Accountable authentication with privacy protection: The Larch system for universal login

Emma Dauterman
UC Berkeley

Danny Lin
Woodinville High School

Henry Corrigan-Gibbs
MIT

David Mazières
Stanford

Abstract. Credential compromise is hard to detect and hard to mitigate. To address this problem, we present larch, an accountable authentication framework with strong security and privacy properties. Larch protects user privacy while ensuring that the larch log server correctly records every authentication. Specifically, an attacker who compromises a user's device cannot authenticate without creating evidence in the log, and the log cannot learn which web service (relying party) the user is authenticating to. To enable fast adoption, larch is backwards-compatible with relying parties that support FIDO2, TOTP, and password-based login. Furthermore, larch does not degrade the security and privacy a user already expects: the log server cannot authenticate on behalf of a user, and larch does not allow relying parties to link a user across accounts. We implement larch for FIDO2, TOTP, and password-based login. Given a client with four cores and a log server with eight cores, an authentication with larch takes 150ms for FIDO2, 91ms for TOTP, and 74ms for passwords (excluding preprocessing, which takes 1.23s for TOTP).

1 Introduction

Account security is a perennial weak link in computer systems. Even well-engineered systems with few bugs become vulnerable once human users are involved. With poorly engineered or configured systems, account compromise is often the first of several cascading failures. In general, 82% of data breaches involve a human element, with the most common methods including use of stolen credentials (40%) and phishing (20%) [79].

When users and administrators identify stolen credentials, it is challenging to determine the extent of the damage. Not knowing what an attacker accessed can lead to either inadequate or overly extensive recovery. LastPass suffered a breach in November 2022 because they didn't fully recover from a compromise the previous August [72]. Conversely, Okta feared 366 organizations might have been accessed when an attacker gained remote desktop access at one of their vendors. It took a three-month investigation to determine that, in fact, only two organizations, not 366, had really been victims of the breach [31].

Single sign-on schemes, such as OpenID [74] and "Sign in with Google," can keep an authentication log and thereby determine the extent of a credential compromise. However, these centralized systems represent a security and privacy risk: they give a third party access to all of a user's accounts and to a trace of their authentication activity.

An ideal solution would give the benefits of universal authentication logging without the security and privacy drawbacks of single-sign-on systems. For security, the logging service shouldn't be able to authenticate on behalf of a user. For privacy, the logging service should learn no information about a user's authentication history: the log service should not even learn if the user is authenticating to the same web service twice or to two separate web services.

In this paper, we propose larch ("login archive"), an accountable authentication framework with strong security and privacy properties. Authentication takes place between a user and a service, which we call the *relying party*. In larch, we add a third party: a user-chosen larch log service. The larch log service provides the user with a complete, comprehensive history of her authentication activity, which helps users detect and recover from compromises. Once an account is registered with larch, even an attacker who controls the user's client cannot authenticate to the account without the larch log service storing a record that allows the user to recover the time and relying-party name.

The key challenge in larch is allowing the log service to maintain a complete authentication history *without becoming a single point of security or privacy failure*. A malicious larch log service cannot access users' accounts and learns no information about users' authentication histories. Only users can decrypt their own log records.

Larch works with any relying party that supports one of three standard user authentication schemes: FIDO2 [36] (popularized by Yubikeys and Passkeys [3]), TOTP [68] (popularized by Google Authenticator), and password-based login. FIDO2 is the most secure but least widely deployed of the three options.

A larch deployment consists of two components: a browser add-on, which manages the user's authentication secrets, and one or more larch log services, which store authentication logs on behalf of a set of users. At a high level, larch provides four operations. (1) Upon deciding to use larch, a user performs a one-time *enrollment* with a log service. (2) For each account to use with larch, the user runs *registration*. To relying parties, registration looks like adding a FIDO2 security key, adding an authenticator app, or setting a password. (3) The user then performs *authentication* with larch as necessary to access registered accounts. Finally, (4) at any point the user can *audit* login activity by downloading and decrypting the complete history of authentication events to all accounts. The client can use auditing for intrusion detection or to evaluate the extent of the damage after a client has been compromised.

All authentication mechanisms require generating an authentication credential based on some secret. In FIDO2, the secret is a signature key and the credential is a digital signature; the signed payload depends on the name of the relying party and a fresh challenge, preventing both phishing and credential reuse. With TOTP, the secret is an HMAC key and the credential an HMAC of the current time, which prevents credential reuse in the future. With passwords, the credential is simply the password, which has the disadvantage that it can be reused once a malicious client obtains it.

Larch splits the authentication secret between the client and log service so that both parties must participate in authentication. We introduce split-secret authentication protocols for FIDO2, TOTP, and password-based login. At the end of each protocol, the log service holds an encrypted authentication log record and the client holds a credential. Larch ensures that if the client obtains a valid credential, the log service also obtains a well-formed log record, even if the client is compromised and behaves maliciously. At the same time, the log service learns no information about the relying parties that the user authenticates to.

We design larch to achieve the following (informal) security and privacy goals:

- *Log enforcement against a malicious client:* An attacker that compromises a client cannot authenticate to an account that the client created before compromise without the log obtaining a well-formed, encrypted log record.
- *Client privacy and security against a malicious log:* A malicious log service cannot authenticate to the user's accounts or learn any information about the relying parties to which the user has authenticated, including whether two authentications are for the same account or different accounts.
- *Client privacy against a malicious relying party:* Colluding malicious relying parties cannot link a user across accounts.

Larch's FIDO2 protocol uses zero-knowledge proofs [43] to convince the log that an encrypted authentication log record generated by the client is well-formed relative to the digest of a FIDO2 payload. If it is, the client and log service sign the digest with a new, lightweight two-party ECDSA signing protocol tailored to our setting. For TOTP, larch executes an authentication circuit using an existing garbled-circuit-based multiparty computation protocol [87, 84]. For password-based login, the client privately swaps a ciphertext encrypting the relying party's identity for the log's share of the corresponding password using a discrete-log-based protocol [46].

In the event that a user's device is compromised, a user can revoke access to all accounts—even accounts she may have forgotten about—by interacting only with the log service. At the same time, involving the log service in every authentication could pose a reliability risk (just as relying on OpenID does). We show how to split trust across multiple log service providers to strengthen availability guarantees, making larch strictly better than OpenID for all three of security, privacy, and availability.

We expect users to perform many password-based authentications, some FIDO2 authentications, and a comparatively small number of TOTP authentications. Given a client with four cores and a log server with eight cores, an authentication with larch takes 150ms for FIDO2, 91ms for TOTP, and 74ms for passwords (excluding preprocessing, which takes 1.23s for TOTP). One authentication requires 1.73MiB of communication for FIDO2, 65.2MiB for TOTP, and 3.25KiB for passwords. TOTP communication costs are comparatively high because we use garbled circuits [84]; however, all but 202KiB of the communication can be moved into a preprocessing step.

Larch shows that it is possible to achieve privacy-preserving authentication logging that is backwards compatible with existing standards. Moreover, larch provides new paths for FIDO2 adoption, as larch users can authenticate using FIDO2 without dedicated hardware tokens, which could motivate more relying parties to deploy FIDO2. Users who do own hardware tokens can use them to authenticate to the larch log service, providing strong security guarantees for relying parties that do not yet support FIDO2 (albeit without the anti-phishing protection). We also suggest small changes to the FIDO standard that would substantially reduce the overheads of larch while providing the same security and privacy properties.

2 Design overview

We now give an overview of larch.

2.1 Entities

A larch deployment involves the following entities:

Users. We envision a deployment with millions of users, each of which has hundreds of accounts at different online services—shopping websites, financial institutions, news sites, and so on. Each user has an account at a larch log service, secured by a strong, unique password and optionally (but ideally) strong second-factor authentication such as a FIDO2 hardware security key. (In Section 6, we describe how a user can create accounts with multiple log services in order to protect against faulty logs.) A user also has a set of devices (e.g. laptop, phone, tablet) running larch client software and storing larch secrets, including cryptographic keys and passwords.

Relying parties. A relying party is any website that a user authenticates to (e.g., a shopping website or bank). Larch is compatible with any relying party that supports authentication via FIDO2 (U2F) [36, 80], time-based one-time passwords (TOTP) [68], or standard passwords. The strength of larch's security guarantees depends on the strength of the underlying authentication method.

Log service. Whenever the user authenticates to a relying party, the client must communicate with the log service. We envision a major service provider (e.g. Google or Apple) deploying this service on behalf of their customers. The log service:

- keeps an encrypted record of the user's authentication history, but
- learns no information about which relying party the user authenticates to.

At any time, a client can fetch this authentication record from the log service and decrypt it to see the user's authentication history. That is, if an attacker compromises one of Alice's devices and authenticates to `github.com` as Alice, the attacker will leave an indelible trace of this authentication in the larch log. At the same time, to protect Alice's privacy, the log service learns no information about which relying parties Alice has authenticated to. A production log service should consist of multiple, georeplicated servers to ensure high availability.

2.2 Protocol flow

Background. We use two-out-of-two *additive secret sharing* [75]: to secret-share a value $x \in \{0, \ldots, p-1\}$, choose random values $x_1, x_2 \in \{0, \ldots, p-1\}$ such that $x_1 + x_2 = x \bmod p$. Neither x_1 nor x_2 individually reveals any information about x. We also use a cryptographic *commitment scheme*: to commit to a value $x \in \{0,1\}^*$, choose a random value $r \in \{0,1\}^{256}$ (the commitment *opening*) and output the hash of $(x \| r)$ using a cryptographic hash function such as SHA-256. For computationally bounded parties, the commitment reveals no information about x, but makes it impractical to convince another party that the commitment opens to a value $x' \neq x$.

The client's interaction with the log service consists of four operations.

Step 1: Enrollment with a log service. To use larch, a user must first *enroll* with a larch log service by creating an account. In addition to configuring traditional account authentication (i.e., setting a password and optionally registering FIDO2 keys), the user's client generates a secret *archive key* for each authentication method supported. For FIDO2 and TOTP, the archive key is a symmetric encryption key, and the client sends the log service a commitment to this key. For passwords, the archive key is an ElGamal private encryption key, so the client sends the log service the corresponding public key. The client subsequently encrypts log records using these archive keys, while the log service verifies these log records are well-formed using the corresponding commitment or public key.

Step 2: Registration with relying parties. After the user has enrolled with a log service, she can create accounts at relying parties (e.g., `github.com`) using larch-protected credentials. We call this process *registration*. Registration works differently depending on which authentication mechanism the relying party uses: FIDO2 public-key authentication, TOTP codes, or standard passwords. All generally follow the same pattern where at the conclusion of the registration protocol:

- the log service holds an encryption of the relying party's identity under a key that only the client knows,
- the log service and client jointly hold the account's authentication secret using two-out-of-two *secret sharing* [75],
- the relying party is unaware of larch and holds the usual information necessary to verify account access: an ECDSA public key (for FIDO2), an HMAC secret key (for TOTP), or a password hash (for password-based login), and
- the log service learns nothing about the identity of the relying party.

By splitting the user's authentication secret between the client and the log, we ensure that the log service participates in all of the user's authentication attempts, which allows the log service to guarantee that every authentication attempt is correctly logged.

The underlying authentication mechanisms (FIDO2, TOTP, and password-based login) only provide security for a given relying party if the user's device was uncompromised at the time of registration; larch provides the same guarantees.

Step 3: Authentication to a relying party. Registering with a relying party lets the user later authenticate to that relying party (Figure 1). At the conclusion of an authentication operation, larch must ensure that:

- authentication succeeds at the relying party,
- the log service holds a record of the authentication attempt that *includes the name of the relying party*, encrypted under the archive key known only to the client, and
- the log service learns *no information* about the identity of the relying party involved.

The technical challenge here is guaranteeing that a compromised client cannot successfully authenticate to a relying party without creating a valid log record. In particular, the log service must verify that the log record contains a valid encryption of the relying party's name under the archive key *without* learning anything about the relying party's identity.

To achieve these goals, we design *split-secret authentication protocols* that allow the client and log to use their split authentication secrets to jointly produce an authentication credential. Our split-secret authentication protocols are essentially special-purpose two-party computation protocols [88]. In a two-party computation, each party holds a secret input, and the protocol allows the parties to jointly compute a function on their inputs while keeping each party's input secret from the other. Our split-secret authentication protocols follow a general pattern, although the specifics depend on the underlying authentication mechanism in use (FIDO2, TOTP, or password-based login):

- The client algorithm takes as input the identity of the relying party, the client's share of the corresponding authentication secret, the archive key, and the opening for the log service's commitment to the archive key.

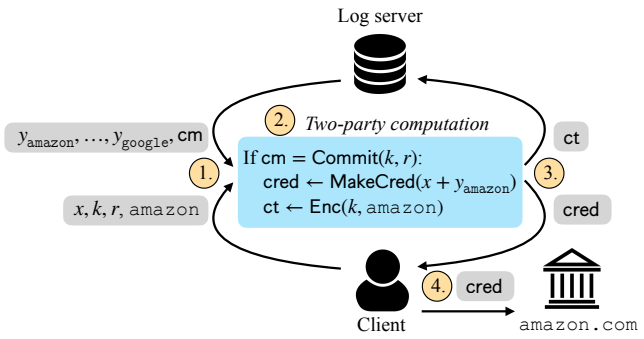

Figure 1: The client and log service run split-secret authentication where the client obtains the credential for `amazon.com` and the log service obtains an encryption of `amazon.com` under the client's key. The client's inputs are its share x of the authentication secret, the archive key k, a random nonce r, and the string `amazon.com`. The log's inputs are its shares $y_{\texttt{amazon}}, \ldots, y_{\texttt{google}}$ of all the client's authentication secrets and the commitment cm to the archive key generated at enrollment. The MakeCred function takes extra inputs for FIDO2 and TOTP.

- The log algorithm takes as input its shares of authentication secrets and the client's commitment to the archive key (which it received at enrollment).
- The client algorithm outputs an authentication credential: a signature (for FIDO2), an HMAC code (for TOTP), or a password (for password-based login).
- The log algorithm outputs an encryption of the relying party identifier under the archive key.

In this way, the client and log service jointly generate authentication credentials while guaranteeing that every successful authentication is correctly logged. The client and log do not learn any information beyond the outputs of the computation. We use this general pattern to construct split-secret authentication protocols for FIDO2 (Section 3), TOTP (Section 4), and password-based login (Section 5).

Step 4: Auditing with the log. Finally, at any time, the user can ask the log service for its collection of log entries encrypted under the archive key. A user could do this when she suspects that an attacker has compromised her credentials. The user's client could also perform this auditing in the background and notify the user if it ever detects anomalous behavior. The client uses the encryption key it generated during enrollment to decrypt log entries.

2.3 System goals

We now describe the security goals of larch (Figure 2).

Goal 1: Log enforcement against a malicious client. Say that an honest client enrolls with an honest log service and then registers with a set of relying parties. Later on, an attacker compromises the client's secrets (e.g., by compromising one of the user's devices and causing it to behave maliciously). Every successful authentication attempt that the attacker makes using credentials managed by larch will appear in the client's authentication log stored at the larch log service. Furthermore, the honest client can decrypt these log entries using its secret key.

Goal 2: Client privacy and security against a malicious log. Even if the log service deviates arbitrarily from the prescribed protocol, it learns no information about (a) the client's authentication secrets (meaning that the log service cannot authenticate on behalf of the client) or (b) which relying parties a client has interacted with.

Goal 3: Client privacy against a malicious relying party. A set of colluding malicious relying parties learn no information about which registered accounts belong to the same client. That is, relying parties cannot link a client across multiple relying parties using information they learn during registration or authentication.

To be usable in practice, larch should additionally achieve the following functionality goal:

Goal 4: No changes to the relying party. Relying parties that support FIDO2 (U2F), TOTP, or password authentication do not need to be aware of larch. Clients can unilaterally register authentication credentials such that all future authentications are logged in larch.

2.4 Non-goals and extensions

Availability against a compromised log service. Larch does not provide availability if the log service refuses to provide service. We discuss defenses against availability attacks in Section 6.

Privacy against colluding log and relying party. If the log service colludes with a relying party, they can always use timing information to map log entries to authentication requests. Therefore, larch makes no effort to obscure the relationship between private messages seen by the two parties and only guarantees privacy when the relying party and log service do not collude.

