ZKPoK.Prove}(\mathcal{C}, \mathsf{cm_{st}}, w)$ for $w = (k, i, \pi_{\mathsf{inc}})$ and $x = (\mathsf{addr}, \mathsf{cm_{st}})$.

$\mathsf{VPS.Append}(\mathsf{st_{old}}, \mathsf{addr}, v, \pi_{\mathsf{appd}}) \rightarrow \mathsf{st_{new}}$
- Compute $\mathsf{cm_{st}} \leftarrow \mathsf{ADS.Commit}(\mathsf{st_{old}})$.
- Run $b \leftarrow \mathsf{ZKPoK.Verify}(\mathcal{C}, (\mathsf{addr}, \mathsf{cm_{st}}), \pi_{\mathsf{appd}})$.
- If $b = 1$, output $\mathsf{ADS.Append}(\mathsf{st_{old}}, \mathsf{addr}, v)$.
- Otherwise, output $\mathsf{st_{old}}$.

Figure 2: Our verifiable private sequences construction.

4. The Passlog protocol

We now describe the core Passlog protocol. We show how to use verifiable private sequences to build a verifiable authentication logging protocol. We also introduce a mechanism for detecting log misbehavior (Section 4.1).

4.1. Verifiable authentication logging

We would like to use verifiable private sequences in order to store log records. Using this tool, the log service can play the role of the server described in Section 3 to verify that clients are maintaining sequences correctly without learning the mapping of appends to sequences.

However, we also need to ensure that clients can only authenticate if they are logging well-formed authentication records. The log service cannot perform this check because it should not know where the client is authenticating. Our approach is to allow the relying party to verify this without revealing any information that could allow the relying party to link a client across accounts.

Set up client state. The client starts by sampling a verifiable private sequence key k_{seq}, which it will use to identify its sequence of log records. The client also samples another key k_{enc}, which is a symmetric authenticated encryption key for encrypting log records. The client then fetches the log service's public key $\mathsf{pk_{log}}$.

Registration with a relying party. To register with a relying party, the client samples two commitment openings, $r_{\mathsf{seq}}, r_{\mathsf{enc}}$ and uses them to commit to $k_{\mathsf{seq}}, k_{\mathsf{enc}}$ respectively to generate $\mathsf{cm_{seq}}, \mathsf{cm_{enc}}$. The client then sends these commitments to the relying party, along with the log's public key $\mathsf{pk_{log}}$, and saves $r_{\mathsf{seq}}, r_{\mathsf{enc}}$. The client will prove that future authentication records are correct relative to these commitments. To ensure that the client state does not grow with the number of relying parties, the client can compress $r_{\mathsf{seq}}, r_{\mathsf{enc}}$ using a PRG seed.

Authentication to a relying party. At a high level, authentication proceeds in four steps:

1) The relying party sends the client an authentication challenge (a random string).
2) The client generates an encrypted authentication record.
3) The client appends the authentication record to a verifiable private sequence with the log service.
4) The client convinces the relying party that the authen-

tication record is logged and well-formed relative to $\mathsf{cm}_{\mathsf{seq}}$ and $\mathsf{cm}_{\mathsf{enc}}$.

We now describe authentication in more detail. After receiving the challenge string chal, the client generates a record string m using a format chosen by the relying party. The string m could include just the relying party's name, but could also include any sensitive actions made by the user (e.g., changing a password). The client generates the encrypted log record as $\mathsf{ct} \leftarrow \mathsf{Enc}(k_{\mathsf{enc}}, m)$ where Enc is a symmetric authenticated encryption scheme.

The client then appends ct to its verifiable private sequence keyed by k_{seq} at location i. When it sends the append request to the log service, the client includes the challenge string chal. If the log service successfully appends ct at address addr, it signs $(\mathsf{addr}, \mathsf{ct}, \mathsf{chal})$ and sends the signature σ back to the client.

The client then needs to generate a proof that $\mathsf{addr}, \mathsf{ct}$ are correctly linked to the $\mathsf{cm}_{\mathsf{seq}}, \mathsf{cm}_{\mathsf{enc}}$ commitments that the client sent at registration. To do this, the client generates a zero-knowledge proof π with public instance $(\mathsf{addr}, \mathsf{ct}, \mathsf{cm}_{\mathsf{seq}}, \mathsf{cm}_{\mathsf{enc}}, m)$ and private witness $(k_{\mathsf{seq}}, k_{\mathsf{enc}}, r_{\mathsf{seq}}, r_{\mathsf{enc}}, i)$ certifying that:

- $\mathsf{ct} = \mathsf{Enc}(k_{\mathsf{enc}}, m)$
- $\mathsf{addr} = \mathsf{PRF}(k_{\mathsf{seq}}, i)$
- $\mathsf{cm}_{\mathsf{seq}} = \mathsf{Commit}(k_{\mathsf{seq}}, r_{\mathsf{seq}})$
- $\mathsf{cm}_{\mathsf{enc}} = \mathsf{Commit}(k_{\mathsf{enc}}, r_{\mathsf{enc}})$

This proof allows the relying party to check that the log record encrypts the correct data using the client's original encryption key, and that the log record is included in the sequence that the client committed to at registration. This ensures that an adversary cannot corrupt or hide the log contents by logging the wrong information, using the wrong key, or appending the record to a different sequence.

The client sends $\pi, \sigma, \mathsf{addr}, \mathsf{ct}$ to the relying party. The relying party verifies π and checks that σ verifies under the $\mathsf{pk}_{\mathsf{log}}$ that the client sent at registration time. If those checks pass, then the relying party authenticates the client.

Looking up records with the log service. To retrieve the contents of its log, the client needs to fetch the values in its verifiable private sequence from the log state and decrypt them using k_{enc}. In Section 5.2, we show how to use private information retrieval to look up a value while hiding the corresponding address. If the client does not hide the address, an attacker could potentially use request timing and ordering to infer that multiple addresses belong to the same sequence.

Proving log misbehavior. In the design described above, a malicious log service can lie to the relying party about adding an authentication record to the public log state. More precisely, it can sign $(\mathsf{addr}, \mathsf{ct}, \mathsf{chal})$ without adding $(\mathsf{addr}, \mathsf{ct})$ to the log state, which could allow a malicious log server to authenticate to a user's account without creating a log record. We would like to ensure that if this ever happens, a user and relying party can together catch this misbehavior (although they cannot prevent this behavior in real time). The ability to catch misbehavior raises the stakes of omitting records—if a log service is caught misbehaving in a publicly verifiable way, this could damage the log service's reputation.

To perform this checking, relying parties store the signatures under the log's public key that are sent by the client to authenticate. If a user ever suspects misbehavior, it can request the authentication signatures for its account from the relying party. If the log service ever signed a $(\mathsf{addr}, \mathsf{ct})$ pair without including it in the public log state, the client can use the signature to prove misbehavior. To minimize the storage overhead, relying parties can retain these signatures for some time period (e.g., 48 hours) and then delete them. The client can request the signatures at any point during this interval.

Recovering after compromise. When a user detects suspicious login activity, she can lock down all of her accounts by revoking her Passlog logging sequence. More precisely, a client can request to log a record with ct set as a special codeword REVOKE as in a regular authentication. In future authentications, the log service can check in zero-knowledge if an append request is trying to append to a REVOKE record and, if so, refuse to complete the authentication.

5. Instantiating the Passlog log service

We explain how we implement our public ledger of records and present two ways a Passlog log service could be instantiated.

5.1. Log data structure

We now describe how the log service stores log records in a publicly verifiable way. We use this authenticated data structure to implement verifiable private sequences.

We leverage a sparse Merkle tree [52], [59], which we call L_{R}, to efficiently store millions of $(\mathsf{addr}, \mathsf{ct})$ pairs while supporting efficient membership and non-membership proofs. Using a sparse Merkle tree, the ciphertexts are stored at the leaf nodes specified by their addresses.

The log server inserts a batch of $(\mathsf{addr}, \mathsf{ct})$ pairs every epoch (on the order of seconds [19], [53], [57], [59]). At every epoch, the log server commits to the updated contents of L_{R} and posts the resulting commitment to a separate append-only ledger L_{C}. We implement L_{C} as a binary Merkle tree. In this construction, a proof of inclusion of $(\mathsf{addr}, \mathsf{ct})$ inserted in epoch i is simply (1) a proof of inclusion of addr in the L_{R} tree from epoch i committed to by cm_i, and (2) a proof of inclusion of cm_i in L_{C}.

