Acsesor: A New Framework for Auditable Custodial Secret Storage and Recovery

Melissa Chase[1], Hannah Davis[2], Esha Ghosh[1], and Kim Laine[1]

[1]Microsoft Research
[2]Seagate*
{*melissac,esha.ghosh,kim.laine*}*@microsoft.com* *hannah.e.davis@seagate.com*

Abstract

Custodial secret management services provide a convenient centralized user experience, portability, and emergency recovery for users who cannot reliably remember or store their own credentials and cryptographic keys. Unfortunately, these benefits are only available when users compromise the security of their secrets and entrust them to a third party. This makes custodial secret management service providers ripe targets for exploitation, and exposes valuable and sensitive data to data leaks, insider attacks, and password cracking, etc. In non-custodial solutions (utilized by some password managers and cryptocurrency wallets), the users are in charge of a high-entropy secret, such as a cryptographic secret key or a long passphrase, that controls access to their data. While these solutions have a stronger security model, the obvious downside here is the usability: it is very difficult for people to store cryptographic secrets reliably.

We present Acsesor: a new framework for auditable custodial secret management with decentralized trust. Our framework offers a middle-ground between a fully custodial and fully non-custodial recovery system: it enhances custodial recovery systems with cryptographically assured access monitoring and a distributed trust assumption. In particular, the Acsesor framework distributes the recovery process across a set of (user-chosen) guardians. However, the user is never required to interact directly with the guardians during recovery, which allows us to retain the high usability of centralized custodial solutions. By allowing the guardians to implement flexible user-chosen response policies, Acsesor can address a broad range of problem scenarios in classical secret management solutions. Finally, we also instantiate the Acsesor framework with a base protocol built of standard primitives: standard encryption schemes, commitment schemes, and privacy-preserving transparency ledgers.

1 Introduction

The problem of secret management is a fundamental one. When a user wishes to store a secret for later use, they generally have three options: they can remember it themselves, write it down or store it on a local device, or entrust it to a third-party. Secret management solutions where the user is responsible for storing and managing their secret are called *non-custodial* (or *self-custodial*), whereas solutions where a third-party service manages the secret on the user's behalf are called *custodial*.

Two conflicting feature requirements for any data management service are colloquially known as the "hammer" and "toilet" tests: if you want to prevent anyone from accessing your data, can you do so by intentionally physically destroying your personal devices? In contrast, if you accidentally de-

*work done while at Microsoft Research

stroy the same devices, are your digital assets recoverable? It is clearly impossible to simultaneously pass both tests. After all, any data that can be recovered post-catastrophe by an honest and helpful custodian can also be recovered by a malicious or compromised custodian. Worse, when servers are compromised or their data leaked, users may be unaware that their credentials have been exposed.

Remembering high-entropy secrets, such as strong passwords or passphrases, is very challenging for people. Writing down such secrets may be an option, but for high-value secrets may not be reliable or available enough. Papers and storage devices may go missing or get stolen, people travelling may be unable to find any safe place to store their valuable secrets, and medical conditions or simply advanced age may make remembering secrets or their locations impractical. Nevertheless, credentials with immense personal value [1, 28] and vast amounts of wealth in cryptocurrency wallets [10, 35, 43, 13, 14] are protected by self-managed passwords, passphrases, and keys, and often consequently lost. When required to remember secrets, people mostly resort to using weak passwords and reusing the same password across multiple services. Indeed, in 2019, a survey done by the security company Avast found that 83% of Americans are using weak passwords [4]. In the same year, a survey commissioned by Google found that 52% of Americans reuse passwords for multiple accounts, and 13% reuse the same password for all accounts [22]. A study in Behavior and Information Technology [40] identified the convenience-security tradeoff as the primary motivator of weak password choices.

Many services today offer custodial secret management (*e.g.*, password managers, custodial wallets, secure cloud storage). On the upside, these custodial systems have full control over the user's account and can help their users with password reset, an array of authentication methods, and detecting suspicious access attempts to the account. On the downside, the users have to place immense trust on these custodians. Even if such custodians have no malicious intent, they may be compelled to provide access to law enforcement under a subpoena, or unwittingly to hackers in case of a security breach. Thus, custodians end up becoming valuable targets and single points of failure for security. For example, recently, in a security breach in LastPass, an unauthorized party was able to gain access to some of their customers' information [29]. For some services, such as cryptocurrency wallets, custodial secret management is also problematic due to the liability risks and anti-money laundering regulations that financial servers need to comply with.

1.1 Acsesor

In this paper, we construct Acsesor: a general framework for auditable custodial secret management with decentralized trust. Our framework offers a middle-ground between a fully custodial (centralized) and fully non-custodial (user-managed/distributed) recovery system. It enhances custodial recovery systems with cryptographically assured access monitoring and a distributed trust assumption. This allows Acsesor to support the usability, availability and flexibility of a custodial system, with the greater resilience against attacks that a non-custodial system can provide.

