Encrypted Access Logging for Online Accounts: Device Attributions without Device Tracking

Carolina Ortega Pérez*
Cornell University

Alaa Daffalla*
Cornell University

Thomas Ristenpart
Cornell Tech

Abstract

Despite improvements in authentication mechanisms, compromise of online accounts remains prevalent. Therefore, technologies to detect compromise retroactively are also necessary. Service providers try to help users diagnose the security status of their accounts via account security interfaces (ASIs) that display recent logins or other activity. Recent work showed how major services' ASIs are untrustworthy because they rely on easily manipulated client-provided values. The reason is a seemingly fundamental tension between accurately attributing accesses to particular devices and the need to prevent online services from tracking devices.

We propose client-side encrypted access logging (CSAL) as a new approach that navigates the tension between tracking privacy and ASI utility. The key idea is to add to account activity logs end-to-end (E2E) encrypted device identification information, leveraging OS support and FIDO2-style attestations. We detail a full proposal for a CSAL protocol that works alongside existing authentication mechanisms and provide a formal analysis of integrity, privacy, and unlinkability in the face of honest-but-curious adversaries. Interestingly, a key challenge is characterizing what is feasible in terms of logging in this setting. We discuss security against active adversaries, provide a proof-of-concept implementation, and overall show feasibility of how OS vendors and service providers can work together towards improved account security and user safety.

1 Introduction

Protecting online accounts from compromise is critical to user security. While increasingly sophisticated mechanisms for authentication offer proactive security for accounts, in some cases compromise cannot be avoided, and reactive solutions are also needed. For instance, this is the case for at-risk users [69], for whom attackers routinely compromise accounts; a common example being abusive intimate partners who have physical access to devices and/or who can compel disclosure of authentication credentials (e.g., [39,45,60]). Services cannot distinguish between such illicit and legitimate client accesses (with existing methods), so users need to be able to have insight into access to their accounts. Services have tried to meet this need by deploying what Daffalla et al. [39] refer to as account security interfaces (ASIs): web pages or application screens that give users information about accesses to their account, such as the set of devices with authenticated sessions.

Current ASIs for online accounts have significant limitations. They rely on untrustworthy information such as user agent (UA) strings (configurable by users) or IP addresses (changeable via VPN) to communicate to the account owner what devices have been accessing the account. As shown in [39], even for prominent services such as Google and Facebook, adversaries can hide accesses completely from ASIs or "spoof" ASI data to make accesses appear benign, thereby undermining the secure attribution of accesses to particular devices. This is not just bad implementations, but rather seemingly fundamental: web and native clients on most platforms cannot access static device identifiers such as serial or IMEI numbers [54] to prevent malicious services from exploiting them for user tracking (see, e.g., [35,43,47,52,55,56,61]).

The status quo is therefore that while the web architecture does make device tracking harder, we simultaneously are leaving users vulnerable due to missing, poor, or misleading ASI data. This suggests the following question: Is it possible to re-architect clients and services so that users can obtain trustworthy device attributions for accesses, but without enabling device tracking by web services?

We initiate answering this open question, in particular by exploring what we call client-side-encrypted access logging (CSAL). At a high level, the idea is to associate with actions logged by the service (e.g., logging in, changing a password) an end-to-end (E2E) encryption of device identifiers. These encrypted identifiers should be decryptable only by logged in devices and not by the service provider.

In more detail, our approach to CSAL is as follows. Authentication takes place between a client application (browser

*Authors contributed equally to this work.

with web app or native app) and a service, which we call the relying party (RP). For current ASIs, a plaintext, RP-visible client-to-service-to-client channel is used to communicate access information between authenticated clients. We add an E2E encrypted channel to communicate device identifiers between authenticated client devices. The channel's endpoints are not the clients, but rather a new type of OS service running on client devices that we call encryptors. The encryptor can display ASI logs to users (like other OS interfaces listing account information, e.g., [15]). We require OS support, but as discussed above, this seems fundamental given the privacy risks associated with revealing device identifiers to client apps. Architecturally, our setup takes inspiration from FIDO2 [19]. That said, our approach is agnostic to the authentication protocol used by the RP and now provides integrity against client devices offering spoofed data, which is missing from current ASIs, while keeping tracking information private.

When authenticating with an RP, clients can request the encryptor to generate a new CSAL session, which produces fresh public keys along with a FIDO2-style attestation that can be used to verify that the public key was generated by a trusted OS vendor. The RP can relay attested public keys between clients, who can in turn hand them to encryptors to verify that they are from an allowed OS before encrypting device identifiers using them. This approach provides strong security for honest-but-curious (HBC) RPs, but an actively malicious RP can try to add an endpoint — this will be detectable, disincentivizing RPs from doing so. Moreover, we build into our protocols the ability of encryptors to verify requests as originating with particular RPs, meaning that OS vendors could enroll only RPs that have procedural mechanisms in place to prevent such bad RP behavior.

A key technical question is how to define correctness for CSAL protocols. We could ask for full correctness: all device-attributed actions are relayed to all authenticated clients. However, the approach outlined above does not ensure this: a client that logs in, performs some actions, and logs out cannot perform the full back-and-forth key exchange required to transmit its device information to future sessions. To understand if this is fundamental, we formalize CSAL protocols with definitions for correctness (even with adversarial client inputs, providing a form of integrity), log privacy, and session unlinkability. The privacy and unlinkability notions capture an HBC RP.

We first use this formal model to show that privacy and full correctness are fundamentally at odds. An HBC RP can break privacy for any fully correct protocol because, intuitively, communicating private information from an already inactive session to a future session is impossible — the RP knows the same as the future recipient. We then relax the correctness goal via a graph-theoretic reachability condition capturing how log entries should be recoverable by a client if there was the possibility of any, even indirect, prior communication from the action-originating client. This correctness condition assumes that the encryptor on a device can communicate privately "for free" between different sessions on that device, including (for some period of time managed by the encryptor) logged-out sessions.

Achieving this new better correctness therefore requires taking advantage of the fact that sessions on a device can communicate. Consider a device that has an old, logged-out session and an active session. To achieve the new correctness, the active session must relay key material for the old session to active sessions on other devices. Doing so by only sending (encrypted) secret keys when necessary would, however, reveal to the RP that these two sessions were on the same device, violating unlinkability. We resolve this by "smuggling" secret keys known to a session to other sessions, using suitable amounts of dummy plaintext to prevent linkability attacks. Putting it all together, we prove that our CSAL protocol achieves the better correctness, log privacy, and unlinkability against HBC RPs. We leave formalizing fully malicious RP privacy and unlinkability to future work, but provide a detailed, structured exploration of active attacks and how they impact our CSAL protocol.

Finally, we provide a proof-of-concept implementation of our CSAL approach and show that while it adds overheads, they would not be prohibitive in practice. Our prototype is available as a public, open-source project[1].

Summary. In summary, we initiate an investigation into the challenge of improving ASIs, which are critically important for users facing attacks. In this work, we:

- propose CSAL protocols as a way to navigate the tension between logging with device attributions and the privacy threat of device tracking by RPs;
- formalize CSAL protocols and show that there are fundamental trade-offs between log privacy and log correctness;
- detail a full CSAL protocol and provide formal analyses to prove that it achieves correctness, log privacy, and unlinkability for honest-but-curious adversaries; and
- report on a proof-of-concept CSAL implementation.

Our implementation shows that CSAL protocols are efficient enough for practice. Yet, deployment will require cooperation between OS vendors and web service developers, which requires buy-in, time, and standardization efforts. Hence we see our main contribution as a feasibility result that new architectures that enable better ASIs for users can be built. We close with a discussion on limitations and deployment challenges.

2 Related Work

Now, we explore prior work on log and account security and provide background on E2E encrypted applications.

System and access logs. System logs record important system information such as system events, configurations, and

[1] zenodo.org/records/14737179

resource usage. Logs provide a valuable view of past and current states of almost any complex system. Among NIST-suggested system and audit monitoring techniques is the periodic review of system-generated logs, which can detect security problems [67]. Because of their forensic value, system logs represent an obvious attack vector. Thus, protecting these logs (e.g., [62, 67]) and their integrity [30, 31, 58] has always been one of the most critical tasks in a computer system.

Single-system logging schemes have been using symmetric cryptography [30, 58, 65], public key cryptography [51, 58], tamper-resistant hardware [37, 53], and identity-based encryption [63, 70]. Since all these focus on system logs for a single system, they are not suitable for web account access logging.

Account security. Some prior work focused on increasing authentication security. Works on two-factor authentication [5, 14], or, more recently, on passkeys [9, 10, 13] help prevent phishing and brute-force attacks. However, they do not address threat models where a malicious party might be an abusive partner who has access to the user's devices, or might even share devices with them. Addressing this requires more reactive measures to detect malicious activities post-factum.

To help users diagnose if a compromise has occurred, platforms have deployed solutions such as email notifications [59] and ASIs [39]. Detecting compromise of familiar devices is hard. We focus on enabling more granular and integral ASIs, but our solution works alongside email notifications.

Web account access logging. Current web services primarily log accesses on the server side, and while a long literature on authentication mechanisms seeks to ensure only users holding the correct credentials (e.g., passwords or 2FA) can log in, account compromise is still prevalent. Thus, services complement authentication protocols with mechanisms to inform users about access.

Hammann et al. [49] introduced a graph-based formalism to evaluate the security and recoverability of online accounts, and conducted a user study [48] of real-world user accounts leveraging their formalism. But their work doesn't address logging accesses in a secure way to enable building their account access graphs; which our CSAL approach could provide.

Daffalla et al. [39] showed that ASIs play an important role in communicating to users about account accesses. They highlight how this is particularly important for at-risk users, i.e., those who face an elevated risk of an attack on their digital safety (e.g., survivors of intimate partner violence, journalists, refugees, etc.) [32, 69]. At-risk users especially rely on the ability to determine if and from where a compromise has occurred. They also experimentally show that ASIs deployed by popular services are vulnerable to spoofing and hiding attacks, which undermine the integrity of these logs, limiting users' ability to identify malicious accesses and account activity.

A recent work on securely logging web access is Larch [41]. It introduces a private login logging system design that forces authentication to involve both a relying party (to be logged into) and a separate third-party service called the login archive service. They use split-secret authentication [66] where a secret is shared between the archive service and the client to enforce access logging. This approach cryptographically guarantees log entries for each login without modifications to the relying party. However, Larch does not address device attribution, relying on client-provided (and often erroneous) names for devices, nor does it allow for granular logging of post-login actions, which are likely to be of forensic importance, as victims of account compromise invariably want to know what was done with illicit access [39].

E2E encrypted applications. A growing body of research studies the design and analysis of E2E encrypted protocols (e.g., [27, 36, 38, 44, 46]) and their use in various applications (e.g., [6, 7, 12, 33]). At first glance, using an existing E2E encrypted group messaging protocol like MLS [27], with each device as a group member, might seem enough for encrypting device identifiers. However, this fails in our setting because group E2E messaging is designed for transient messages, with granular forward secrecy. Thus new group members cannot read old messages. We need a way of granularly giving them access without breaking unlinkability. We believe one could build CSAL starting from MLS, but (1) it would be more expensive than our direct approach; (2) our negative results from Section 5 still apply; and (3) work is needed to figure out how to ensure proper functionality and security.

Secure cloud storage and encrypted backups end-to-same-end architectures face some challenges similar to the ones in our system. One of the main challenges is making cryptographic secrets available across multiple devices while balancing usability and security. Some systems rely on human-chosen PINs, with backup attempts limited by hardware security modules [23, 40]; we avoid using PINs or other secrets that users must memorize or store separately.

Meta recently announced [24] an encrypted storage protocol called Labyrinth for Facebook Messenger. It relies on symmetric keys that rotate each epoch. New sessions learn the current epoch for encryption but require some recovery key material, which can be obtained using recovery codes, user-chosen PINs, etc. In CSAL, the recovery of previous entries is automatic, and doesn't require additional interaction from the user. This is especially important in our system because, while Facebook's sessions can be long-lived, we consider other systems where sessions last only a day, as is the case for many enterprise accounts.

3 Overview of Our Approach

In this section, we describe our approach to improving ASIs via client-side access logging (CSAL). We explain the architecture, goals, and threat model.

Device tracking and ASIs. Current ASIs are easy to spoof [39], so they do not provide integrity against mali-

cious clients. This makes them ineffective for distinguishing whether a device's activity is honest or malicious. To solve this, our goal is to enable ASIs that provide granular information to users about account accesses, ideally including static device identifiers (e.g., device serial numbers) as they cannot be spoofed and are easy to associate with a given device. This way, ASIs would help users to identify their devices and sessions and attribute actions to particular devices, which may or may not belong to the user.

A key challenge with such granular ASIs is that such useful access logging stands in tension with privacy under the service-based *device tracking* threat model. Currently, web services and mobile apps are restricted by browsers and OSs from accurately identifying devices. This is on purpose: allowing clients (web/mobile apps) to access static, unique device identifiers would trivially enable device tracking (i.e., associating different requests to the same device). This is a well-known threat model and decades of research (see [35, 52, 56, 61]) and standardization have gone into architecting the web to prevent RPs from maliciously tracking users (e.g., for invasive targeted advertising). There is an ongoing shift away from the user agent strings current ASIs rely upon; instead, browsers may only support client hints [16, 22] that only reveal to web services the features supported by a client. Thus, the current trajectory is that future privacy mechanisms will further degrade ASI utility.

We will resolve this tension by suggesting a new architecture for handling ASI information, combining RP-oblivious information with RP-visible information.

Client-side encrypted access logging. Our underlying insight is that providing information for ASIs visible to users doesn't fundamentally require it to be visible to RPs. The former occurs on the client, so we can handle sensitive information, like device identifiers, in an RP-oblivious manner via E2E encrypted channels. To describe our approach, we start with a high-level abstraction of the architectural components and information flows. We discuss practical realizations of this architecture later.