Updating client proofs of inclusion. To authenticate at a later epoch j ($j > i$), the client must present a proof of inclusion of $(\mathsf{addr}, \mathsf{ct})$ from its previous authentication in epoch i. (Note that we require two authentications by the same client to be spaced at least an epoch apart in order to allow the client to supply a proof of inclusion.) Requiring a proof of inclusion from a past authentication leads to a privacy challenge: the client must present a proof of inclusion that validates in epoch j, but the client should not reveal that its prior authentication took place in epoch i. To address this problem, we have the client maintain a locally updated

inclusion proof corresponding to its last authentication. After the client authenticates in epoch i, it requests a proof of inclusion of cm_i in $\mathsf{L_C}$. For each subsequent epoch, the client will fetch the new append to $\mathsf{L_C}$ (we call this background fetching process *subscription*), which enables the client to generate an updated proof of inclusion of cm_i for the current $\mathsf{L_C}$. This way, the client can provide an updated proof of inclusion in epoch j without revealing that the client's last authentication was in epoch i.

Garbage collection. To prevent the log's storage requirements from growing indefinitely, we can allow the log service to initialize a new, empty ledger at regular intervals (e.g., monthly) while retaining the previous ledger for a longer archival period (e.g., one year). During authentication, the client proves that its previous record is included in either the current ledger or an archived one. The root of the corresponding ledger becomes a public input in the log service's zero-knowledge proof, which the log service crosschecks with its archive. Using this approach, the log service learns the month of the client's last authentication. To hide this information, the public input in the log's zero-knowledge proof could include all available roots, and the client's private witness could specify which of the roots contains the last authentication.

5.2. Private lookups in the log

In Passlog, a client needs to be able to request values from its verifiable private sequence in order to recover the contents of its authentication log. A client also needs to be able to check if a new entry has been appended to its verifiable private sequence in order to potentially detect compromise. A key challenge in supporting these lookups is privacy—while verifiable private sequences ensure that the public state and contents of appends provide strong privacy, a series of reads can reveal private information. For example, if a client makes two back-to-back requests for different addresses, the log server could infer that these addresses belong to the same sequence.

Private information retrieval (PIR) [21] is a useful tool for preventing this leakage. A PIR protocol allows a client to fetch an item from a server without revealing the identity of the item to the server. However, when looking up entries in a verifiable private sequence, the client needs not only privacy, but also integrity—the client needs to check that its result is consistent with log server's commitment. Using only standard PIR, the log service could tamper with the data in order to try to learn which element the client is retrieving (a selective-failure attack [48]).In order to provide both privacy and integrity, we need an authenticated PIR scheme.

Following the approach of Colombo et al. [24], we use a two-server linear PIR scheme with Merkle proofs in order to protect against server tampering. Using Merkle proofs makes it possible not only to defend against selective-abort attacks from the server, but also to allow the client to check the validity of its response relative to a publicly available commitment. In our setting, the two servers could be the log service and an auditor (in a deployment with a centralized log service), or two parties maintaining the log service (in a deployment with a decentralized log service). The client can check this commitment with other auditors if it wishes.

However, while Colombo et al. propose using Merkle trees for authenticated PIR, their techniques do not extend directly to fetching membership or non-membership proofs in sparse Merkle trees. One challenge of this setting is that the client does not know which nodes of the sparse Merkle tree have been populated and so does not know which nodes along the Merkle path have been inserted and which are "empty" nodes. Solving this problem would seem to require a number of PIR queries equal to the tree depth to allow the client to learn which nodes are present.

Instead, we show how to take advantage of an existing tool for privately encoding a path in a tree in order to privately fetch a proof of membership or non-membership in a single round. Specifically, we leverage incremental distributed point functions (DPFs). While DPFs are commonly used for private information retrieval [14], [37], this is, to our knowledge, the first use of incremental DPFs for private tree lookups.

Tool: Incremental Distributed Point Function (DPF). An incremental DPF [12] secret-shares the values of nodes in a depth n binary tree such that the tree contains a single non-zero path. The client can generate DPF keys of size roughly $O(\lambda \cdot n)$ that encode the location of this path. A key allows a server to construct secret shares of the node values in this tree, but without learning the location of the path.

Incremental DPFs for sparse Merkle tree lookups. To construct a query for addr, the client starts by generating incremental DPF keys that correspond to the path in a tree of length d (here d is the bit length of our addresses, which is 256 in our implementation). The client sends a key to each of the two servers. The servers then need to use their keys in order to fetch the membership or non-membership proof corresponding to the path. The servers need to return the siblings that are present along the path encoded in their incremental DPF keys. (Note that in the membership proof, addr is present, and in the non-membership proof, the path to addr simply terminates in an empty node.) The servers cannot evaluate their keys on all possible nodes in the depth d tree, as $d = 256$ in our implementation and this would be prohibitively expensive. Instead, the servers can simply evaluate their keys for all non-empty nodes and all nodes with non-empty siblings (Figure 3). The servers then multiply the DPF evaluation for a given node with the value for that node's sibling. At the leaf nodes, the servers can multiply the DPF evaluation by the ciphertext in order to fetch the corresponding ciphertext. The servers sum the results across each level and return secret shares for each level, along with the signed log roots. The client can check that the log roots match and then reconstruct the secret share to recover the proof of membership (or non-membership) and the ciphertext (if present). In this way, the client can privately fetch the membership or non-membership proof in a single round.

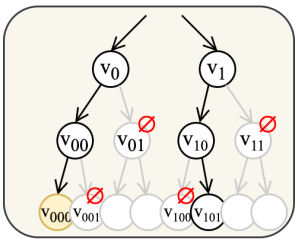

Figure 3: Performing lookups on a sparse Merkle tree via incremental DPFs. In a sparse Merkle tree of depth 3, suppose two nodes v_{000} and v_{101} are inserted. This modifies v_0, v_{00}, v_1, and v_{10} and leaves all other nodes as default hashes ($\emptyset$). To fetch the proof of inclusion for v_{000}, the server first evaluates its DPF key for all non-empty nodes ($v_0, v_{00}, v_{000}, v_1, v_{10}, v_{101}$) and all nodes with non-empty siblings that have not been evaluated yet ($v_{01}, v_{001}, v_{11}, v_{100}$). Then the server multiplies each evaluation with the node's sibling's value (using a default hash if the sibling does not exist).

5.3. Centralized log deployment

We now describe how to deploy Passlog with a central log server and multiple auditors. In this model, the log server maintains the data structure described in Section 5.1 such that the auditors can verify that it is updated correctly.

At a high level, the auditors must check that the posted commitment is consistent with updates to $\mathsf{L_R}$. At every epoch, the log service appends the new root of $\mathsf{L_R}$ to $\mathsf{L_C}$ and then sends its auditors the following: the signed new root of $\mathsf{L_C}$, the new root of $\mathsf{L_R}$ and its proof of inclusion in $\mathsf{L_C}$, and all new appends made to $\mathsf{L_R}$ in the last epoch with proofs of inclusion in $\mathsf{L_R}$. The auditors then need to check that $\mathsf{L_C}$ in epoch i is simply the product of applying the posted appends to the state of $\mathsf{L_C}$ in epoch $i-1$. Notably, the log server should not be able to erase any log records without detection. To check this, the auditors can simply replay the work and check that the resulting roots match—if they do not, then the auditor has publicly verifiable evidence of misbehavior. We ensure that any misbehavior is detected if at least one honest auditor checks $\mathsf{L_C}$ every epoch. Auditors can also store copies of the log state so that if the log service refuses to serve read queries, clients can still access their log records.

5.4. Decentralized log deployment

To avoid trusting a single log server to preserve log entries, multiple log servers run by different organizations can replicate the log using Byzantine-fault tolerant (BFT) consensus. The servers can use threshold signatures to prove the validity of $\mathsf{cm_{st}}$. However, traditional BFT consensus protocols have the disadvantage of closed membership.

A more appealing distributed architecture would be to maintain the log as a public blockchain with open membership. Though we don't want to rely on cryptocurrency mining and staking for security, Passlog is well suited to the federated Byzantine agreement model demonstrated by Stellar [55]: if each organization running replicas chooses other organizations it wants to agree with, transitively any two replicas most people care about will end up in agreement. Since all replicas sign the ledger root hash, users can at registration tell the relying party which sets of log server signature keys they consider sufficient to validate $\mathsf{cm_{st}}$.

6. Implementation

We implemented Passlog in both the centralized and decentralized log models in roughly 2,300 lines of Rust and 110 lines of Noir. We wrote our zero-knowledge proofs in Noir [2] on the scalar field of BN254 [68]. We proved them using the Barretenberg [1] proof system with an UltraHonk prover. The log service and relying party each requires a one-time setup that generates a verification key for their respective zero-knowledge proof circuit. Our private lookups use an existing incremental DPF implementation [26]. We will make our implementation open source at time of publication.

We evaluated addresses of records using a Poseidon sponge construction [40] with the user's sequence key and the index of each address in the user's verifiable private sequence as two inputs. We use a PRG seed to compress the client's commitment openings, as well as authenticated encryption nonces. Commitments are the Poseidon sponge function evaluated on the user's sequence key and its commitment opening. For authenticated encryption, we instantiated the DuplexSponge framework [10] with the Poseidon hash function. In all instances of Poseidon, we used a rate of 4, a capacity of 1, and the x^5 S-box which give approximately 128 bits of security against both collision and preimage attacks.