The Acsesor approach. At a high level, the Acsesor approach is as follows: the user maintains an account at a *server (or server)*, which provides a single point of access. If a user wants to store a secret, they choose a set of *guardians* among whom to distribute trust. For example, users could select a combination of third-party guardians and their own trusted devices. The user also chooses a policy stating the conditions under which the guardians should aid in secret recovery, for example, a delay period, or a required second factor. We discuss guardians in Section 2.3 and various policies in Section 5. They use this information to encode their secret, and store the resulting blob with the server. When the user wants to recover their secret, they contact the server with their request; the server authenticates the user and then passes the request to the appropriate guardians who aid in the recovery.

To protect against a server that tries to initiate a recovery attempt without the user's knowledge or an attacker that compromises the server's authentication, Acsesor requires the server to log all recovery requests. The guardians are responsible for ensuring that request they are responding to is logged before responding to it.

Acsesor asks its users (client devices) to periodically monitor their accounts (using their own devices) to detect any fraudulent recovery requests. If the user is logged in on their device, the monitoring is run by that device automatically; any requests not originating on that device should result in a notification to the user. Thus, the user is only alerted when some suspicious activity is detected, providing a similar interface to the already common "Did you log in from a new device?" alerts.

This approach is inspired by the recent advances in logging based transparency technology (*e.g.*, key transparency, binary transparency, software transparency, credential transparency) in the industry [5, 21, 32, 18, 16], government [45], standardization bodies [25] and academic literature [34, 7, 41, 42, 8, 31, 32]. The philosophy that underpins much of the growing trust in transparency systems, is that a service with transparency guarantees is more accountable. This enforces better behaviour from the servers, and increased public trust. Note, also that, it obviously improves over the current custodial solutions where in most cases, a compromise is not detected until the server discloses it.

The ledger. The construction above requires some way for the guardians and users to ensure that they are seeing the same view, so that if the guardians check that a request is logged they can be sure that that request will still be present later when the user audits the log. This inherently requires some additional root of trust; a blockchain, a trusted party, trusted hardware, etc. In our construction we will abstract this out as a ledger functionality, which allows the server to record requests and then present the request history for a given user and convince the recipient that it is complete. We also summarize two instantiations based on prior work: one based on a trusted bulletin board and one based on trusted hardware. This allows us to focus on the aspects of our protocol that are new and specific to the problem of secret storage.

Construction overview. The Acsesor framework consists of a lightweight base protocol with three phases: registration, secret recovery, and monitoring. The protocol delegates authentication entirely to the server, so that it may be compatible with any existing identity systems.

The protocol proceeds roughly as follows, omitting some subtleties which we will discuss in Section 3.1. In the registration phase, the user registers with the Acsesor server. They choose a set of guardian nodes to share responsibility for securing the secret. Next, the user chooses a one-time cryptographic key, encrypts their secret under it, and secret-shares the key; each guardian will have access to one share. Each share is encrypted under the corresponding guardian's public key and sent to the server for storage, along with the encrypted secret. The server logs the blob it stores, on the ledger. The user verifies that the blob was correctly logged.

To recover their secrets, the user authenticates with the server and requests to initiate a recovery, at which time the server posts a receipt of the request in the ledger, along with a desired one-time public key the user has chosen. When the guardians see this request, they first ensure that it has been logged, then download the encrypted key pieces from the server, decrypt them, re-encrypt under the user's posted public key, and route them back to the user through the server. The user downloads the encrypted secret from the server, as well as the encrypted shares of their encryption key, reconstructs the key, and decrypts the secret. They also check that the posted public key is correct (*i.e.*, the same as the one they locally generated). The user will also continuously monitor their account on the ledger to detect potential fraudulent recovery attempts done on their behalf.

Acsesor policies. The most basic application of Acsesor would have no policy, so the guardians would immediately reply to any request forwarded to them by the server, and a corrupt server could easily learn the user's secret. Note that this would still be an improvement on traditional custodial systems in that it would provide transparency - the user would be able to detect when any access had occurred.

However, in many cases we would like to provide stronger guarantees. In those cases it make sense for the user to set a policy controlling when the guardians should process a request. What policy the user should choose depends on security/availability/latency needs of the application and the user's preferences. For example, to provide higher security, the policy might require that the user present a second factor as part of the recovery request; the guardian would verify this second factor and only then release the secret. This could be a strong factor like a hardware token, or something weaker like a PIN/password that the user would remember; in the latter case, we could additionally have the guardians enforce rate limiting. Alternatively, the policy could require out-of-band confirmation, where the guardian would send a message to the user through an out of band channel specified in the policy (e.g. email address or SMS number) and require that the user respond over the same channel to confirm the request.