In our architecture, we consider three entities: the relying party, the client applications, and the client OS. The RP is remote and accessible from clients via standard encrypted network connections (e.g., HTTPS). Clients authenticate to the RP in standard ways (e.g., password-based login, passkeys [19], SSO [8]). Abstractly speaking, current ASIs utilize a plaintext, RP-visible channel to communicate client accesses information between authenticated clients. This information can include UA strings, other HTTP headers, public IP addresses, and more. Our architecture retains this plaintext channel. More broadly, our approach does not sacrifice any existing functionality provided by the RP.

Where our design deviates from current practice is in adding an E2E encrypted channel to transmit RP-oblivious access information between the different authenticated client devices. Because OSs do not trust client applications with static device identifiers, the endpoints of our encrypted channel are OS services. We refer to these services as *encryptors*, the name being reminiscent of the authenticators used in FIDO2. The encryptors can also serve as decryptors, helping populate an OS-managed ASI that communicates access logs with device attributions to users. The UI design itself is out of scope for this paper, and future work will be needed to develop good ASI design patterns.

System goals. One overarching goal is to ensure that logging does not reduces service functionality or forces adoption of new authentication protocols. Thus, we want our system to work alongside existing RP-provided functionality and authentication mechanisms, and to operate in the background without interfering with user experience. The only change is that the ASI will be OS-based (e.g., accessible through an account security UI in a settings app) rather than web- or native-client based.

Our security goals are twofold: privacy and integrity of the access logs. We focus on computationally-limited adversaries. While our formalism focuses on an HBC RP, we discuss mitigations against malicious RP activities in Section 6. From the client side, we target both, UI-bound adversarial users and malicious client software. We assume the OS is trusted, as it already knows the information we want to keep private. We assume secure communication channels (e.g., HTTPS) exist between clients and RPs. This prevents attacks by traditional network adversaries that intercept communications.

Privacy: One of our primary security goals is to prevent RPs and application clients from accessing privileged information sent via the encrypted channels. In particular, static device identifiers that we will associate with log entries. A system achieves *log privacy* if even in the face of malicious clients and RPs, the logs' encrypted content remains private from the relevant parties. A related goal is *unlinkability*: preventing the RP from detecting whether two logins come from the same device. Unlinkability is also relevant for malicious clients in the sense that client software that gets uninstalled and then reinstalled on a device should not be able to distinguish this case from being installed on a new device.

As mentioned in the introduction, a malicious RP might seek to add a malicious end-point and violate log privacy. This will always be possible in our setting since the RP controls account authentication. To defend against such attacks, our target is to ensure that endpoints receiving plaintext log entries are detectable to other clients. This is the same level of guarantee achieved in practice by most E2E encrypted messaging systems. We discuss this threat further in Section 6.

Integrity: One motivation is ensuring that malicious users can't hide accesses or spoof how their devices are identified by ASIs. We refer to this as preserving *log integrity*: protecting the content of the logs from being manipulated by a malicious user, client, or RP. As we will see, by design, logs

will combine plaintexts provided by the RP, client, and OS. An actively malicious RP or client will be able to modify respective plaintext portions (with some limits on clients, discussed Section 6), so here the guarantee is simply that the the OS-provided plaintexts have integrity, allowing among other things attributing such log entries to a particular device. Note that one fundamental limit in our setting is that an actively malicious RP—which, in our setting, controls all messages passed between clients—can always split the view of different clients, and we can at best ensure that those clients can detect inconsistencies should future messages be passed between them. A related goal is *session authenticity*, meaning RPs or the application client cannot impersonate other sessions.

Non-goals: A malicious RP can always deny service to clients; it does not even need to deploy support for CSAL in the first place. So we do not target availability. Given that we are building a logging system, not an ephemeral messaging system, we need content, and the keys to recover it, to be available on clients for a given system-defined duration. Therefore, standard forward secrecy and post-compromise security goals from E2E encrypted messaging settings (e.g., [26, 34, 38]) do not apply here. At last, ASIs are dual-use technologies; while they help legitimate users detect malicious activity in their accounts, they could also enable an attacker that compromise an account to monitor the owner. As argued in [39], the forensic benefits to victims should outweigh the potential risks, but we leave exploration of how different types of ASI data impact abuse scenarios to future work.

4 A Full CSAL Protocol

In this section, we describe a full CSAL protocol Ψ, including the interactions between OS, clients, and RPs.

Cryptographic primitives. Our CSAL protocol uses a few standard cryptographic schemes. We use public-key encryption (PKE) in a key transport kind of mode, and digital signatures (DS). When encrypting, we always sign as well. We denote this by the notation $\{N, M\}^{ssk}_{epk}$, which means encrypting M under public key epk and signing the resulting ciphertext along with some associated data N. In more conventional notation this would be written as $\mathsf{Sign}(ssk, N \,\|\, \mathsf{Enc}(epk, M))$ (note that this is randomized). We also combine signature verification and decryption: for a ciphertext $C \leftarrow\!\!\$\ \{N, M\}^{ssk}_{epk}$, decryption takes in the verification key spk, the decryption key esk, and C. It returns the message: $M \leftarrow \mathsf{Dec}(spk, esk, C)$. For simplicity, we assume C includes N. We denote signing a message by $\{M\}^{ssk}$. We also use symmetric encryption, specifically, authenticated encryption with associated data (AEAD) and sign AEAD ciphertexts. So $\{N, M\}^{ssk}_{K}$ means $\mathsf{Sign}(ssk, N \,\|\, \mathsf{AE}(K, N, M))$, where N is treated as AD and also signed (in case AE is not context committing [29]). Similarly, decryption implicitly includes verification and takes three values as input $M \leftarrow \mathsf{Dec}(spk, K, C)$. We use variable name conventions (K vs. epk) to distinguish AEAD versus PKE. Each session will be associated with a freshly generated PKE key pair, DS key pair, and secret key K, we denote generation of these keys as $((epk, esk), (spk, ssk), K) \leftarrow\!\!\$\ \mathsf{Kg}$.

Session initialization. We start by describing how our CSAL protocol Ψ initializes a new session on a client (also referred to as login algorithm or $\mathcal{L}$). A detailed diagram of session initialization appears in Figure 1. We assume a standard username, password authentication protocol and defer details on other authentication mechanisms to later in this section. A session here corresponds to an HTTP(S) session, which is managed using cookies. A session is active as long as it has a valid cookie, whether it is online or offline. The duration of a session corresponds to the lifetime of a persistent cookie as set by the RP, and as retained by a client — we discuss session lifetime further towards the end of the section. Our CSAL session lifetime is similarly defined.

Our protocol proceeds in two stages. First, a client, call it A, performs a standard authentication with the RP. In our running example, this would be submitting a username and password to the RP, followed by verification of these submitted values. If verification fails, the client receives a login failure notification. Otherwise, the RP generates a message consisting of (1) a short-lived cookie (indicating that the authentication succeeded); (2) a challenge N that should never repeat; (3) a parameter string indicating the RP-supported cryptographic algorithms; (4) the RP's certificate $\mathrm{CERT}_{\mathrm{rp}}$ (containing a signing public key for the RP); (5) a (possibly empty) set of public keys $\mathcal{PK} = \{pk_1, pk_2, \ldots\}$ associated to other sessions (on other devices or, possibly on this device) along with the values[2] needed to check their attestations (which are stored by the RP in a table T_{CERT}); (6) a table T_{KEM} of previously received PKE ciphertexts (used for key transport); and (7) a signature over the resulting message $\sigma_{\mathrm{req}} = \{N \,\|\, \mathrm{params} \,\|\, \mathrm{CERT}_{\mathrm{rp}} \,\|\, \mathcal{PK} \,\|\, T_{\mathrm{KEM}}\}^{sk_{\mathrm{rp}}}$.

The client stores the cookie and then makes an API call to the device's encryptor service for session initialization, passing to the encryptor all the (non-cookie) components of the RP's response $(N, \mathrm{params}, \mathrm{CERT}_{\mathrm{rp}}, \mathcal{PK}, T_{\mathrm{KEM}}, \sigma_{\mathrm{req}})$. It also includes in the call a byte string CL of client-provided information that should be associated with the session in the log (e.g., browser version or other details about the client software instance provided by the client itself).

The encryptor checks the validity of $\mathrm{CERT}_{\mathrm{rp}}$ and uses it to verify σ_{req}. It also checks that all the keys in $\mathcal{PK}$ have valid attestations. If so, the encryptor creates a new session ID sid, unique across all sessions for the user (a sufficiently large random value), and performs four further tasks:

- *Session key generation:* The encryptor checks that it supports the cryptographic algorithms acceptable to the RP,

[2] For notational brevity, we slightly abuse notation and let $\mathcal{PK}$ denote both the public keys, attestation signatures, and associated certificates. We assume algorithms can parse this appropriately.

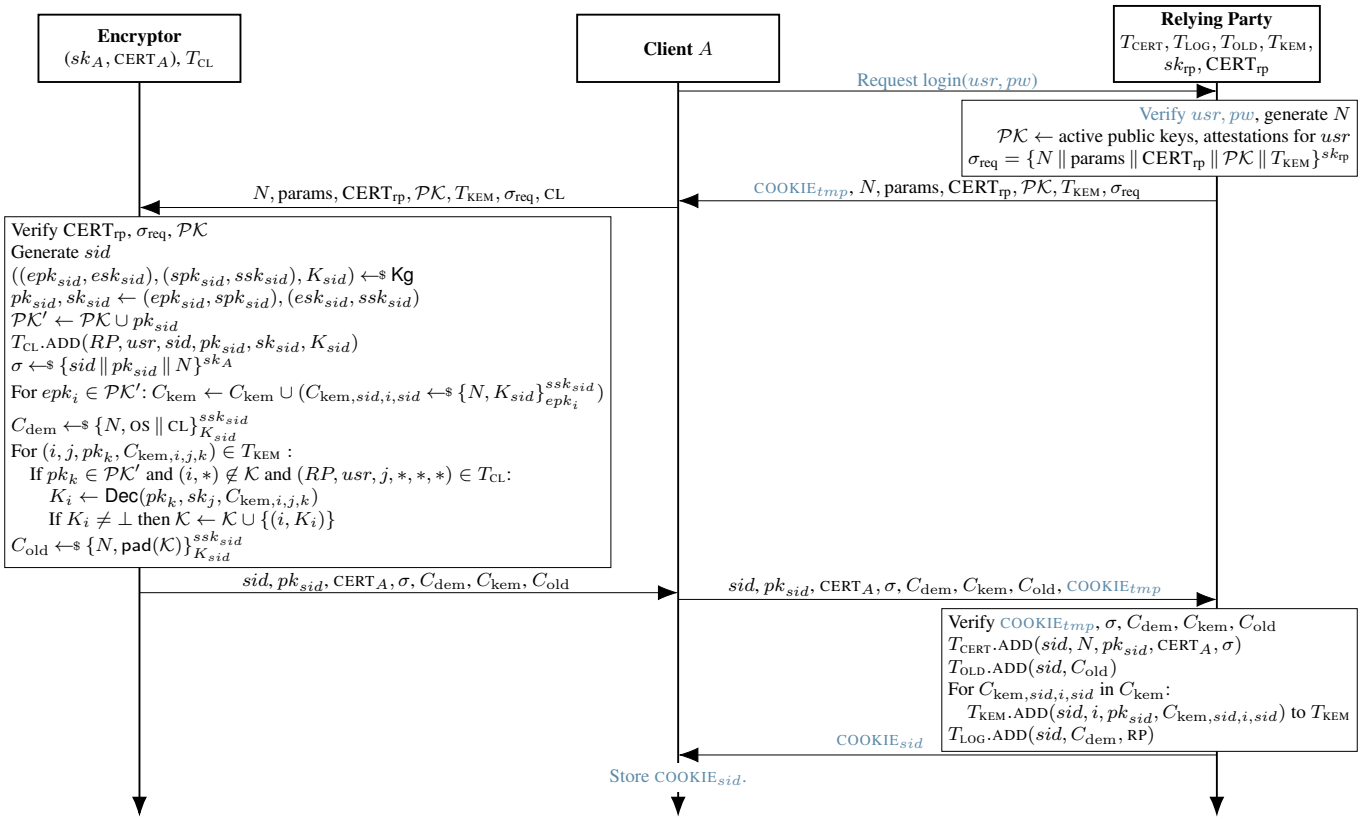

Figure 1: CSAL protocol Ψ login flow ($\mathcal{L}$). If any server-side verification fails, the relying party deletes the relevant COOKIE_{tmp} and returns $\perp$. The values OS, CL, and RP are plaintext log entry byte strings generated by the encryptor, the client, and the relying party, respectively. We use the symbol '$*$' to represent matching any value.

defaulting to some default cryptographic algorithms if not. It then uses one of these algorithms to generate new keys for encrypting and signing:

$$(epk_{sid}, esk_{sid}), (spk_{sid}, ssk_{sid}), K_{sid} \leftarrow\$ \mathsf{Kg} \ .$$

Let $pk_{sid} = (epk_{sid}, spk_{sid})$. The new public key is added to the public key set via $\mathcal{PK}' = \mathcal{PK} \cup \{pk_{sid}\}$. The encryptor creates a new entry in a table T_{CL} that records the RP, the user, the sid, and the new keys. It also sets a timeout on the keys, after which it should delete them.

- *Session key attestation*: The encryptor also generates an attestation for the new session public key pk_{sid}. We denote this by $\sigma \leftarrow\$ \{sid \,\|\, pk_{sid} \,\|\, N\}^{sk_A}$. A vendor-specific attestation certificate CERT_A can be used to verify that the session public key was generated by an encryptor service associated to a known vendor (but not to a specific device). For σ and CERT_A, we rely on FIDO2-style attestations, which we discuss below.

- *Encrypted entry generation*: The encryptor encrypts the key K_{sid} under each $pk_i \in \mathcal{PK}'$, which we denote by $C_{\text{kem},i} := \{N, K_{sid}\}^{ssk_{sid}}_{pk_i}$. Additionally, the encryptor generates an initial encrypted log entry $C_{\text{dem}} := \{N, \text{OS} \,\|\, \text{CL}\}^{ssk_{sid}}_{K_{sid}}$, where OS is a byte string that can include static device identifiers (e.g., a device serial number).