Log signatures are Boneh-Lynn-Shacham signatures [13] on the BLS12-381 curve. In our decentralized implementation with two log servers, we used the scheme's multi-signatures and public key aggregation method. We instantiated the ledger $\mathsf{L_R}$ of records as a sparse Merkle tree of depth 256 that can index all possible 32-byte Poseidon hash outputs. We instantiated the ledger $\mathsf{L_C}$ of commitments as a binary Merkle tree of depth 10—this number is not required by our system but has to be fixed to compile zero-knowledge proofs. Unless otherwise specified, epochs are 3 seconds long.

Unless otherwise specified, we assume Passlog has 10M users that each logs 20 authentications daily. Every hour, each user performs lookups on all her records that have been logged in the past period. We instantiate the garbage collection method (Section 5) by having the log service initialize a new ledger every month and archive the old ledger for one year.

7. Evaluation

We evaluate Passlog's performance during authentication in both the centralized and decentralized (two-server) modes. We also evaluate its performance during lookup in the two-server setting. For lookup latency experiment with 10M leaves in the log tree, we run the two log servers on a Google Compute Engine n2d-standard-16 instance with AMD Milan CPU, 8 cores, and 256GB of memory. In all other experiments, we run the log server on an n2d-custom-16-32768 instance with AMD Milan CPU, 8 cores, and 32GB

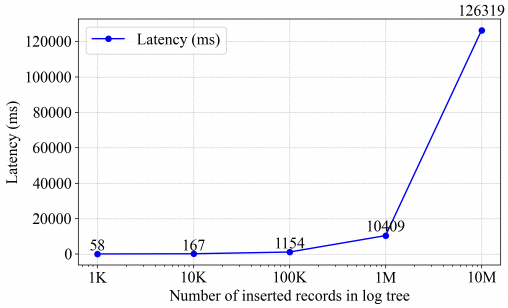

Figure 4: Lookup latency increases nearly linearly as the number of leaves in the log tree increases. The x-axis is shown in log scale. For a four core client sending split lookup requests to two serves, each lookup takes 58ms when there are 1K records in the ledger and 126s with 10M records.

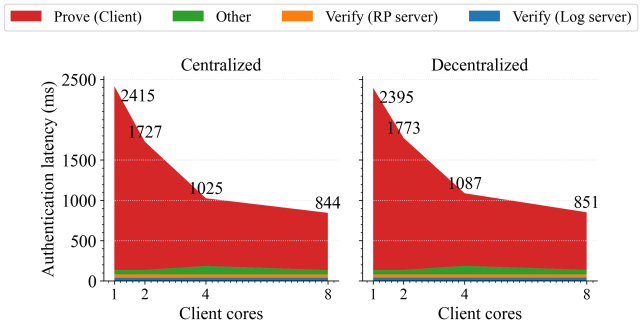

Figure 5: Client latency for each authentication decreases as the number of client cores increases in both the centralized log (left) and decentralized log (right) models. For a four-core client, each authentication takes 1,025ms in the single-server setting (840ms of which is spent on zero-knowledge proof generation) and 1,087ms in the two-server setting. 'Other' is predominantly networking time (e.g., transmitting a 14KB proof over a 100Mbps bandwidth takes 20ms, in addition to the default 20ms RTT).

of memory. For calculating networking and storage costs, we assume all log machines are located in the us-central1 region. We run the client, relying party, and auditor on e2-custom-8-16384 instances with a default CPU, 4 cores, and 16GB of memory, comparable to a commodity laptop. For latency experiments, we configure the communication links between all parties to have a 20ms RTT and a bandwidth of 100Mbps.

7.1. End-user cost

Latency. Registration takes 22ms for a client on four cores. Thereafter, with one log server, the client can complete each authentication in 2415ms on one core, or 1025ms on four cores (Figure 5). The heaviest computation performed by the client is generating zero-knowledge proofs, which takes 870ms on four cores. It takes the log server 42ms to verify the proof, and the relying party 38ms. The log-side proof requires most constraints for proving that a previous record exists in the ledger, and the relying party proof requires most constraints for proving authenticated encryption (Table 6). Although our implementation uses the Barretenberg proof system, Passlog can be instantiated with any zero-knowledge proof system, depending on deployment requirements. As zero-knowledge proofs become more performant, so too will Passlog (e.g., faster proof generation would decrease authentication latency). Authenticating with two log servers in the decentralized model marginally increases latency; it takes 2395ms on one core and 1087ms on four cores.

Each lookup takes 167ms when there are 10K records in the log tree and 126s with 10M records (Figure 4). To reduce per-client overhead, we require that clients perform lookups over all their new appends at regular intervals. We envision these regular lookups will happen in the background through some client-side application. Clients may also request individual lookups manually, provided the records fall in the current lookup window. 8.33M is the number of records that the log server has to scan for each lookup if we assume 10M users, 20 authentications daily per user, and lookups over all new appends every hour. To minimize computation during

	Proof predicate	Num. gates
Log	Check first authentication	11
	Check REVOKE	18
	$\mathsf{addr}_i = \mathsf{PRF}(k_{\mathsf{seq}}, i)$	95
	$\mathsf{addr}_{i-1} = \mathsf{PRF}(k_{\mathsf{seq}}, i-1)$	96
	Check addr_{i-1} included in L_R	21,600
	Check of $\mathsf{cm}_{\mathsf{st}}$ included in L_C	4,349
	Full circuit	**26,389**
RP	$\mathsf{cm}_{\mathsf{seq}} = H(k_{\mathsf{seq}}, r_{\mathsf{seq}})$	95
	$\mathsf{cm}_{\mathsf{enc}} = \mathsf{PRF}(k_{\mathsf{enc}}, r_{\mathsf{enc}})$	94
	$\mathsf{addr}_i = \mathsf{PRF}(k_{\mathsf{seq}}, i)$	95
	Authenticate ct encryption	172
	Full circuit	**406**

TABLE 6: Breakdown of logical gates in the log and relying party zero-knowledge proofs for the i-th authentication. The number of constraints for individual predicates adds up to more than the size of the full circuit, because Noir applies optimizations like shared gate elimination when the circuit is compiled as a whole. The commitment tree L_C is instantiated with 10 levels here, but it is not fixed by our implementation (e.g., depth 256 would require 68,576 total gates in the log proof).

lookup, the log service converts all non-empty nodes in the tree that fall in the lookup window to types that distributed point functions can evaluate more efficiently. The lookup process can be further parallelized across cores and machines with existing techniques [21], [42].

Communication. Client's communication cost during registration is 256 bytes (Table 7). Thereafter, for every authentication, the client sends a total of 28.9KB to the log server and relying party, 28.4KB of which are zero-knowledge proofs. During each lookup, the client sends two 11.4KB big DPF keys, one each to its log server and audit server.

Comparison to existing solutions. The state-of-the-art authentication logging solution, larch, takes 150ms to authenticate to relying parties using FIDO2, 91ms for TOTP [62], and 74ms for password-based log-ins when the client runs

	Item	Comm. (B)
Registration	$\mathsf{pk}_{\mathsf{log}}$	192 (C→R)
	$\mathsf{cm}_{\mathsf{seq}}$	32
	$\mathsf{cm}_{\mathsf{enc}}$	32
Authentication	Proof	14.1KB (C→L)
	Record	64
	chal	16
	Signature	96 (L→C)
	chal	16 (R→C)
	ct plaintext	32
	Proof	14.3KB (C→R)
	Record	64
	Signature	96
Audit	Root of L_R	32
	New appends to L_R	32/append
	New append indices	32/append
	Incl. proofs in L_R	$32d$/append
	Root of L_C	32
	New append to L_C	32
	New append index	1
	Incl. proof in L_C	$32d'$
	Signature	96
Lookup	DPF key	11.4KB (C→L)
	Path secret shares	$12d$ (L→C)
	Root secret share	12
	cm secret share	12
	Signature	96
Subscription	New append to L_C	32 (L→C)
	New append index	4
	Root of L_C	32
	Inl. proof in L_C	$32d'$
	Signature	96

TABLE 7: Breakdown of communication cost during each Passlog operation. 'C' is client, 'L' is log, 'R' is RP, and '→' is the direction of communication (e.g., C→L means client sending to log). d is the log tree depth and d' is the commitment tree depth. For instance, during each audit, when $d = 256$, $d' = 10$, and 100 new appends were made in the last epoch, the log server sends 826KB to each auditor. Our implementation fixes the size of a ct plaintext to 32B as the witness size needs to be constant for zero-knowledge proofs.

on four cores and the log server on eight cores. While Passlog and larch both provide universal authentication logging, Passlog achieves stronger privacy guarantees against a colluding log server and relying party.