On the other hand, if we want to guarantee that the secret remains available (assuming the server is trusted for availability, mode discussion on this in Section Section 2), the policy might specify that the guardians wait a certain amount of time from the time the message was posted on the log to when they should process the recovery, thus allowing the user time to respond before their secret is compromised. Of course, in some cases there isn't any way for the user to respond to compromise even given plenty of warning; in those cases the policy should require a reasonable second factor on top of the wait time.

The two strategies discussed above could be combined in to one policy as well to allow quick recovery in most cases, but slower recovery when the user loses their primary second factor. This flexibility should allow Acsesor to be tailored for a variety of applications; we discuss appropriate policies for applications to password management and cryptocurrency wallets in Section 6.

In some cases, the user may not want to ever actually recover their secret, but simply have the guardians use it for something, such as creating a digital signature. This is common functionality with some cryptocurrency wallets, where the wallet provider's servers, and possibly the user's device, hold signature key shares that are used in a threshold signature protocol. Acsesor's guardians can enact similar threshold signing policies, as long as they do not require the guardians to interact with each other. See Section 5 for more discussion.

1.2 Contributions

We summarize our main contributions below.

Acsesor framework. Our first contribution is the introduction of the new framework (Acsesor) that combines the usability, availability and flexibility of a custodial system, with the distributed trust of a non-custodial system. To the best of our knowledge, Acsesor is the first system that combines these properties with strong auditability guarantees. The benefits of this design are the following: *Usability:* Acsesor retains the usability of custodial fully centralized systems, as the user only interacts with the server. The server can allow authentication via weak secrets like passwords or pins, and allow account recovery via SMS or a live customer service call. It can also utilize state-of-the-art fraud detection mechanisms to determine when to allow access and when to require additional authentication.
Flexibility: Because we provide the user flexibility in choosing the policy, we support a variety of different applications depending on the security, availability, and latency requirements of the application and preferences of the user.
Security: We obtain significantly stronger guarantees than custodial systems, in particular for the case where the server is corrupt.

Formal definition. Defining the security and privacy properties of Acsesor is the most involved and interesting technical aspect of this work. We define 6 security properties (each of them is formalized as a security game). The security and privacy guarantees that Acsesor provides, gracefully degrades depending on the corruption model. In our threat modeling, we assume that there are always some honest and some malicious users in the system. The properties are the following.
User privacy: Malicious users using the system

should not be able to learn any information about the honest users (including, when they make recovery requests).
Confidentiality: Malicious parties should not be able to get an honest user's secret.
Policy enforcement: Malicious parties cannot learn an honest user's secret unless there is an adversarial request for which the user's policy has been satisfied.
Transparency: The adversary cannot learn an honest user's secret without logging at least one recovery access beyond those corresponding to the honest user's recovery requests. In other words, if the adversary learns the secret, it will be visible to the honest user when they audit their history.
History consistency: If the user verifies his recovery history multiple times, the earlier recovery context lists will always appear as prefixes of the later context lists, even if the user has lost state between the two checks.
Robustness: The user cannot be convinced that a recovery has been successful unless it has correctly produced the user's original stored secret.

Acsesor achieves all the security properties listed above if the server and both guardians are honest. In the case where all the parties (the server and both guardians) are potentially malicious, Acsesor achieves *History Consistency and Robustness.* This might seem surprising at first. But Acsesor is able to achieve these properties even in such a strong corruption setting because it uses the ledger functionality.

The security gradually degrades when some of the parties are malicious. Acsesor achieves *Confidentiality* if at least the server is honest – one or both of the guardians can be compromised. If the server is malicious, but at least one of the guardians is honest (the other guardian could be compromised), Acsesor inherently cannot achieve *Confidentiality* because the server can make its own recovery requests, but it still guarantees *Policy Enforcement and Transparency.* We give a high-level intuition for the security games we define to capture each property in Section Section 2.2 and defer the formal definitions to the Appendix.

Protocol. We provide a concrete instantiation of the Acsesor framework and rigorously prove that it satisfies the security and privacy definitions discussed above. Our construction is built from standard cryptographic building blocks: a Public Key Encryption (PKE) scheme, a symmetric encryption scheme, and a cryptographic commitment scheme. In addition to this, our construction uses an ideal Ledger functionality (that we define). We discuss two different instantiations of the Ledger functionality based on existing works: OPTIKS PAHD [31] and Nimble [2].

Performance. In Table 1 we report performance numbers of Acsesor, both for an OPTIKS PAHD-based instantiation, as well as the Nimble-based instantiation, showing that both instatiations are practical. Generally, the Nimble-based instantiation is much faster, but both can serve thousands of user monitoring requests per second. This is necessary, as monitoring is the most commonly occurring event. The number of recovery requests these instantiations support in the hundreds or thousands per second, depending quite strongly on the number of users for the OPTIKS PAHD-based instantiation. We believe this to be easily sufficient, as recovery is expected to be an uncommon event.