- *Smuggling ciphertext*: The encryptor's last task is to check if it has access to any secret keys K_i from prior sessions. It loops over the KEM ciphertexts in T_{KEM}, checking if it can decrypt them using keys previously generated by this device. Naively, this would be expensive, since T_{KEM} grows quadratically in the number of sessions, but T_{KEM} has only $|\mathcal{PK}|$ different encrypted session keys. The resulting secret keys are collected into a set $\mathcal{K}$ that is then padded to a size fixed by the number of current sessions $m = |\mathcal{PK}'|$ and encrypted via $C_{\text{old}} \leftarrow\$ \{N, \mathsf{pad}(\mathcal{K})\}^{ssk}_{K_{sid}}$.

The encryptor then returns to the client the tuple

$$sid, pk_{sid}, \text{CERT}_A, \sigma, C_{\text{dem}}, C_{\text{old}}, C_{\text{kem},1}, \ldots, C_{\text{kem},m} \ .$$

The client sends this to the RP, along with the temporary authentication cookie. The RP does validity checks, verifying the attestation (see below) and the signatures. It also ensures the number of KEMs is correct. If all checks pass, login is complete, and the RP sends a new session cookie to the client.

The RP inserts $(sid, N, pk_{sid}, \text{CERT}_A, \sigma)$ into a sessions table T_{CERT}; inserts (sid, C_{old}) into a table for "smuggled"

old session keys T_{OLD}; inserts each $C_{\text{kem},i}$ into the table for KEMs T_{KEM} (with information of who encrypted, to whom, and who signed the KEM); and finally inserts $(sid, C_{\text{dem}}, \text{RP})$ into a table T_{LOG} for encrypted log entries. Here RP is plaintext data that the RP wants to associate with this session (e.g., UA string, client IP address). The RP adds pk_{sid} to its set of active public keys associated to active sessions.

This login procedure requires two round trips to the RP, whereas username, password login requires just one. While it's tempting to merge into a single round trip, this would require sending $\mathcal{PK}$ to an unauthenticated client, revealing to arbitrary malicious clients the number of active sessions at the RP. Whether we can improve performance here is an open question, but we expect that in practice as login is infrequent the extra round trip will not be prohibitive.

Attestations. We use OS attestations to ensure encryptors are not being impersonated by actively malicious clients or RPs (such as a compromised browser). At the same time, we don't want certificates from these attestations to allow device fingerprinting (see unlinkability in Section 5 and full version). We adopt FIDO2-style attestations, because they have similar user tracking prevention goals. For example, FIDO2 recommends that authenticator manufacturers ship authenticators in batches and that these batches are sufficiently large. Authenticators in a batch share the same attestation certificate and thus include all those devices in a single anonymity set [17]. Alternatively, they can implement a surrogate attestation model using the authentication key itself to sign or the direct anonymous attestation (DAA) protocol [2, 4]. To verify an attestation, RPs verify the formatting of the attestation statement and the attestation certificate chain.

"Smuggling" old session keys. The login protocol includes a ciphertext C_{old} that encrypts as many old session keys for which the current client device has access. We refer to this as "smuggling" these secret keys, since we're hiding from the RP the fact that this client session has access to these keys. Smuggling is important for correctness, in particular helping us ensure that new sessions on a device can help other devices access the log data for old sessions.

To explain, consider a setting where a user needs to re-authenticate each day with the RP; authenticated sessions only last 24 hours. The user normally uses a primary device A, but occasionally logs in from a second device B. Then it is likely that this user rarely has multiple sessions authenticated at the same time. While device A, which has access to the secret keys for prior sessions from A, could recover them, device B won't be able to recover old sessions from A even if A and B both have an active session during the same time period.

This is where C_{old} comes into play. By encrypting old session keys to which A has access, and also encrypting to all active sessions the current session's secret key, the session on B can see the entire history. Of course if A is never active at the same time as B, this won't work. In fact, in Section 5 we prove that this is a fundamental limitation in our setting where one must perform a key exchange to make encrypted logs accessible to a new client.

Note that the client always prepares a C_{old} of the same size (by way of pad), regardless of how many session secret keys the client can recover. Omitting it or reducing its size would mean the RP can distinguish between sessions initiated from a fresh device versus an old one. Smuggling does use up some bandwidth as T_{KEM} can grow to be quadratic in m (the active number of sessions) and C_{old} is linear in m. Whether this can be improved (or proving it can't be) is a good open question.

Action logging. Once a session has been initialized, the client can generate log entries in response to user (or other) actions. For example, if a user wants to change their authentication methods, the client can log this action in the background. The granularity of logged actions is up to the RP.

The protocol for action logging ($\mathcal{N}$) is essentially identical to the second phase of session initialization, because the client is assumed to have an authenticated session. Figure 4 in Appendix A shows the action logging protocol flow. The RP first generates a new challenge N and a request signature $\sigma_{\text{req}} = \{sid \,\|\, N\}^{sk_{\text{rp}}}$. It sends $(sid, N, \sigma_{\text{req}})$ to the client. To perform an action, the client first forwards that information along with CL to the encryptor. Notice that our protocol tolerates lag between receiving $(sid, N, \sigma_{\text{req}})$ and it being used by the client, which allows the RP to stage fresh challenge requests with the client (e.g., embedded in the change password HTML) to avoid needing two round trips over the network.

Upon receiving the request, the encryptor verifies σ_{req}. It then generates the OS information string OS and ciphertext $C_{\text{dem}} \leftarrow\!\!\$ \{N, \text{OS}\|\text{CL}\}^{ssk_{sid}}_{K_{sid}}$. It returns C_{dem} to the client, who forwards (sid, C_{dem}) to the RP along with the action request (e.g., the HTTP request with old and new passwords). Similar to session initialization, the RP verifies the digital signature in C_{dem}, checks N, and logs a new entry $(sid, C_{\text{dem}}, \text{RP})$ in T_{LOG}. The value RP is as before RP-provided plaintext information about the logged action, e.g., "password change". Finally, the RP may reject a log entry based on its application logic: if an action such as changing the password fails (the client tried to use a disallowed password), the RP can discard the log entry since the action was not, in fact, taken.

Periodic re-encryption. As described so far, only login enables performing key exchange between clients. Our solution is to include a periodic re-encryption ($\mathcal{R}$) mechanism that allows older sessions to grant newer sessions access to (their or other clients') symmetric keys. One can view this as a simple gossip protocol for sharing the secret key K_{sid} associated with a session. As it will be somewhat expensive, we decouple this from logging actions which are on the critical path for user experience, and instead let RP decide the pace at which re-encryptions occur in the background.

Conceptually, the goal is to have all active sessions receive the secret keys used by all the other active sessions. Letting m

be the number of currently active sessions, consider an $m \times m$ matrix whose rows and columns correspond to session IDs $\alpha_1, \ldots, \alpha_m$. Cell i, j is either empty or has a KEM ciphertext $C_{\text{kem},i,j,k} = \{N, K_i\}^{ssk_k}_{pk_j}$ for some $k \in [1..m]$. In words, each entry is a KEM encryption of the secret key associated to the session indicated by the row i, to the public key of the session indicated by the column j. The KEM ciphertext is produced by some session associated with the index k. During a login, a new row and column $m+1$ are added, and the KEM ciphertexts submitted fill out the row for all columns with entries where $k = m+1$. So if we add a sequence of sessions with no further activity, the matrix is a lower triangular matrix, and no session j can read log entries from a session $i < j$ (ignoring, for now, smuggled session keys).

Towards distributing session keys further, we utilize a synchronization mechanism that works as follows. Periodically, clients communicate with the RP, via what we refer to as sync or re-encryption requests (Figure 5 in Appendix A), even if they are not logging any action. A sync request doesn't pass any additional information to the RP (beyond session cookies required to authenticate it); here we assume the RP can look up the session ID associated to the request, call it l, which we will use interchangeably as an index of its row/column number in the matrix. The RP then investigates the current matrix, looking for any cells $[i, j]$ that are empty (j does not yet have the key for i) but for which $[i, l]$ is non-empty (l has the key for i). The RP then generates a re-encryption request message that includes, first, a fresh challenge N. This is followed by a list L of re-encryption request tuples $(i, j, pk_k, C_{\text{kem},i,l,k})$, where k whichever session shared K_i with session l (if shared during login, $k = i$) and $C_{\text{kem},i,l,k}$ is the associated KEM. Finally, the RP generates a request signature $\sigma_{\text{req}} = \{sid \,\|\, N \,\|\, L \,\|\, T_{\text{CERT}}\}^{sk_{\text{rp}}}$.

The client passes this request on to its encryptor, which first checks the request signature validity. Then for each tuple in L, it checks the certificates and public key attestations. It then decrypts (and verifies) $C_{\text{kem},i,l,k}$ to recover K_i, and finally generate a new KEM ciphertext $C_{\text{kem},i,j,l} = \{N, K_i\}^{ssk_l}_{epk_j}$. The encryptor returns a list of the resulting KEM ciphertexts L_{KEM}, which the client sends back to the RP to update T_{KEM}.

As more sessions come online, they can pass along secret keys via re-encryptions. Notice that when one session gains access to the secret key of an active session, it also gains access to session keys smuggled by that session (if any).

Retrieving logs. A client can request that logs be retrieved and shown to the user (history algorithm $\mathcal{H}$). A diagram appears in Figure 6 in Appendix A. To explain, an authenticated client, such as A in our example above, sends a request to the RP to fetch the encrypted logs. The RP returns the identifier sid, the tables T_{CERT}, T_{OLD}, and T_{LOG}, and the column of KEMs from T_{KEM} for that user. Denote the latter by $T_{\text{KEM}}[*, sid]$. It also signs the information it is sending via

$$\sigma_{\text{req}} \leftarrow\!\!\$ \{sid \,\|\, T_{\text{CERT}} \,\|\, T_{\text{OLD}} \,\|\, T_{\text{LOG}} \,\|\, T_{\text{KEM}}[*, sid]\}^{sk_{\text{rp}}} .$$

The client then makes a history API call to the encryptor service, passing along all the values. The encryptor verifies the attestation and keys and then performs three actions:

- *Encapsulated keys recovery:* Using the verification keys in T_{CERT} and the account's secret keys, the encryptor decapsulates each KEM available in T_{KEM}, by verifying the signatures over the keys, the KEM, and then decrypting the ciphertext. It aggregates the account's symmetric keys and the newly recovered key into a set of keys $\mathcal{K}$.
- *Smuggled keys recovery:* The encryptor traverses T_{OLD} from the last to the first entry, verifies and decrypts any ciphertext it can, using keys in $\mathcal{K}$, ignoring dummy padding. Any recovered smuggled keys are added to $\mathcal{K}$ before moving on to the next ciphertext.
- *Log entry recovery:* The encryptor goes over the entries in T_{LOG}, verifies each entry's signature and, if valid, decrypts the entry with the appropriate key in $\mathcal{K}$. Then, it assembles a record for that entry $(\text{RP}_i, \text{CL}_i, \text{OS}_i)$ and adds it to an ordered list $\mathcal{T}$ of these log entry tuples. If it cannot decrypt the entry, it adds instead $(*, *, \text{OS}_i)$ to $\mathcal{T}$.

These plaintext logs are then shown to the user via a privileged OS interface. We cannot return decrypted logs to the client; this would enable malicious client software to learn static identifiers such as serial numbers included in OS entries. Thus, we suggest that the ASI data be shown via OS-managed interfaces, e.g., within an account security page within a system settings app. Future work will be required to determine how to build a usable interface for logs, e.g., via user studies.

Session termination. Sessions can terminate for a variety of reasons. A session may be explicitly terminated because the user asks the client to logout, in which case, the client performs an API call to the encryptor to record that the session has ended, and can also notify the RP about the termination. Sessions may also time out or be remotely ended (e.g. log out through other devices). In these cases, the RP refuses further updates from that session. A user may also delete cookies without notifying the RP. Like in normal web session management, the RP will retain information in the session table until that session times out.

We set two configurable parameters: maximum session duration Δ_{sess} and log visibility duration Δ_{log}. For example $\Delta_{sess} = 90$ days is a standard timeout used in web session management; the user must log in again after Δ_{sess} days. For Δ_{log}, applications can configure how long log files should be retained. No matter how a session sid ends, for correctness, the encryptor and the RP retain the table entries associated with sid for $\Delta_{log} + \Delta_{sess}$ after it becomes invalid. This ensures that no other active session requires any information provided by sid to recover a key or log entry.

Other authentication mechanisms. In our example above, we used standard password-based login. But our protocol is agnostic to authentication mechanisms, requiring just that the

RP first performs authentication (whether it be via password, passkey, TOTP, MFA, etc.) before generating the CSAL login challenge. Thus, it is plug-and-play with existing protocols.

One potentially confusing point is that our encryptor architecture is modeled after the authenticator architectures used in FIDO2 [17]. But in a deployment, these would be distinct protocol flows and must happen sequentially (first authenticator flows, then encryptor). That said, eventual standards and implementations might target unifying encryptors and authenticators, e.g., by extending the client-to-authenticator (CTAP) protocol [1] to include a CSAL encryptor API so that a single OS service could handle both.

5 Formal Treatment of CSAL

In this section, we introduce a formal model for CSAL protocols. Our goals are twofold: (1) to explore basic, fundamental limitations on client-side encrypted logging in our setting and (2) to initiate rigorous security analyses for CSAL protocols. We start by formally defining CSAL protocol syntax and semantics, followed by security notions including integrity, privacy, and unlinkability. We then show an impossibility result that integrity and privacy cannot be achieved simultaneously. Next, we show that we can relax the integrity notion in order to avoid the negative result, and finally prove that our CSAL protocol Ψ (from Section 4) simultaneously enjoys privacy, unlinkability, and this relaxed notion of integrity.

CSAL protocols. We consider client-server CSAL protocols consisting of initialization, action logging, re-encryption, and history protocols between a client and the RP. We model passive adversaries that observe message transcripts and intermediate values, but do not deviate from protocols (i.e., are semi-honest or honest-but-curious) — see the next section for a discussion on fully malicious adversaries. Our formalization takes inspiration from the literature on two-party computation secure against HBC (or semi-honest) adversaries that can corrupt some inputs (e.g., [50]). Thus, each algorithm captures the full execution of the relevant client-server protocol.