Some existing SSO solutions provide logging but with weaker privacy guarantees. For instance, each authentication with OpenID takes 165ms [67], but the service shares users' complete authentication history with third parties.

We also compare Passlog's latency with that of existing authentication solutions without logging history. Password-based authentication systems often use password hashing to store user passwords securely at the server. The Argon2 [72] password hashing function takes 0.5s on two cores. Bcrypt [70] has adaptable performance and is typically 2-3 times faster than Argon2 at comparable security levels.

7.2. Cost to deploy a Passlog service

We evaluate the cost to deploy a Passlog service in the centralized log model.

Storage. In the BLS signature scheme, a (compressed) public key is 48 bytes, a (uncompressed) secret key is 32 bytes, and a (compressed) signature is 96 bytes. Public keys and signatures are compressed for communication and storage, and are uncompressed for signing and signature verification. The log server stores its own compressed key pair, the client stores a public key for each log server that participates in authentication, and the relying party stores one public key for each distinct log server its users use (Table 8). In the ledger of records implemented as a sparse Merkle tree, for n inserted records and N bits in each hash value, the tree stores at most $n(N - \log_2 n + 2) - 1$ non-empty hashes [65]. For a depth d tree, we additionally store d default hashes. For instance, when each hash is 32 bytes (i.e., $N = 256$), 10M users log 20 authentications each every day, and the log archives the ledger once every month, the size of one ledger at the end of the month is 1.34TB. Furthermore, we could store each archived ledger for one year; for ledgers that are older than one month, we store only the leaf appends. Clients who have not authenticated in more than one month would experience a higher latency, as the log service has to compute intermediary nodes in the tree during authentication time. In this scenario, the total storage requirement for all live and archived ledgers is 4.57TB. We could optimize storage by not storing intermediary nodes in the current tree, but this comes at the cost of increased authentication latency for clients, as the log service would have to compute the unsaved hashes during each authentication.

Each relying party stores all authentication records, 64B each, that clients send to it, along with a log server's signature. Suppose a relying party has 10M users that each authenticate 20 times every day. Say that relying party keeps records around for 9 days to give clients sufficient time to retrieve them. Then the relying party stores a maximum 288GB of records and log signatures at any given time.

Throughput. Authentication throughput of the relying party is 36.1 auths/core/s. Authentication throughput of the log server is 20.0 auths/core/s when epochs are 3 seconds and 20.7 auths/core/s when epochs are 120 seconds (Figure 9). This illustrates a tradeoff: shorter epochs reduce the size of audit requests, as there are fewer new events appended to the ledger during each epoch, but increase their frequency. In contrast, longer epochs reduce the total number of audit requests but increase their size. This tradeoff results in a non-linear relation between the log throughput and epoch size, as the overhead of sending many smaller requests does not diminish proportionally as epoch size grows. Lookup throughput of the log server is 4.91 lookups/core/s with 1K records in the tree, 1.10 with 10K, 0.21 with 100K, and 0.11 with 1M. To achieve higher throughput, multiple lookup requests can be batched so the database is scanned once for all of them. Auditor throughput is 211.1 audits/core/s.

	Item	Storage (B)
Client	k_{seq}	32
	k_{enc}	32
	PRG seed	16
	addr_{i-1}	32
	Root of $\mathsf{L_R}$	32
	Incl. proof in $\mathsf{L_C}$	$32d'$
	$\mathsf{L_C}$ new appends	32/append
RP	$(\mathsf{addr}, \mathsf{ct})$	$\leq$ 64/auth/user
	Signature	$\leq$ 96/auth/user
	$\mathsf{cm}_{\mathsf{seq}}$	32/user
	$\mathsf{cm}_{\mathsf{enc}}$	32/user
	$\mathsf{pk}_{\mathsf{log}}$	$\leq$ 48/user
	ZK verifying key	1825
Log	Current $\mathsf{L_R}$	$\leq 32n(258 - \log_2 n) - 1$
	Archive $\mathsf{L_R}$	$\leq 32n(258 - \log_2 n) - 1$ $+352n$
	Default hashes	$32d$
	$\mathsf{cm}_{\mathsf{st}}$	$32(2^{d'+1} - 1)$
	$\mathsf{pk}_{\mathsf{log}}$	48
	$\mathsf{sk}_{\mathsf{log}}$	32
	ZK verifying key	1825

TABLE 8: Storage requirements at each party in Passlog. d is the log tree depth, d' is the commitment tree depth, and n is the number of records in $\mathsf{L_R}$. There are 11 archive $\mathsf{L_R}$ after garbage collection. After each authentication, clients erase appends to $\mathsf{L_C}$ in previous epochs that they have received through subscription. If there are multiple users using the same log service, relying parties only store distinct $\mathsf{pk}_{\mathsf{log}}$. They also periodically erase old $(\mathsf{addr}, \mathsf{ct})$ and log signatures after clients have reviewed them.

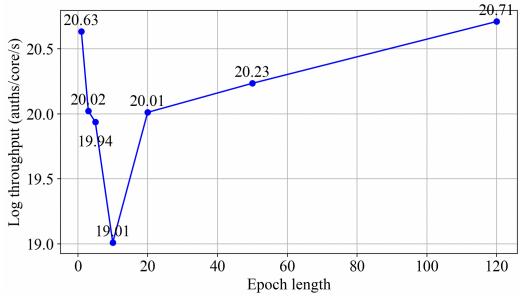

Figure 9: Log achieves minimum authentication throughput when epochs are 5 seconds and maximum when epochs are 120 seconds. Clients have to wait for at least one epoch between authentications.

Cost. Running a single core on an n2d-custom-16-32768 machine with a 10GB disk and 32GB of memory takes $0.22 hourly, and on an e2-custom-8-16384 machine with 16GB of memory takes $0.09 hourly. [22]. Running a single core on an n2d-standard-16 instance (used for lookup latency experiments with 10M records) costs $1.75 hourly. Transferring data to a Google Cloud Platform machine is free, and transferring data out to the Internet costs $0.045-$0.085/GiB in the Standard Tier pricing model, depending on how much total data is transferred [39]. A log server only meets the minimum data transfer size for billing at over 190.1M authentications (a $17 data transfer fee).

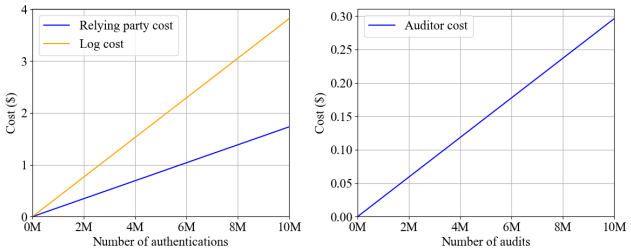

Figure 10: Computation costs to deploy log and relying party servers to support the first 10M authentications (left), and to deploy an auditor to perform the first 10M audits (right). After the log state grows past 10M appends, storing the ledger including all intermediary nodes costs $0.014 monthly (Section 7.2).

Processing 10M authentications with 3-second epochs takes 138.7 log core hours and 76.95 relying party core hours. It costs $8.15 for a Passlog log server to support 10M authentications, and $4.23 for a relying party (Figure 10). In one year, if a user requests 7,300 authentications (20 everyday), it takes a log server $0.0061 to support the user, and a relying party $0.0031. Performing 10M audits takes 13.16 auditor core hours and costs $1.18. Supporting 10M lookups on the most recent 1M records takes 50,794 total log core hours and costs $44,444 per lookup server.

The log state is dominated by the current and archived log trees and the commitment tree. Storing 10M appends and intermediary nodes in the current log tree and the most recent archive tree costs $0.74 monthly in Google Cloud Storage's Nearline Storage [23]. Each archived ledger is stored for 12 months; storing 10M leaf nodes from each of the total 11 ledgers in Coldline Storage costs $0.014 monthly. Storing one commitment ledger as a fully populated binary tree of depth 32 in Archive Storage costs $0.175 monthly. We do not account for the cost of reading from old memory, such as looking up a record in an archived ledger, or writing to storage, such as archiving a ledger.

8. Discussion

Handling device loss. If a user loses all her devices, she needs a way to recover her authentication secrets. Users can keep the secrets encrypted under a separate key and back up this key using an existing secret recovery scheme [18], [25], [27], [47], [54], [56], [81], [84]. We also describe a backup authentication mechanism, offline FIDO, below.

Account recovery with offline FIDO. Common practice for allowing users to recover accounts with lost 2FA credentials is asking them to store per-account one-time recovery codes. Unfortunately, with many accounts, managing such recovery codes can be burdensome, which encourages users to store them in an encrypted file, which is no longer a second factor.