Concrete applications and policies. We give two applications for Acsesor: a password manager and a crypto wallet. We describe Acsesor-supported versions for both of these applications in detail and discuss how that improves security and privacy over the state-of-the-art for these applications. We also give concrete instantiations of the *policy* for these.

2 The Acsesor Framework

We begin by describing the most basic version of our framework, which we call AcsesorCore. Here we consider a user who will store their secret using two guardians, and in this basic scheme, the user can only store one secret. We will discuss extensions in Section 5.

As described above, our protocols will make use of a ledger functionality; we will briefly describe it before describing AcsesorCore.A ledger functionality provides an append-only and tamper-proof dictionary of (label, value) pairs. Only the service provider can add new entries to the dictionary. The server also gives external parties access permission to lookup certain labels on the ledger. We discuss the need for a ledger in Section 1.1, define the functionality more formally in Section 2.4, and discuss two possible instantiations of it in Section 3.2. Now we are ready to describe AcsesorCore. The algorithms are described in detail in Section 2.1.

System Setup. The service provider will run ServerInit to initialize it's state and produce public parameters (which will include public parameters of an empty ledger). Each guardian will run GuardianKeyGen to generate it's own key pair. (See Section 2.3 for a discussion of how Guardian public keys could be certified and distributed.)

Storing Secrets. The user will run UserStoreSecret to generate an encrypted blob encoding their secret for their choice of guardians, under their choice of policy. They will upload this blob to the service provider to be stored under their username. The service provider will store the blob with the ledger and send back another blob *receipt* to the user. The user will run UserVerifyStorage with *receipt* to verify that their secret was indeed stored correctly. While we assume that they must somehow authenticate to the service provider, this authentication may not be cryptographic (*e.g.*, phone authentication), and we do not model it here.

Recovering Secrets. When the user wants to recover their secret, they will use UserRequestRecovery to generate a request message *msg*. They contact the service provider, authenticate via some potentially non-cryptographic mechanism, and send this request message to the service provider.

The service provider will add the recovery request to the ledger. Then, it will forward each request message to the appropriate guardians along with (some portion of) the user's stored blob and a handle to find the request on the ledger.

The guardians will run GuardianResponse to check that the request was added to the ledger and then produce a partially decrypted blob.

The responses from all the guardians will be returned to the user who will combine them using UserCompleteRecovery to reconstruct their secret. The service provider also returns a handle to the users so that they can check their recovery request was added to the ledger and that the response from the guardians are consistent with the recovery request message *msg* and their initial stored blob.

Monitoring the recovery request history. The user's device will periodically ask the service provider to provide a list of all of the access requests that have been made under their username. Again, we assume they authenticate with the service provider, but do not model the authentication here. The service provider will provide this list along with a handle to confirm this list with the ledger. The user will verify this proof using VerifyRecoveryHistory.

Contexts. So far we have described logging requests, but without specifying what exactly gets logged. In the rest of our proposal, we denote the information that gets logged as the "context" of the request. This could be simply the time that the request is made, or it could be some additional information, like the reason a request was necessary ("new device: iphone 15", "OS reinstall to recover from malware" etc). The user is responsable for choosing the correct context when he makes a request, and we will guarantee that any adversarial request will be reflected as an additional context in the user's recovery history, which we assume the user will be able to detect. (E.g. the user sees that there where 2 recovery requests on a given date, when he only made one, or there is a "new device: iphone 15" entry when he does not own such a device.)

For example, if we use time as a context, and we assume the user runs a recovery history query after every recovery, then to audit his history later, he need only 1) check that the *contexts* list is increasing, 2) check that there have been no recovery requests since his latest recovery. This means that instead of having to remember contexts for all of his recoveries, the user need only remember the time of his latest recovery.

Availability. Because the service provider provides a single point of contact for the user, if the service provider is unavailable (either maliciously or otherwise), the user will be unable to reconstruct their secret. This may often be worthwhile in exchange for the usability advantages discussed above.

A more subtle point is that the service provider also gates access to the ledger: if the service provider does not respond, then the user will be unable to monitor the ledger. In this case, the user must consider their account as potentially compromised. More generally, systems that do not require a central service provider to coordinate requests and perform authentication (such as [37, 26]), provide stronger availability guarantees at the cost of weaker security of authentication or useability.

2.1 The Algorithms

In more detail, our system requires the following algorithms:

$\underline{\mathsf{GuardianKeyGen}(1^k) \rightarrow (gpk, gsk)}$: The guardians will each run this algorithm once to generate a long term key pair. We assume that the public key comes along with a certificate that the user can verify, but that is outside of our model.

$\underline{\mathsf{ServerInit}(1^k) \rightarrow params}$: The service provider will run this before the system starts to initialize its state st_S. *params* will be available to all clients and will define the system. *params* is an implicit input to all the algorithms below.