Formally, a CSAL scheme $\Pi = (\mathcal{I}_s, \mathcal{I}_c, \mathcal{L}, \mathcal{N}, \mathcal{R}, \mathcal{H})$ is a tuple of six algorithms. The RP initialization algorithm $\mathcal{I}_s$ and the client initialization algorithm $\mathcal{I}_c$ take no inputs and output an initial RP and client state, respectively. The other four algorithms capture login ($\mathcal{L}$), action ($\mathcal{N}$), re-encryption ($\mathcal{R}$), and history ($\mathcal{H}$). Each takes a pair of inputs, one tuple from the client and one tuple from the RP. Login outputs a fresh session ID sid and updated client and RP states. The action and re-encryption algorithms outputs a new client and RP state. The history algorithm outputs a transcript $\mathcal{T}$ consisting of tuples[3] $(t, sid, \text{OS}, \text{CL}, \text{RP})$ indicating a transcript entry type $t \in \{\text{INIT}, \text{ACTION}, \text{REENC}\}$, a session ID, and

[3]Note that in the prior section we only output the last four values: in practice the other records should indicate the entry type, but it's convenient for defining security to not make that assumption here.

$\underline{\mathcal{L}((st_{\text{clt}}, \text{OS}, \text{CL}), (st_{\text{rp}}, \text{RP}))}$:

01 $(T_{\text{CERT}}, T_{\text{LOG}}, T_{\text{OLD}}, T_{\text{KEM}}) \leftarrow st_{\text{rp}}$ // RP challenge generation
02 For row in T_{CERT}:
03 $(i, N_i, pk_i, \text{CERT}_i) \leftarrow$ row
04 $\mathcal{PK} \leftarrow \mathcal{PK} \cup pk_i$
05 $N \leftarrow\!\!\$\ \{0,1\}^n$

06 $(sk_A, \text{CERT}_A, T_{\text{CL}}) \leftarrow st_{\text{clt}}$ // Encryptor response
07 $sid \leftarrow\!\!\$\ \{0,1\}^n$
08 $((epk_{sid}, esk_{sid}), (spk_{sid}, ssk_{sid}), K_{sid}) \leftarrow\!\!\$\ \mathsf{Kg}$
09 $(pk_{sid}, sk_{sid}) \leftarrow (epk_{sid}, spk_{sid}), (esk_{sid}, ssk_{sid})$
10 $\mathcal{PK} \leftarrow \mathcal{PK} \cup pk_{sid}$
11 For $pk_i \in \mathcal{PK}$: $C_{\text{kem},sid,i,sid} \leftarrow\!\!\$\ \{N, K_{sid}\}^{ssk_{sid}}_{pk_i}$
12 $C_{\text{dem}} \leftarrow\!\!\$\ \{N, \text{OS} \,\|\, \text{CL}\}^{ssk_{sid}}_{K_{sid}}$
13 For $(i, j, pk_k, C_{\text{kem},i,j,k}) \in T_{\text{KEM}}$
14 If $(j, *, *, *) \in T_{\text{CL}}$:
15 $K_i \leftarrow \mathsf{Dec}(pk_k, sk_j, C_{\text{kem},i,j,k})$
16 If $K_i \neq \perp$ then $\mathcal{K} \leftarrow \mathcal{K} \cup \{(i, K_i)\}$
17 $C_{\text{old}} \leftarrow\!\!\$\ \{N, \mathsf{pad}(\mathcal{K})\}^{ssk_{sid}}_{K_{sid}}$
18 Add $(sid, pk_{sid}, sk_{sid}, K_{sid})$ to T_{CL}
19 $st_{\text{clt}} \leftarrow (sk_A, \text{CERT}_A, T_{\text{CL}})$

20 Add $(sid, N, pk_{sid}, \text{CERT}_A)$ to T_{CERT} // RP receiving response
21 Add (sid, C_{old}) to T_{OLD}
22 For pk_i in $\mathcal{PK}$: Add $(sid, i, pk_{sid}, C_{\text{kem},sid,i,sid})$ to T_{KEM}
23 Add $(sid, C_{\text{dem}}, \text{RP})$ to T_{LOG}
24 $st_{\text{rp}} \leftarrow (T_{\text{CERT}}, T_{\text{LOG}}, T_{\text{OLD}}, T_{\text{KEM}})$
25 Return $(sid, st_{\text{clt}}, st_{\text{rp}})$

Figure 2: Client-server session initialization for our protocol Ψ as a single algorithm $\mathcal{L}$. We omit some active-attack countermeasures that are irrelevant to our analyses for brevity.

the plaintext entries. In some cases plaintext entries can be missing, which we denote with the symbol '$*$'. For simplicity, we assume that history does not modify the client or RP states; it is easy to modify our treatment to allow such modifications.

To make all this concrete, we describe a simplified version of our protocol Ψ from Section 4 omitting some elements for brevity (such as signatures and their verifications) that are not relevant to our analyses. Pseudocode for the login algorithm appears in Figure 2. Due to space constraints we relegate the other algorithms to Appendix B.

CSAL log integrity. We formalize correctness as a security property, meaning it holds even for a client choosing adversarial sequences of interactions. Figure 3 gives a pseudocode game capturing log integrity security for a CSAL scheme. In our games, all counter variables are implicitly initialized to zero, tables to be everywhere $\perp$, and bit strings to be empty. The game LIS is parameterized by a CSAL scheme $\Pi = (\mathcal{I}_s, \mathcal{I}_c, \mathcal{L}, \mathcal{N}, \mathcal{R}, \mathcal{H})$, a pruning function prune, and an adversary $\mathcal{A}$. Looking ahead, prune is a way to modify the level of integrity guarantee that a CSAL scheme must achieve.

The game allows the adversary $\mathcal{A}$ to initiate new clients via an InitClient oracle. The adversary can then instruct each client to perform a new login to initiate a new session for that client, have the client perform an action within a session, ask it to perform a re-encryption, or have it access its history. The game keeps track of an ideal transcript $\mathcal{T}^*$, which corresponds to the sequence of client logins and actions. It is a list of tuples of the calls in the game, including for which clients and what sessions, as well as the adversarially-input encryptor, client, and RP plaintext log strings.

Along the way, the CSAL protocols are executed — their goal is to similarly keep track of this transcript $\mathcal{T}$. A single RP instance is maintained throughout, by way of updates to a common state st_{rp} that is updated with each RP invocation. Note that all variables in the game are global, but not directly accessible to the adversary except for what is returned by procedures explicitly. This captures adversaries that can monitor all state on the client side.

When the adversary calls the History oracle, specifying a particular client and session via the parameters c, sid, we run the CSAL's history routine. We then compare its output $\mathcal{T}$, to the ideal transcript $\mathcal{T}^*$, after processing each entry on $\mathcal{T}^*$ with the transcript pruning algorithm $\mathsf{prune}(\mathcal{T}^*, sid)$. The latter outputs an ordered list of entries $(t_i, sid_i, \text{OS}_i, \text{CL}_i, \text{RP}_i)$.

As mentioned above, pruning functions control the level of integrity required of a scheme. At one extreme, if prune outputs nothing, then integrity is trivial to achieve. At the other extreme, if prune simply processes each $\mathcal{T}^*$ entry by removing the client identifier, it requires that the CSAL protocol can ensure History outputs information on every single login and action—a strong guarantee that, as we will see, cannot be achieved by CSAL schemes that provide log privacy. We refer to this latter pruning function as the identity pruning function, denoted by $\mathsf{pruneId}$. Later in the section we will define other pruning functions that lie between these two extremes.

The adversary wins if the client's history does not match the pruned, ideal transcript. We define the $\text{LIS}_{\Pi,\mathsf{prune}}$-advantage of adversary $\mathcal{A}$ as the probability that it can win the game:

$$\mathsf{Adv}^{\text{lis}}_{\Pi,\mathsf{prune}}(\mathcal{A}) = \Pr\left[\text{LIS}_{\Pi,\mathsf{prune}}(\mathcal{A}) \Rightarrow 1\right]$$

where the probability space is defined over the random coins of the game and $\mathcal{A}$.

Definition 5.1 *$\mathcal{A}$. We say that a CSAL scheme Π is ϵ-correct for prune if for all $\text{LIS}_{\Pi,\mathsf{prune}}$-adversaries $\mathcal{A}$ it holds that $\mathsf{Adv}^{\text{lis}}_{\Pi,\mathsf{prune}}(\mathcal{A}) \leq \epsilon$.*

CSAL privacy from RPs. We are interested in CSAL schemes that preserve the privacy of device identifiers from the RP. To formalize this, we use a left-or-right privacy notion that tasks an adversary with distinguishing between transcripts of the protocol applied to one of two randomly chosen sequences of log entries. The privacy security game is shown in Figure 3. The game starts by choosing a challenge bit b, initializing the RP's state st_{rp}, and then running the adversary who gets access to the st_{rp} and to oracles for client initialization, logins, actions, and re-encryption. For the login oracle, the adversary queries on a client c, two chosen device plaintexts OS_0, CL_0 and OS_1, CL_1, and the RP plaintext log entry RP. For the action oracle, the adversary queries on a session identifier, two pairs of client-side log entries, and a RP plaintext log entry. The CSAL login and action algorithms are executed using one of the two identifiers chosen based on the challenge bit b. The adversary receives the new session ID sid and updated RP state st_{rp}. We require queries are such that $|\text{OS}_0| = |\text{OS}_1|$ and $|\text{CL}_0| = |\text{CL}_1|$.

The adversary can also invoke re-encryption queries for adversarially-chosen client and session. The adversary does *not* have the ability to query a history oracle — this would lead to trivially violating log privacy. See Section 6 for more discussion of this threat model in which an RP colludes with a malicious client.

We define the PRIV_Π-advantage of an adversary $\mathcal{D}$ as the probability that it outputs 1, namely:

$$\mathsf{Adv}^{\text{priv}}_{\Pi}(\mathcal{D}) = 2 \cdot \Pr\left[\text{PRIV}_\Pi(\mathcal{D}) \Rightarrow 1\right] - 1 \ .$$

The probability space above is defined over the random coins of the game and $\mathcal{D}$.

CSAL unlinkability. Due to space constraints, we formalize an unlinkability security game in Appendix C. There, the adversary must distinguish between sessions from different versus the same client. We prove our scheme Ψ achieves it in the full version of the paper.

Privacy limits integrity. There is an inherent tension between log integrity and privacy. Intuitively, no private scheme can ensure that all clients have the same view of the transcript $\mathcal{T}$: in our setting, where clients do not shared secrets before communicating, they need to somehow exchange cryptographic key material in order to communicate privately. We have the following theorem (see full version for proof).

Theorem 5.1 *Let Π be an CSAL scheme that is ϵ-correct for the identity pruning function $\mathsf{pruneId}$. We give $\mathcal{D}$, a PRIV_Π-adversary, such that $1-2\epsilon \leq \mathsf{Adv}^{\text{priv}}_{\Pi}(\mathcal{D})$. Adversary $\mathcal{D}$ makes two queries and runs in a small constant amount of time.*

Achievable integrity plus privacy. This raises the question of what level of log integrity (correctness) is achievable when we want privacy, which we can characterize with the pruning functions that modify entries to what should be communicable without compromising privacy. In the example above, if prune replaced OS and CL in the first entry with '$*$', then the negative result would no longer work. As we will show, we can in fact build a scheme (ours) that achieves LIS and PRIV security for that pruning function and sequence of logins and actions.

To generalize this we need to specify a pruning function that converts the ideal transcript into one that is achievable,

$\underline{\text{LIS}_{\Pi,\mathsf{prune}}(\mathcal{A})}$
$st_{\text{rp}} \leftarrow\!\!\$\ \mathcal{I}_s$; win $\leftarrow 0$
$\mathcal{A}^{\mathcal{O}}$
Return win

$\underline{\text{InitClient}}$
$\mu \leftarrow \mu + 1$; $st_\mu \leftarrow\!\!\$\ \mathcal{I}_c$

$\underline{\text{Login}(c, \text{OS}, \text{CL}, \text{RP})}$
If $c > \mu$ then Return $\perp$
$(sid, st_c, st_{\text{rp}}) \leftarrow\!\!\$\ \mathcal{L}((st_c, \text{OS}, \text{CL}), (st_{\text{rp}}, \text{RP}))$
$\gamma_{c,sid} \leftarrow$ True
$q \leftarrow q + 1$
$\mathcal{T}^*[q] \leftarrow (\text{INIT}, c, sid, \text{OS}, \text{CL}, \text{RP})$
Return (sid, st_c)

$\underline{\text{Action}(c, sid, \text{OS}, \text{CL}, \text{RP})}$
If $\gamma_{c,sid} = \perp$ then Return $\perp$
$(st_c, st_{\text{rp}}) \leftarrow\!\!\$\ \mathcal{N}((st_c, sid, \text{OS}, \text{CL}), (st_{\text{rp}}, sid, \text{RP}))$
$q \leftarrow q + 1$
$\mathcal{T}^*[q] \leftarrow (\text{ACTION}, c, sid, \text{OS}, \text{CL}, \text{RP})$
Return st_c

$\underline{\text{Re-encryption}(c, sid)}$
If $\gamma_{c,sid} = \perp$ then Return $\perp$
$(st_c, st_{\text{rp}}) \leftarrow\!\!\$\ \mathcal{R}((st_c, sid), (st_{\text{rp}}, sid))$
$\mathcal{T}^*[q] \leftarrow (\text{REENC}, c, sid)$
Return st_c

$\underline{\text{History}(c, sid)}$
If $\gamma_{c,sid} = \perp$ then Return $\perp$
$(\mathcal{T}, st_c) \leftarrow\!\!\$\ \mathcal{H}((st_c, sid), (st_{\text{rp}}, sid))$
If $\mathcal{T} \neq \mathsf{prune}(\mathcal{T}^*, sid)$ then
  win $\leftarrow 1$
Return $\mathcal{T}$