As a better alternative, we implemented a virtual offline FIFO2 authenticator called *ofido*. *ofido* only needs to be set up once by each user for an arbitrary number of recoverable accounts. During setup, *ofido* generates a secret 18-word seed

phrase and a a master-public-key file containing an ES256 public signature key. *ofido* can be run with only this public key, and it uses BIP-32 hierarchical key derivation to generate new public keys for each account registration. Of course, *ofido* can't authenticate the user unless it is run in recovery mode and given the 18-word seedphrase, which users should store on paper and only enter in case of emergency.

ofido is technically not FIDO2 compliant because browsers request non-user-present signatures during credential registration. However, the purpose of this signature is just to check whether or not a particular credential corresponds to a particular authenticator. No one checks if the non-user-present signature is valid, so *ofido* returns a syntactically valid signature that won't verify. This approach works with every relying party we've tried.

Adoption via FIDO2. Deploying a Passlog service requires modifying relying parties. To ease the path to adoption, we can design the relying party's proof verification process to be compatible with FIDO2 [6], which many relying parties already support. For instance, the proof could be packaged as a new signature type that adheres to the Client to Authenticator Protocol [15], which governs communication between browsers and FIDO2 authenticators. A new client credential format can also be introduced to align with the W3C WebAuthn [43] specification.

9. Related work

Privacy-preserving authentication logging. Single sign-on systems allow users to authenticate to all their accounts from multiple devices using a single identity. Services like BrowserID [3] ensure that the identity provider does not learn the identity of relying parties. Furthermore, services like EL PASSO [86], IRMA [7], PseudoID [32], and the IBM identity mixer [45] ensure that the identity provider does not learn the identity of users as well as relying parties. These services, however, do not offer logging, and some of them do not prevent colluding relying parties from linking users across accounts. Perez et al. also proposed a system for client-side encrypted access logging [69]; they introduce mechanisms for trustworthy device attribution, but provide weaker integrity guarantees for malicious parties than Passlog.

Larch is the state-of-the-art in privacy-preserving server-side authentication logging. Like Passlog, it logs a user's authentication history without learning anything about the relying party based on protocol messages. Larch does not prevent a colluding log server and relying party from linking a user across accounts. This arises partly from larch's objective of achieving compatibility with existing authentication frameworks like TOTP [62] and FIDO2. In Passlog, we sacrifice backward compatibility and provide additional security against colluding log and relying parties.

Transparency logs. Like Passlog, transparency logs detect attacks on a system by logging sensitive actions [8], [51], [52], [53], [57], [59], [63], [71]. Key transparency logs provide privacy by hiding client identities and, in some schemes, sequences of updates [19], [53], [57], [59]. Similarly, Passlog's public state is encrypted and a user can only identify and decrypt records that are generated with her key.

Blockchain-based authentication. Prior work has explored using blockchain for authentication (summarized in these surveys [4], [61]). Like Passlog, these systems involve a large network of parties in authentications. Unlike Passlog, the focus of much of this work is on verifying user identities, rather than creating encrypted authentication records.

Proving properties of pseudorandom values. Our verifiable private sequences provide proofs that pseudorandom values satisfy certain constraints without revealing anything about the pseudorandom values or the sequence they belong to. Verifiable random functions [60] provide proofs that a pseudorandom output was correctly computed from a given input and key (but do not hide the inputs). Publicly verifiable secret sharing [20], [73] allows a dealer to distribute secret shares to participants along with proofs that certify the shares have been correctly distributed. Verifiable identity families [28] also produce unique, per-identity public keys from a master secret key, along with publicly verifiable proofs that each key really corresponds to the given identity.

Proving properties of encrypted data. Verifiable encryption [16], [74] proves properties of a ciphertext to a third-party verifier via public-key cryptography. Recent developments in verifiable encryption include a generic compiler that can turn a large class of zero-knowledge proofs based on MPC-in-the-head [46] into secure verifiable encryption protocols [76]. Zero-knowledge middleboxes also rely on clients to prove properties of encrypted data: in this setting, clients prove that their encrypted network packets satisfy middlebox policies [41], [85].

Authenticated private information retrieval. Early works on authenticated private information retrieval [82], [87] in the single-server setting only guaranteed that a query is consistent with *some* database, not necessarily the intended one. A proposal by Colombo et al. [24] overcomes this limitation in the single- and two-server settings. The former requires that the server honestly commits to the database. Dietz et al. [33] show how to defend against a server that provides a malformed database commitment. Furthermore, Falk et al. [35] present a generic compiler that transforms any private information retrieval scheme to a maliciously secure one. Separately, VeriSimplePIR [30] is a verifiable version of the semi-honest SimplePIR protocol [42]. Distributed point functions [14], [37] can also provide integrity guarantees for private information retrieval.

10. Conclusion

Passlog records a user's complete authentication history in a public ledger while providing strong privacy and security. A Passlog log service ensures that every authentication is correctly logged, but the authentication requests do not reveal the identity of the user or relying party. This design makes it possible to run the log service using one or more parties and allow many parties to audit the log.

References

[1] Barretenberg. https://github.com/AztecProtocol/barretenberg.

[2] Noir documentation. https://noir-lang.org/docs.

[3] Persona. https://github.com/mozilla/persona.

[4] Sohail Abbas, Manar Abu Talib, Afaf Ahmed, Faheem Khan, Shabir Ahmad, and Do-Hyeun Kim. Blockchain-based authentication in internet of vehicles: A survey. *Sensors*, 21(23):7927, 2021.

[5] Shashank Agrawal and Srinivasan Raghuraman. Kvac: Key-value commitments for blockchains and beyond. In *Advances in Cryptology–ASIACRYPT 2020: 26th International Conference on the Theory and Application of Cryptology and Information Security, Daejeon, South Korea, December 7–11, 2020, Proceedings, Part III 26*, pages 839–869. Springer, 2020.

[6] The FIDO Alliance. User authentication specifications overview, Jun 2024. https://fidoalliance.org/specifications/.

[7] Gergely Alpár, Fabian Van Den Broek, Brinda Hampiholi, Bart Jacobs, Wouter Lueks, and Sietse Ringers. Irma: practical, decentralized and privacy-friendly identity management using smartphones. In *10th Workshop on Hot Topics in Privacy Enhancing Technologies (HotPETs 2017)*, pages 1–2, 2017.

[8] Michael P Andersen, Sam Kumar, Moustafa AbdelBaky, Gabe Fierro, John Kolb, Hyung-Sin Kim, David E Culler, and Raluca Ada Popa. WAVE: A decentralized authorization framework with transitive delegation. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 1375–1392, 2019.

[9] Mihir Bellare and Phillip Rogaway. Random oracles are practical: A paradigm for designing efficient protocols. In *Proceedings of the 1st ACM Conference on Computer and Communications Security*, pages 62–73, 1993.

[10] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. Duplexing the sponge: single-pass authenticated encryption and other applications. In *International Workshop on Selected Areas in Cryptography*, pages 320–337. Springer, 2011.

[11] Manuel Blum, Paul Feldman, and Silvio Micali. Non-interactive zero-knowledge and its applications. In *Providing Sound Foundations for Cryptography: On the Work of Shafi Goldwasser and Silvio Micali*, pages 329–349. 2019.

[12] Dan Boneh, Elette Boyle, Henry Corrigan-Gibbs, Niv Gilboa, and Yuval Ishai. Lightweight techniques for private heavy hitters. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 762–776, 2021.

[13] Dan Boneh, Ben Lynn, and Hovav Shacham. Short signatures from the weil pairing. In Colin Boyd, editor, *Advances in Cryptology — ASIACRYPT 2001*, pages 514–532, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg.

[14] Elette Boyle, Niv Gilboa, and Yuval Ishai. Function secret sharing: Improvements and extensions. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, page 1292–1303, New York, NY, USA, 2016. Association for Computing Machinery.

[15] John Bradley, Jeff Hodges, Michael B. Jones, Akshay Kumar, rolf Lindemann, and Johan Verrept. Client to authenticator protocol (CTAP), Feb 2025. https://fidoalliance.org/specs/fido-v2.2-ps-20250228/fido-client-to-authenticator-protocol-v2.2-ps-20250228.pdf.

[16] Jan Camenisch and Ivan Damgård. Verifiable encryption, group encryption, and their applications to separable group signatures and signature sharing schemes. In *Proceedings of the 6th International Conference on the Theory and Application of Cryptology and Information Security: Advances in Cryptology*, ASIACRYPT '00, pages 331–345, Berlin, Heidelberg, 2000. Springer-Verlag.

[17] Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation*, OSDI '99, pages 173–186, USA, 1999. USENIX Association.

[18] Melissa Chase, Hannah Davis, Esha Ghosh, and Kim Laine. Acsesor: A new framework for auditable custodial secret storage and recovery. Cryptology ePrint Archive, Paper 2022/1729, 2022.

[19] Melissa Chase, Apoorvaa Deshpande, Esha Ghosh, and Harjasleen Malvai. Seemless: Secure end-to-end encrypted messaging with less trust. In *Proceedings of the 2019 ACM SIGSAC conference on computer and communications security*, pages 1639–1656, 2019.