$\underline{\mathsf{UserStoreSecret}(s, u, policy, gpk_1, gpk_2) \rightarrow (storedblob, st)}$: The user will run UserStoreSecret to generate encryption (*storedblob*) of their secret s to store with the service provider. u is the user's username, *policy* tells the guardians about any additional conditions that must be met before decryption, and gpk_1, gpk_2 are the guardians' public keys. The policy could, for example, specify a wait time the guardians must wait before responding, or an SMS number to contact for confirmation. These are other policies are discussed in Section 5.

$\underline{\mathsf{UserVerifyStorage}(st, receipt) \rightarrow \mathsf{success/failure}}$: The service provider sends a *receipt* to the user; the user will run UserVerifyStorage(st, *receipt*) using the state st output by UserStoreSecret to verify that their secret was indeed stored correctly.

$\underline{\mathsf{UserRequestRecovery}(u, context) \rightarrow (st_u, msg)}$: The user will run this algorithm to prepare a recovery request message *msg*. The only information required is their username u and a *context* which the user can later use to recognize this recovery. (See above for discussion of contexts.) st_u is a state that the user will later use to extract the secret from the response to their request.

$\underline{\mathsf{ServerStoreandLog}(st_s, (u, storedblob)) \rightarrow (st'_s, receipt)}$: The service provider runs this algorithm when user u requests to store secret encoded in *storedblob*. It takes as input an initial state st_s and produces an updated state st'_s, and a receipt *receipt* to be sent back to the user and which the user will use to verify that their data has been correctly stored and logged.

$\underline{\mathsf{ServerProcessRequest}(st_s, (u, msg)) \rightarrow (st'_s, \mathsf{servermsg}_1, \mathsf{servermsg}_2, receipt)}$: The service provider runs this algorithm to update the ledger to include the additional recovery request (u, msg). The service provider's initial state is st_s, and its state after the update is st'_s. It generates messages $\mathsf{servermsg}_1, \mathsf{servermsg}_2$ to be sent to guardians 1 and 2 respectively along with the user message *msg*. It also generates a receipt *receipt* which will be returned to the user along with the guardian responses; *receipt* helps the user verify that the service provider has performed correctly.

$\underline{\mathsf{GuardianResponse}(\mathsf{sk}, u, msg, \mathsf{servermsg}) \rightarrow (policy, decblob)}$ The guardians run this algorithm to process a user's recovery request. It takes as input the guardian's secret key sk, the user's username u, the recovery request message *msg*, and the message servermsg from the server. It produces a policy *policy* that must be satisfied before the secret is released, along with a partially decrypted ciphertext *decblob* which will it the guardian will release once it determines that the policy has been satisfied.

$\underline{\mathsf{UserCompleteRecovery}(st_u, decblob_1, decblob_2, receipt) \rightarrow s or \perp}$: The user runs this algorithm to complete their recovery and recover the secret. It takes as input the state the user generated as part of their recovery request, some auxiliary data produced by the server (to check that the recovery request was correctly logged) and the partially decrypted ciphertexts produced by the two guardians. It produces either the secret s, or $\perp$ indicating that some error has occurred.

$\underline{\mathsf{GetUserRecoveryHistory}(st_s, u) \rightarrow (contexts, receipt)}$: The server runs this algorithm when the user requests the list of recovery requests for their username. st_s is the server's state, u is the user's username. The output is a list *contexts* corresponding to recovery requests against the user's account and a receipt that will be returned to the user.

$\underline{\mathsf{VerifyRecoveryHistory}(u, contexts, receipt) \rightarrow 0 or 1}$: The user runs this algorithm to verify that they have received the complete list of recovery requests. u is the user's username, *contexts* is the list of contexts that the server claims represents the recovery requests for the user's account, and *receipt* is the server's proof that this list is complete.

2.2 Security and Confidentiality Guarantees

Our system guarantees the following properties. As discussed in Section 2.4 our instantiation will rely on a ledger functionality, hence for our construction the properties below will rely on the trust assumptions required for our ledger instantiation as well as the trust assumptions listed below. We define the security and privacy properties in our system in the game based paradigm. Each of the definitions below captures a different security/privacy property of the Acsesor framework. For each property we give in brackets the minimum trust assumptions required. The formal definitions are deferred to Section C.1.

User privacy [honest SP, honest guardians]. This property aims to capture the following property: a malicious user using the system should not be able to learn any information about the behavior of honest users, if both the server and guardians are honest. In particular, while a malicious user may learn that an action has occurred, they do not learn what action it was nor who requested it.

To capture this formally, we define a game where we give the adversary access to an oracle to which he can submit pairs of users and actions (either new secret registrations or secret recovery requests on behalf of users of his choice). Depending on a hidden bit, the game will either take the first or the second user/action in every pair. To prevent trivial attacks the actions are structured so that the adversary can never produce an invalid sequence of actions. Finally, we also provide oracles to allow the adversary to interact with the service on behalf of malicious users. The scheme provides anonymity if the adversary has negligible advantage in producing the hidden bit.