Limitations of underlying authentication schemes. Larch provides security guarantees that match the security of the underlying authentication schemes. FIDO2 provides the strongest security, followed by TOTP, and then followed by passwords. For TOTP and password-based login, larch provides no protection against *credential breaches*: if an attacker steals users' authentication secrets (MAC keys or passwords) from the relying party, the attacker can use those secrets to authenticate without those authentications appearing in the log. FIDO2 defends against credential breaches because the relying party only ever sees the client's public key.

Larch does protect against *device compromise* for all three authentication mechanisms: even if an attacker gains control of a user's device, generating any of the user's larch-protected credentials requires communicating with the log service and results in an archived log record. If the user discovers the device break-in later on, she can recover from the log a list

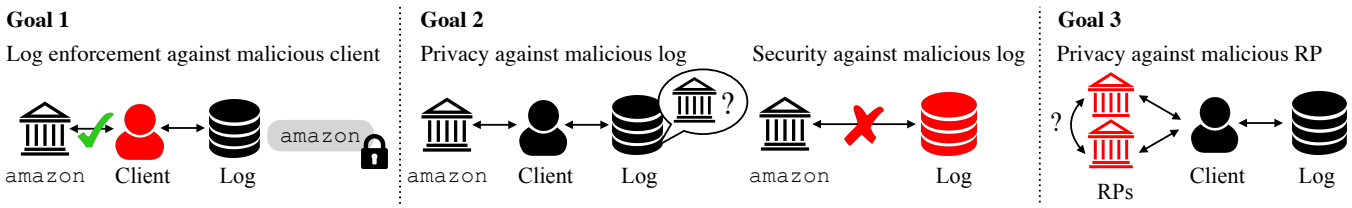

Figure 2: Larch security goals.

of authentications and take steps to remediate the effects of compromise (contacting the affected relying parties, etc.).

An attacker who compromises an account can often disable two-factor authentication or add its own credentials to a compromised account. Therefore, only an attacker's first successful access to a given relying party is guaranteed to be archived in larch. That said, many relying parties send out notifications, require step-up authentication, or revoke access to logged in clients on credential updates, all of which could complicate an attack or alert legitimate users to a problem. Hence, it is valuable to ensure that all accesses with the original account credentials are logged. Larch can make this guarantee for FIDO2, where every authentication requires a unique two-party signature. It does not provide this guarantee with passwords, as the attacker learns the password as part of the authentication process: only the attacker's first authentication to a given relying party will be logged. With TOTP, each generated code produces a larch log record. Some relying parties implement a TOTP replay cache, in which case one code allows one login. Other relying parties allow a single TOTP code to be used for arbitrarily many authentications in a short time period (generally about a minute).

Fortunately, when recovering from compromise, a user is most interested in learning whether an attacker has accessed an account zero times or more than zero times. For larch-generated credentials, users will always be able to learn this information from the larch log. However, if users import passwords that are not unique into larch, this guarantee does not hold. By default, the larch client software generates a unique random password for every relying party, but it also allows user to import existing legacy passwords, which might not be unique. In the event of password reuse, the attacker can generate a single log record to obtain the password and then use it to authenticate to all affected relying parties.

3 Logging for FIDO2

3.1 Background

FIDO2 protocol. The FIDO2 protocol [36, 80] allows a client to authenticate using cryptographic keys stored on a device (e.g., a Yubikey hardware token or a Google passkey). To register with a relying party (e.g., `github.com`), the client generates an ECDSA keypair, stores the secret key, and sends the public key to the relying party. When the client subsequently wants to authenticate to relying party `github.com`, Github's server sends the client a random challenge. The client then signs the hash of the string `github.com` and the Github-chosen challenge using the secret key the client generated for `github.com` at registration. If the signature is valid, the Github server authorizes the client. Because the message signed by the client is bound to the name `github.com`, FIDO2 provides a strong defense against phishing attacks. The FIDO2 protocol supports passwordless, second-factor, and multi-factor authentication.

Zero-knowledge arguments. Informally, zero-knowledge arguments allow a prover to convince a verifier that a statement is true without revealing *why* the statement is true [43]. More precisely, we consider non-interactive zero-knowledge argument systems [13, 35] in the random-oracle model [10]. Both the prover and verifier hold the description of a computation C and a public input x. The prover's goal is to produce a proof π that convinces the verifier that there exists a witness w that causes $C(x, w) = 1$, without revealing the witness w to the verifier. We require the standard notions of completeness, soundness, and zero knowledge [13, 43]. Throughout the paper, we will refer to this type of argument system as a "zero-knowledge proof."

We use the ZKBoo protocol [54, 42, 20] for proving statements about computations expressed as Boolean circuits. Our system could also be instantiated with succinct non-interactive arguments of knowledge, which would decrease proof size and verification time, but at the cost of increasing proving time and requiring large parameters generated via a separate setup algorithm [12, 39, 45, 71].

Threshold signatures. A two-party threshold signature scheme [27, 28] is a set of protocols that allow two parties to jointly generate a single public key along with two shares of the corresponding secret key and then jointly sign messages using their secret key shares such that the signature verifies under the joint public key. Informally, no malicious party should be able to subvert the protocols to extract another party's share of the secret key or forge a signature on a message other than the honest party's message. We would ideally instantiate our system using BLS multisignatures [14]. Unfortunately, the FIDO2 standard limits the choice of signing algorithms to ECDSA and RSASSA [67]. For backwards-compatibility, we present a construction for two-party ECDSA signing with preprocessing tailored to our setting in Section 3.3.

3.2 Split-secret authentication

We now describe our split-secret authentication protocol for FIDO2 where the authentication secret is split between the larch client software and the log service. The key challenge is achieving log enforcement and log privacy simultaneously: every successful authentication should result in a valid log entry encrypting the identity of the relying party, but the log should not learn the identity of the relying party.

We use threshold signing to ensure that both the client and log participate in every successful authentication. A natural way to use threshold signing would be to have the client and log each generate a new threshold signing keypair at every registration. Unfortunately, if the log service used a different key share for each relying party, it would know which authentication requests correspond to the same relying party, violating Goal 2 (privacy against a malicious log). Instead, we have the log use the *same signing-key share for all relying parties*. The client still uses a different signing-key share per party, ensuring the public keys are unlinkable across relying parties. To authenticate to a relying party with identifier id and challenge chal, the client computes a digest $\mathsf{dgst} = \mathsf{Hash}(\mathsf{id}, \mathsf{chal})$ that hides id. The client and log then jointly sign dgst.

We also need to ensure that the log service obtains a correct record of every authentication. In particular, the log should only participate in threshold signing if it obtains a valid encryption ct of the relying-party identifier id [77].

To be valid, a ciphertext ct must (1) decrypt to id under the archive key k established for that client, and (2) be correctly related to the digest dgst that the log will sign (i.e., $\mathsf{Dec}(k, \mathsf{ct}) = \mathsf{id}$ and $\mathsf{dgst} = \mathsf{Hash}(\mathsf{id}, \mathsf{chal})$). To allow the log service to check that the client is using the right archive key without learning the key, we use a commitment scheme. During enrollment, the client generates a commitment cm to the archive key k using random nonce r and sends cm to the log service. During authentication, the client uses a zero-knowledge proof to prove to the log that it knows a key k, randomness r, relying-party identifier id, and authentication challenge chal such that ciphertext ct, digest dgst, and commitment cm from enrollment meet the following conditions:

(a) $\mathsf{cm} = \mathsf{Commit}(k, r)$,

(b) $\mathsf{id} = \mathsf{Dec}(k, \mathsf{ct})$, and

(c) $\mathsf{dgst} = \mathsf{Hash}(\mathsf{id}, \mathsf{chal})$.

The public inputs are the ciphertext ct, digest dgst, and commitment cm (known to the client and log); the witness is the archive key k (known only to the client), commitment opening r, relying-party identifier id, and challenge chal.

Final protocol. We now outline our final protocol.

Enrollment. During enrollment, the client samples a symmetric encryption key k as the archive key and commits to it with some random nonce r. The client sends the commitment cm to the log, and the log generates a signing-key share for the user. The log sends the client the public key corresponding to its signing-key share to allow the client to derive future keypairs for relying parties.

Registration. At registration, the client generates a new signing-key share for that relying party. The client then aggregates the log's public key with its new signing-key share and sends the resulting public key to the relying party. No interaction with the log service is required.

Authentication. To authenticate to id with challenge chal, the client computes $\mathsf{dgst} \leftarrow \mathsf{Hash}(\mathsf{id}, \mathsf{chal})$ and $\mathsf{ct} \leftarrow \mathsf{Enc}(k, \mathsf{id})$. The client then generates a zero-knowledge proof π that it knows an archive key k, commitment nonce r, relying-party identifier id, and authentication challenge chal such that dgst and ct are correctly related relative to the commitment cm that the client generated at enrollment. The client sends dgst, ct, and π to the log service. The log service checks the proof and, if it verifies, runs its part of the threshold signing protocol. The log service stores ct and returns its signature share to the client. The log service also stores the current time and client IP address with ct, allowing the user to obtain additional metadata by auditing. Finally, the client completes the threshold signature and sends it to the relying party.

Auditing. To audit the log, the client requests the list of ciphertexts and metadata from the log service and decrypts all of the relying-party identifiers.

3.3 Two-party ECDSA with preprocessing

Section 3.2 shows how to implement larch for any two-of-two threshold signing scheme that cryptographically hashes input messages. However, FIDO2 compatibility forces us to use ECDSA, which is more cumbersome than BLS to threshold. We present a concretely efficient protocol for ECDSA signing between the client and log.

There is a large body of prior work on multi-party ECDSA signing [30, 61, 22, 4, 23, 18, 48, 41, 40, 19]. However, existing protocols are orders of magnitude more costly than the one we present here [61, 41, 40, 18, 19]. The efficiency gain for us comes from the fact that we may assume that the client is *honest at enrollment time and only later compromised*. In contrast, standard schemes for two-party ECDSA signing must protect against the compromise of either party at any time. Prior protocols provide this stronger security property at a computational and communication cost. In our setting, we need only ensure that an honest client can run an enrollment procedure with the log service such that if the client is later compromised, the attacker cannot subvert the signing protocol.

We leverage the client to split signing into two phases:

1. During an *offline phase*, which takes place during enrollment, the client performs some preprocessing to produce a "presignature." Security only holds if the client is honest during the offline phase.
2. During an *online phase*, which takes place during authentication, the client and log service use the presignature to

perform a lightweight, message-dependent computation to produce an ECDSA signature. Security holds if either the client or log service is compromised during the online phase.

Prior work also splits two-party signing into an offline and online phase. However, prior work performs this partitioning to reduce the online time at the expense of a more costly offline phase [22, 85, 23, 18]. (The offline phase in these schemes is expensive since the protocols do not assume that both parties are honest during the offline phase.) We split the signing scheme into an offline and online phase to take advantage of the fact that we may assume that the client is honest in the offline phase and so can reduce the total computation time this way.

An additional requirement in our setting is that the log should *not* learn the public key that the signature is generated under. Because the public key is specific to a relying party, hiding the public key is necessary for ensuring that the log cannot distinguish between relying parties. The signing algorithm can take as input a relying-party-specific key share from the client and a relying-party-independent key share from the log.

Background: ECDSA. For a group $\mathbb{G}$ of prime order q with generator g, fixed in the ECDSA standard, an ECDSA secret key is of the form $\mathsf{sk} \in \mathbb{Z}_q$, where $\mathbb{Z}_q$ denotes the ring of integers modulo q. The corresponding ECDSA public key is $\mathsf{pk} = g^{\mathsf{sk}} \in \mathbb{G}$. ECDSA uses a hash function $\mathsf{Hash}\colon \{0,1\}^* \to \mathbb{Z}_q$ and a "conversion" function $f : \mathbb{G} \to \mathbb{Z}_q$. To generate an ECDSA signature on a message $m \in \{0,1\}^*$ with secret key $\mathsf{sk} \in \mathbb{Z}_q$, the signer samples a signing nonce $r \xleftarrow{R} \mathbb{Z}_q$ and computes

$$r^{-1} \cdot (\mathsf{Hash}(m) + f(g^r) \cdot \mathsf{sk}) \quad \in \mathbb{Z}_q.$$

We give the ECDSA signing algorithm in detail in Appendix A.

Our construction. We now describe our construction for a two-party ECDSA signing protocol with presignatures. (See Appendix C for technical details.) To generate the log keypair, the log samples $x \xleftarrow{R} \mathbb{Z}_q$, sets its secret key to $x \in \mathbb{Z}_q$, and sets its public key to $X = g^x \in \mathbb{G}$. Then to generate a keypair from the log public key, the client samples $y \xleftarrow{R} \mathbb{Z}_q$ and sets the relying-party-specific public key to $\mathsf{pk} = X \cdot g^y \in \mathbb{G}$. For each public key of the form $g^{x+y} \in \mathbb{G}$, the log has one share $x \in \mathbb{Z}_q$ of the secret key that is the same for all public keys and the client has the other share $y \in \mathbb{Z}_q$ of the secret key that is different for each public key.

We split the signature-generation process into two parts:

1. *Offline phase*: a message-independent, key-independent "presignature" algorithm that the client runs, and
2. *Online phase*: a message-dependent, key-dependent signing protocol that the log and client run jointly.

To generate the presignature in the offline phase, the client samples a signing nonce $r \xleftarrow{R} \mathbb{Z}_q$, computes $R \leftarrow g^r \in \mathbb{G}$, and splits r^{-1} into additive secret shares: $r^{-1} = r_0 + r_1 \in \mathbb{Z}_q$. The log's portion of the presignature is $(f(R), r_0) \in \mathbb{Z}_q^2$, and the client's portion is $(f(R), r_1) \in \mathbb{Z}_q^2$. Then, to produce a signature on a message in the online phase, the client and log simply perform a single secure multiplication to compute

$$r^{-1} \cdot (\mathsf{Hash}(m) + f(R) \cdot \mathsf{sk}) \in \mathbb{Z}_q$$

where $r^{-1} \in \mathbb{Z}_q$ (signing nonce) and $\mathsf{sk} \in \mathbb{Z}_q$ (signing key) are secret-shared between the client and log.

To perform this multiplication over secret-shared values, we use Beaver triples [9]. A Beaver triple is a set of one-time-use shares of values that the log and client can use to efficiently perform a two-party multiplication on secret-shared values. Traditionally, generating Beaver triples is one of the expensive portions of multiparty computation protocols (e.g., in prior work on threshold ECDSA [22]). In our setting, the client at enrollment time can generate a Beaver triple as part of the presignature. Note that the client and log can use each signing nonce and Beaver triple exactly once. That is, the client and log must use a fresh presignature to generate each signature.

Malicious security. By deviating from the protocol, neither the client nor the log should be able to learn secret information (i.e., the other party's share of the secret key or signing nonce) or produce a signature for any message apart from the one that the protocol fixes. We describe how to accomplish this using traditional tools for malicious security (e.g. information-theoretic MACs [24]) in Appendix C.

Formalizing and proving security. We define and prove security in Appendix B and Appendix C.

Implications for system design. Our preprocessing approach increases the client's work at enrollment: the client generates some number of presignatures (e.g., 10K) and sends the log's presignature shares to the log. To reduce storage burden on the log, the client can store encryptions of the log's presignature shares.

When the client is close to running out of presignatures, it can authenticate with the log, generate more presignatures, and send the log's presignature shares to the log service. If the log service does not receive an objection after some period of time, it will start using the new presignatures. An honest client periodically checks the log to see whether any unexpected presignatures (created by an attacker) appear in its log. If the client learns that a new batch of presignatures was generated that the client did not authorize, the client authenticates to the log service and objects.

If the client runs out of presignatures and the log service rejects the client's presignatures, the client and the log can temporarily use a more expensive signing protocol that does not require presignatures [41, 61, 30, 85]. The client could run out of presignatures and be forced to use the slow multisignature protocol in the following cases:

1. The attacker compromised the user's credentials with the log service, allowing the attacker to object to the new presignatures. In this case, the attacker could change the user's credentials and permanently lock the user out of her account.