$\underline{\text{PRIV}_{\Pi}(\mathcal{D})}$
$st_{\text{rp}} \leftarrow\!\!\$\ \mathcal{I}_s$; $b \leftarrow\!\!\$\ \{0,1\}$
$b' \leftarrow\!\!\$\ \mathcal{D}^{\mathcal{O}}(st_{\text{rp}})$
Return $(b' = b)$

$\underline{\text{InitClient}}$
$\mu \leftarrow \mu + 1$; $st_\mu \leftarrow\!\!\$\ \mathcal{I}_c$

$\underline{\text{LoginLR}(c, \text{OS}_0, \text{CL}_0, \text{OS}_1, \text{CL}_1, \text{RP})}$
If $c > \mu$ then Return $\perp$
$(sid, st_c, st_{\text{rp}}) \leftarrow\!\!\$\ \mathcal{L}((st_c, \text{OS}_b, \text{CL}_b), (st_{\text{rp}}, \text{RP}))$
$\gamma_{c,sid} \leftarrow$ True
Return (sid, st_{rp})

$\underline{\text{ActionLR}(c, sid, \text{OS}_0, \text{CL}_0, \text{OS}_1, \text{CL}_1, \text{RP})}$
If $\gamma_{c,sid} = \perp$ then Return $\perp$
$(st_c, st_{\text{rp}}) \leftarrow\!\!\$\ \mathcal{N}((st_c, sid, \text{OS}_b, \text{CL}_b), (st_{\text{rp}}, sid, \text{RP}))$
Return st_{rp}

$\underline{\text{Re-encryption}(c, sid)}$
If $\gamma_{c,sid} = \perp$ then Return $\perp$
$(st_c, st_{\text{rp}}) \leftarrow\!\!\$\ \mathcal{R}((st_c, sid), (st_{\text{rp}}, sid))$
Return st_{rp}

Figure 3: **(Left)** Logging integrity security (LIS) game for a CSAL scheme Π, pruning function prune, and adversary $\mathcal{A}$. Here $\mathcal{O} = (\text{InitClient}, \text{Login}, \text{Action}, \text{Re-encryption}, \text{History})$ represents the set of oracles the adversary can call. **(Right)** Logging RP privacy (PRIV) game for a CSAL scheme Π and adversary $\mathcal{D}$. Here $\mathcal{O} = (\text{InitClient}, \text{LoginLR}, \text{ActionLR}, \text{Re-encryption})$.

given the pattern of accesses so far made in the course of the LIS game. Towards this, consider a graph $G = (V, E)$, where V consists of nodes labeled by sessions, and a directed edge (u, v) between nodes means session u can access log entries made by session v. We can build this graph iteratively by processing the ideal transcript $\mathcal{T}^*$ which is an ordered list of six-tuples $(t_i, c_i, sid_i, \text{OS}_i, \text{CL}_i, \text{RP}_i)$ for $1 \le i \le \tau$. Here we denote the entry type by a label $t_i \in \{\text{INIT}, \text{ACTION}, \text{REENC}\}$.

If $t_i = \text{INIT}$, then we add a new node labeled by sid_i to the graph. We add an edge (u, sid_i) and (sid_i, u) for every u whose client is the same as c_i: all sessions on the same client are fully connected (clients in our model are stateful and so can decrypt logs from any session on that device). We include here the edge (sid_i, sid_i). Once that is done, we then an add edge (u, sid_i) for all sessions u on other clients. If instead $t_i = \text{REENC}$, then we add an edge (u, sid_i) for all sessions u that do not already have an edge to sid_i. If finally $t_i = \text{ACTION}$, we do nothing to the graph. Finally, we can compute the transitive closure of G, and use the result as the graph G for processing subsequent entries.

Then, our pruning function pruneR, that takes reachability into account, processes a transcript $\mathcal{T}^*$ and session id sid as follows. It first generates a reachability graph G as described above, by processing $\mathcal{T}^*$. Then, for each entry $(t_i, c_i, sid_i, \text{OS}_i, \text{CL}_i, \text{RP}_i)$ where $t_i \in \{\text{INIT}, \text{ACTION}\}$, it adds the entry $(t_i, sid_i, \text{OS}, \text{CL}, \text{RP}_i)$, to the output transcript $\mathcal{T}$, where $(\text{OS}, \text{CL}) = (\text{OS}_i, \text{CL}_i)$ if $(sid, sid_i) \in G$ (an edge from sid to sid_i exists), else $(\text{OS}, \text{CL}) = (*, *)$. Recall that '$*$' denotes a pruned portion of the log entry. RP_i are never pruned.

Now we introduce our first pair of positive results: our CSAL scheme Ψ is ϵ-correct for pruning function pruneR with $\epsilon = 0$ (perfectly correct) and it achieves privacy, assuming the security of the underlying encryption schemes. We conjecture that no private scheme can do better than pruneR on LIS correctness. This would mean that our protocol is optimally correct, but we leave proof of this fact to future work. The proofs of the following theorems are in the full version.

Theorem 5.2 *Our CSAL scheme Ψ is ϵ-correct for pruning function pruneR with $\epsilon = 0$.*

To account for resource usage in our reductions, let $T^{\mathsf{PKE}}_{\mathsf{Enc}}(\ell)$ denote the worst-case runtime of PKE on input plaintext with length ℓ and $T^{\mathsf{AE}}_{\mathsf{Enc}}(\ell)$ be the worst-case runtime of AE encryption with plaintexts of length ℓ. Let ℓ_1 be the maximum length of the encoding of a log entry $(\text{OS} \,\|\, \text{CL})$. We assume κ is the length of symmetric keys used in the scheme (i.e., $\kappa = 128$). We assume oracle calls, fixed-length randomness generation, and table lookups and use $\mathcal{O}$ to hide small constants.

Theorem 5.3 *Let Ψ be our CSAL scheme built using PKE scheme PKE and AEAD AE. Let $\mathcal{D}$ be a PRIV_{Ψ}-adversary making at most q queries. Then we give $\text{MU-IND-CPA}_{\mathsf{PKE}}$-adversary $\mathcal{D}_{pke}$ and $\text{IND-CPA}_{\mathsf{AE}}$-adversary $\mathcal{D}_{ae}$ such that*

$$\mathsf{Adv}^{\text{priv}}_{\Psi}(\mathcal{D}) \le 2 \cdot \mathsf{Adv}^{\text{mu-ind-cpa}}_{\mathsf{PKE}}(\mathcal{D}_{pke}) + 2q \cdot \mathsf{Adv}^{\text{ind-cpa}}_{\mathsf{AE}}(\mathcal{D}_{ae}) \ .$$

Adversary $\mathcal{D}_{pke}$ makes at most $q^2 + q$ oracle queries, while $\mathcal{D}_{ae}$ makes at most $q+1$ oracle queries. Their run times are at most $\mathcal{O}(q^3 + q \cdot T^{\mathsf{AE}}_{Enc}(\ell_1) + q \cdot T^{\mathsf{AE}}_{Enc}(q\kappa))$ and $\mathcal{O}(q^3 \cdot T^{\mathsf{PKE}}_{Enc}(\ell_1))$.

6 Security Beyond Passive Adversaries

In Section 5 we formalized passive security for our CSAL protocol, showing that an HBC RP cannot infer information about client-side plaintext log entries, nor can it link two sessions to the same device. This addresses the most pressing threats to user privacy, but leaves open the question of what actively malicious RPs might achieve. Future work is needed to formally model these more complicated threats; here we provide a structured discussion of both what actively malicious attacks we believe our protocol prevents and which attacks it does not prevent.

For each security goal we identify actively malicious attacks. For each attack we discuss either how our protocol mitigates them, what mitigations could be added (and their trade-offs in terms of complexity or performance), or how some of the malicious attacks appear to be fundamental—no scheme in our setting will be able to prevent them, though protocols might be able to detect them. While we cannot claim that our analysis predicts all possible attacks, we believe we have covered the most pressing issues.

Log privacy. We start with active attacks targeting log privacy. Recall that we want RP's to be blinded from the client- and encryptor-provided log entries (CL and OS, respectively).

RP adds a malicious client device. A malicious RP can allow a malicious client device to log into a user account. This is similar to the "ghost user" approach suggested for covert,platform-facilitated monitoring of encrypted messaging [57]. This attack occurs because the RP controls authentication and appears to be fundamental in any setting where this is the case, so no CSAL protocol can fully mitigate it.

We do note that our protocol does add some friction to success of this attack, by ensuring that. First, fully automating the attack would require the malicious RP to subvert some device's OS encryptor (to decrypt log entries), i.e., rooting the device and if the encryptor is protected by hardware subverting those defenses as well. While such attacks exist (see [68]), they may be out-of-reach for some adversaries. Second, honest client devices will be aware of the inserted session because the client needs the public key of the malicious device to share information with it. While the client will not know, a priori, whether the public key is associated with a malicious client, users could, in theory, determine that a particular RP is always adding additional devices. The attack's detectability should strongly disincentivize RPs from behaving poorly. Finally, in our protocol encryptors can authenticate the RP. Hence, OS vendors can maintain quality control over which RPs they support, rejecting encryptor requests for RPs with a bad reputation (like how browsers manage root TLS certificates).

RP deploys a malicious client. An RP may also ship malicious client software to an (honest) client device, whether it be malicious javascript running in the browser or a malicious native app, to learn static identifiers and fingerprint devices. This threat is not hypothetical: malicious RPs are known to deploy tracking or device fingerprinting code [25, 42, 43, 47, 54, 55, 61, 71], which is why OS's restrict them from having access to static device identifiers. Our protocol maintains the same level of protections as current OS's, by restricting access to the OS-supplied log information OS to the OS-managed encryptor, combined with end-point attestations, preventing the client from performing a meddler-in-the-middle attack against the encrypted log entries.

RP reuses nonces: Our protocol uses RP-chosen nonces to prevent a malicious client from replaying prior log files. A malicious RP could repeat nonces, allowing the replay attack—we discuss this below regarding log integrity. For log privacy, replaying the nonces has no impact, as privacy is provided by the client-side chosen randomness used within encryption.

Unlinkability. A passive RP only learns the vendor identity (from the encryptor certificate), and cannot learn anything else about whether two sessions originate from the same device (see Appendix C). The same RP-added malicious device attack discussed above would undermine unlinkability, which, again, seems fundamental but with the partial mitigations applying here as well.

Malicious client linking attacks: A malicious client may try to fingerprint a device. Our encryptor-based protocol should make this harder, though it will require that the OS encryptor implementations isolate sessions by application (i.e., prevent cross-application requests) and clear application-related sessions state when an app is uninstalled.

Log integrity. To address log integrity, we break our discussion up into two integrity threat models: (1) compromised client software (and an honest RP), and (2) a malicious RP (including possibly malicious client software).

Malicious client log hiding or spoofing: A malicious client might try to hide its sessions or some of its actions, for example by refusing to send ciphertexts, sending bogus ciphertexts that fail decryption, or removing public keys from $\mathcal{PK}$. For the first attack, an honest RP will prevent login or any action without a properly formatted and digitally signed CSAL response. While by design the RP cannot verify the contents of the encrypted log entry, recall from Section 4, that we rely on FIDO2-style attestations and only the encryptor can sign the ciphertexts. Hence, we prevent the second threat of sending bogus ciphertexts because the OS encryptor is the only entity that can generate an RP-accepted ciphertext. So at best the malicious client can change the client log info string CL, but, importantly, the OS-provided log string OS will be correct. The encryptor might be able to further prevent encrypting junk CL by performing sanity checks and rejecting requests that do not satisfy those checks (e.g., that an application description within CL matches the calling application). The third issue here is a malicious client removing public keys from $\mathcal{PK}$, to, for example, selectively deny some clients from learning specific log entries. This is prevented because the RP signs request, which the OS encryptor verifies.

If an adversary compromises the OS, then our protocol does not prevent spoofed encrypted log entries — we are not suggesting full hardware-based attestation of the OS. Partly, because we believe the most pressing threat to log integrity is from UI-bound adversaries that will not even compromise a client, let alone the OS. Also, because full attestation is difficult to deploy. Even here, our protocol prevents the compromised encryptor from impersonating other existing sessions and ensures that honest clients receive any maliciously-generated ciphertext. This means that the malicious access cannot be fully hidden and users will see a log entry with (potentially spoofed) OS and client information strings, but a valid RP-provided string.

Malicious RP selectively dropping entries: An actively malicious RP can always deny log access, or even entry access, to any client session. Like in encrypted group messaging, attacks that split the view of a group are unavoidable, and we can at most force the malicious RP to be consistent: if it removes clients or particular client actions from the view of another client, it cannot allow communication between those clients subsequently. Our current protocol does not yet achieve this, because clients do not check for consistency across interactions with the RP nor is there any way for a client to detect that an action log ciphertext C_{dem} was dropped.

We can prevent such attacks with small changes to our current protocol. First, clients can record during login and re-encryption, what public key set they observed and use this to check for consistency of $\mathcal{PK}$. Second, to prevent individual actions from being omitted, each log entry can include as part of its associated data, a hash of the previous log entry ciphertext C_{dem} from that session. Recipients can then verify a total ordering over log entries (per session), hence the RP can at best truncate sessions (which is always possible). This would also prevent a related attack, reordering log entries, even without reliance on client-side time stamps. These additional mitigations might cause performance issues for complex, asynchronous web applications, such as those that allow parallel calls from the client to the RP.

7 Implementation and Evaluation

We implemented a prototype of our CSAL protocol to evidence feasibility of the approach. Now we discuss some details and performance of our proof-of-concept.

Implementation components. Our CSAL prototype consists of a CSAL-supporting RP, a CLI client, and an encryptor service, all built in Python. Communication between the client and RP is via HTTPS, and between the client and encryptor uses the Python subprocess module. Storage on the RP and encryptor uses SQLite. The encryptor currently runs as an unprivileged process to ease development.

For the public key encryption, we utilize HPKE [11] as implemented in the PyHPKE library [21]. For digital signatures and AEAD, we use RSA-PSS and Fernet AEAD from the Python cryptography library [20].