[20] Benny Choc, Shafi Goldwasser, Silvio Micali, and Baruch Awerbuch. Verifiable secret sharing and achieving simultaneity in the presence of faults. In *Annual Symposium on Foundations of Computer Science (Proceedings)*, pages 383–395, 1985.

[21] Benny Chor, Eyal Kushilevitz, Oded Goldreich, and Madhu Sudan. Private information retrieval. *J. ACM*, 45(6):965–981, November 1998.

[22] Google cloud. https://cloud.google.com/compute/vm-instance-pricing.

[23] Google Cloud. Cloud storage pricing.

[24] Simone Colombo, Kirill Nikitin, Henry Corrigan-Gibbs, David J. Wu, and Bryan Ford. Authenticated private information retrieval. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 3835–3851, Anaheim, CA, August 2023. USENIX Association.

[25] Graeme Connell, Vivian Fang, Rolfe Schmidt, Emma Dauterman, and Raluca Ada Popa. Secret key recovery in a Global-Scale End-to-End encryption system. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 703–719, 2024.

[26] Henry Corrigan-Gibbs. Code that accompanies the paper "lightweight techniques for private heavy hitters" at IEEE S&P 2021, 2023. https://github.com/henrycg/heavyhitters.

[27] Emma Dauterman, Henry Corrigan-Gibbs, and David Mazières. SafetyPin: Encrypted backups with Human-Memorable secrets. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 1121–1138. USENIX Association, November 2020.

[28] Emma Dauterman, Henry Corrigan-Gibbs, David Mazières, Dan Boneh, and Dominic Rizzo. True2F: Backdoor-resistant authentication tokens. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 398–416. IEEE, 2019.

[29] Emma Dauterman, Danny Lin, Henry Corrigan-Gibbs, and David Mazières. Accountable authentication with privacy protection: The larch system for universal login. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, pages 81–98, Boston, MA, July 2023. USENIX Association.

[30] Leo de Castro and Keewoo Lee. VeriSimplePIR: Verifiability in SimplePIR at no online cost for honest servers. In *33rd USENIX Security Symposium (USENIX Security 24)*, pages 5931–5948, 2024.

[31] Alfredo De Santis, Silvio Micali, and Giuseppe Persiano. Non-interactive zero-knowledge proof systems. In Carl Pomerance, editor, *Advances in Cryptology — CRYPTO '87*, pages 52–72, Berlin, Heidelberg, 1988. Springer Berlin Heidelberg.

[32] Arkajit Dey and Stephen Weis. PseudoID: Enhancing privacy in federated login. In *Hot Topics in Privacy Enhancing Technologies*, pages 95–107, 2010.

[33] Marian Dietz and Stefano Tessaro. Fully malicious authenticated PIR. Cryptology ePrint Archive, Paper 2023/1804, 2023.

[34] Roger Dingledine, Nick Mathewson, and Paul Syverson. Tor: the second-generation onion router. In *Proceedings of the 13th Conference on USENIX Security Symposium - Volume 13*, SSYM'04, page 21, USA, 2004. USENIX Association.

[35] Brett Falk, Pratyush Mishra, and Matan Shtepel. Malicious security for PIR (almost) for free. Cryptology ePrint Archive, Paper 2024/964, 2024.

[36] Ariel Gabizon, Zachary J Williamson, and Oana Ciobotaru. Plonk: Permutations over lagrange-bases for oecumenical noninteractive arguments of knowledge. *Cryptology ePrint Archive*, 2019.

[37] Niv Gilboa and Yuval Ishai. Distributed point functions and their applications. In *Advances in Cryptology–EUROCRYPT 2014: 33rd Annual International Conference on the Theory and Applications of Cryptographic Techniques, Copenhagen, Denmark, May 11-15, 2014. Proceedings 33*, pages 640–658. Springer, 2014.

[38] Shafi Goldwasser, Silvio Micali, and Charles Rackoff. The knowledge complexity of interactive proof systems. *SIAM Journal on Computing*, 18(1):186–208, 1989.

[39] Google. All network pricing. https://cloud.google.com/vpc/network-pricing?hl=en.

[40] Lorenzo Grassi, Dmitry Khovratovich, Christian Rechberger, Arnab Roy, and Markus Schofnegger. Poseidon: A new hash function for zero-knowledge proof systems. Cryptology ePrint Archive, Paper 2019/458, 2019.

[41] Paul Grubbs, Arasu Arun, Ye Zhang, Joseph Bonneau, and Michael Walfish. Zero-Knowledge middleboxes. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 4255–4272, 2022.

[42] Alexandra Henzinger, Matthew M. Hong, Henry Corrigan-Gibbs, Sarah Meiklejohn, and Vinod Vaikuntanathan. One server for the price of two: Simple and fast Single-Server private information retrieval. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 3889–3905, Anaheim, CA, August 2023. USENIX Association.

[43] Jeff Hodges, J.C. Jones, Michael B. Jones, Akshay Kumar, and Emil Lundberg. Web authentication: an API for accessing public key credentials, Apr 2021. https://www.w3.org/TR/webauthn-2/.

[44] Yuncong Hu, Kian Hooshmand, Harika Kalidhindi, Seung Jin Yang, and Raluca Ada Popa. Merkle^2: A low-latency transparency log system. Cryptology ePrint Archive, Paper 2021/453, 2021.

[45] Eine Partnerschaft in Nanotechnologie von IBM Research und ETH Zürich. IBM Identity mixer (Authentication without identification). https://www.zurich.ibm.com/pdf/csc/Identity_Mixer_Nov_2015.pdf.

[46] Yuval Ishai, Eyal Kushilevitz, Rafail Ostrovsky, and Amit Sahai. Zero-knowledge from secure multiparty computation. In *Proceedings of the thirty-ninth annual ACM symposium on Theory of computing*, pages 21–30, 2007.

[47] Ivan Krstic. Behind the scenes with iOS security, 2016. https://www.blackhat.com/docs/us-16/materials/us-16-Krstic.pdf.

[48] E. Kushilevitz and R. Ostrovsky. Replication is not needed: single database, computationally-private information retrieval. In *Proceedings 38th Annual Symposium on Foundations of Computer Science*, pages 364–373, 1997.

[49] Leslie Lamport. The part-time parliament. In *Concurrency: the works of Leslie Lamport*, pages 277–317. 2019.

[50] Leslie Lamport, Robert Shostak, and Marshall Pease. The byzantine generals problem. *ACM Trans. Program. Lang. Syst.*, 4(3):382–401, July 1982.

[51] Adam Langley, Emilia Kasper, and Ben Laurie. Certificate transparency, 2013. https://datatracker.ietf.org/doc/html/rfc6962.

[52] Ben Laurie and Emilia Kasper. Revocation transparency. *Google Research, September*, 33, 2012.

[53] Julia Len, Melissa Chase, Esha Ghosh, Kim Laine, and Radames Cruz Moreno. OPTIKS: An optimized key transparency system. In *33rd USENIX Security Symposium (USENIX Security 24)*, pages 4355–4372, 2024.

[54] Ryan Little, Lucy Qin, and Mayank Varia. Secure account recovery for a privacy-preserving web service. In *Proceedings of the 33rd USENIX Conference on Security Symposium*, SEC '24, USA, 2024. USENIX Association.

[55] Marta Lokhava, Giuliano Losa, David Mazières, Graydon Hoare, Nicolas Barry, Eli Gafni, Rafał Malinowsky, and Jed McCaleb. Fast and secure global payments with Stellar. In *27th ACM Symposium on Operating Systems Principles*, Huntsville, Ontario, October 2019.

[56] Joshua Lund. Technology preview for secure value recovery, 2019. https://signal.org/blog/secure-value-recovery/.

[57] Harjasleen Malvai, Lefteris Kokoris-Kogias, Alberto Sonnino, Esha Ghosh, Ercan Oztürk, Kevin Lewi, and Sean Lawlor. Parakeet: Practical key transparency for end-to-end encrypted messaging. In *Network and Distributed System Security (NDSS) Symposium 2023*, 2023.

[58] Anthony Mpho Matlala. Setup Ceremonies. 2021. https://zkproof.org/2021/06/30/setup-ceremonies/.

[59] Marcela S Melara, Aaron Blankstein, Joseph Bonneau, Edward W Felten, and Michael J Freedman. CONIKS: Bringing key transparency to end users. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 383–398, 2015.

[60] Silvio Micali, Michael Rabin, and Salil Vadhan. Verifiable random functions. In *Proceedings of the 40th Annual Symposium on the Foundations of Computer Science*, pages 120–130, New York, NY, October 1999. IEEE.

[61] Ali H Mohsin, AA Zaidan, BB Zaidan, Osamah Shihab Albahri, Ahmed Shihab Albahri, MA Alsalem, and KI Mohammed. Blockchain authentication of network applications: Taxonomy, classification, capabilities, open challenges, motivations, recommendations and future directions. *Computer Standards & Interfaces*, 64:41–60, 2019.