2. The honest client was close to running out of presignatures, generated new presignatures, and then ran out of presignatures while waiting for a possible objection. This scenario only occurs when the honest client makes an unexpectedly large number of authentications in a short period of time. The client only needs to pay the cost of the slow multisignature protocol for a short period of time.

An attacker that has compromised the log service can also deny service, as we discussed in Section 2.4.

4 Logging for time-based one-time passwords

We now show how larch can support time-based one-time passwords (TOTP).

4.1 Background: TOTP

TOTP is a popular form of second-factor authentication that authenticator apps (Authy, Google Authenticator, and others [68]) implement. When a client registers for TOTP with a relying party, the relying party sends the client a secret cryptographic key. Then, to authenticate, the client and the relying party both compute a MAC on the current time using the secret key from registration. The client sends the resulting MAC tag to the server. If the client's submitted tag matches the one that the server computes, the relying party authorizes the client. TOTP uses a hash-based MAC (HMAC).

4.2 Split-secret authentication for TOTP

At a high level, in our split-secret authentication protocol for TOTP, both the client and log service have as private input additive secret shares of the TOTP secret key. At the conclusion of the split-secret authentication, the client holds a TOTP code and the log service holds a ciphertext. We now give the details of our protocol.

Enrollment. At enrollment, just as with FIDO2, the client generates and stores a long-term symmetric-encryption archive key k and random nonce r. Then, the client sends the commitment $\mathsf{cm} = \mathsf{Commit}(k, r)$ to the log service.

Registration. To register a client, a relying party generates and sends the client a secret MAC key k_{id} for TOTP. The client samples a random identifier id for the relying party and then splits the TOTP secret key k_{id} into additive secret shares $\mathsf{klog}_{\mathsf{id}}$ and $\mathsf{kclient}_{\mathsf{id}}$. The client sends $(\mathsf{id}, \mathsf{klog}_{\mathsf{id}})$ to the log service and locally stores $(\mathsf{id}, \mathsf{kclient}_{\mathsf{id}})$ alongside a name identifying the relying party (e.g., `user@amazon.com`).

Authentication. In order to authenticate to the relying party id at time t, the client needs to compute $\mathsf{HMAC}(\mathsf{k}_{\mathsf{id}}, t)$ with the help of the log service. Let n be the number of relying parties with which the client has registered. To authenticate, the client and log service run a secure two-party computation where:

- The client's input is its long-term symmetric archive key k and commitment opening r from enrollment, the relying-party identifier id, and the client's share of the TOTP key $\mathsf{kclient}_{\mathsf{id}}$.
- The log service's input is the commitment cm from enrollment, the list of relying-party identifiers that the client has registered with $(\mathsf{id}_1, \ldots, \mathsf{id}_n)$, and the log service's TOTP key shares $(\mathsf{klog}_{\mathsf{id}_1}, \ldots, \mathsf{klog}_{\mathsf{id}_n})$—one per relying party.
- The client outputs the TOTP code $\mathsf{HMAC}(\mathsf{k}_{\mathsf{id}}, t)$.
- The log outputs an encrypted log record: an encryption of the relying-party identifier id under the archive key k.

We execute this two-party computation using an off-the-shelf garbled-circuit-based multiparty computation protocol. Garbled circuits allow two parties to jointly execute any Boolean circuit on private inputs, where neither party learns information about the other's input beyond what they can infer from the circuit's output [87]. We use the protocol from Wang et al. [84], which provides malicious security, meaning that the protocol remains secure even if one corrupted party deviates arbitrarily from the protocol. As long as either the client or the log service is honest, the log service does not learn any information about the client's authentication secrets, and the client learn no information about the TOTP secret, apart from the single TOTP code that the protocol outputs. Because we use an off-the-shelf garbled-circuit protocol, the communication overhead is much higher than in the special-purpose protocols we design for FIDO2 and passwords (Section 8). TOTP is challenging to design a special-purpose protocol for because the authentication credential must be generated via the SHA hash function which, unlike the authentication credentials for FIDO2 and passwords, does not have structure we can exploit. Clients can ask the log service to delete registrations for unused accounts to speed up the two-party computation.

Auditing. To audit the log, the client simply requests the list of ciphertexts from the log service. The client decrypts each ciphertext with its archive key k and then, using its mapping of id values to relying party names, outputs the resulting list of relying party names.

5 Logging for passwords

We now describe how larch can support passwords.

5.1 Protocol overview

We construct a split-secret authentication protocol that takes place between the client and the log service. In particular, we show how the client can compute the password to authenticate to a relying party in such a way that (a) the log service does not learn the relying party's identity and (b) the client's authentication attempt is logged. At the start of the authentication protocol run:

- the client holds a secret key, the log service's public key, and the identity id^* of the relying party it wants to authenticate to, and
- the log service holds its own secret key, the client's public key, and a list of relying-party identities $(\mathsf{id}_1, \ldots, \mathsf{id}_n)$ at which the client has registered.

At the end of the authentication protocol run:

- the client holds a password derived as a pseudorandom function of the client's secret, the log's secret, and the relying party identity id^*, and
- the log service holds a ciphertext encrypting the relying party's identity id^* under the client's public key.

Limitations inherent to passwords. As we discussed in Section 2.4, larch for passwords does not protect against credential breaches, but does defend against device compromise.

5.2 Split-secret authentication for passwords

The larch scheme for password-based authentication uses a cyclic group $\mathbb{G}$ of prime order q with a fixed generator $g \in \mathbb{G}$. Our implementation uses the NIST P-256 elliptic-curve group.

When using password-based authentication in larch, the client and log service after registration each hold a secret share of the password for each relying party. In particular, the password for a relying party with identity $\mathsf{id} \in \{0,1\}^*$ is the string $\mathsf{pw}_{\mathsf{id}} = k_{\mathsf{id}} \cdot \mathsf{Hash}(\mathsf{id})^k \in \mathbb{G}$, where:

- $k_{\mathsf{id}} \in \mathbb{Z}_q$ is a per-relying-party secret share held by the client,
- $\mathsf{Hash}\colon \{0,1\}^* \to \mathbb{G}$ is a hash function, and
- $k \in \mathbb{Z}_q$ is a per-client secret key held by the log service.

Thus, computing $\mathsf{pw}_{\mathsf{id}}$ requires both the client's per-site key k_{id} and the log's secret key k.

The technical challenge is to construct a protocol that allows the client to compute the password $\mathsf{pw}_{\mathsf{id}}$ while (a) hiding id from the log service and (b) ensuring that the log service completes the interaction holding an encryption of id under the client's public key.

Protocol. We describe the protocol steps:

Enrollment. The client samples an ElGamal secret key $x \in \mathbb{Z}_q$ as the archive key and sends the corresponding public key $X = g^x \in \mathbb{G}$ to the log service. The log service samples a Diffie-Hellman secret key $k \in \mathbb{Z}_q$ and sends its public key $K = g^k \in \mathbb{G}$ to the client.

Registration. The client samples a per-relying-party random identifier $\mathsf{id} \xleftarrow{R} \{0,1\}^{128}$, saves id locally alongside the name of the relying party (e.g., `user@amazon.com`), and sends id to the log service. The log service saves the string $\mathsf{Hash}(\mathsf{id})$ and replies with $\mathsf{Hash}(\mathsf{id})^k \in \mathbb{G}$. To generate a new strong password $\mathsf{pw}_{\mathsf{id}}$ (the recommended use), the client samples and saves a random key share $k_{\mathsf{id}} \xleftarrow{R} \mathbb{G}$ and sets $\mathsf{pw}_{\mathsf{id}} \leftarrow k_{\mathsf{id}} \cdot \mathsf{Hash}(\mathsf{id})^k \in \mathbb{G}$. To import a legacy password $\mathsf{pw}_{\mathsf{id}}$ (less secure), the client computes and stores $k_{\mathsf{id}} \leftarrow \mathsf{pw}_{\mathsf{id}} \cdot \left(\mathsf{Hash}(\mathsf{id})^k\right)^{-1} \in \mathbb{G}$. The client then deletes $\mathsf{Hash}(\mathsf{id})^k$ and $\mathsf{pw}_{\mathsf{id}}$. Note that the log server can discard id, which it only uses to avoid providing h^k for arbitrary h. When the client samples id and k_{id} randomly in the recommended usage, the password $\mathsf{pw}_{\mathsf{id}}$ for each relying party is random and distinct.

Authentication. During authentication, the client must recompute the password $\mathsf{pw}_{\mathsf{id}}$. To do so, the client first sends the log service an encryption of $\mathsf{Hash}(\mathsf{id})$ under the public ElGamal archive key g^x: the client samples $r \xleftarrow{R} \mathbb{Z}_q^*$ and computes the ciphertext $(c_1, c_2) = (g^r, \mathsf{Hash}(\mathsf{id}) \cdot g^{xr}) \in \mathbb{G}^2$. In addition, the client sends a zero-knowledge proof to the log service attesting to the fact that (c_1, c_2) is an encryption under the client's public key X of $\mathsf{Hash}(\mathsf{id})$ for $\mathsf{id} \in \{\mathsf{id}_1, \mathsf{id}_2, \ldots, \mathsf{id}_n\}$—the set of relying-party identifiers that the client sent to the log service during each of its registrations so far. The client executes this proof using the technique from Groth and Kohlweiss [46]. The proof size is $O(\log n)$ and the prover and verifier time are both $O(n)$. (See Appendix D for implementation details.)

The log service saves the ciphertext as a log entry, checks the zero-knowledge proof, and returns the value $h = c_2^k = \mathsf{Hash}(\mathsf{id})^k \cdot g^{xrk} \in \mathbb{G}$ to the client. The client can then compute

$$\mathsf{pw}_{\mathsf{id}} = k_{\mathsf{id}} \cdot h \cdot K^{-xr} = k_{\mathsf{id}} \cdot \mathsf{Hash}(\mathsf{id})^k \quad \in \mathbb{G}.$$

Crucially, the client deletes $\mathsf{pw}_{\mathsf{id}}$ after authentication to ensure that future authentications must again interact with the log service.

Auditing. To audit the log, the client downloads the ElGamal ciphertexts and can decrypt each ciphertext to recover a list of hashed identities: $(\mathsf{Hash}(\mathsf{id}_1), \mathsf{Hash}(\mathsf{id}_2), \ldots)$. The client uses its stored mapping of ids to relying-party identifiers to recover the plaintext names of the relying parties in the log.

6 Protecting against log misbehavior

The larch log service must participate in each of the user's authentication attempts. If the log service goes offline, the user will not be able to authenticate to any of her larch-enabled relying parties. In a real-world deployment, the log service could consist of multiple servers replicated using standard state-machine replication techniques to tolerate benign failures [58, 70]. However, users might also worry about intentional denial-of-service attacks on the part of the log.

To defend against availability attacks, a user can split trust across multiple logs. At enrollment time, the user can enroll with n logs. Then at registration, the user can set a threshold t of logs that must participate in authentication. Thus, the user can authenticate to her accounts so long as t logs are online, and she can audit activity so long as $n - t + 1$ logs are available. We need $n - t + 1$ logs to be available for auditing in order to guarantee that at least one of the t logs that participated in authentication is online. To ensure that colluding logs cannot authenticate on behalf of a client, the user's client

can run $n+1$ logical parties, and $n+t+1$ parties can generate an authentication credential. In the setting with multiple log services, we need to adapt our two-party protocols to threshold multi-party protocols. Although we present our techniques for two parties (the client and a single log), our techniques generalize to multiple parties in a straightforward way.

For FIDO2 and passwords, the client now sends a zero-knowledge proof to each of the n logs. In the password case, the client can then retrieve (t,n) Shamir shares of the password [75], and in the FIDO2 case, the client can run any existing multi-party threshold signing protocol that does not take the public key as input [76, 22]. For TOTP, the client and the n logs can execute the same circuit using any malicious-secure threshold multi-party computation protocol [11].

Note that for relying parties that support FIDO2, users can optionally register a backup hardware FIDO2 device to allow them to bypass the log. In this case, the user can authenticate either via larch or via her backup FIDO2 key. While registering a backup hardware device protects against availability attacks, if an attacker obtains this hardware device, they can authenticate as the user without interacting with the log.

7 Implementation

We implemented larch for FIDO2, TOTP, and passwords with a single log service. We use C/C++ with gRPC and OpenSSL with the P256 curve (required by the FIDO2 standard). We wrote approximately 5,700 lines of C/C++ and 50 lines of Javascript (excluding tests and benchmarks). Our implementation is available at `https://github.com/edauterman/larch`.

For our FIDO2 implementation, we implemented a ZKBoo [42] library for arbitrary Boolean circuits. Our ZKBoo implementation (with optimizations from ZKB++ [20]) uses emp-toolkit to support arbitrary Boolean circuits in Bristol Fashion [83]. To support the parallel repetitions required for soundness error $< 2^{-80}$, we use SIMD instructions with a bitwidth of 32 and run 5 threads in parallel. For the proof circuit, we use AES in counter mode for encryption and SHA-256 for commitments (SHA-256 is necessary for backwards compatability with FIDO2). We built a log service and client that invoke the ZKBoo library, as well as a Chrome browser extension that interfaces with our client application and is compatible with existing FIDO2 relying parties. We built our browser extension on top of an existing extension [56].

Our TOTP implementation uses a maliciously secure garbled-circuit construction [84] implemented in emp-toolkit [83]. We generated our circuit using the CBMC-GC compiler [37] with ChaCha20 for encryption and SHA-256 for commitments.

For our passwords implementation, we implemented Groth and Kohlweiss's proof system [46].

Our implementation uses a single log server for the log service, does not encrypt communication between the client and the log service, and does not require the client to authenticate with the log service. A real-world deployment would use multiple servers for replication, use TLS between the client and the log service, and authenticate the client before performing any operations.

Optimizations. We use pseudorandom generators (PRGs) to compress presignatures: the log stores 6 elements in $\mathbb{Z}_q$ and the client stores 1 element. Also, instead of running an authenticated encryption scheme (e.g. AES-GCM) inside the circuit for FIDO2 or TOTP, we run an encryption scheme without authentication (e.g. AES in counter mode) inside the circuit and then sign the ciphertext (client has the signing key, log has the verification key). The log can check the integrity of the ciphertext by verifying the signature, which is must faster than checking in a zero-knowledge proof or computing the ciphertext tag jointly in a two-party computation.

8 Evaluation

In this section, we evaluate the cost of larch to end users and the cost of running a larch log service.

Experiment setup. We run our benchmarks on Amazon AWS EC2 instances. Unless otherwise specified, we run the log service on a c5.4xlarge instance with 8 cores (2 hyperthreads per core) and 32GB of memory and, for latency benchmarks, the client on a c5.2xlarge instance with 4 cores and 16GB of memory, comparable to a commodity laptop. We configure the network connection between the client and log service to have a 20ms RTT and a bandwidth of 100 Mbps.

8.1 End-user cost

We show larch authentication latency and communication costs for FIDO2, TOTP, and passwords.

8.1.1 FIDO2

Latency. The client for our FIDO2 scheme can complete authentication in 303ms with a single CPU core, or 117ms when using eight cores (Figure 3). Loading a webpage often takes a few seconds because of network latency, so the client cost of larch authentication is minor by comparison. The client's running time during authentication is independent of the number of relying parties. The heaviest part of the client's computation is proving to the log service that its encrypted log entry is well formed.

At enrollment, the client must generate many "presignatures," which it later uses to run our authentication protocol with the log. Generating 10,000 presignatures for 10,000 future FIDO2 authentications takes 885ms. When the client runs out of presignatures, it generates new presignatures it can use after a waiting period (see Section 3.3).

Communication. During enrollment, the client must send the log 1.8MiB worth of presignatures. Thereafter, each authenti-

cation attempt requires 1.73MiB worth of communication: the bulk of this consists of the client's zero-knowledge proof of correctness, and 352B of it comes from the signature protocol. By using a different zero-knowledge proof system, we could reduce communication cost at the expense of increasing client computation cost.