We set challenge nonces and session IDs to be 16 B (matching the length required by FIDO2's WebAuthn [18]). Certificates are standard self-signed X.509 PEM format; cookies are Base64 encoded 32-byte values that are URL safe. We set OS to be the system's serial number, and CL to be a standard ASCII-encoded UA string. For both the RP and encryptor certificates we add a certificate generation routine that populates the certificate with random subject name details, issuer name, validity period, and certificate serial number. HTTP payloads for protocol messages between the client and RP are encoded as byte strings using Python's pickle library.

Evaluation. We performed micro-benchmarks on the CSAL login, re-encryption, and log retrieval payloads, to measure the CPU costs and bandwidth utilization under a single session, and analyze the growth based on having more sessions. We did not evaluate action logging because its payload size and computation is independent of the number of sessions. For convenience, we run the prototype locally with the RP as a localhost web service. While we could measure the protocol over a wide-area network, our overheads are small enough that such measurements would be dominated by the network's performance profile. CPU timing measurements use the Python timing library on a MacBook Pro with an M1 CPU, 8 GB of RAM running Sonoma v.14.6.

Login: For session initialization we first measure the bandwidth cost of login without key smuggling. Sending $(N, \text{params}, \text{CERT}_{\text{rp}}, \mathcal{PK}, \text{COOKIE}, \sigma_{\text{req}})$ to the client takes 1,696 B and from client to encryptor as 1,825 B including the value of CL. In this first login, $\mathcal{PK}$ and CERT_{rp} are empty.

Each additional public key in $\mathcal{PK}$ adds 1,753 B to the payload; 69 B for the pickled HPKE public key, 451 B for the RSA-PSS public key in PEM format, and 1,233 B for the encryptor certificate in PEM format. Running 10 CSAL logins back-to-back the total payload size is 36,834 B to the client and 38,112 B to the encryptor, and running 40 CSAL logins result in a payload size of 788,896 B to the client. This is reasonable given the infrequency of logins.

The RP stores a row for each login containing a user id, an account id (e.g., email address), the session id retrieved from the encryptor, the corresponding public key associated with the session id, and the ciphertexts (C_{kem}, C_{dem}). Thus, storage for one such entry is 515 B, measured in bytes without any encoding. For the ciphertexts created by the encryptor, C_{dem} measures 292 B and each C_{kem} has size 60 B.

On the encryptor's end, a single populated row on the encryptor's database is 2,263 B and contains the sid, user account info (e.g., email address), RP info (e.g., facebook.com), and the public, private, and symmetric keys associated with the session. Given that HPKE provides an encryption key encapsulated with the public key, we also store the encapsulated key on the database for log decryption later on. The size of

the entire sqlite database with a single row is 8 KB; it incurs additional storage associated with indices, metadata, etc.

To ensure that new sessions can access data belonging to old sessions, the encryptor smuggles old secret keys to new sessions during login. During the smuggling, the encryptor iterates over T_{KEM} and attempts to decrypt all ciphertexts using the secret keys it has stored. For T_{KEM} with a single entry and public key, $\mathsf{pad}(\mathcal{K})$ is fixed at 44 B and C_{old} is 164 B. For two entries, $\mathsf{pad}(\mathcal{K})$ has size 88 B and C_{old} 328 B, and so on.

Thus, the encryptor's response is proportional to the number of public keys in $\mathcal{PK}$. The entire response to the client for an empty $\mathcal{PK}$ is 2,885 B. Its breakdown is 16 B for sid, 65 B for pk, 1,233 B for CERT_A, 292 B for C_{dem} and 60 B for C_{kem}. In addition to this, each of the ciphertexts are signed thus resulting in a 256 B digital signature, and a 451 B RSA public key in PEM format. σ is also 256 B. For reference, an initial password authentication request to Amazon (without CSAL) embeds a payload of 14,027 B.

The entire time elapsed for a round trip CSAL login is 7 ms for an empty $\mathcal{PK}$. This averaged over 10 CSAL logins increases to 146.307 ms. We do note that we used the *time.sleep()* function to introduce some delays as data is received from the client and sent back to the RP to avoid incomplete or truncated data. However, the numbers we report here are without factoring in the delays.

Re-encryption: For our analysis we simulate the first client device to log into the account, such that its key K is contained in only one entry in T_{KEM}, and with a growing number of sessions that do not know K. In the case where there are two sessions, the RP payload sent to the encryptor has size 2,653 B, including L and the two entries in T_{CERT}. The encryptor payload is 382 B for L_{KEM}. The sizes of L and L_{KEM} grow linear to the number of re-encryptions requested. For n sessions, the worst case requires $O(n^2)$ re-encryptions, but each missing KEM only needs to be encrypted once. Since re-encryption can be performed in the background, its overheads will not affect user experience.

Log retrieval: For log retrieval, the protocol forwards tables $T_{\text{CERT}}, T_{\text{LOG}}, T_{\text{OLD}}$, the corresponding entries for sid in T_{KEM}, and a signature σ_{req} to the client. Here we assume each table has a single session's record from a client login, and that session requested to view their logs. The resulting payload sent to the user has size of 2,482 B (no smuggling) and 2,631 B for the smuggling case — this will grow proportionally to the number of active sessions.

On the encryptor's end multiple operations are carried out to display the final log (see Section 4). Decryption and signature verification takes an average of 0.63 ms of CPU time, taken over 10 runs. With more sessions and entries, this time will increase linearly. While our experimental results are preliminary, they indicate that CSAL protocols will not incur prohibitive overheads in practice.

8 Discussion

Here we discuss remaining challenges, paths, and considerations for the deployment of our protocol.

Deployment considerations. Our prototype shows that the performance cost of deploying our CSAL protocol is not prohibitive. The major deployment obstacles come from needing support from OS vendors and other infrastructure challenges, specially to provide security against actively malicious RPs.

Reliance on FIDO2: Our protocol relies on FIDO2-style attestations for unlinkability and encryptor authenticity in the face of an actively malicious client or RP. FIDO2 is already deployed, and adopting their attestation mechanisms will speed deployment of CSAL. Unfortunately, we inherit this approach's limitations, such as depending on vendors implementing certificates properly to achieve unlinkability.

Certificate management: CSAL requires public key infrastructures (PKI) to manage certificates from both the encryptors and the RPs. From the OS side, we would like to be able to list and revoke RPs based on their behavior, while from the RP side, we must handle expired certificates, etc. For the management of the encryptor certificates, deployments could use a system similar to FIDO MDS [3], which provides the RP with the attestation key in FIDO attestation protocols. Meanwhile, for the RPs, one could use an infrastructure similar to the one for TLS certificates [64].

OS-vendor support: A key component of our system is the OS encryptor. Deploying it would require collaboration between OS vendors and browsers or other clients, similar to what was required for FIDO2. We acknowledge this is not an easy task and would likely take years. However, seeing how far FIDO2 has come, we believe it is possible. Additionally, it's possible that parts of FIDO2 specifications and implementations could be reused for CSAL.

Limitations and future work. As discussed earlier, attestation is difficult to deploy and there might be devices that do not support it. If attestation is unavailable, we provide at least as much information as current ASIs and provide integrity against HBC RPs, which current ASIs do not do.

While we provide a discussion of malicious security, we leave the formalization of a malicious RP for future work. Additionally, we assumed a client-server architecture, future work could look into alternative architectures, such as peer-to-peer. Finally, usability of ASIs in general remains an open question. Future work will be needed to explore what CSAL-enabled ASIs should show to users and how that information should be displayed, in order to help the legitimate account owner understand their security. At the same time, we will also need to review such designs to assuage safety concerns should CSAL-enabled ASIs become available to an attacker that successfully compromises the account.

9 Ethics and Open Science Policy

Our paper tackles an important problem: building account security systems that ensure trustworthy information for users. Our research and experiments consist of a prototype implementing an encryptor for a local application, which synthetic account data. As such we had no ethical concerns. Our prototype is available as a public, open-source project.

Acknowledgements

This work was funded in part by NSF grant CNS-2120651 and a generous gift from Google.

References

[1] Client to authenticator protocol (ctap). https://fidoalliance.org/specs/fido-v2.1-ps-20210615/fido-client-to-authenticator-protocol-v2.1-ps-20210615.html. (Accessed on 04/29/2024).

[2] Fido 2.0: Key attestation format. https://fidoalliance.org/specs/fido-v2.0-ps-20150904/fido-key-attestation-v2.0-ps-20150904.html. (Accessed on 04/25/2024).

[3] Fido alliance metadata service. hhttps://fidoalliance.org/metadata/. (Accessed on 09/04/2024).

[4] fidoalliance.org/specs/fido-v2.0-id-20180227/fido-ecdaa-algorithm-v2.0-id-20180227.pdf. https://fidoalliance.org/specs/fido-v2.0-id-20180227/fido-ecdaa-algorithm-v2.0-id-20180227.pdf. (Accessed on 04/29/2024).

[5] Get a Verification Code and Sign in With Two-factor Authentication - Apple Support. https://support.apple.com/en-us/HT204974. (Accessed on 12/04/2022).

[6] How encrypted email works | proton. https://proton.me/blog/encrypted-email. (Accessed on 01/27/2024).

[7] icloud data security overview - apple support. https://support.apple.com/en-us/102651. (Accessed on 01/27/2024).

[8] Introduction to federated identity and the fedid cg | federated identity community group. https://www.w3.org/community/fed-id/2022/04/21/introduction-to-federated-identity-and-the-fedid-cg/. (Accessed on 01/28/2024).

[9] Passkeys. https://fidoalliance.org/passkeys/. (Accessed on 01/11/2024).

[10] Passkeys. https://developer.apple.com/passkeys/. (Accessed on 01/11/2024).

[11] Rfc 9180 - hybrid public key encryption. https://datatracker.ietf.org/doc/rfc9180/. (Accessed on 04/25/2024).

[12] Securitywhitepaper.pdf. https://mega.nz/SecurityWhitepaper.pdf. (Accessed on 01/27/2024).

[13] The simplest, most secure way to sign into your accounts without a password. https://safety.google/authentication/passkey/. (Accessed on 01/11/2024).

[14] Two-factor authentication (2fa). https://duo.com/product/multi-factor-authentication-mfa/two-factor-authentication-2fa. (Accessed on 01/11/2024).

[15] Use your internet accounts on mac - apple support. https://support.apple.com/guide/mac-help/add-your-email-and-other-accounts-mh35565/mac. (Accessed on 04/27/2024).

[16] User-agent client hints. https://wicg.github.io/ua-client-hints/. (Accessed on 01/28/2024).

[17] Web authentication: An api for accessing public key credentials - level 2. https://www.w3.org/TR/webauthn-2/#batch-attestation. (Accessed on 04/29/2024).

[18] Web authentication: An api for accessing public key credentials - level 2. https://www.w3.org/TR/webauthn-2/#sctn-spec-roadmap. (Accessed on 03/05/2024).

[19] Web authentication: An api for accessing public key credentials level 1. https://www.w3.org/TR/webauthn-1/. (Accessed on 01/28/2024).

[20] Welcome to pyca/cryptography. https://cryptography.io/en/latest/. (Accessed on 09/04/2024).

[21] Welcome to pyhpke — pyhpke 0.5.3 documentation. https://pyhpke.readthedocs.io/en/stable/index.html. (Accessed on 04/25/2024).

[22] What is user-agent reduction? | privacy sandbox | google for developers. https://developers.google.com/privacy-sandbox/protections/user-agent. (Accessed on 01/28/2024).

[23] Security of end-to-end encrypted backups, 2021.

[24] The labyrinth encrypted message storage protocol, 2023.

[25] Gunes Acar, Marc Juarez, Nick Nikiforakis, Claudia Diaz, Seda Gürses, Frank Piessens, and Bart Preneel. Fpdetective: dusting the web for fingerprinters. In *CCS*, 2013.

[26] Joël Alwen, Sandro Coretti, and Yevgeniy Dodis. The double ratchet: security notions, proofs, and modularization for the signal protocol. In *EUROCRYPT*, 2019.

[27] R Barnes, B Beurdouche, R Robert, J Millican, E Omara, and K Cohn-Gordon. RFC 9420: The Messaging Layer Security (MLS) protocol, 2023.

[28] Mihir Bellare, Alexandra Boldyreva, Anand Desai, and David Pointcheval. Key-privacy in public-key encryption. In *ASIACRYPT*, 2001.

[29] Mihir Bellare and Viet Tung Hoang. Efficient schemes for committing authenticated encryption. In *EUROCRYPT*, 2022.

[30] Mihir Bellare and Bennet Yee. Forward Integrity for Secure Audit Logs. Technical report, 1997. Preprint.

[31] Mihir Bellare and Bennet Yee. Forward-security in Private-key Cryptography. In *Topics in Cryptology—CT-RSA*, 2003.

[32] Rosanna Bellini, Emily Tseng, Noel Warford, Alaa Daffalla, Tara Matthews, Sunny Consolvo, Jill Palzkill Woelfer, Patrick Gage Kelley, Michelle L Mazurek, Dana Cuomo, et al. Sok: Safer digital-safety research involving at-risk users. In *IEEE Security & Privacy*, 2024.

[33] Josh Blum, Simon Booth, Brian Chen, Oded Gal, Maxwell Krohn, Julia Len, Karan Lyons, Antonio Marcedone, Mike Maxim, Merry Ember Mou, et al. Zoom cryptography whitepaper. 2022.

[34] Nikita Borisov, Ian Goldberg, and Eric Brewer. Off-the-record communication, or, why not to use PGP. In *Proceedings of the 2004 ACM workshop on Privacy in the electronic society*, pages 77–84, 2004.

[35] Justin Brookman, Phoebe Rouge, Aaron Alva, and Christina Yeung. Cross-Device Tracking: Measurement and Disclosures. *PoPETs*, 2017.

[36] Ran Canetti, Palak Jain, Marika Swanberg, and Mayank Varia. Universally composable end-to-end secure messaging. In *CRYPTO*, 2022.

[37] Cheun Ngen Chong, Zhonghong Peng, and Pieter H Hartel. Secure audit logging with tamper-resistant hardware. In *IFIP SEC*, 2003.