[62] David M'Raihi, Johan Rydell, Mingliang Pei, and Salah Machani. TOTP: Time-Based One-Time Password Algorithm. RFC 6238, May 2011.

[63] Kirill Nikitin, Eleftherios Kokoris-Kogias, Philipp Jovanovic, Nicolas Gailly, Linus Gasser, Ismail Khoffi, Justin Cappos, and Bryan Ford. CHAINIAC: Proactive Software-Update transparency via collectively signed skipchains and verified builds. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 1271–1287, Vancouver, BC, August 2017. USENIX Association.

[64] Diego Ongaro and John Ousterhout. raft. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, USENIX ATC'14, page 305–320, USA, 2014. USENIX Association.

[65] Rasmus Östersjö. *Sparse Merkle Trees: Definitions and space-time trade-offs with applications for Balloon*. PhD thesis, Karlstad University, 2016.

[66] Charalampos Papamanthou, Roberto Tamassia, and Nikos Triandopoulos. Authenticated hash tables based on cryptographic accumulators. *Algorithmica*, 74:664–712, 2016.

[67] OpenID Foundation Helping people assert their identity wherever they choose. Openid.

[68] Geovandro C. C. F. Pereira, Marcos A. Simplício Jr, Michael Naehrig, and Paulo S. L. M. Barreto. A family of implementation-friendly BN elliptic curves. Cryptology ePrint Archive, Paper 2010/429, 2010.

[69] Carolina Ortega Pérez, Alaa Daffalla, Thomas Ristenpart, and Cornell Tech. Encrypted access logging for online accounts: Device attributions without device tracking. In *USENIX Security Symposium*, 2025.

[70] Niels Provos and David Mazières. A future-adaptable password scheme. In *Proceedings of the FREENIX Track: 1999 USENIX Annual Technical Conference*, 1999.

[71] Mark D Ryan. Enhanced certificate transparency and end-to-end encrypted mail. *Cryptology ePrint Archive*, 2013.

[72] Hynek Schlawack. What is argon2? https://argon2-cffi.readthedocs.io/en/stable/argon2.html.

[73] Markus Stadler. Publicly verifiable secret sharing. In Ueli Maurer, editor, *Advances in Cryptology — EUROCRYPT '96*, pages 190–199, Berlin, Heidelberg, 1996. Springer Berlin Heidelberg.

[74] Markus Stadler. Publicly verifiable secret sharing. In Ueli Maurer, editor, *Advances in Cryptology — EUROCRYPT '96*, pages 190–199, Berlin, Heidelberg, 1996. Springer Berlin Heidelberg.

[75] Apple Support. About icloud private relay. https://support.apple.com/en-euro/102602, Aug 2023.

[76] Akira Takahashi and Greg Zaverucha. Verifiable encryption from mpc-in-the-head. *IACR Communications in Cryptology*, 04 2024.

[77] Roberto Tamassia. Authenticated data structures. In *Algorithms-ESA 2003: 11th Annual European Symposium, Budapest, Hungary, September 16-19, 2003. Proceedings 11*, pages 2–5. Springer, 2003.

[78] Alin Tomescu, Ittai Abraham, Vitalik Buterin, Justin Drake, Dankrad Feist, and Dmitry Khovratovich. Aggregatable subvector commitments for stateless cryptocurrencies. In *International Conference on Security and Cryptography for Networks*, pages 45–64. Springer, 2020.

[79] Alin Tomescu, Vivek Bhupatiraju, Dimitrios Papadopoulos, Charalampos Papamanthou, Nikos Triandopoulos, and Srinivas Devadas. Transparency logs via append-only authenticated dictionaries. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 1299–1316, 2019.

[80] Nirvan Tyagi, Ben Fisch, Andrew Zitek, Joseph Bonneau, and Stefano Tessaro. Versa: Verifiable registries with efficient client audits from rsa authenticated dictionaries. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, pages 2793–2807, 2022.

[81] Shabsi Walfish. Google Cloud Key Vault Service. Google, 2018. https://developer.android.com/about/versions/pie/security/ckv-whitepaper.

[82] Xingfeng Wang and Liang Zhao. Verifiable single-server private information retrieval. In *Information and Communications Security: 20th International Conference, ICICS 2018, Lille, France, October 29-31, 2018, Proceedings*, pages 478–493. Springer, 2018.

[83] Jess Weatherbed. Lastpass reveals attackers stole password vault data by hacking an employee's home computer. *The Verge*, February 2023. https://www.theverge.com/2023/2/28/23618353/lastpass-security-breach-disclosure-password-vault-encryption-update.

[84] WhatsApp. Security of end-to-end encrypted backups, 2021. https://www.whatsapp.com/security/WhatsApp_Security_Encrypted_Backups_Whitepaper.pdf.

[85] Collin Zhang, Zachary DeStefano, Arasu Arun, Joseph Bonneau, Paul Grubbs, and Michael Walfish. Zombie: Middleboxes that Don't snoop. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, pages 1917–1936, Santa Clara, CA, April 2024. USENIX Association.

[86] Zhiyi Zhang, Michał Król, Alberto Sonnino, Lixia Zhang, and Etienne Rivière. EL PASSO: Privacy-preserving, asynchronous single sign-on, February 2020.

[87] Liang Zhao, Xingfeng Wang, and Xinyi Huang. Verifiable single-server private information retrieval from LWE with binary errors. *Information Sciences*, 546:897–923, 2021.

Appendix

1. Verifiable private sequence properties

We provide formal definitions for the properties we informally defined in (Section 3.2) for verifiable private sequences. Throughout, an efficient algorithm is one that runs in probabilistic polynomial time.

Completeness. Completeness informally states that address-value pairs that have been correctly appended should be returned in read requests. To formalize completeness, we define a completeness experiment parameterized by security parameter λ (Figure 11).

Definition 1. (Completeness) A VPS scheme parameterized by security parameter λ and value space $\mathcal{V}$ is complete if, for all efficient adversaries $\mathcal{A}$, the output of the game in Experiment 1 is 0 with negligible probability.

Experiment 1: VHS completeness. We define a completeness game parameterized by an adversary $\mathcal{A}$, verifiable private sequences scheme VPS, security parameter λ, and value space $\mathcal{V}$. The game proceeds as follows:

- The challenger runs $\mathsf{st} \leftarrow \mathsf{VPS.Init}(1^\lambda)$, initializes $\mathsf{vals} = \emptyset$, and sets $b = 1$, $\mathsf{ctr} = 0$.
- The adversary can issue KeyGen, Append, and Read queries to the challenger:
  - On $\mathsf{KeyGen}()$, the challenger:
    - $k_{\mathsf{ctr}} \leftarrow \mathsf{VPS.KeyGen}(1^\lambda)$
    - $\pi_{\mathsf{inc}}^{(\mathsf{ctr})} \leftarrow \perp$, $\mathsf{len}_{\mathsf{ctr}} \leftarrow 0$, $\mathsf{ctr} \leftarrow \mathsf{ctr} + 1$
  - On $\mathsf{Append}(u, v)$ for user $u \in \mathbb{N}$ and value $v \in \mathcal{V}$, the challenger:
    - Check if $u < \mathsf{ctr}$. If not, then return.
    - $\mathsf{cm}_{\mathsf{st}} \leftarrow \mathsf{VPS.GetCm}(\mathsf{st})$.
    - $\mathsf{addr} \leftarrow \mathsf{VPS.GetAddr}(k_u, \mathsf{len}_u)$
    - $\pi_{\mathsf{appd}} \leftarrow \mathsf{VPS.ProveAppend}(\mathsf{cm}_{\mathsf{st}}, k_u, \mathsf{len}_u, \mathsf{addr}, \pi_{\mathsf{inc}}^{(u)})$
    - $\mathsf{st} \leftarrow \mathsf{VPS.Append}(\mathsf{st}, \mathsf{addr}, v, \pi_{\mathsf{appd}})$
    - Store $\mathsf{vals} = \mathsf{vals} \cup \{(u, \mathsf{len}_u, v)\}$
    - $(\_, \pi_{\mathsf{inc}}^{(u)}) \leftarrow \mathsf{VPS.Read}(\mathsf{st}, \mathsf{addr})$
    - $\mathsf{len}_u \leftarrow \mathsf{len}_u + 1$
  - On $\mathsf{Read}(u, i)$ for user $u \in \mathbb{N}$ and index $i \in \mathbb{N}$, the challenger:
    - Check if $u < \mathsf{ctr}$ and $i < \mathsf{len}_u$. If not, then return.
    - $\mathsf{addr} \leftarrow \mathsf{VPS.GetAddr}(k_u, i)$
    - $(v, \pi_{\mathsf{inc}}) \leftarrow \mathsf{VPS.Read}(\mathsf{st}, \mathsf{addr})$
    - $\mathsf{cm}_{\mathsf{st}} \leftarrow \mathsf{VPS.GetCm}(\mathsf{st})$
    - If $\mathsf{VPS.VerifyRead}(\mathsf{cm}_{\mathsf{st}}, \mathsf{addr}, v, \pi_{\mathsf{inc}}) = 0$, set $b = 0$.
    - If $(u, i, v) \notin \mathsf{vals}$, set $b = 0$.
- When the adversary has finished making queries, the challenger outputs b.