Comparison to existing two-party ECDSA. For comparison, a state-of-the-art two-party ECDSA protocol [85] that does not require presignatures from the client and uses Paillier requires 226ms of computation at signing time (the authors' measurements exclude network latency, which we estimate would add 80ms) and 6.3KiB of per-signature communication. In contrast, our signing protocol only requires 0.5KiB per-signature communication (including the log presignature and the signing messages) and takes 61ms time at signing, almost all of which is due to network latency and can be run in parallel with proving and verifying as the computational overhead is minimal (1ms).

8.1.2 TOTP

Latency. In Figure 3 (right), we show how TOTP authentication latency increases with the number of relying parties the user registered with. Because we implement TOTP authentication using garbled circuits [84], we can split authentication into two phases: an "offline", input-independent phase and an "online", input-dependent phase (the log service and client communicate in both phases). Both phases are performed once per authentication. However, the offline phase can be performed in advance of when the user needs to authenticate to their account, and so it does not affect the latency that the user experiences. For 20 relying parties, the online time is 91ms and the offline time is 1.23s. For 100 relying parties, the online time is 120ms and the offline time is 1.39s.

Communication. Communication costs for our TOTP authentication scheme are large: for 20 relying parties, the total communication cost is 65MiB, and for 100 relying parties, the total communication is 93MiB. The online communication costs are much smaller: for 20 relying parties, the online communication is 202KiB and for 100 relying parties, the online communication is 908KiB. We envision clients running the offline phase in the background while they have good connectivity. While these communication costs are much higher than those associated with FIDO2 or passwords, we expect users to authenticate with TOTP less frequently because TOTP is only used for second-factor authentication.

8.1.3 Passwords

Latency. In Figure 3 (center), we show how password authentication latency increases with the number of registered relying parties. With 16 relying parties, authentication takes 28ms, and with 512 relying parties, it takes 245ms: the authentication time grows linearly with the number of relying parties. The proof system we use requires padding the number of relying parties to the nearest power of two, meaning that registering at additional relying parties does not affect the latency or communication until the number of relying parties reaches the next power of two.

Communication. In Figure 5, we show how communication increases logarithmically with the number of relying parties. This behavior is due to the fact that proof size is logarithmic in the number of relying parties. With 16 relying parties, the communication is 1.47KiB, and with 512 relying parties, it is 4.14KiB.

8.2 Cost to deploy a larch service

If successful, larch can become much simpler and more efficient with a little support from future FIDO specifications (see Section 9). Nonetheless, we show larch is already practical by analyzing the cost of deploying a larch service today (Table 6). We expect a larch log service to perform many password-based authentications, some FIDO2 authentications, and a comparatively small number of TOTP authentications. This is because the majority of relying parties only support passwords, and relying parties typically require second-factor authentication only from time to time.

Throughout this section, we consider password-based authentication with 128 relying parties (based on the fact that the average user has roughly 100 passwords [73]) and TOTP-based authentication with 20 relying parties (based on the fact that Yubikey's maximum number of TOTP registrations is 32 [2]). The authentication overhead of FIDO2 in larch is independent of the number of relying parties the user has registered with.

Storage. For each of the three protocols, the log service must store authentication records (timestamp, ciphertext, and signature). FIDO2 and TOTP have 88B authentication records, and passwords have 138B records (due to the size of ElGamal ciphertexts). The FIDO2 protocol additionally requires the client to generate presignatures for the log, each of which is 192B. For 10K presignatures, the log service must store 1.83MiB. In Figure 4 (left), we show how per-client log storage actually decreases as presignatures are consumed and replaced by authentication records. To minimize storage costs, the log service can encrypt its presignatures and store them at the client. The log service then simply needs to keep a counter to prevent presignature re-use.

Throughput. In Table 6, we show the number of auths/s a single log service core can support assuming 128 passwords and 20 TOTP accounts. We achieve the highest throughput for passwords (47.62 auths/cores/s), which are the most common authentication mechanism. For FIDO2, which can be used as either a first or second authentication factor and is supported by fewer relying parties than passwords, we achieve 6.18 auths/core/s. Finally, for TOTP, which is only used as a second factor, we achieve 0.73 auths/core/s.

Our FIDO2 protocol can be instantiated with any NIZK proof system to achieve a different tradeoff between authentication latency and log service throughput. For example, we instan-

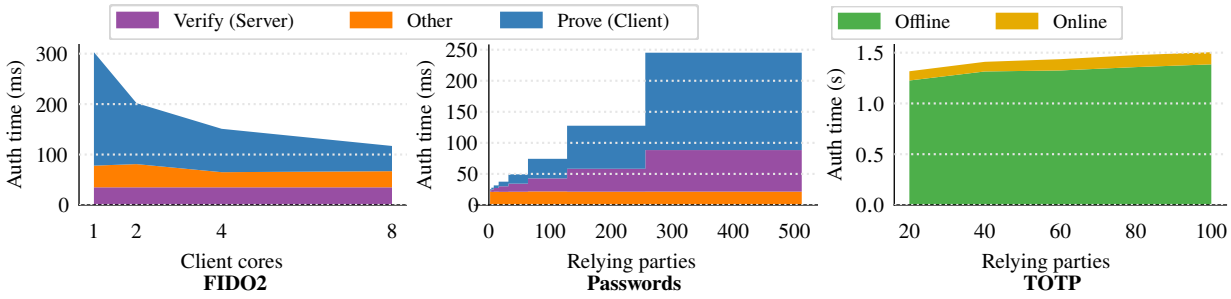

Figure 3: On the left, larch FIDO2 latency decreases as the number of client cores increases (latency is independent of the number of relying parties). In the center, larch password latency grows with the number of relying parties, with the majority of the time spent on client proof generation. On the right, larch TOTP latency grows with the number of relying parties, with the majority of the time spent in an input-independent "offline" phase as opposed to the input-dependent "online" phase (both phases require network communication).

tiate our system with ZKBoo, but could also use Groth16 [45] to reduce communication and verifier time (increasing log throughput). We measure the performance of Groth16 on our larch FIDO2 circuit on the BN-128 curve using ZoKrates [91] with libsnark [57] with a single core (we only measure the overhead of SHA-256, which dominates circuit cost, to provide a performance lower bound). While the verifier time is much lower (8ms) and the proof is much smaller (4.26KiB), (1) the trusted setup requires the client to store 19.86MiB and the log service to store 9.2MiB per client, and (2) the proving time is 4.07s, meaning that authentication latency is much higher.

Cost. We now quantify the cost of running a larch log service. The cost of one core on a c5 instance is \$0.0425-\$0.085/hour depending on instance size [1]. Data transfer to AWS instances is free, and data transfer from AWS instances costs \$0.05-\$0.09/GB depending on the amount of data transferred per month [1]. In Table 6, we show the cost of supporting 10M authentications for each authentication method with larch.

Supporting 10M authentications requires 450 log core hours for FIDO2, 3,832 log core hours for TOTP, and 59 log core hours for passwords. Compute for 10M authentications costs \$19.13-\$38.25 for FIDO2, \$162.86-\$325.72 for TOTP, and \$2.51-\$5.02 for passwords. Communication for 10M authentications costs \$0.10-\$0.19 for FIDO2, \$17,923-\$32,262 for TOTP, and \$0.015-\$0.027 for passwords. The high cost for TOTP is due to the large amount of communication required at authentication: the log service must send the client 36.8MiB for every authentication. In both the FIDO2 and password protocols, the vast majority of the communication overhead is due to the proof sent from the client to the log service, which incurs no monetary cost. We show how cost increases with the number of authentications for each of the the authentication methods in Figure 4 (right).

TOTP is substantially more expensive than FIDO2 or passwords. However, we expect a relatively small fraction of authentication requests to be for TOTP.

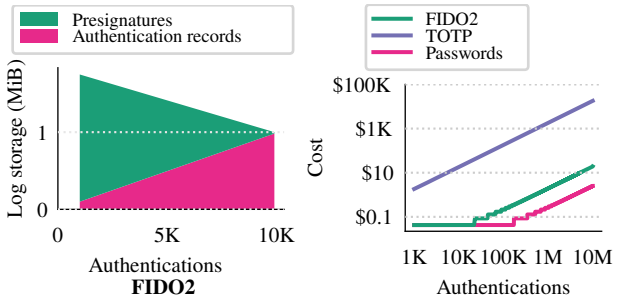

Figure 4: On the left, per-client storage overhead at the log decreases as presignatures are replaced with authentication records (client enrolls with 10K presignatures). On the right, minimum cost of supporting more authentications with passwords, (128 relying parties), FIDO2, and TOTP (20 relying parties). Both axes use a logarithmic scale.

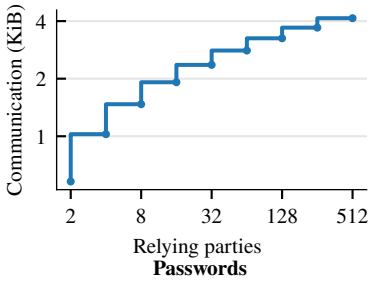

Figure 5: Communication for larch with passwords increases logarithmically with the number of relying parties (both axes use a logarithmic scale).

9 Discussion

Deployment strategy. Because larch supports passwords, TOTP, and FIDO2, people can use it with the vast majority of web services. In addition, larch offers users many of the benefits of FIDO2 without a dedicated hardware security token, particularly FIDO2's protection against phishing. The flexibility for users to choose log services can foster an ecosystem of new security products, such as log services that request login confirmation via a mobile phone app, apps that monitor the log to notify users of anomalous behavior, or enterprise security products that monitor access to arbitrary

	FIDO2	TOTP	Password
Online auth time	150 ms	91 ms	74 ms
Total auth time	150 ms	1.32 s	74 ms
Online auth comm.	1.73 MiB	201 KiB	3.25 KiB
Total auth comm.	1.73 MiB	65 MiB	3.25 KiB
Auth record	88 B	88 B	138 B
Log presignature	192 B	∅	∅
Log auths/core/s	6.18	0.73	47.62
10M auths min cost	$19.19	$18,086	$2.48
10M auths max cost	$38.37	$32,588	$4.96

Table 6: Costs for larch with FIDO2, TOTP (20 relying parties), and passwords (128 relying parties). We take the cost of one core on a c5 instance to be $0.0425-$0.085/hour (depending on instance size) and data transfer out of AWS to cost $0.05-$0.09/GB (depending on amount of data transferred) [1]. For comparison, the Argon2 password hash function should take 0.5s using 2 cores.

third-party services that a company could contract with.

FIDO improvements. Larch can benefit from enhancements we hope to see considered for future versions of the FIDO specification. One simple improvement would be to support BLS signatures, which are easier to threshold and so eliminate larch's need for presignatures [14].

Future versions of FIDO could also directly support secure client-side logging by allowing the relying party to compute the encrypted log record itself. The relying party could then ensure that the log service receives the correct encrypted log record by checking for the log record in the signing payload. Specifically, the signature payload could have the form:

$$\mathsf{Hash}(\text{log-record-ciphertext}, \mathsf{Hash}(\text{remaining-FIDO-data})) \ .$$

The log server can then take the outer hash preimage as input without needing to verify anything else about the log record.

We want to allow the relying party to generate the encrypted log record without making it possible to link users across relying parties. Instead of giving the relying party the user's public key directly at registration, which would link a user's identity across relying parties, we instead give the relying party a key-private, re-randomizable encryption of the relying party's identifier (we can achieve this using ElGamal encryption). At authentication, the relying party can re-randomize the ciphertext to generate the encrypted log record.

We also hope that future FIDO revisions standardize and promote authentication metadata as part of the challenge and hypothetical log record field. For users with multiple accounts at one relying party, it would be useful to include account names as well as relying party names in signed payloads. It would furthermore improve security to allow distinct types of authentication log records for different security-sensitive operations such as authorizing payments and changing or removing 2FA on an account. An app monitoring a user's log can then immediately notify the user of such operations.

Multiple devices. Clients need to authenticate to their accounts across multiple devices, which requires synchronizing a small amount of dynamic, secret state across devices. Cross-device state could be stored encrypted at the log, or could be disseminated through existing profile synchronization mechanisms in browsers. There is a danger of the synchronization mechanism maliciously convincing two devices to use the same presignature. Therefore, presignatures should be partitioned between devices in advance, and devices should employ techniques such as fork consistency [65] to detect and deter any rollback attacks. Existing tools can help a user recover if she loses all of her devices [25, 55, 81, 62].

Enforcing client-specific policies. We can extend larch in a straightforward way to allow the log to enforce more complex policies on authentications. The client could submit a policy at enrollment time, and the log service could then enforce this policy for subsequent authentications. If the policy decision is based on public information, the log service can apply the policy directly (e.g., rate-limiting, sending push notifications to a client's mobile device). Other policies could be based on private information. For example, if we used larch for cryptocurrency wallets, the log could enforce a policy such as "deny transactions sending more than $10K to addresses that are not on the allowlist." For policies based on private information, the client could send the log service a commitment to the policy at enrollment, and the log service could then enforce the policy by running a two-party computation or checking a zero-knowledge proof.

Revocation and migration. If a client loses her device or wants to migrate her authentication secrets from an old device to a new device, she needs a way to easily and remotely invalidate the secrets on the old device. Larch allows her to do this easily. To migrate credentials to a new device, the client and log simply re-share the authentication secrets. To invalidate the secrets on the old device, the client asks the log to delete the old secret shares (client must authenticate with the log first).

Account recovery. In the event that a client loses all of her devices, she needs some way to recover her larch account. To ensure that she can later recover her account, the client can encrypt her larch client state under a key derived from her password and store the ciphertext with the larch service. The security of the backup is only as good as the security of the client's password. Alternatively, the client could choose a random key to encrypt her client state and then back up this key using her password and secure hardware in order to defend against password-guessing attacks [25].

Limitations. If an attacker compromises the client's account with the log, the attacker can access the client's entire authentication history. To mitigate this damage, the log could delete old authentication records (e.g., records older than one week) or re-encrypt them under a key that the user keeps offline.

10 Related work

Privacy-preserving single sign-on. Like larch, existing privacy-preserving single sign-on systems hide the relying party from the identity provider. Unlike larch, these systems do not protect users' accounts from a malicious attacker that compromises the identity provider, and they do not privately log the identity of the relying party. BrowserID [33] (implemented in Mozilla Persona and Firefox Accounts) and SPRESSO [34] are single sign-on services that ensure that the identity provider does not learn the identity of the relying parties. However, neither prevents colluding relying parties from linking a user's accounts across relying parties. EL PASSO [90], UnlimitID [53], UPRESSO [50], PseudoID [29] and Hammann et al. [51] show how to build single sign-on services that protect clients from curious identity providers while ensuring that relying parties cannot link users' accounts.

Separately, Privacy Pass allows a user to obtain anonymous tokens for completing CAPTCHAs, which she can then spend at different relying parties without allowing them to link her across sites [26]. Like larch, Privacy Pass does not link users across accounts, but unlike larch, Privacy Pass does not provide a mechanism for logging authentications.

Threshold signing. Our two-party ECDSA with preprocessing protocol builds on prior work on threshold ECDSA. MacKenzie and Reiter proposed the first threshold ECDSA protocol for a dishonest majority specific to the two-party setting [64]. Genarro et al. [41] and Lindell [61] subsequently improved on this protocol. Doerner et al. show how to achieve two-party threshold ECDSA without additional assumptions [30]. Another line of work supports threshold ECDSA using generic multi-party computation over finite fields [76, 22]. A number of works show how to split ECDSA signature generation into online and offline phases [23, 18, 48, 47, 38, 19, 85, 4]; in many, the offline phase is signing-key-specific, allowing for a non-interactive online signing phase, whereas we need an offline phase that is signing-key-independent. Abram et al. show how to reduce the bandwidth of the offline phase via pseudorandom correlation generators [4]. Aumasson et al. provide a survey of prior work on threshold ECDSA [8]. Arora et al. show how to split trust across a group of FIDO authenticators to enable account recovery using a new group signature scheme [6].

Proving properties of encrypted data. Larch's split-secret authentication protocol for FIDO2 and passwords relies on proving properties of encrypted data, which is also explored in prior work. Verifiable encryption was first proposed by Stadler [77], and Camenisch and Damgard introduced it as a well-defined primitive [15]. Subsequent work has designed verifiable encryption schemes for limited classes of relations (e.g. discrete logarithms) [16, 7, 86, 63, 69]. Takahashi and Zaverucha introduced a generic compiler for MPC-in-the-head-based verifiable encryption [78]. Lee et al. [60] contribute a SNARK-based verifiable encryption scheme that decouples the encryption function from the circuit by using a commit-and-prove SNARK [17]. This approach does not work for us for FIDO2 authentication because the ciphertext must be connected to a SHA-256 digest.