Confidentiality [honest SP]. This definition captures the confidentiality Acsesor provides for honest users' secrets when either or both of the guardians are malicious, but the server is honest. The malicious guardians can collude with the malicious users in the system as well. Acsesor guarantees that any honest user's secret will remain secure. For this definition to be meaningful, we do assume that the honest server can effectively authenticate the honest user and detect recovery attempts by adversaries.

In the security game, the adversary has access to several oracles which let it instruct an honest user to store a secret or request a recovery. The adversary also has oracles to store and recover secrets on behalf of malicious users. The game flips a bit and decides to either encrypt the honest user's secret (provided by the adversary when it invokes the oracle for storing the honest user's secret) or encrypt a random string. The adversary's goal is to guess the bit. We say that the adversary wins the game if it can guess the bit with non-negligible advantage over 1/2.

Policy Enforcement [1 honest guardian]. This property captures the confidentiality Acsesor can provide for an honest user's secret when the server is malicious and one of the guardians is malicious. The malicious parties can potentially collude, and they can collude with malicious users in the system. In this case, Acsesor guarantees that the adversary cannot learn an honest user's secret unless there is an adversarial request for which the honest guardian determines that the user's policy has been satisfied.

Defining the formal security game to capture this property is more involved then the confidentiality property above. As in that previous game, we construct oracles for the adversary which lets it instruct an honest user to store a secret or request a recovery. In addition to that, we give the adversary oracles for observing the honest guardian's responses (for both malicious user requests and for honest user's request, when the policy is not satisfied). We wish to capture any policy, including those that cannot easily be formally modeled, like a wait time, or the out-of-band confirmation. To do this, we provide two oracles to capture the honest guardian' response process. First, the adversary gets access to an oracle where they can submit recovery requests to the honest guardian. In response, the guardian will extract the policy. Then, the adversary can choose to call a second oracle through which the guardian will return its response; this models the case where the guardian determines that the policy has been satisfied. Of course, if the adversary can make a malicious request which satisfies the policy used in the challenge, then it will be able to trivially learn the secret. Thus, we constrain the second oracle to return $\perp$ when the request is adversarial and the policy matches that in the challenge. Note that we no longer need any oracle for storing secrets for malicious users, as the adversary (which includes the malicious server) could do that by itself. As in the previous game, the game decides whether to encrypt the real secret, provided by the adversary or a random string, depending on a bit flip. The adversary wins the game if it can guess the bit with non-negligible advantage over 1/2.

Transparency [1 honest guardian]. This property guarantees that the adversary cannot learn an honest user's secret without logging at least one access beyond those corresponding to the honest user's recoveries. This, in turn, implies that if the adversary learns the secret, it will be visible to the honest user when they audit their recovery history. Acsesor provides transparency as long as at least one of the guardians is honest. The server and the other guardian can be malicious and colluding with malicious users of the system.

Defining the security game for transparency is a bit involved. To capture the property that the adversary should not be able to guess the honest user's secret (unless the recovery attempt has been logged), we let the game flip a bit and decide whether to encrypt the real secret (provided by the adversary through its oracle calls) or a random string. We also initialize a game variable *caught* to 0. We give oracle access to the adversary which lets it store a secret for an honest user and instruct that user to request a recovery.

In addition, we give the adversary an oracle to verify recovery history for the honest user with a list of recovery *contexts*. The purpose of this oracle is to capture the scenario where the adversary's recovery attempt has been recorded in the list of recovery *contexts*. If so, this oracle sets the variable *caught* to 1.

At the end of the game, the adversary outputs its guessed bit b' to indicate its guess about whether the real secret was stored or a random string. However, if $caught = 1$, the adversary will get the secret trivially. So, in this case, the game outputs a random bit. If $caught = 0$, then the adversary should not have access to the honest user's secret, so its guess is meaningful. In this case, the game outputs b'. *Transparency* is achieved if the advantage of this game in producing the bit b is negligible.

History consistency [none]. This property guarantees that if the user verifies his recovery history multiple times, the earlier context lists will always appear as prefixes of the later context lists. Acsesor guarantees this property even when the server and both guardians are malicious. Notice that, this, together with the *Transparency* property means that if the user can recognize the context for his most recent recovery, when he does a recovery history query he need only make sure 1) that there is only one such context in that list and 2) that it appears at the end of the list. This is nice in that it means the user does not need to remember context information for all of his past recoveries, only the most recent one.

The security game for this property is fairly simple. The adversary is given a single oracle where it provides a list of *contexts, receipts* for the target user. If VerifyRecoveryHistory accepts, this list of contexts is recorded. If at some point VerifyRecoveryHistory accepts a list that does not contain the previous list as a prefix, then the adversary wins the game.

Robustness [none]. We already discussed how all confidentiality is lost when both guardians and the server are malicious. However, interestingly, we can still achieve *robustness* in this case. This property guarantees that the user cannot be convinced that a recovery has been successful unless it has correctly produced the user's original stored secret.