[38] Katriel Cohn-Gordon, Cas Cremers, Benjamin Dowling, Luke Garratt, and Douglas Stebila. A formal security analysis of the signal messaging protocol. *Journal of Cryptology*, 33:1914–1983, 2020.

[39] Alaa Daffalla, Marina Bohuk, Nicola Dell, Rosanna Bellini, and Thomas Ristenpart. Account security interfaces: important, unintuitive, and untrustworthy. In *USENIX Security*, 2023.

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

[41] Emma Dauterman, Danny Lin, Henry Corrigan-Gibbs, and David Mazières. Accountable authentication with privacy protection: The larch system for universal login. In *OSDI*, 2023.

[42] Peter Eckersley. How Unique is Your Web Browser? In *PETS*, 2010.

[43] Manuel Egele, Christopher Kruegel, Engin Kirda, and Giovanni Vigna. Pios: Detecting privacy leaks in ios applications. In *NDSS*, 2011.

[44] Robert E Endeley. End-to-end encryption in messaging services and national security—case of whatsapp messenger. *Journal of Information Security*, 9(1):95–99, 2017.

[45] Diana Freed, Jackeline Palmer, Diana Minchala, Karen Levy, Thomas Ristenpart, and Nicola Dell. "A Stalker's Paradise" How Intimate Partner Abusers Exploit Technology. In *CHI*, 2018.

[46] Christina Garman, Matthew Green, Gabriel Kaptchuk, Ian Miers, and Michael Rushanan. Dancing on the lip of the volcano: Chosen ciphertext attacks on apple {iMessage}. In *USENIX Security*, 2016.

[47] Michael C. Grace, Wu Zhou, Xuxian Jiang, and Ahmad-Reza Sadeghi. Unsafe exposure analysis of mobile in-app advertisements. In *ACM WISEC*, New York, NY, USA, 2012.

[48] Sven Hammann, Michael Crabb, Sasa Radomirovic, Ralf Sasse, and David Basin. I'm surprised so much is connected. In *CHI*, 2022.

[49] Sven Hammann, Saša Radomirović, Ralf Sasse, and David Basin. User account access graphs. In *CCS*, 2019.

[50] Carmit Hazay and Yehuda Lindell. *Efficient secure two-party protocols: Techniques and constructions*. Springer Science & Business Media, 2010.

[51] Jason E Holt and Kent E Seamons. Logcrypt: forward security and public verification for secure audit logs. *Cryptol. ePrint Archive*, 2005.

[52] Thomas Hupperich, Davide Maiorca, Marc Kührer, Thorsten Holz, and Giorgio Giacinto. On the Robustness of Mobile Device Fingerprinting: Can Mobile Users Escape Modern Web-Tracking Mechanisms? In *ACSAC*, 2015.

[53] Vishal Karande, Erick Bauman, Zhiqiang Lin, and Latifur Khan. Sgx-log: Securing system logs with sgx. In *Asia CCS*, 2017.

[54] Andreas Kurtz, Hugo Gascon, Tobias Becker, Konrad Rieck, and Felix Freiling. Fingerprinting mobile devices using personalized configurations. *PoPETs*, 2016.

[55] Andreas Kurtz, Andreas Weinlein, Christoph Settgast, and Felix Freiling. Dios: Dynamic privacy analysis of ios applications. 2014.

[56] Adam Lerner, Anna Kornfeld Simpson, Tadayoshi Kohno, and Franziska Roesner. Internet Jones and the Raiders of the Lost Trackers: An Archaeological Study of Web Tracking from 1996 to 2016. In *USENIX Security*, 2016.

[57] Ian Levy and Crispin Robinson. Principles for a more informed exceptional access debate. *The Lawfare Institute*, 2018.

[58] Di Ma and Gene Tsudik. A New Approach to Secure Logging. *ACM Transactions on Storage*, 2009.

[59] Philipp Markert, Leona Lassak, Maximilian Golla, and Markus Dürmuth. Understanding users' interaction with login notifications. In *CHI*, 2024.

[60] Tara Matthews, Kathleen O'Leary, Anna Turner, Manya Sleeper, Jill Palzkill Woelfer, Martin Shelton, Cori Manthorne, Elizabeth F Churchill, and Sunny Consolvo. Stories From Survivors: Privacy & Security Practices When Coping With Intimate Partner Abuse. In *CHI*, 2017.

[61] Nick Nikiforakis, Alexandros Kapravelos, Wouter Joosen, Christopher Kruegel, Frank Piessens, and Giovanni Vigna. Cookieless monster: Exploring the ecosystem of web-based device fingerprinting. In *IEEE Security & Privacy*, 2013.

[62] United States. Department of Defense. *Department of Defense Trusted Computer System Evaluation Criteria*. Department of Defense, 1987.

[63] Yasuhiro Ohtaki. Partial disclosure of searchable encrypted data with support for boolean queries. In *ARES*. IEEE, 2008.

[64] Eric Rescorla. The Transport Layer Security (TLS) Protocol Version 1.3. RFC 8446, 2018.

[65] Bruce Schneier and John Kelsey. Cryptographic support for secure logs on untrusted machines. In *USENIX Security Symposium*, 1998.

[66] Adi Shamir. How to share a secret. *Commun. ACM*, 1979.

[67] Marianne Swanson and Barbara Guttman. *Generally Accepted Principles and Practices for Securing Information Technology Systems*. NIST, 1996.

[68] Stephan van Schaik, Alex Seto, Thomas Yurek, Adam Batori, Bader AlBassam, Christina Garman, Daniel Genkin, Andrew Miller, Eyal Ronen, and Yuval Yarom. SoK: SGX.Fail: How stuff get eXposed. 2022.

[69] Noel Warford, Tara Matthews, Kaitlyn Yang, Omer Akgul, Sunny Consolvo, Patrick Gage Kelley, Nathan Malkin, Michelle L Mazurek, Manya Sleeper, and Kurt Thomas. Sok: A framework for Unifying At-risk User Research. In *IEEE Security & Privacy*, 2022.

[70] Brent R Waters, Dirk Balfanz, Glenn Durfee, and Diana K Smetters. Building an encrypted and searchable audit log. In *NDSS*, 2004.

[71] Ting-Fang Yen, Yinglian Xie, Fang Yu, Roger Peng Yu, and Martin Abadi. Host fingerprinting and tracking on the web: Privacy and security implications. In *NDSS*, 2012.

A Further Diagrams

Due to space constraints, we include three flow diagrams here in the appendix: one for action logging, one for re-encryption, and one for log retrieval (also referred to as obtaining the history). See Figure 4, Figure 5, and Figure 6.

B Algorithms Pseudocode

Here we include the pseudocode for the remaining CSAL algorithms based on our protocol from Section 4. Figure 7 includes the pseudocode for the CSAL action algorithm $\mathcal{N}$. The client state is a tuple containing a client table T_{CL} and the encryptor keys. The RP state is a tuple of tables. The RP generates a challenge and then the client encrypts a new DEM using this challenge. Finally, the RP adds the DEM to T_{LOG}, which updates the RP state. It returns the client and RP states.

Next, Figure 8 shows the pseudocode for the CSAL history algorithm $\mathcal{H}$. The algorithm first goes over all the KEMs encrypted to some session in the current device, decrypts them and aggregates all the keys into a set $\mathcal{K}$. Next, it iterates over T_{OLD} in reverse, and recovers the old symmetric keys

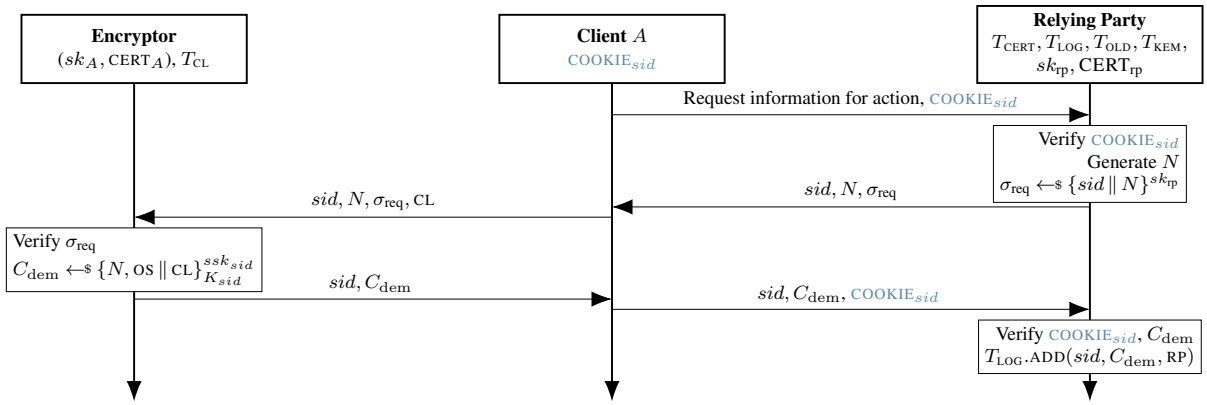

Figure 4: CSAL protocol Ψ action log flow ($\mathcal{N}$). If any RP verification fails, it would delete relevant COOKIE_{sid} and return $\perp$.

it can using the keys in $\mathcal{K}$. At the same time, it adds each recovered key to $\mathcal{K}$. Finally, it goes over T_{LOG} to reconstruct the activity log $\mathcal{T}$. For each entry on the table, if the key to decrypt it is in $\mathcal{K}$, it creates an entry OS, CL, RP, else the entry only contains RP. It adds each entry to $\mathcal{T}$ and finally returns $\mathcal{T}$ and the unmodified client state.

Finally, the algorithm for the CSAL re-encryption $\mathcal{R}$ is shown in Figure 9. The RP again generates a challenge N. The client recovers the KEMs it can decrypt and re-encrypts each one to any other session that did not yet have access to such KEM, creating new KEMs. The new KEMs are stored in T_{KEM}. The client state doesn't change, but the new T_{KEM} updates the RP state. The algorithm returns the two states.

C Unlinkability Formalization

We show a pseudocode game capturing unlinkability in Figure 10. The purpose of the unlinkability game is to express the idea that the server should not be able to tell whether two different sessions come from the same or from different devices. At the start, the challenger selects a random bit b, initializes the server state st_{rp}. Then, it runs the distinguisher $\mathcal{D}$, who has access to st_{rp} and to the oracles for client initialization, client login, client action, client re-encryption, and challenge. For the challenge oracle, $\mathcal{D}$ provides information for two clients. The challenge then runs a login on one of the two clients at random (based on b). The adversary can observe the sid and the final server state output across all the oracles. With this information, $\mathcal{D}$ wins if it guesses the bit b.

We define the UNLINK_Π advantage of an adversary $\mathcal{D}$ as the probability that it outputs 1, namely:

$$\mathsf{Adv}^{\text{unlink}}_{\Pi}(\mathcal{D}) = 2 \cdot \Pr\left[\text{UNLINK}_\Pi(\mathcal{D}) \Rightarrow 1\right] - 1.$$

Intuitively, our CSAL scheme inherits the unlinkability of the underlying attestation mechanisms because our ciphertexts' length are independent of the previous use of that client device and the protocol generates fresh public keys and session identifiers independently of the client device. Specifically, we don't need our underlying PKE to be anonymous (in the sense of [28]). Hence, assuming CERT_A does not reveal information that allows device fingerprinting, our scheme prevents linking two sessions to the same device, As discussed in Section 4, we rely on anonymity of FIDO2 attestations, which they achieve via vendor-defined anonymity sets.

Theorem C.1 *Let Π be our CSAL scheme with sid bit length n and built using PKE scheme* PKE *and AEAD* AE. *Let $\mathcal{D}$ be an* UNLINK_Π*-adversary that makes at most q queries to its oracles. Then we give an* $\text{MU-IND-CPA}_{\mathsf{PKE}}$*-adversary $\mathcal{D}_{pke}$ and* $\text{IND-CPA}_{\mathsf{AE}}$*-adversary $\mathcal{D}_{ae}$ such that*

$$\begin{aligned}\mathsf{Adv}^{\text{unlink}}_{\Pi}(\mathcal{D}) \leq & 2\mathsf{Adv}^{\text{mu-ind-cpa}}_{\mathsf{PKE}}(\mathcal{D}_{pke}) \\ & + 2q \cdot \mathsf{Adv}^{\text{ind-cpa}}_{\mathsf{AE}}(\mathcal{D}_{ae}) + q^2/2^{n-1}\ .\end{aligned}$$

Adversary $\mathcal{D}_{pke}$ makes at most $q^2 + q$ oracle queries, while $\mathcal{D}_{ae}$ makes at most $q+1$ oracle queries. Their run times are at most $\mathcal{O}(q^3 + q \cdot T^{\mathsf{AE}}_{Enc}(\ell_1) + q \cdot T^{\mathsf{AE}}_{Enc}(q\kappa))$ and $\mathcal{O}(q^3 \cdot T^{\mathsf{PKE}}_{Enc}(\kappa))$.

At first glance, the UNLINK theorem and the game look similar to those of PRIV, however there are some subtle differences. First, throughout the game, the different client states are only indexed by the sid, not the client. This is to allow the adversary to query the oracles on the responses from the challenge oracle; however, it does not know the client from such session. Hence, we can only query by sid.

Moreover, in PRIV, we require that the adversarially generated inputs OS_0, CL_0 and OS_1, CL_1 have the same length. This means that the output ciphertext C_{dem} would have the same length, while the size of the other plaintexts is independent of the inputs. However, here, what gets encrypted can depend on the device selected. For instance, a device c with two sessions would have more smuggling keys than two sessions on different devices. This is why calling the pad function in $\Pi.\mathcal{L}$ is important.