Grubbs et al. introduce zero-knowledge middleboxes, which enforce properties on encrypted data using SNARKs [49]. Wang et al. show how to build blind certificate authorities, enabling a certificate authority to validate an identity and generate a certificate for it without learning the identity [82]. DECO allows users to prove that a piece of data accessed via TLS came from a particular website and, optionally, prove statements about the data in zero-knowledge [89].

Transparency logs. Like larch, transparency logs detect attacks rather than prevent them, and they achieve this by maintaining a log recording sensitive actions [66, 44, 52, 21, 59, 5, 25]. However, transparency logs traditionally maintain public, global state. For example, the certificate transparency log records what certificates were issued and by whom in order to track when certificates were issued incorrectly [59]. In contrast, the larch log service maintains encrypted, per-user state about individual users' authentication history.

11 Conclusion

Larch is an authentication manager that logs every successful authentication to any of a user's accounts on a third-party log service. It guarantees log integrity without trusting clients. It furthermore guarantees account security and privacy without trusting the log service. Larch works with any existing service supporting FIDO2, TOTP, or password-based login. Our evaluation shows the implementation is practical and cost-effective.

Acknowledgements. We thank the anonymous reviewers and our shepherd Ittay Eyal for their feedback. We also thank Raluca Ada Popa for her support, as well as students in the Sky security group for giving feedback that improved the presentation of this paper. The RISELab and Sky Lab are supported by NSF CISE Expeditions Award CCF-1730628 and gifts from the Sloan Foundation, Alibaba, Amazon Web Services, Ant Group, Ericsson, Facebook, Futurewei, Google, Intel, Microsoft, Nvidia, Scotiabank, Splunk, and VMware. Emma Dauterman was supported by an NSF Graduate Research Fellowship and a Microsoft Ada Lovelace Research Fellowship. This work was funded in part by the Stanford Future of Digital Currency Initiative as well as gifts from Capital One, Facebook, Google, Mozilla, Seagate, and MIT's FinTech@CSAIL Initiative. We also received support under NSF Awards CNS-2054869 and CNS-2225441.

References

[1] Amazon EC2 On-Demand Pricing. `https://aws.amazon.com/ec2/pricing/on-demand/`, accessed December 7, 2022.

[2] How many accounts can I register my YubiKey with?, 2020. https://support.yubico.com/hc/en-us/articles/360013790319-How-many-accounts-can-I-register-my-YubiKey-with-FIDO2.

[3] Passkeys. Google, 2022. https://developers.google.com/identity/passkeys.

[4] Damiano Abram, Ariel Nof, Claudio Orlandi, Peter Scholl, and Omer Shlomovits. Low-bandwidth threshold ECDSA via pseudorandom correlation generators. In *IEEE Security & Privacy*, 2022.

[5] 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 *USENIX Security*, 2019.

[6] Sunpreet S Arora, Saikrishna Badrinarayanan, Srinivasan Raghuraman, Maliheh Shirvanian, Kim Wagner, and Gaven Watson. Avoiding lock outs: Proactive fido account recovery using managerless group signatures. *Cryptology ePrint Archive*, 2022.

[7] Giuseppe Ateniese. Verifiable encryption of digital signatures and applications. *ACM Transactions on Information and System Security (TISSEC)*, 7(1):1–20, 2004.

[8] Jean-Philippe Aumasson, Adrian Hamelink, and Omer Shlomovits. A survey of ecdsa threshold signing. *Cryptology ePrint Archive*, 2020.

[9] Donald Beaver. Efficient multiparty protocols using circuit randomization. In *CRYPTO*, 1991.

[10] Mihir Bellare and Phillip Rogaway. Random oracles are practical: A paradigm for designing efficient protocols. In *CCS*, pages 62–73, 1993.

[11] M Ben-Or, S Goldwasser, and A Wigderson. Completeness theorems for non-cryptographic fault-tolerant distributed computing. In *STOC*, pages 1–10, 1988.

[12] Nir Bitansky, Ran Canetti, Alessandro Chiesa, and Eran Tromer. From extractable collision resistance to succinct non-interactive arguments of knowledge, and back again. In *ITCS*, pages 326–349, 2012.

[13] Manuel Blum, Paul Feldman, and Silvio Micali. Non-interactive zero-knowledge and its applications. In *ACM STOC*. 1988.

[14] Dan Boneh, Manu Drijvers, and Gregory Neven. BLS multi-signatures with public-key aggregation. https://crypto.stanford.edu/~dabo/pubs/papers/BLSmultisig.html, Accessed 23 May 2022, March 2018.

[15] Jan Camenisch and Ivan Damgård. Verifiable encryption, group encryption, and their applications to separable group signatures and signature sharing schemes. In *ASIACRYPT*, pages 331–345. Springer, 2000.

[16] Jan Camenisch and Victor Shoup. Practical verifiable encryption and decryption of discrete logarithms. In *CRYPTO*, pages 126–144. Springer, 2003.

[17] Matteo Campanelli, Dario Fiore, and Anaïs Querol. Legosnark: Modular design and composition of succinct zero-knowledge proofs. In *CCS*, pages 2075–2092, 2019.

[18] Ran Canetti, Rosario Gennaro, Steven Goldfeder, Nikolaos Makriyannis, and Udi Peled. UC non-interactive, proactive, threshold ECDSA with identifiable aborts. In *CCS*, pages 1769–1787, 2020.

[19] Ran Canetti, Nikolaos Makriyannis, and Udi Peled. UC non-interactive, proactive, threshold ECDSA. *Cryptology ePrint Archive*, 2020.

[20] Melissa Chase, David Derler, Steven Goldfeder, Claudio Orlandi, Sebastian Ramacher, Christian Rechberger, Daniel Slamanig, and Greg Zaverucha. Post-quantum zero-knowledge and signatures from symmetric-key primitives. In *CCS*, pages 1825–1842, 2017.

[21] Melissa Chase, Apoorvaa Deshpande, Esha Ghosh, and Harjasleen Malvai. Seemless: Secure end-to-end encrypted messaging with less trust. In *CCS*, pages 1639–1656, 2019.

[22] Anders Dalskov, Claudio Orlandi, Marcel Keller, Kris Shrishak, and Haya Shulman. Securing DNSSEC keys via threshold ECDSA from generic MPC. In *European Symposium on Research in Computer Security*, pages 654–673. Springer, 2020.

[23] Ivan Damgård, Thomas Pelle Jakobsen, Jesper Buus Nielsen, Jakob Illeborg Pagter, and Michael Bæksvang Ostergård. Fast threshold ECDSA with honest majority. In *SCN*, pages 382–400. Springer, 2020.

[24] Ivan Damgård, Valerio Pastro, Nigel Smart, and Sarah Zakarias. Multiparty computation from somewhat homomorphic encryption. In *CRYPTO*, pages 643–662. Springer, 2012.

[25] Emma Dauterman, Henry Corrigan-Gibbs, and David Mazières. SafetyPin: Encrypted backups with Human-Memorable secrets. In *OSDI*, pages 1121–1138, 2020.

[26] Alex Davidson, Ian Goldberg, Nick Sullivan, George Tankersley, and Filippo Valsorda. Privacy pass: Bypassing internet challenges anonymously. *Proc. Priv. Enhancing Technol.*, 2018(3):164–180, 2018.

[27] Yvo Desmedt. Society and group oriented cryptography: A new concept. In *EUROCRYPT*, pages 120–127. Springer, 1987.

[28] Yvo Desmedt and Yair Frankel. Threshold cryptosystems. In *EUROCRYPT*, pages 307–315. Springer, 1989.

[29] Arkajit Dey and Stephen Weis. PseudoID: Enhancing privacy in federated login. In *HotPETS workshop*, 2010.

[30] Jack Doerner, Yashvanth Kondi, Eysa Lee, and Abhi Shelat. Secure two-party threshold ECDSA from ECDSA assumptions. In *IEEE Security & Privacy*, pages 980–997. IEEE, 2018.

[31] Corin Faife. Okta ends lapsus$ hack investigation, says breach lasted just 25 minutes. *The Verge*, April 2022. `https://www.theverge.com/2022/4/20/23034360/okta-lapsus-hack-investigation-breach-25-minutes`.

[32] Manuel Fersch, Eike Kiltz, and Bertram Poettering. On the provable security of (EC)DSA signatures. In *CCS*, 2016.

[33] Daniel Fett, Ralf Küsters, and Guido Schmitz. Analyzing the BrowserID SSO system with primary identity providers using an expressive model of the web. In *European Symposium on Research in Computer Security*, pages 43–65. Springer, 2015.

[34] Daniel Fett, Ralf Küsters, and Guido Schmitz. Spresso: A secure, privacy-respecting single sign-on system for the web. In *CCS*, pages 1358–1369, 2015.

[35] Amos Fiat and Adi Shamir. How to prove yourself: Practical solutions to identification and signature problems. In *EUROCRYPT*, pages 186–194. Springer, 1986.

[36] FIDO Alliance. FIDO alliance specifications: Overview. `https://fidoalliance.org/specifications/`, Accessed 20 May 2022.

[37] Martin Franz, Andreas Holzer, Stefan Katzenbeisser, Christian Schallhart, and Helmut Veith. CBMC-GC: An ANSI C Compiler for Secure Two-Party Computations. In *Compiler Construction: 23rd International Conference*, volume 8409, page 244, 2014.

[38] Adam Gągol and Damian Straszak. Threshold ecdsa for decentralized asset custody. Technical report, Tech. rep., Cryptology ePrint Archive, Report 2020/498, 2020.

[39] Rosario Gennaro, Craig Gentry, Bryan Parno, and Mariana Raykova. Quadratic span programs and succinct nizks without pcps. In *EUROCRYPT*, pages 626–645. Springer, 2013.

[40] Rosario Gennaro and Steven Goldfeder. Fast multiparty threshold ECDSA with fast trustless setup. In *CCS*, pages 1179–1194, 2018.

[41] Rosario Gennaro, Steven Goldfeder, and Arvind Narayanan. Threshold-optimal DSA/ECDSA signatures and an application to bitcoin wallet security. In *ACNS*, pages 156–174. Springer, 2016.

[42] Irene Giacomelli, Jesper Madsen, and Claudio Orlandi. ZKBoo: Faster zero-knowledge for boolean circuits. In *USENIX Security*, pages 1069–1083, 2016.

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

[44] Trillian. `https://github.com/google/trillian`.

[45] Jens Groth. On the size of pairing-based non-interactive arguments. In *EUROCRYPT*, pages 305–326. Springer, 2016.

[46] Jens Groth and Markulf Kohlweiss. One-out-of-many proofs: Or how to leak a secret and spend a coin. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 253–280. Springer, 2015.

[47] Jens Groth and Victor Shoup. Design and analysis of a distributed ECDSA signing service. *Cryptology ePrint Archive*, 2022.

[48] Jens Groth and Victor Shoup. On the security of ECDSA with additive key derivation and presignatures. In *EUROCRYPT*, 2022.

[49] Paul Grubbs, Arasu Arun, Ye Zhang, Joseph Bonneau, and Michael Walfish. Zero-knowledge middleboxes. *Cryptology ePrint Archive*, 2021.

[50] Chengqian Guo, Jingqiang Lin, Quanwei Cai, Fengjun Li, Qiongxiao Wang, Jiwu Jing, Bin Zhao, and Wei Wang. UPPRESSO: Untraceable and unlinkable privacy-preserving single sign-on services. *arXiv preprint arXiv:2110.10396*, 2021.

[51] Sven Hammann, Ralf Sasse, and David Basin. Privacy-preserving OpenID connect. In *ASIACCS*, pages 277GS22–289, 2020.

[52] Yuncong Hu, Kian Hooshmand, Harika Kalidhindi, Seung Jin Yang, and Raluca Ada Popa. Merkle 2: A low-latency transparency log system. In *IEEE Security & Privacy*, pages 285–303. IEEE, 2021.

[53] Marios Isaakidis, Harry Halpin, and George Danezis. Unlimitid: Privacy-preserving federated identity management using algebraic macs. In *Proceedings of*

the 2016 ACM on Workshop on Privacy in the Electronic Society, pages 139–142, 2016.

[54] Yuval Ishai, Eyal Kushilevitz, Rafail Ostrovsky, and Amit Sahai. Zero-knowledge from secure multiparty computation. In *STOC*, pages 21–30, 2007.

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

[56] Krypton. kr-u2f. https://github.com/kryptco/kr-u2f, Accessed 17 May 2022.

[57] SCIPR Lab. libsnark. https://github.com/scipr-lab/libsnark, Accessed 30 May 2022.

[58] Leslie Lamport. The part-time parliament. In *TOCS*, pages 133–169, 1998.

[59] Adam Langley, Emilia Kasper, and Ben Laurie. Certificate transparency. *Internet Engineering Task Force*, 2013. https://tools.ietf.org/html/rfc6962.

[60] Jiwon Lee, Jaekyoung Choi, Jihye Kim, and Hyunok Oh. SAVER: Snark-friendly, additively-homomorphic, and verifiable encryption and decryption with rerandomization. *Cryptology ePrint Archive*, 2019.

[61] Yehuda Lindell. Fast secure two-party ECDSA signing. In *CRYPTO*, pages 613–644. Springer, 2017.

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

[63] Vadim Lyubashevsky and Gregory Neven. One-shot verifiable encryption from lattices. In *EUROCRYPT*, pages 293–323. Springer, 2017.

[64] Philip MacKenzie and Michael K Reiter. Two-party generation of DSA signatures. In *CRYPTO*, pages 137–154. Springer, 2001.

[65] David Mazières and Dennis Shasha. Building secure file systems out of Byzantine storage. In *21st Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, pages 108–117, July 2002.

[66] Marcela S Melara, Aaron Blankstein, Joseph Bonneau, Edward W Felten, and Michael J Freedman. CONIKS: Bringing key transparency to end users. In *USENIX Security*, 2015.

[67] Kathleen Moriarty, Burt Kaliski, Jakob Jonsson, and Andreas Rusch. PKCS# 1: RSA cryptography specifications version 2.2. *Internet Engineering Task Force, Request for Comments*, 8017:72, 2016.

[68] D. M'Raihi, S. Machani, M. Pei, and J. Rydell. TOTP: Time-Based One-Time Password Algorithm. RFC 6238, May 2011.

[69] Jonas Nick, Tim Ruffing, Yannick Seurin, and Pieter Wuille. MuSig-DN: Schnorr multi-signatures with verifiably deterministic nonces. In *CCS*, pages 1717–1731, 2020.

[70] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *USENIX ATC*, pages 305–319, 2014.

[71] Bryan Parno, Jon Howell, Craig Gentry, and Mariana Raykova. Pinocchio: Nearly practical verifiable computation. In *IEEE Security & Privacy*, pages 238–252. IEEE, 2013.

[72] Emma Roth. LastPass' latest data breach exposed some customer information. *The Verge*, November 2022. https://www.theverge.com/2022/11/30/23486902/lastpass-hackers-customer-information-breach.

[73] Adam Rowe. Study reveals average person has 100 passwords, November 2021. https://tech.co/password-managers/how-many-passwords-average-person.

[74] N. Sakimura, J. Bradley, M. Jones, B. de Medeiros, and C. Mortimore. OpenID connect core 1.0 incorporating errata set 1. https://openid.net/specs/openid-connect-core-1_0.html, November 2014.

[75] Adi Shamir. How to share a secret. *Communications of the ACM*, 22(11):612–613, 1979.

[76] Nigel P Smart and Younes Talibi Alaoui. Distributing any elliptic curve based protocol. In *IMA International Conference on Cryptography and Coding*, pages 342–366. Springer, 2019.

[77] Markus Stadler. Publicly verifiable secret sharing. In *EUROCRYPT*, pages 190–199. Springer, 1996.

[78] Akira Takahashi and Greg Zaverucha. Verifiable encryption from MPC-in-the-Head. *Cryptology ePrint Archive*, 2021.