At a high level, the security game is defined as follows. We give oracle access to the adversary which lets it store a secret for an honest user and instruct that user to verify a recovery. The game stores the honest user's stored secret (sent by the adversary through the oracle call) in a table. If the adversary is able to make the UserCompleteRecovery algorithm output a different secret from the one stored in the game's table, the adversary wins the game.

2.3 Guardians

The guardians must be trusted not to collude with the provider or one another. This raises the question of who can be trusted to operate guardians. We propose several possibilities, which may be more or less suited for specific applications. The guardians need to store their private keys, and depending on the kinds of policies they support, possibly a small amount of information per user (see Section 5 for more details). The guardians' operations are limited to decrypting and re-encrypting key shares and verifying the presence of recovery attempts in the ledger. Accordingly, guardians are relatively lightweight, and we imagine they could be run efficiently on even low-powered devices.

User-managed devices. Letting users run guardians explicitly places secrets back in their control. Of course, this also places responsibility for disaster recovery on the user, and if the guardians went offline in a burglary, fire, or flood, the secrets would be destroyed. This could be partially mitigated by moving to a threshold design, where there are n guardians and any t of them are required to recover the secret.

There are still some advantages over purely user-managed systems: most of the storage is offloaded onto the provider, and the threshold architecture means that secrets could be recoverable even when one or more devices are offline.

Similarly, the user could consider devices owned by friends, family members, or administrators; since Acsesor allows for many guardians, the user could require a consortium of trusted people, so that no one person must be granted full trust.

Independent organizations. Users may not trust a single corporation with their most high-value secrets, but they might be more likely to trust a set of them not to collude. This is especially promising for enterprise applications, where there is contractual recourse for malicious behavior, and for settings in which organizational, departmental and/or geographic diversity adds value.

PKI. We assume a public-key infrastructure that the user can rely on to get authentic and valid public keys for their chosen guardians. If the guardians are public entities as in the last option above, this can be the standard certificate based PKI. In the case of user devices the user will have to ensure that they has the appropriate public keys for their devices.

2.4 Ledger

Here we define the Ledger functionality that we alluded to before. The functionality accepts the following commands. In Section 3.2, we discuss two possible Ledger instantiations. Here $\mathcal{L}$ is a stateful leakage function used to parameterize the Ledger Functionality.

On command Init() from a server Server, the ledger initializes a key-value store D, and a table T and records the identity of Server. It returns *params*.
On command Store(x, v), if $D(x) = \perp$ from Server, the ledger updates $D(x) := v$. If $D(x) = \vec{v}$, the ledger updates $D(x) := \vec{v}||v$. Delete all entries in T. It also notifies the adversary that a successful store command has been performed

(but not what the inputs were) and sends the adversary $\mathcal{L}(\mathsf{Store}, x, v)$.
On command $\mathsf{GetLookupHandle}(x)$ from Server, the ledger generates a handle h, stores $(h, x, D(x))$ sends h to the Server.
On command $\mathsf{Lookup}(h)$ from P, if $(h, x, \vec{v})$ is stored in T for some $x, \vec{v}$, then return $x, \vec{v}$. It also sends $\mathcal{L}(\mathsf{Lookup}, x, D(x))$ to P if it is adversarial. Else return $\perp$.

3 Our construction

We give an overview of our construction here, and defer the more detailed description to the Appendix.

3.1 Construction overview

Roughly, our construction proceeds as follows:

System setup. This consists of the guardians generating their keypair and publishing their public keys and the server calling ledger Init to get and publish the public parameters *params*.

Storing Secrets. We will gradually build up the protocol.

Attempt 1: As a first attempt, we can try to directly create two shares of the client's secret s, such that $s_1 \oplus s_2 = s$. Then, the client can encrypt each share to each guardian along with their username, the policy under which it should be released and stores the ciphertexts with the server. But, now, if both guardians are malicious, they will immediately get the secret as soon as it gets the ciphertexts from the server as part of a recovery request. Thus, we will not be able to achieve confidentiality the secret.

Attempt 2: To overcome this challenge, we add a layer of encryption on top of the secret. To store s, the client first picks a random key sk_s which it uses to encrypt the secret s to form ct_s. Then it creates two secret shares of sk_s such that $\mathsf{sk}_1 \oplus \mathsf{sk}_2 = \mathsf{sk}_s$. Now, in the case where both guardians are malicious, they would learn sk_s if a recovery attempt has been made. But they still won't get s as long as they don't have the ct_s stored with the server.