Figure 11: VHS completeness experiment.

Privacy. Append requests are distributed independently of the content of the appends.

Definition 2. (Privacy) Let the adversary's view for a sequence of append operations $u_1, \ldots, u_n$ where $u_i \in \mathbb{N}$ for $i \in [n]$ be defined as follows:

- $\mathsf{st} \leftarrow \mathsf{VPS.Init}(1^\lambda)$
- $\mathsf{keys} = \{\}$
- For $i \in [n]$:
  - If $u_i \notin \mathsf{keys}$:
    - $k_{u_i} \leftarrow \mathsf{VPS.KeyGen}(1^\lambda)$
    - $\mathsf{len}_{u_i} \leftarrow 0$
    - $\pi_{\mathsf{inc}}^{(u_i)} \leftarrow \perp$

- $\mathsf{cm}_{\mathsf{st}} \leftarrow \mathsf{VPS.GetCm}(\mathsf{st})$
- $\mathsf{addr} \leftarrow \mathsf{VPS.GetAddr}(k_{u_i}, \mathsf{len}_{u_i})$
- $\pi_{\mathsf{appd}} \leftarrow \mathsf{VPS.ProveAppend}(\mathsf{cm}_{\mathsf{st}}, k_{u_i}, \mathsf{len}_{u_i}, \mathsf{addr}, \pi_{\mathsf{inc}}^{(u_i)})$
- $(\_, \pi_{\mathsf{inc}}^{(u_i)}) \leftarrow \mathsf{VPS.Read}(\mathsf{st}, \mathsf{addr})$
- Output $(\mathsf{addr}, \pi_{\mathsf{appd}})$

Then a VPS scheme parameterized by security parameter λ and value space $\mathcal{V}$ provides privacy if, for any two sequences of append operations of length n and for all efficient adversaries $\mathcal{A}$, the adversary's views generated by the two sequences are computationally indistinguishable in λ.

Append soundness. Elements must be added to a sequence in order and keyed by the correct k. We capture a more precise definition in the experiment in Figure 12. At a high level, the adversary can issue requests to start and append to sequences where the keys are held by honest clients, send read requests, and submit arbitrary append requests. Then the adversary can submit an append request, and the adversary wins if the append either writes to a challenger-controlled client's sequence, or the adversary writes out-of-order in an adversary-controlled sequence.

Definition 3. (Append Soundness) A VPS scheme parameterized by security parameter λ and value space $\mathcal{V}$ has append soundness if, for all efficient adversaries $\mathcal{A}$, the probability that the output of Experiment 2 is the bit $b = 0$ is negligible in λ.

Read soundness. The definition of read soundness follows the definition of read soundness from authenticated data structures.

Definition 4. (Read soundness) A VPS scheme parameterized by security parameter λ and value space $\mathcal{V}$ has read soundness if, for all efficient adversaries $\mathcal{A}$, $\mathcal{A}$ can generate values $\mathsf{addr} \in \{0,1\}^\lambda, v \in \mathcal{V}$, $v' \in \mathcal{V}$, π_{inc}, π'_{inc}, and $\mathsf{cm}_{\mathsf{st}}$ such that both of the following statements hold:

$$\mathsf{VPS.VerifyRead}(\mathsf{cm}_{\mathsf{st}}, \mathsf{addr}, v, \pi_{\mathsf{inc}}) = 1$$

$$\mathsf{VPS.VerifyRead}(\mathsf{cm}_{\mathsf{st}}, \mathsf{addr}, v', \pi'_{\mathsf{inc}}) = 1$$

Experiment 2: VHS append soundness. This game is parameterized by an adversary $\mathcal{A}$, verifiable private sequences scheme VPS, security parameter λ, and value space $\mathcal{V}$. It proceeds as follows.

- The challenger runs $\mathsf{st} \leftarrow \mathsf{VPS.Init}(1^\lambda)$ and sets $\mathsf{ctr} = 0$, $\mathsf{addrs} = \emptyset$.
- The adversary can make the following queries:
  - On $\mathsf{KeyGen}()$, the challenger:
    - $k_{\mathsf{ctr}} \leftarrow \mathsf{VPS.KeyGen}(1^\lambda)$
    - $\pi_{\mathsf{inc}}^{(\mathsf{ctr})} \leftarrow \perp$, $\mathsf{len}_{\mathsf{ctr}} \leftarrow 0$, $\mathsf{ctr} \leftarrow \mathsf{ctr} + 1$
  - On $\mathsf{Append}(u, v)$, for $u \in \mathbb{N}$ and value $v \in \mathcal{V}$, the challenger:
    - Check if $u < \mathsf{ctr}$. If not, then return.
    - $\mathsf{cm}_{\mathsf{st}} \leftarrow \mathsf{VPS.GetCm}(\mathsf{st})$
    - $\mathsf{addr} \leftarrow \mathsf{VPS.GetAddr}(k_u, \mathsf{len}_u, \mathsf{addr}, \pi_{\mathsf{inc}}^{(u)})$
    - $\pi_{\mathsf{appd}} \leftarrow \mathsf{VPS.ProveAppend}(\mathsf{cm}_{\mathsf{st}}, k_u, \mathsf{len}_u, \mathsf{addr}, \pi_{\mathsf{inc}}^{(u)})$
    - $\mathsf{st} \leftarrow \mathsf{VPS.Append}(\mathsf{st}, \mathsf{addr}, v, \pi_{\mathsf{appd}})$
    - $(\_, \pi_{\mathsf{inc}}^{(u)}) \leftarrow \mathsf{VPS.Read}(\mathsf{st}, \mathsf{addr})$
    - $\mathsf{len}_u \leftarrow \mathsf{len}_u + 1$
    - $\mathsf{addrs} \leftarrow \mathsf{addrs} \cup \{\mathsf{addr}\}$
    - Returns $(\mathsf{addr}, \pi_{\mathsf{appd}}, \pi_{\mathsf{inc}}^{(u)})$ to $\mathcal{A}$.
  - On $\mathsf{Read}(\mathsf{addr})$, the challenger:
    - $\mathsf{addrs} \leftarrow \mathsf{addrs} \cup \{\mathsf{addr}\}$
    - Returns $\mathsf{VPS.Read}(\mathsf{st}, \mathsf{addr})$ to $\mathcal{A}$.
  - On $\mathsf{Append}(\mathsf{addr}, v, \pi_{\mathsf{appd}})$ for $\mathsf{addr} \in \{0,1\}^\lambda$, $v \in \mathcal{V}$, and proof π_{appd}, the challenger:
    - $\mathsf{addrs} \leftarrow \mathsf{addrs} \cup \{\mathsf{addr}\}$
    - $\mathsf{st} \leftarrow \mathsf{VPS.Append}(\mathsf{st}, \mathsf{addr}, v, \pi_{\mathsf{appd}})$
- Now, $\mathcal{A}$ must send the challenger $(\mathsf{addr}^*, v^*, \pi_{\mathsf{appd}}*)$.
- The challenger runs:
  - $\mathsf{st} \leftarrow \mathsf{VPS.Append}(\mathsf{st}, \mathsf{addr}^*, v^*, \pi^*_{\mathsf{appd}})$
  - $(v, \pi_{\mathsf{inc}}) \leftarrow \mathsf{VPS.Read}(\mathsf{st}, \mathsf{addr}^*)$
  - $\mathsf{cm}_{\mathsf{st}} \leftarrow \mathsf{VPS.GetCm}(\mathsf{st})$
  - $b \leftarrow \mathsf{VPS.VerifyRead}(\mathsf{cm}_{\mathsf{st}}, \mathsf{addr}^*, v, \pi_{\mathsf{inc}})$
- $\mathcal{A}$ wins if:
  - $b = 1$,
  - $v = v^*$ $(v \neq \perp)$
  - $\mathsf{addr}^* \notin \mathsf{addrs}$
  - Either:
    - $\mathcal{A}$ sends (u, i) where $\mathsf{addr}^* = \mathsf{VPS.GetAddr}(k_u, i)$
    - $\mathcal{A}$ sends (k, i) such that for
      - $\mathsf{addr}_{i-1} \leftarrow \mathsf{VPS.GetAddr}(k, i-1)$
      - $(v, \_) = \mathsf{VPS.Read}(\mathsf{st}, \mathsf{addr}_{i-1})$

      $v = \perp$ and $\mathsf{addr}^* = \mathsf{VPS.GetAddr}(k, i)$

Figure 12: VHS append soundness experiment.