[79] Verizon. *DBIR Data Breach Investigations Report*, 2022 edition. https://www.verizon.com/business/resources/T3cd/reports/dbir/2022-data-breach-investigations-report-dbir.pdf.

[80] W3C. Web authentication: An api for accessing public key credentials level 2, April 2021. https://www.w3.org/TR/webauthn-2/, Accessed 20 May 2022.

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

[82] Liang Wang, Gilad Asharov, Rafael Pass, Thomas Ristenpart, and Abhi Shelat. Blind certificate authorities. In *IEEE Security & Privacy*, pages 1015–1032. IEEE, 2019.

[83] Xiao Wang, Alex J. Malozemoff, and Jonathan Katz. EMP-toolkit: Efficient MultiParty computation toolkit. `https://github.com/emp-toolkit`, 2016.

[84] Xiao Wang, Samuel Ranellucci, and Jonathan Katz. Authenticated garbling and efficient maliciously secure two-party computation. In *Proceedings of the 2017 ACM SIGSAC conference on computer and communications security*, pages 21–37, 2017.

[85] Haiyang Xue, Man Ho Au, Xiang Xie, Tsz Hon Yuen, and Handong Cui. Efficient online-friendly two-party ECDSA signature. In *CCS*, pages 558–573, 2021.

[86] Shota Yamada, Nuttapong Attrapadung, Bagus Santoso, Jacob CN Schuldt, Goichiro Hanaoka, and Noboru Kunihiro. Verifiable predicate encryption and applications to CCA security and anonymous predicate authentication. In *PKC*, pages 243–261. Springer, 2012.

[87] Andrew C Yao. Protocols for secure computations. In *23rd annual symposium on foundations of computer science (sfcs 1982)*, pages 160–164. IEEE, 1982.

[88] Andrew Chi-Chih Yao. How to generate and exchange secrets. In *FOCS*, pages 162–167. IEEE, 1986.

[89] Fan Zhang, Deepak Maram, Harjasleen Malvai, Steven Goldfeder, and Ari Juels. DECO: Liberating web data using decentralized oracles for TLS. In *CCS*, pages 1919–1938, 2020.

[90] Zhiyi Zhang, Michal Król, Alberto Sonnino, Lixia Zhang, and Etienne Rivière. EL PASSO: Efficient and lightweight privacy-preserving single sign on. *PoPETS*, 2021(2):70–87, 2021.

[91] ZoKrates. ZoKrates. `https://github.com/Zokrates/ZoKrates`, Accessed 30 May 2022.

A The ECDSA signature scheme

We include a short description of ECDSA following Fersch et al. [32]. The ECDSA signature scheme over message space $\mathcal{M}$ uses a fixed group $\mathbb{G} = \langle g \rangle$ of prime order q. The scheme also uses a hash function $\mathsf{Hash} : \mathcal{M} \to \mathbb{Z}_q$ and a conversion function $f : \mathbb{G} \to \mathbb{Z}_q$. Secret keys are of the form $x \in \mathbb{Z}_q$ and public keys of the form $X = g^x \in \mathbb{G}$. The algorithms of the signature scheme are:

- $\mathsf{ECDSA.Gen}() \to (\mathsf{sk}, \mathsf{pk})$:
  - Sample $x \xleftarrow{R} \mathbb{Z}_q$.
  - Output (x, g^x).
- $\mathsf{ECDSA.Sign}(\mathsf{sk} = x, m) \to \sigma$:
  - $r \xleftarrow{R} \mathbb{Z}_q^*$
  - $R \leftarrow g^r$
  - $s \leftarrow r^{-1} \cdot (\mathsf{Hash}(m) + f(R) \cdot x) \in \mathbb{Z}_q$
  - Output $\sigma = (f(R), s)$
- $\mathsf{ECDSA.Verify}(\mathsf{pk} = X, m, \sigma) \to \{0, 1\}$
  - Parse σ as (c, s).
  - If $c = 0$ or $s = 0$ output 0.
  - Compute $R' \leftarrow (g^{\mathsf{Hash}(m)} X^c)^{1/s} \in \mathbb{G}$
  - Output 1 iff $R' \neq 1 \in \mathbb{G}$ and $f(R') = c$.

In practice, Hash is a cryptographic hash function (e.g. SHA-256) and f is the function that interprets a group element $g \in \mathbb{G}$ as an elliptic-curve point and outputs the value of the x-coordinate of the point in $\mathbb{Z}_q$.

B ECDSA with additive key derivation and presignatures

We now define security for a variant of ECDSA where (1) the adversary can choose "tweaks" to the signing key, and (2) the adversary can request presignatures that are later used to generate presignatures. In the rest of this section, we will show that the advantage of the adversary in this modified version of ECDSA is negligible. In Appendix C, we will argue that a two-party protocol that achieves the ideal functionality of this modified version of ECDSA is secure.

Throughout all algorithms implicitly run in time polynomial in the security parameter.

We define a security experiment for ECDSA with preprocessing and additive key derivation in Experiment 1 in Figure 7. The security experiment models the fact that the client's and log's secret key shares are not authenticated, and so the adversary can query for signatures under a signing key with adversarially chosen "tweaks". However, the final signature must verify under a fixed set of tweaks (in our setting, these correspond to public keys that the client generated at registration). The presignature queries allow us to capture the client preprocessing that we take advantage of.

Theorem 1. *Let* $\mathsf{ECDSAAdv}[\mathcal{A}, \mathbb{G}, \ell, N]$ *denote the adversary* $\mathcal{A}$*'s advantage in Experiment 1 with group* $\mathbb{G}$ *of prime order* q*,* ℓ Gen *queries, and* N *total* PreSign *and* Sign *queries. Then*

$$\mathsf{ECDSAAdv}[\mathcal{A}, \mathbb{G}, \ell, N] \leq O(N \cdot \ell / q) \ .$$

Proof. In order to prove the above theorem, we use an additional experiment, Experiment 2 in Figure 8. Let $\mathsf{GSECDSAAdv}[\mathcal{B}, \mathbb{G}, N, \mathcal{E}]$ denote the advantage of adversary $\mathcal{B}$ in Experiment 2 with a set of tweaks $\mathcal{E}$ of size ℓ. By Lemma 2,

$$\mathsf{GSECDSAAdv}[\mathcal{A}, \mathbb{G}, N, \mathcal{E}] \leq O(N \cdot |\mathcal{E}| / q) \ ,$$

and by Lemma 3,

$$\mathsf{ECDSAAdv}[\mathcal{A},\mathbb{G},\ell,N] \leq \mathsf{GSECDSAAdv}[\mathcal{B},\mathbb{G},N,\mathcal{E}] \ ,$$

and so

$$\mathsf{ECDSAAdv}[\mathcal{A},\mathbb{G},\ell,N] \leq O(N\cdot|\mathcal{E}|/q) \ .$$

□

For the intermediate security experiment, we use the security game from Groth and Shoup, which we include for reference as Experiment 2 in Figure 8. The intermediate security experiment allows us to leverage Groth and Shoup's security analysis for ECDSA with additive key derivation and presignatures [48]. They define a security game that is similar to but slightly different from Experiment 1 that we define to match our setting. Experiment 2 models the variant of additive key derivation where the signing tweak is not constrained to lie in the set of tweaks that the adversary must produce a forgery for (only the forging tweak is constrained; see Note 1 in Section 6 of Groth and Shoup [48]).

At a high level, the differences between Experiment 1 and Experiment 2 are:

- In Experiment 1, the adversary makes Gen queries to receive public keys corresponding to the set of tweaks, whereas in Experiment 2, the adversary simply receives the set of tweaks directly at the beginning of the experiment (these are an experiment parameter).
- Signing queries in Experiment 1 take as input a share of the tweak rather than the entire signing tweak.
- The Experiment 1 challenger enforces an order on Gen, PreSign, and Sign queries. The Experiment 2 challenger does not enforce an order.

Lemma 2. *Let* $\mathsf{GSECDSAAdv}[\mathcal{A},\mathbb{G},N,\mathcal{E}]$ *denote adversary* $\mathcal{A}$*'s advantage in Experiment 2 with group* $\mathbb{G}$ *of prime order* q*, number of presignature and signing queries* $N\in\mathbb{N}$*, and a set of tweaks* $\mathcal{E}$*. Then if* Hash *is collision resistant and* $\mathbb{G}$ *is a random oracle,*

$$\mathsf{GSECDSAAdv}[\mathcal{A},\mathbb{G},N,\mathcal{E}] \leq O(N\cdot|\mathcal{E}|/q) \ .$$

We refer the reader to Groth and Shoup's Theorem 4 for the analysis proving Lemma 2 in the generic group model. In our setting, the number of public keys and therefore size of $\mathcal{E}$ is polynomial.

All that remains is to prove that the the adversary in the modified Experiment 1 does not have a greater advantage than the adversary in the original Experiment 2.

Lemma 3. *Let* $\mathsf{ECDSAAdv}[\mathcal{A},\mathbb{G},\ell,N]$ *denote the adversary* $\mathcal{A}$*'s advantage in Experiment 1 with group* $\mathbb{G}$ *and* $\ell,N\in\mathbb{N}$*. Let* $\mathsf{GSECDSAAdv}[\mathcal{B},\mathbb{G},N,\mathcal{E}]$ *denote the adversary* $\mathcal{B}$*'s advantage in Experiment 2 with a set of tweaks* $\mathcal{E}$ *of size* ℓ*. Then given an adversary* $\mathcal{A}$ *in Experiment 1, we construct an adversary* $\mathcal{B}$ *for Experiment 2 that runs in time linear in* $\mathcal{A}$ *such that for all groups* $\mathbb{G}$*,* $N\in\mathbb{N}$*, and randomly sampled set of tweaks* $\mathcal{E}$ *of size* ℓ*,*

$$\mathsf{ECDSAAdv}[\mathcal{A},\mathbb{G},\ell,N] \leq \mathsf{GSECDSAAdv}[\mathcal{B},\mathbb{G},N,\mathcal{E}] \ .$$

Proof. We prove the above theorem by using an adversary $\mathcal{A}$ in Experiment 1 to construct an adversary $\mathcal{B}$ in Experiment 2. We then show that the adversary $\mathcal{B}$ has an advantage greater than or equal to the adversary $\mathcal{A}$.

We construct $\mathcal{B}$ in the following way:

Experiment 1: ECDSA with presignatures and additive key derivation. The experiment is parameterized by a number of Gen queries $\ell\in\mathbb{N}$, a number of PreSign and Sign queries $N\in\mathbb{N}$, a group $\mathbb{G}$ of prime order q with generator g, message space $\mathcal{M}$, a hash function $\mathsf{Hash}:\mathcal{M}\to\mathbb{Z}_q$, a conversion function $f:\mathbb{G}\to\mathbb{Z}_q$.

- The challenger initializes state $\mathsf{initdone}=0$, $\mathsf{presigdone}=0$.
- The adversary can make ℓ Gen queries and N PreSign and Sign queries.
- $\mathsf{Gen}()\to\mathsf{pk}$:
  - If $\mathsf{initdone}=0$, $k=1$; otherwise $k\leftarrow k+1$.
  - Sample $\mathsf{sk}_k \xleftarrow{R} \mathbb{Z}_q$.
  - Set $\mathsf{ctr}_{\mathsf{presig}},\mathsf{ctr}_{\mathsf{auth}}\leftarrow 0$.
  - Set $\mathsf{initdone}\leftarrow 1$, $\mathsf{presigdone}\leftarrow 0$.
  - Output g^{sk_k}.
- $\mathsf{PreSign}()\to R$:
  - If $\mathsf{initdone}=0$ or $\mathsf{presigdone}=1$, output $\perp$.
  - Sample $r_{\mathsf{ctr}_{\mathsf{presig}}} \xleftarrow{R} \mathbb{Z}_q^*$.
  - Output $g^{r_{\mathsf{ctr}_{\mathsf{presig}}}}$ and set $\mathsf{ctr}_{\mathsf{presig}}\leftarrow\mathsf{ctr}_{\mathsf{presig}}+1$.
- $\mathsf{Sign}(m,\omega,j)\to\sigma$:
  - If $\mathsf{initdone}=0$, $\mathsf{ctr}_{\mathsf{presig}}<\mathsf{ctr}_{\mathsf{auth}}$, or $j>k$ or $j<1$, output $\perp$.
  - Let $R\leftarrow g^{r_{\mathsf{ctr}_{\mathsf{auth}}}}$
  - Let $s\leftarrow r^{-1}_{\mathsf{ctr}_{\mathsf{auth}}}\cdot(\mathsf{Hash}(m)+(\mathsf{sk}_j+\omega)\cdot f(R))$.
  - Set $\mathsf{presigdone}\leftarrow 1$, $\mathsf{ctr}_{\mathsf{auth}}\leftarrow\mathsf{ctr}_{\mathsf{auth}}+1$.
  - Output $(s,f(R))$.

The output of the experiment is "1" if:

- the signature σ^* on m^* verifies under pk,
- m^* was not an input to Sign, and
- pk was an output of Gen.

The output is"0" otherwise.

Figure 7: Our experiment for security of ECDSA with additive key derivation and presignatures.

Experiment 2: Groth-Shoup ECDSA with presignatures and additive key derivation [48].
We recall the security experiment from Groth and Shoup [48] in Figure 4 for ECDSA with presignatures with the modifications described for additive key derivation.
The experiment is parameterized by the number of total queries the adversary can make $N \in \mathbb{N}$, a group $\mathbb{G}$ of prime order q with generator g, message space $\mathcal{M}$, a hash function $\mathsf{Hash} : \mathcal{M} \to \mathbb{Z}_q$, a conversion function $f : \mathbb{G} \to \mathbb{Z}_q$, and a set of tweaks $\mathcal{E} \subseteq \mathbb{Z}_q$.

- The challenger initializes state:
  - $k \leftarrow 0$, $K \leftarrow \emptyset$
  - $d \xleftarrow{R} \mathbb{Z}_q$, $D \leftarrow g^d \in \mathbb{G}$.
- The adversary can make N total queries (presign or sign).
- Presignature query:
  - $k \leftarrow k+1$, $r_k \xleftarrow{R} \mathbb{Z}_q^*$
  - $R_k \leftarrow g^{r_k} \in \mathbb{G}$
  - $t_k \leftarrow f(R_k) \in \mathbb{Z}_q$. If $t_k = 0$, output $\perp$
  - Return R_k
- Signing request for message m with presignature $k \in K$ and tweak $\omega \in \mathbb{Z}_q$:
  - $K \leftarrow K \backslash \{k\}$
  - $h_k \leftarrow \mathsf{Hash}(m) \in \mathbb{Z}_q$
  - If $h_k + t_k d + t_k \omega = 0$, output $\perp$
  - $s_k \leftarrow r_k^{-1}(h_k + t_k d + t_k \omega) \in \mathbb{Z}_q$
  - Return (s_k, t_k, R_k)
- After making N queries, the adversary must output $(m^*, \sigma^*, \omega^*)$.

The output of the experiment is "1" if:

- the signature σ^* on m^* verifies under $D \cdot g^{\omega^*}$,
- m^* was not an input to a previous signing query, and
- $\omega^* \in \mathcal{E}$.

The output is "0" otherwise.

Figure 8: Security experiment for ECDSA with additive key derivation and presignatures from Groth and Shoup [48].

- Rather than sending $\mathcal{A}$ the set of tweaks $\mathcal{E}$ immediately, $\mathcal{B}$ keeps the set of tweaks to use to respond to Gen queries. On the ith invocation of Gen, $\mathcal{B}$ sends $D \cdot g^{\omega_i}$ where $\omega_i \in \mathcal{E}$.
- $\mathcal{B}$ simply forwards presignature requests from $\mathcal{A}$ to the Experiment 2 challenger and sends the responses back to $\mathcal{A}$.
- $\mathcal{B}$ takes signing requests from $\mathcal{A}$ with an index j and a tweak ω. $\mathcal{B}$ then computes $\omega_j + \omega = \omega'$ where ω_j was the value returned by the jth call to Gen (if $\mathcal{A}$ has not made j calls to Gen, $\mathcal{B}$ outputs $\perp$). $\mathcal{B}$ then forwards the signing request with ω' to the Experiment 2 challenger.
- $\mathcal{B}$ additionally enforces that all presignature queries must be made before signing queries and that presignatures must be used in order. If $\mathcal{A}$ sends queries that do not meet these requirements, $\mathcal{B}$ outputs $\perp$.