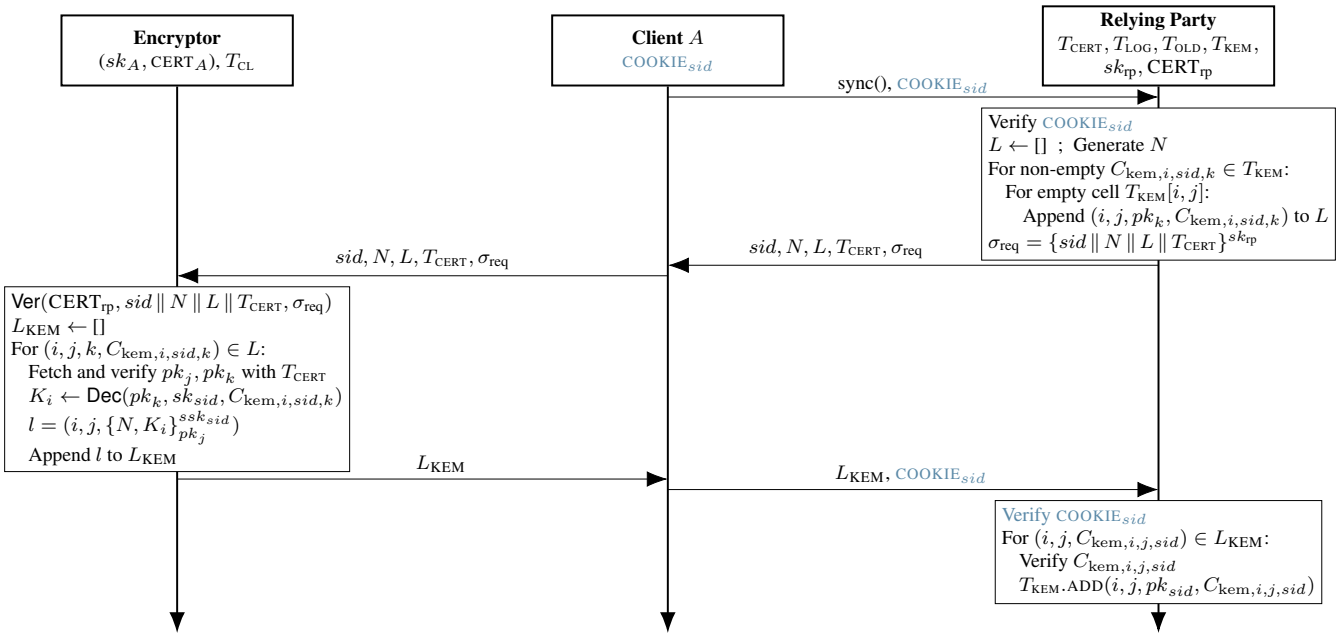

Figure 5: CSAL protocol Ψ re-encryption flows ($\mathcal{R}$), in which a device can pass along to other sessions some additional session secret keys to which the device has access.

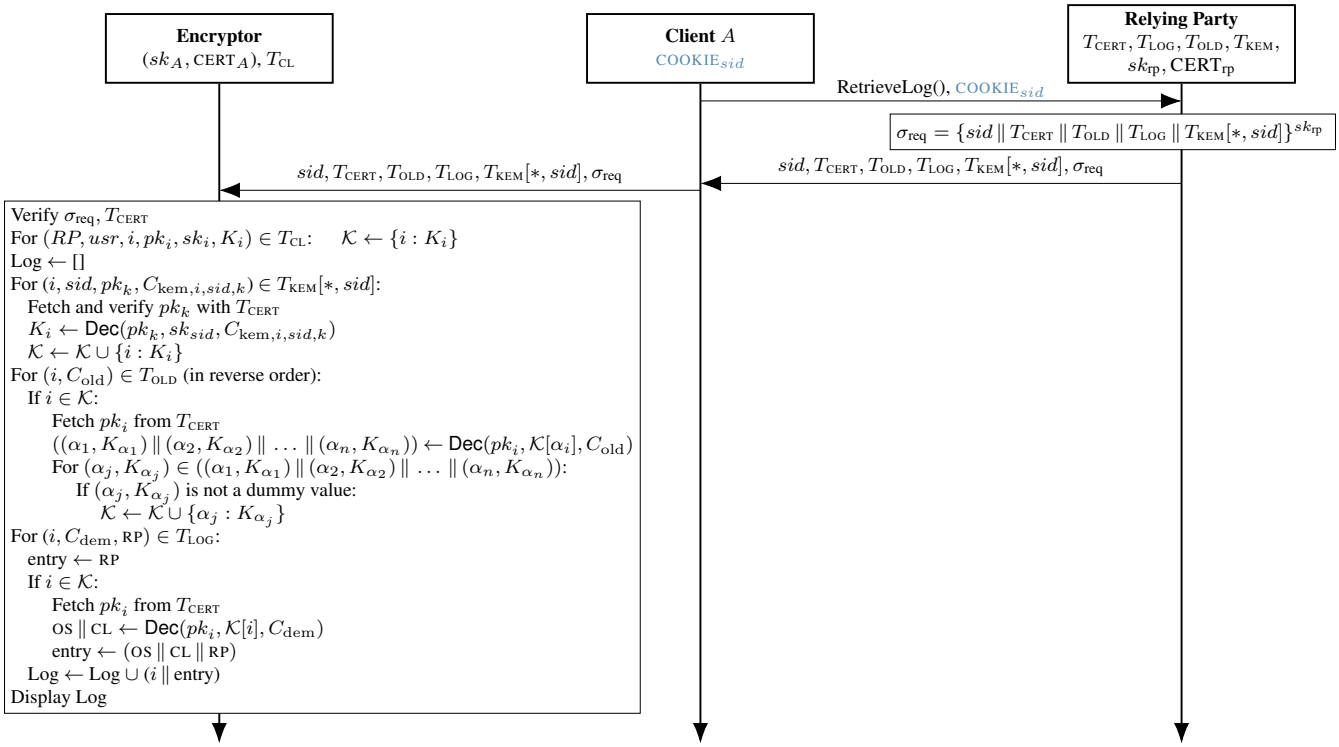

Figure 6: CSAL protocol Ψ log retrieval flow ($\mathcal{H}$).

$\underline{\mathcal{N}((st_{\text{clt}}, sid, \text{OS}, \text{CL}), (st_{\text{rp}}, sid', \text{RP})) :}$

If $sid \neq sid'$ return $\perp$

$(T_{\text{CERT}}, T_{\text{LOG}}, T_{\text{OLD}}, T_{\text{KEM}}) \leftarrow st_{\text{rp}}$ // RP challenge generation
$N \leftarrow\!\!\$\ \{0,1\}^n$

$(sk_A, \text{CERT}_A, T_{\text{CL}}) \leftarrow st_{\text{clt}}$ // Client response
$(pk_{sid}, sk_{sid}, K_{sid}) \leftarrow st_{\text{clt}}[sid]$
$C_{\text{dem}} \leftarrow\!\!\$\ \{N, \text{OS} \parallel \text{CL}\}^{ssk_{sid}}_{K_{sid}}$
$st_{\text{clt}} \leftarrow (sk_A, \text{CERT}_A, T_{\text{CL}})$

Add $(sid, C_{\text{dem}}, \text{RP})$ to T_{LOG} // RP receiving response
$st_{\text{rp}} \leftarrow (T_{\text{CERT}}, T_{\text{LOG}}, T_{\text{OLD}}, T_{\text{KEM}})$

Return $(st_{\text{clt}}, st_{\text{rp}})$

Figure 7: Client-server action for our protocol Ψ as a single algorithm $\mathcal{N}$.

$\underline{\mathcal{H}((st_{\text{clt}}, sid), (st_{\text{rp}}, sid')) :}$

If $sid \neq sid'$ return $\perp$

$(T_{\text{CERT}}, T_{\text{LOG}}, T_{\text{OLD}}, T_{\text{KEM}}) \leftarrow st_{\text{rp}}$ // RP

$(sk_A, \text{CERT}_A, T_{\text{CL}}) \leftarrow st_{\text{clt}}$ // Client response
$(pk_{sid}, sk_{sid}, K_{sid}) \leftarrow st_{\text{clt}}[sid]$
For (i, pk_i, sk_i, K_i) in T_{CL}:
  $\mathcal{K} \leftarrow \mathcal{K} \cup \{i : K_i\}$
For $(i, sid, pk_k, C_{\text{kem},i,sid,k})$ in $T_{\text{KEM}}[*, sid]$:
  $K_i \leftarrow \mathsf{Dec}(pk_k, sk_{sid}, C_{\text{kem},i,sid,k})$
  $\mathcal{K} \leftarrow \mathcal{K} \cup \{i : K_i\}$
For (i, C_{old}) in T_{OLD}.reversed():
  If $\{i : K_i\}$ in $\mathcal{K}$:
    $(i, N_i, pk_i, \text{CERT}_i) \leftarrow T_{\text{CERT}}[i]$
    $((\alpha_1, K_{\alpha_1}) \parallel \ldots \parallel (\alpha_n, K_{\alpha_n})) \leftarrow \mathsf{Dec}(pk_i, K_i, C_{\text{old}})$
    For (j, K_j) in $((\alpha_1, K_{\alpha_1}) \parallel \ldots \parallel (\alpha_n, K_{\alpha_n}))$:
      If $K_j \neq 0^n$:
        $\mathcal{K} \leftarrow \mathcal{K} \cup \{j : K_j\}$
For $(i, C_{\text{dem}}, \text{RP})$ in T_{LOG}:
  entry $\leftarrow$ RP
  If $\{i : K_i\}$ in $\mathcal{K}$:
    $(i, N_i, pk_i, \text{CERT}_i) \leftarrow T_{\text{CERT}}[i]$
    OS $\parallel$ CL $\leftarrow \mathsf{Dec}(pk_i, K_i, C_{\text{dem}})$
    entry $\leftarrow$ (OS $\parallel$ CL $\parallel$ RP)
  $\mathcal{T} \leftarrow \mathcal{T} \cup (i \parallel \text{ entry})$
Return $(\mathcal{T}, st_{\text{clt}})$

Figure 8: Client-server history recovery for protocol Ψ as a single algorithm $\mathcal{H}$.

$\underline{\mathcal{R}((st_{\text{clt}}, sid), (st_{\text{rp}}, sid')) :}$

If $sid \neq sid'$ return $\perp$

$(T_{\text{CERT}}, T_{\text{LOG}}, T_{\text{OLD}}, T_{\text{KEM}}) \leftarrow st_{\text{rp}}$ // RP challenge generation
For row in T_{CERT}:
  $(i, N_i, pk_i, \text{CERT}_i) \leftarrow$ row
  $\mathcal{S} \leftarrow \mathcal{S} \cup i$
$N \leftarrow\!\!\$\ \{0,1\}^n$

$(sk_A, \text{CERT}_A, T_{\text{CL}}) \leftarrow st_{\text{clt}}$ // Client response
$(pk_{sid}, sk_{sid}, K_{sid}) \leftarrow st_{\text{clt}}[sid]$
For $(*, sid, *, C_{\text{kem},i,sid,k})$ in T_{KEM}:
  $K_i \leftarrow \mathsf{Dec}(pk_k, sk_{sid}, C_{\text{kem},i,sid,k})$
  For j in $\mathcal{S}$:
    If $(i, j, *, *)$ not in T_{KEM}:
      $C_{\text{kem},i,j,sid} \leftarrow\!\!\$\ \{N, K_i\}^{ssk_{sid}}_{pk_k}$
      $L_{\text{KEM}} \leftarrow L_{\text{KEM}} \cup (i, j, pk_{sid}, C_{\text{kem},i,j,sid})$
$st_{\text{clt}} \leftarrow (sk_A, \text{CERT}_A, T_{\text{CL}})$

For row in L_{KEM}: Add row to T_{KEM} // RP receiving response
$st_{\text{rp}} \leftarrow (T_{\text{CERT}}, T_{\text{LOG}}, T_{\text{OLD}}, T_{\text{KEM}})$
Return $(st_{\text{clt}}, st_{\text{rp}})$

Figure 9: Client-server re-encryption for our protocol Ψ as a single algorithm $\mathcal{R}$

$\underline{\text{UNLINK}_{\Pi}(\mathcal{D})}$
$b \leftarrow\!\!\$\ \{0,1\}$
$st_{\text{rp}} \leftarrow\!\!\$\ \mathcal{I}_s$
$b' \leftarrow\!\!\$\ \mathcal{D}^{\mathcal{O}}(st_{\text{rp}})$
Return $(b' = b)$

$\underline{\text{InitClient}}$
$\mu \leftarrow \mu + 1$
$st_{\mu} \leftarrow\!\!\$\ \mathcal{I}_c$

$\underline{\text{Login}(c, \text{OS}, \text{CL}, \text{RP})}$
If $c > \mu$ then Return $\perp$
$(sid, st_c, st_{\text{rp}}) \leftarrow\!\!\$\ \mathcal{L}((st_c, \text{OS}, \text{CL}), (st_{\text{rp}}, \text{RP}))$
$\gamma[sid] \leftarrow c$
Return sid, st_{rp}

$\underline{\text{Action}(sid, \text{OS}, \text{CL}, \text{RP})}$
If $\gamma[sid] = \perp$ then Return $\perp$
$c \leftarrow \gamma[sid]$
$(st_c, st_{\text{rp}}) \leftarrow\!\!\$\ \mathcal{N}((st_c, sid, \text{OS}, \text{CL}), (st_{\text{rp}}, sid, \text{RP}))$
Return st_{rp}

$\underline{\text{Re-encryption}(sid)}$
If $\gamma[sid] = \perp$ then Return $\perp$
$c \leftarrow \gamma[sid]$
$(st_c, st_{\text{rp}}) \leftarrow\!\!\$\ \mathcal{R}((st_c, sid), (st_{\text{rp}}, sid))$
Return st_{rp}

$\underline{\text{LoginChallenge}(c_0, c_1, \text{OS}, \text{CL}, \text{RP})}$
If $c_0 > \mu$ or $c_1 > \mu$ then Return $\perp$
$(sid, st_{c_b}, st_{\text{rp}}) \leftarrow\!\!\$\ \mathcal{L}((st_{c_b}, \text{OS}, \text{CL}), (st_{\text{rp}}, \text{RP}))$
$\gamma[sid] \leftarrow c_b$
Return (sid, st_{rp})

Figure 10: Logging server unlinkability (UNLINK) game for a CSAL scheme Π and adversary $\mathcal{D}$. Here $\mathcal{O} = (\text{InitClient}, \text{Login}, \text{Action}, \text{Re-encryption}, \text{LoginChallenge})$.