The adversary $\mathcal{A}$ cannot distinguish between interactions with $\mathcal{B}$ and the Experiment 1 challenger, and so the advantage of $\mathcal{A}$ is less than or equal to that of $\mathcal{B}$, completing the proof. □

Zero-knowledge proof of preimage In larch, the log takes as input a hash of the message rather than the message itself. It is important for security that the log has a zero-knowledge proof of the preimage of the signing digest, as ECDSA with presignatures is completely insecure if the signing oracle signs arbitrary digests directly instead of messages [48]. Because the log checks a zero-knowledge proof certifying that the digest preimage is correctly encrypted before signing, it will not sign arbitrary purported hashes generated by a malicious client (the party submitting the hash must know the preimage for the proof to verify).

C Two-party ECDSA with preprocessing

In this section, we describe the construction of our two-party ECDSA with preprocessing protocol and argue its security. In Appendix C.1, we describe the syntax and construction of our protocol. In Appendix C.2, we explain a sub-protocol and argue that it is secure. Finally in Appendix C.3, we prove that our overall protocol is secure by showing that the protocol achieves the ideal functionality captured by the challenger in Experiment 1 from Appendix B.

C.1 Syntax and construction

For our purposes, a two-party ECDSA signature scheme consists of the following algorithms:

- $\mathsf{LogKeyGen}() \to (\mathsf{sk}_0, \mathsf{pk}_0)$: Generate a log secret key $\mathsf{sk}_0 \in \mathbb{Z}_q$ and corresponding public key $\mathsf{pk}_0 \in \mathbb{G}$. The log runs this routine at enrollment.
- $\mathsf{PreSign}() \to (\mathsf{presig}_0, \mathsf{presig}_1)$: Generate presignature $(\mathsf{presig}_0, \mathsf{presig}_1)$ where each presignature should be used to sign exactly once. The client runs this routine many times at enrollment to generate a batch of presignatures.
- $\mathsf{ClientKeyGen}(\mathsf{pk}_0) \to (\mathsf{sk}_1, \mathsf{pk})$: Given the log public key $\mathsf{pk}_0 \in \mathbb{G}$, output a secret key share $\mathsf{sk}_1 \in \mathbb{Z}_q$ and corresponding public key $\mathsf{pk} \in \mathbb{G}$. The client runs this routine during registration with each relying party.

We additionally define the following signing protocol:

- Π_{Sign}: Both parties take as input the message $m \in \mathcal{M}$, the log takes as input log secret key sk_0 and presignature presig_0, and the client takes as input a secret-key share sk_1 and presignature presig_1. The joint output is a signature σ on message m or $\perp$ (if either misbehaved).

The signing protocol outputs ECDSA signatures that verify under pk output by $\mathsf{ClientKeyGen}$, and so the signature-verification algorithm is exactly as in ECDSA.

The ECDSA signature scheme for message space $\mathcal{M}$ uses a group $\mathbb{G}$ of prime order q and is parameterized by a hash function $\mathsf{Hash} : \mathcal{M} \to \mathbb{Z}_q$ and a conversion function $f : \mathbb{G} \to \mathbb{Z}_q$. An ECDSA keypair is a pair $(y, g^y) \in \mathbb{Z}_q \times \mathbb{G}$ for $y \xleftarrow{R} \mathbb{Z}_q$. We use a secure multiplication protocol Π_{HalfMul} (Figure 10 in Appendix C.2) and a secure opening protocol Π_{Open} that returns the result or $\perp$ ("output" step in Figure 1 of SPDZ [24]).

$\underline{\mathsf{LogKeyGen}() \to (\mathsf{sk}_0, \mathsf{pk}_0)}$:

- Sample $x \xleftarrow{R} \mathbb{Z}_q$ and output (x, g^x).

$\underline{\mathsf{PreSign}() \to (\mathsf{presig}_0, \mathsf{presig}_1)}$:

- Sample $r \xleftarrow{R} \mathbb{Z}_q^*$ and compute $R \leftarrow f(g^r)$.
- Sample $\alpha \xleftarrow{R} \mathbb{Z}_q$ and compute $\hat{r} \leftarrow \alpha \cdot r^{-1}$.
- Split r^{-1} into secret shares r_0, r_1; $\hat{r}$ into $\hat{r_0}, \hat{r_1}$; α into α_0, α_1;
- Output $(R, r_0, \hat{r_0}, \alpha_0)$, $(R, r_1, \hat{r_1}, \alpha_1)$.

$\underline{\mathsf{ClientKeyGen}(\mathsf{pk}_0) \to (\mathsf{sk}_1, \mathsf{pk})}$:

- Sample $y \xleftarrow{R} \mathbb{Z}_q$.
- Output $(y, \mathsf{pk}_0 \cdot g^y)$.

$\underline{\Pi_{\mathsf{Sign}}}$:

We refer to the log as party 0 and the client as party 1. The input of party $i \in \{0, 1\}$ is $(m, \mathsf{sk}_i, \mathsf{presig}_i)$, and the output is a signature σ on m or $\perp$.

For each party $i \in \{0, 1\}$:

- Party i parses presig_i as $(R, r_i, \hat{r_i}, \alpha_i)$.
- Given input $(r_i, \hat{r_i}, \mathsf{sk}_i, \alpha_i)$ for party i, run Π_{HalfMul} to compute shares $(x_i, \hat{x_i}, v_i, \hat{v_i})$ where $\hat{v_i}$ authenticates any intermediate values.
- Party i computes $s_i \leftarrow r_i \cdot \mathsf{Hash}(m) + x_i \cdot R$.
- Party i computes $\hat{s_i} \leftarrow \hat{r_i} \cdot \mathsf{Hash}(m) + \hat{x_i} \cdot R$.
- Parties run Π_{Open} with party i input $\alpha_i, s_i, \hat{s_i}, v_i, \hat{v_i}$ to get s or $\perp$ (if returns $\perp$, output $\perp$).
- Output (R, s).

Figure 9: Two-party ECDSA signing protocol with preprocessing.

We include the constructions for the above algorithms and signing protocols in Figure 9. The construction for the Π_{HalfMul} is in Appendix C.2 and the opening protocol Π_{Open} is the same opening protocol used in SPDZ [24].

C.2 Malicious security with half-authenticated secure multiplication

As part of our signing protocol, we use a half-authenticated secure multiplication sub-protocol. We describe the protocol in Figure 10.

We need to ensure that by deviating from the protocol, neither the client nor the log can learn secret information (i.e. the other party's share of the secret key or signing nonce) or produce a signature for a different message. To use tools for malicious security (e.g. information-theoretic MACs [24]) in a black-box way, we need authenticated shares of the signing nonce and the secret key. We can easily generate authenticated shares of the signing nonce as part of the presignature, but generating authenticated shares of the secret key poses several problems: (1) presignatures are generated at enrollment (before the client has secret-key shares), and (2) we don't want which presignature the client uses to leak which relying party the client is authenticating to.

Ideal functionality. At a very high level, the ideal functionality takes as input additive shares of x, y and outputs shares of $x \cdot y$. In order to perform the multiplication, we use Beaver triples. To authenticate x, we use information-theoretic MAC tags (because there is no MAC tag for y, each party can adjust its share by an arbitrary additive shift without detection). More precisely then, the ideal functionality takes as inputs additive shares of (a, b, c), (f, g, h), $(x, \hat{x}, y)$, and α such that $a \cdot b = c$, $f \cdot g = h$, $(f, g, h) = \alpha \cdot (a, b, c)$, and $\hat{x} = \alpha \cdot x$. Each party outputs intermediate value d and additive shares of $\hat{d}, z, \hat{z}$ where $x \cdot y = z$ and $\hat{d} = \alpha \cdot d$.

Protocol. Our protocol uses information-theoretic MACs for only one of the inputs (the signing nonce). We call this protocol Π_{HalfMul}. Our construction uses authenticated Beaver triples and follows naturally from the SPDZ protocol [24]. We also use a secure opening protocol Π_{Open} for checking MAC tags, which we can instantiate using the SPDZ protocol directly [24]. It is safe to not authenticate one of the key shares due to the fact that the signature scheme is secure if the adversary can request signatures for arbitrary "tweaks" of the secret key (Appendix B).

We present a slightly modified version of the SPDZ protocol [24] for multiplication on authenticated secret-shared inputs in Figure 10. The only difference is that, in our protocol, only one of the inputs is authenticated. This requirement means that we only authenticate one of the intermediate values in the Beaver triple multiplication. We allow the attacker to add arbitrary shifts to the unauthenticated input, but the attacker cannot shift the authenticated input without detection (Claim 4).

The protocol Π_{HalfMul} allows us to model authenticating the signing nonce in ECDSA signing. The signing key is unrestricted (we discuss why this is secure in Appendix B).

We first show the security of our Π_{HalfMul} protocol for secure multiplication where only one of the inputs is authenticated, which is very similar to the multiplication protocol in SPDZ [24]. This allows us to ensure that both parties use the correct signing nonce from the presignature.

Claim 4. *Let $x, y, \alpha, \hat{x} \in \mathbb{Z}_q$ be inputs to Π_{HalfMul} secret-shared across the parties where $\hat{x} = \alpha x$. Then, the probability that an adversary that has statically corrupted one of the parties can cause the protocol Π_{HalfMul} to output shares of $z, \hat{z}, \hat{d}$ where $\hat{z} = \alpha \cdot z$ and $z = (x + \Delta)y$ for some $\Delta \neq 0$ is $1/q$.*

Proof. Let the input Beaver triple be $(a, b, c) \in \mathbb{Z}_q$ such that $a \cdot b = c$ where to multiply values x, y, we use intermediate values $d = x - a$ and $e = y - b$, where $\hat{d}$ is the MAC tag for d.

Π_{HalfMul}:
The protocol is parameterized by a prime q.

Inputs: Party $i \in \{0,1\}$ takes as input an additive share of each of the $\mathbb{Z}_q$ values: (a,b,c), (f,g,h), $(x,\hat{x},y)$, and α, such that:

$$\begin{aligned} a \cdot b &= c && \in \mathbb{Z}_q \\ f \cdot g &= h && \in \mathbb{Z}_q \\ (f,g,h) &= \alpha \cdot (a,b,c) && \in \mathbb{Z}_q^3 \\ \hat{x} &= \alpha \cdot x && \in \mathbb{Z}_q \end{aligned}$$

Outputs: Each party outputs intermediate value $d \in \mathbb{Z}_q$ and additive shares of $\hat{d}$, z, and $\hat{z}$, where:

$$\begin{aligned} x \cdot y &= z && \in \mathbb{Z}_q \\ \hat{d} &= \alpha \cdot d && \in \mathbb{Z}_q. \end{aligned}$$

Protocol. Each party $i \in \{0,1\}$ computes:

$$\begin{aligned} d_i &\leftarrow x_i - a_i && \in \mathbb{Z}_q \\ e_i &\leftarrow y_i - b_i && \in \mathbb{Z}_q \\ \hat{d}_i &\leftarrow \hat{x}_i - f_i && \in \mathbb{Z}_q \end{aligned}$$

and sends d_i, e_i to the other party. Each party i then computes:

$$\begin{aligned} d &\leftarrow d_0 + d_1 && \in \mathbb{Z}_q \\ e &\leftarrow e_0 + e_1 && \in \mathbb{Z}_q \\ z_i &\leftarrow de + db_i + ea_i + c_i && \in \mathbb{Z}_q \\ \hat{z}_i &\leftarrow de \cdot \alpha_i + dg_i + ef_i + h_i && \in \mathbb{Z}_q \end{aligned}$$

and outputs $d \in \mathbb{Z}_q$ and shares $\hat{d}_i, z_i, \hat{z}_i \in \mathbb{Z}_q$.

Figure 10: Π_{HalfMul} protocol.

To avoid detection, the adversary needs to ensure that $\hat{d} = \alpha \cdot d$, so the adversary needs to find some $\Delta_1, \Delta_2 \in \mathbb{Z}_q$ such that

$$\alpha(x + \Delta_1 - a) = \hat{x} + \Delta_2 - \alpha \cdot a \quad \in \mathbb{Z}_q$$

which we can reduce to

$$\alpha(x + \Delta_1) = \hat{x} + \Delta_2 \quad \in \mathbb{Z}_q$$

The probability of the adversary choosing $\Delta_1, \Delta_2 \in \mathbb{Z}_q$ that satisfies this equation is the probability of guessing α, or $1/q$. The value e does not depend on x. Therefore, since the remainder of the operations are additions and multiplications by public values, the attacker can only shift the final output by Δ_3, Δ_4 and needs to ensure the following:

$$\alpha(xy + \Delta_3) = \alpha xy + \Delta_4 \quad \in \mathbb{Z}_q$$

The probability of finding such Δ_3, Δ_4 is the probability of guessing α, which is $1/q$. □

C.3 Security proof for our construction

Recall our construction of our two-party signing scheme in Figure 9.

As the construction in Figure 9 contains separate algorithms for key generation and generating presignatures, we define Π_{Gen} and Π_{PreSign} in terms of the algorithms in Figure 9 below:

- Π_{Gen}:
  - If pk_0 is not initialized, the log runs $(\mathsf{sk}_0, \mathsf{pk}_0) \leftarrow \mathsf{LogKeyGen}()$ and sends pk_0 to the client.
  - If k is not initialized, set to 1; otherwise $k \leftarrow k+1$.
  - The client runs $(\mathsf{sk}_{1,k}, \mathsf{pk}_k) \leftarrow \mathsf{ClientKeyGen}(\mathsf{pk}_0)$
- Π_{PreSign}:
  - Client runs $(\mathsf{presig}_0, \mathsf{presig}_1) \leftarrow \mathsf{PreSign}()$ and sends presig_0 to the log.

Additive key derivation models the fact that the secret key shares are unauthenticated private inputs to Π_{HalfMul}, and so the adversary can run the signing protocol with any secret key share as input. However, to produce a forgery, the adversary must generate a signature that verifies under a small, fixed set of public keys (corresponding to the public keys generated at registration before compromise).

We define the ideal functionality $\mathcal{F}_{\mathsf{ECDSA}}$ as simply the routines the challenger runs to respond to Gen,PreSign, and Sign queries in Experiment 1.

We prove security using the ideal functionality $\mathcal{F}_{\mathsf{Open}}$ for opening values and checking MAC tags from SPDZ [24]. At a high level, $\mathcal{F}_{\mathsf{Open}}$ takes as input the party's output and intermediate shares and MAC tags for output and intermediate shares and outputs the combined output or abort if the MAC tags are not correct. The corresponding simulator $\mathsf{Sim}_{\mathsf{Open}}$ takes as input one party's shares of the output and intermediate values and their corresponding MAC tags, as well as the combined output value.

Theorem 5. *The two-party ECDSA signing protocol* Π *securely realizes (with abort)* $\mathcal{F}_{\mathsf{ECDSA}}$ *in the* $\mathcal{F}_{\mathsf{Open}}$*-hybrid model in the presence of a single statically corrupted malicious party (if the client is the compromised party, it can only be compromised after presigning is complete). Specifically, let* $\mathsf{View}_{\Pi}^{\mathsf{Real}}$ *denote the adversary* $\mathcal{A}$*'s view in the real world. Then there exists a probabilistic polynomial-time algorithm* Sim *where* $\mathsf{View}_{\mathsf{Sim}}^{\mathsf{Ideal}}$ *denotes the view simulated by* Sim *given the outputs of* $\mathcal{F}_{\mathsf{ECDSA}}$ *where* $\mathcal{A}$ *can adaptively choose which procedures to run and the corresponding inputs such that*

$$\mathsf{View}_{\Pi}^{\mathsf{Real}} \approx \mathsf{View}_{\mathsf{Sim}}^{\mathsf{Ideal}}.$$

Proof. Our goal is to construct a simulator where the simulator takes as input the outputs of the ideal functionality $\mathcal{F}$ (as well at the public input message for signing). The adversary $\mathcal{A}$ should then not be able to distinguish between the real world (interaction with the protocol) and the ideal world

(interaction with the simulator where the simulator is given the compromised party's inputs and the outputs of $\mathcal{F}_{\mathsf{ECDSA}}$).

Let i be the index of the compromised party ($i = 0$ for compromised client, $i = 1$ for compromised log). The simulator always generates the presignatures and outputs the presignature share to the adversary, in order to model the fact that we only provide security if the client is malicious at signing time. We construct the simulator as follows:

- $\mathsf{Gen}(\mathsf{pk})$:
  - If $i = 0$:
    - If sk_0 is not initialized, sample $\mathsf{sk}_0 \xleftarrow{R} \mathbb{Z}_q$ and send $\mathsf{pk}_0 \leftarrow g^{\mathsf{sk}_0}$ to $\mathcal{A}$.
    - Otherwise, send nothing to $\mathcal{A}$.
  - Otherwise if $i = 1$:
    - If pk_0 is not initialized, receive pk_0 from $\mathcal{A}$.
    - Output pk.
    - Set $\mathsf{initdone} \leftarrow 1$, $\mathsf{presigdone} \leftarrow 0$.
  - Set $\mathsf{ctr}_{\mathsf{presig}}, \mathsf{ctr}_{\mathsf{auth}} \leftarrow 0$.
  - Set $\mathsf{initdone} \leftarrow 1$.
- $\mathsf{PreSign}(R)$:
  - If $\mathsf{initdone} = 0$ or $\mathsf{presigdone} = 1$, output $\perp$.
  - Sample $\alpha_0, \alpha_1 \xleftarrow{R} \mathbb{Z}_q$, $r_0, r_1 \xleftarrow{R} \mathbb{Z}_q^*$.
  - Set $\alpha^{(\mathsf{ctr}_{\mathsf{presig}})} \leftarrow \alpha_0 + \alpha_1, r^{(\mathsf{ctr}_{\mathsf{presig}})} \leftarrow r_0 + r_1$.
  - Sample $\hat{r_0}, \hat{r_1}$ such that $\alpha^{(\mathsf{ctr}_{\mathsf{presig}})} \cdot r^{(\mathsf{ctr}_{\mathsf{presig}})} = \hat{r_0} + \hat{r_1}$.
  - Sample shares of $(a, b, c), (f, g, h) \in \mathbb{Z}_q^3$ such that $a \cdot b = c$, $f \cdot g = h$, $(f, g, h) = \alpha^{(\mathsf{ctr}_{\mathsf{presig}})}(a, b, c)$, $\hat{x} = \alpha \cdot x$.
  - Let $T_j^{\mathsf{ctr}_{\mathsf{presig}}} = (a_j, b_j, c_j, f_j, g_j, h_j)$ for $j \in \{1, 2\}$.
  - Let $\mathsf{presig}_0^{(\mathsf{ctr}_{\mathsf{presig}})} = (R, r_0, \hat{r_0}, \alpha_0, T_0^{(\mathsf{ctr}_{\mathsf{presig}})})$ and $\mathsf{presig}_1^{(\mathsf{ctr}_{\mathsf{presig}})} = (R, r_1, \hat{r_1}, \alpha_1, T_1^{(\mathsf{ctr}_{\mathsf{presig}})})$.
  - Send $\mathsf{presig}_{1-i}^{(\mathsf{ctr}_{\mathsf{presig}})}$ to $\mathcal{A}$.
  - Set $\mathsf{ctr}_{\mathsf{presig}} \leftarrow \mathsf{ctr}_{\mathsf{presig}} + 1$.
- $\mathsf{Sign}(m, \sigma)$:
  - If $\mathsf{initdone} = 0$ or $\mathsf{ctr}_{\mathsf{presig}} < \mathsf{ctr}_{\mathsf{auth}}$, output $\perp$.
  - Parse σ as $(s, \_)$.
  - Parse $\mathsf{presig}_i^{(\mathsf{ctr}_{\mathsf{auth}})}$ as $(R, r_i, \hat{r_i}, \alpha_i, T_i)$.
  - Parse $T_i^{(\mathsf{ctr}_{\mathsf{auth}})}$ as $(a_i, b_i, c_i, f_i, g_i, h_i)$.
  - Let $x \leftarrow \frac{s - r^{(\mathsf{ctr}_{\mathsf{auth}})} \cdot \mathsf{Hash}(m)}{R}$
  - Let $\mathsf{sk}_i \leftarrow x / r_i$
  - Let $\hat{d_i} = \hat{r_i} - f_i$.
  - Send $d_i = r_i - a_i$ and $e_i = \mathsf{sk}_i - b_i$ to $\mathcal{A}$.
  - Receive d_{1-i} and e_{1-i} from $\mathcal{A}$ and compute $d = d_0 + d_1$ and $e = e_0 + e_1$.
  - Let $x_i \leftarrow de + db_i + ea_i + c_i$
  - Let $\hat{x_i} \leftarrow de\alpha_i + dg_i + ef_i + h_i$
  - Compute $s' \leftarrow r^{(\mathsf{ctr}_{\mathsf{presig}})} \cdot \mathsf{Hash}(m) + x \cdot R$
  - Compute $s_i \leftarrow r_i \cdot \mathsf{Hash}(m) + x_i \cdot R + s - s'$
  - Compute $\hat{s_i} \leftarrow \hat{r_i} \cdot \mathsf{Hash}(m) + \hat{x_i} \cdot R + \alpha^{(\mathsf{ctr}_{\mathsf{auth}})}(s - s')$
  - Run $\mathsf{Sim}_{\mathsf{Open}}$ on $(\alpha_i, s_i, \hat{s_i}, d, \hat{d_i}, s)$; if $\mathsf{Sim}_{\mathsf{Open}}$ aborts, also abort.
  - Set $\mathsf{presigdone} \leftarrow 1$ and $\mathsf{ctr}_{\mathsf{auth}} \leftarrow \mathsf{ctr}_{\mathsf{auth}} + 1$.

We now prove that the view generated by Sim in the ideal world is indistinguishable from the real world.

We start with the real world (Figure 9). We then replace calls to Π_{Open} with $\mathsf{Sim}_{\mathsf{Open}}$. Because we are in the $\mathcal{F}_{\mathsf{Open}}$-hybrid model, the adversary cannot distinguish between these.

Every other message sent to $\mathcal{A}$ is either (1) a value that is random or information theoretically indistinguishable from random, or (2) a value generated by $\mathcal{F}_{\mathsf{ECDSA}}$ (R in $\mathsf{PreSign}$).

The last step is to show that if $\mathcal{A}$ deviates from the protocol when sending d_{1-i} to the simulator, the simulator can detect this and abort. By Claim 4 and the guarantees of $\mathcal{F}_{\mathsf{Open}}$, the adversary cannot send an incorrect value for d_{1-i} without detection except with probability $1/q$. The adversary *can* send any value for e_{1-i}; this is equivalent to the adversary being allowed to choose any signing tweak ω, which the attacker can do in Experiment 1.

Therefore, $\mathcal{A}$ cannot distinguish between the real world and the ideal world except with probability $1/q$, completing the proof. □

D Larch for passwords

D.1 Zero-knowledge proofs for discrete log relations

The protocol of Section 5 requires the client to prove to the log service that the ElGamal decryption of a ciphertext decrypts to one of n values in a set. To do so, the client uses a zero-knowledge proof of discrete-log relations, whose syntax and construction we describe here. The proof system uses a cyclic group $\mathbb{G}$ of prime order q.

The proof system consists of two algorithms:

- $\mathsf{DLProof.Prove}(\mathsf{idx}, x, h, \mathsf{cm}_1, \ldots, \mathsf{cm}_n) \to \pi$:
  Output a proof π asserting that $\mathsf{cm}_{\mathsf{idx}} = h^x \in \mathbb{G}$ for $\mathsf{idx} \in [n]$
- $\mathsf{DLProof.Verify}(\pi, \mathsf{cm}_1, \ldots, \mathsf{cm}_n) \to \{0, 1\}$:
  Check the prover's claim that it knows some $x \in \mathbb{Z}_q$ where $\mathsf{cm}_{\mathsf{idx}} = h^x \in \mathbb{G}$ for $\mathsf{idx} \in [n]$.

We require the standard notions of *completeness*, *soundness* (against computationally bounded provers), and *zero knowledge* [13, 43] (in the random-oracle model [10]). We instantiate DLProof using proof techniques from Groth and Kohlweiss [46].

D.2 Protocol for passwords

We now describe the syntax of our $\mathsf{Larch}_{\mathsf{PW}}$ scheme.

Step #1: Enrollment with log service. At enrollment, the client and log generate cryptographic keys and exchange public keys.

- $\mathsf{Larch}_{\mathsf{PW}}.\mathsf{ClientGen}() \to (x, X)$: The client outputs a secret key $x \in \mathbb{Z}_q$ and a public key $X \in \mathbb{G}$.
- $\mathsf{Larch}_{\mathsf{PW}}.\mathsf{LogGen}() \to (k, K)$: The log service outputs a secret key $k \in \mathbb{Z}_q$ and a public key $K \in \mathbb{G}$.

Step #2: Registration with relying party. Once the client has enrolled with a log service, it can register with a relying party by interacting with the log service.

- $\mathsf{Larch}_{\mathsf{PW}}.\mathsf{ClientRegister}() \to (\mathsf{id}, k_{\mathsf{id}})$: The client outputs an identifier $\mathsf{id} \in \{0,1\}^\lambda$ and a key $k_{\mathsf{id}} \in \mathbb{G}$.
- $\mathsf{Larch}_{\mathsf{PW}}.\mathsf{LogRegister}(k, \mathsf{id}) \to y$: Given the log's secret key k and id produced by $\mathsf{ClientRegister}$, the log outputs $y \in \mathbb{G}$.
- $\mathsf{Larch}_{\mathsf{PW}}.\mathsf{FinishRegister}(k_{\mathsf{id}}, y) \to \mathsf{pw}_{\mathsf{id}}$: Given the key k_{id} generated by $\mathsf{ClientRegister}$ and the value y generated by $\mathsf{LogRegister}$, the client outputs the password $\mathsf{pw}_{\mathsf{id}} \in \mathbb{G}$.

Step #3: Authentication with relying party. After registration, the client and log service perform authentication together.

- $\mathsf{Larch}_{\mathsf{PW}}.\mathsf{ClientAuth}(\mathsf{idx}, x, \mathsf{id}_1, \ldots, \mathsf{id}_n) \to (r, \mathsf{ct}, \pi_1, \pi_2)$: Given an index $\mathsf{idx} \in \{1, \ldots, n\}$, the client's secret key $x \in \mathbb{Z}_q$, and identifier values $\mathsf{id}_1, \ldots, \mathsf{id}_n$ output by $\mathsf{ClientRegister}$, the client outputs $r \in \mathbb{Z}_q^*$, an ElGamal ciphertext $\mathsf{ct} \in \mathbb{G}^2$, and proofs π_1 and π_2.
- $\mathsf{Larch}_{\mathsf{PW}}.\mathsf{LogAuth}(\mathsf{ct}, \pi_1, \pi_2, \mathsf{id}_1, \ldots, \mathsf{id}_n) \to y$: Given a ciphertext $\mathsf{ct} \in \mathbb{G}^2$, proofs π_1 and π_2, and identifiers $\mathsf{id}_1, \ldots, \mathsf{id}_n$ output by $\mathsf{ClientRegister}$, the log service outputs $y \in \mathbb{G}$.
- $\mathsf{Larch}_{\mathsf{PW}}.\mathsf{FinishAuth}(x, K, r, k_{\mathsf{id}}, y) \to \mathsf{pw}_{\mathsf{id}}$: Given the client's secret key $x \in \mathbb{Z}_q$, the log's public key $K \in \mathbb{G}$, the nonce $r \in \mathbb{Z}_q^*$ generated by $\mathsf{ClientAuth}$, the key $k_{\mathsf{id}} \in \mathbb{G}$ generated by $\mathsf{ClientRegister}$, and the value $y \in \mathbb{G}$ from $\mathsf{LogAuth}$, output the password $\mathsf{pw}_{\mathsf{id}} \in \mathbb{G}$.

Step #4: Auditing with log service. Given a ciphertext, the client runs ElGamal decryption to recover the corresponding $\mathsf{Hash}(\mathsf{id})$ value.

We give a detailed description of the larch password-based authentication protocol in Figure 11.

Larch password-based authentication scheme. The protocol is parameterized by: a cyclic group $\mathbb{G}$ of prime order q with generator $g \in \mathbb{G}$, a hash function $\mathsf{Hash}\colon \{0,1\}^* \to \mathbb{G}$, and a zero-knowledge discrete-log proof system DLProof with syntax as in Appendix D.1.

$\mathsf{Larch}_{\mathsf{PW}}.\mathsf{ClientGen}() \to (x, X)$

- Sample $x \xleftarrow{R} \mathbb{Z}_q$.
- Output (x, g^x).

$\mathsf{Larch}_{\mathsf{PW}}.\mathsf{LogGen}() \to (k, K)$:

- Sample $k \xleftarrow{R} \mathbb{Z}_q$.
- Output (k, g^k).

$\mathsf{Larch}_{\mathsf{PW}}.\mathsf{ClientRegister}() \to (\mathsf{id}, k_{\mathsf{id}})$:

- Sample $\mathsf{id} \xleftarrow{R} \{0,1\}^\lambda$.
- Sample $k_{\mathsf{id}} \xleftarrow{R} \mathbb{G}$.
- Output $(\mathsf{id}, k_{\mathsf{id}})$.

$\mathsf{Larch}_{\mathsf{PW}}.\mathsf{LogRegister}(k, \mathsf{id}) \to y$:

- Output $\mathsf{Hash}(\mathsf{id})^k$.

$\mathsf{Larch}_{\mathsf{PW}}.\mathsf{FinishRegister}(k_{\mathsf{id}}, y) \to \mathsf{pw}_{\mathsf{id}}$:

- Output $k_{\mathsf{id}} \cdot y$.

$\mathsf{Larch}_{\mathsf{PW}}.\mathsf{ClientAuth}(\mathsf{idx}, x, \mathsf{id}_1, \ldots, \mathsf{id}_n) \to (r, \mathsf{ct}, \pi_1, \pi_2)$:

- Sample $r \xleftarrow{R} \mathbb{Z}_q^*$
- Compute $c_1 = g^r, c_2 = \mathsf{Hash}(\mathsf{id}_{\mathsf{idx}}) \cdot g^{xr}$.
- Let $h_i = c_2 / \mathsf{Hash}(\mathsf{id}_i)$ for $i \in \{1, \ldots, n\}$.
- Let $\pi_1 \leftarrow \mathsf{DLProof.Prove}(\mathsf{idx}, r, X, h_1, \ldots, h_n)$.
- Let $\pi_2 \leftarrow \mathsf{DLProof.Prove}(\mathsf{idx}, x, c_1, h_1, \ldots, h_n)$.
- Let $\mathsf{ct} = (c_1, c_2)$.
- Output $(r, \mathsf{ct}, \pi_1, \pi_2)$.

$\mathsf{Larch}_{\mathsf{PW}}.\mathsf{LogAuth}(\mathsf{ct}, \pi_1, \pi_2, \mathsf{id}_1, \ldots, \mathsf{id}_n) \to y$:

- Parse ct as (c_1, c_2).
- Let $h_i = c_2 / \mathsf{Hash}(\mathsf{id}_i)$ for $i \in \{1, \ldots, n\}$.
- Let $b_1 \leftarrow \mathsf{DLProof.Verify}(\pi_1, X, h_1, \ldots, h_n)$
- Let $b_2 \leftarrow \mathsf{DLProof.Verify}(\pi_2, c_1, h_1, \ldots, h_n)$
- If $b_1 \neq 1$ or $b_2 \neq 1$, output $\perp$
- Output c_2^k

$\mathsf{Larch}_{\mathsf{PW}}.\mathsf{FinishAuth}(x, K, r, k_{\mathsf{id}}, y) \to \mathsf{pw}_{\mathsf{id}}$:

- Output $k_{\mathsf{id}} \cdot y \cdot K^{-xr}$.

Figure 11: The details of the larch protocol for password-based authentication.