Attempt 3: There is still another caveat. Even one malicious guardian can cause the client to decrypt ct_s under an incorrect (related) key $\mathsf{sk}'_s = \mathsf{sk}'_1 \oplus \mathsf{sk}_2$, where $\mathsf{sk}'_1 \neq \mathsf{sk}_1$. This will violate *Robustness*. One way to get around this, would be to use a stronger encryption scheme that is secure under related key attacks. However, in our scheme, we are already using a ledger for transparency. We can use another instance of the same ledger functionality to store cryptographic commitments to the key shares $\mathsf{sk}_1, \mathsf{sk}_2$. The clients can verify that they got the correct key shares back before decrypting ct_s. This eliminates the need for a stronger encryption scheme. We need hiding cryptographic commitments to ensure that the key shares are hidden from the server, the ledger and the other guardian.

Final Attempt: To store s, the client first picks a random key sk_s which it uses to encrypt the secret s to form ct_s. Then it creates two secret shares of sk_s such that $\mathsf{sk}_1 \oplus \mathsf{sk}_2 = \mathsf{sk}_s$. It also picks randomness $\mathsf{comrand}_1$ and $\mathsf{comrand}_2$ to create commitments h_1, h_2 to the two shares respectively. Then it encrypts one share to each guardian along with their username, the policy under which it should be released and the corresponding commitment randomness, forming ciphertexts ct_1, ct_2. Finally, it creates $blob_1 = (ct_1||h_1), blob_2 = (ct_2||h_2)$. The client requests the server to store $storedblob = (blob_1||blob_2||ct_s)$ in the user's account.

The server logs the storage under the user's account $(u||\texttt{storage})$ with the ledger using $\mathsf{ServerStoreandLog}$, gets back a handle from the ledger and sends it back to the client. The client runs $\mathsf{UserVerifyStorage}$ to check that $(blob_1||blob_2||ct_s)$ was indeed stored in the ledger. This is important, as otherwise, a malicious server could always store a fake blob on a user's account.

Secret Recovery. When the client requests a recovery, it chooses a random key pair $(\mathsf{pk}_u, \mathsf{sk}_u)$, and sends the public key pk_u and some context string *context* (chosen by the user) as the recovery request and stores $(\mathsf{pk}_u, \mathsf{sk}_u, context)$ in its local state.

The server logs this request $(context, \mathsf{pk}_u)$ with the ledger under user account $u||\texttt{recovery}$ and gets back

$\mathsf{handle}_{\texttt{recovery}}$. It sends $\mathsf{handle}_{\texttt{recovery}}$ along with $blob_i$ to guardian i, $i \in \{1, 2\}$.

Each guardian decrypts its share of the random key and re-encrypts it under the public key pk_u from the recovery request. As part of decrypting the share of the random key, it also gets the user's username, the policy under which the response should be released, and the commitment randomness. It checks that this policy has been satisfied and that the request has been properly logged before releasing the re-encrypted ciphertext.

The server looks up the ledger with $u||\texttt{storage}$ to get back the $\mathsf{handle}_{\texttt{storage}}$. It sends back the ciphertexts it received from the guardians, along with $\mathsf{handle}_{\texttt{storage}}$ and $\mathsf{handle}_{\texttt{recovery}}$ to the client.

The client uses $\mathsf{handle}_{\texttt{storage}}$ to get *storedblob* back from the ledger, parses it as $(ct_1||h_1, ct_2||h_2, ct_s)$. It also uses $\mathsf{handle}_{\texttt{recovery}}$ to get back a vector $\vec{v}$ of values from the ledger and checks that $(\mathsf{pk}_u, context) \in \vec{v}$. This is important in detecting any potential Meddler-in-the-Middle (MitM) attack: if a malicious server tries to use its own public key for decryption, the client will notice that the logged key does not match its local key. If the check passes, the client decrypts the ciphertexts in each of the guardian's responses to get back $\mathsf{sk}_i, \mathsf{comrand}_i$, $i \in \{1, 2\}$. It checks that $h_i = com(\mathsf{sk}_i, \mathsf{comrand}_i)$. If all the checks pass, it reconstructs $\mathsf{sk}_s \leftarrow \mathsf{sk}_1 \oplus \mathsf{sk}_2$, and uses sk_s to decrypt ct_s and get s.

Monitoring recovery request history. When the client wants to monitor the history of recovery attempts for user u, it pings the server. The server looks up the ledger for $(u||\texttt{recovery})$. If it gets back $(\mathsf{handle}, \vec{v})$, it sets *contexts* to be the list of context fields from the tuples in $\vec{v}$ and $receipt := \mathsf{handle}$. It sends back *contexts*, *receipt* to the client. The client runs VerifyRecoveryHistory to verify the server's response. It looks up the ledger with *receipt* as the handle and checks that it gets back $u||\texttt{recovery}, \vec{v}$, where the contexts of the fields in $\vec{v}$ match the list *contexts* provided as input.

3.2 Instantiating Ledger functionality.

With PAHD from OPTIKS. Privacy-preserving Authenticated History Dictionary (PAHD) [31] implements the ledger functionality under the assumption that each update is audited by an honest auditor and there is a trusted bulletin board for posting commitments from each update.

We use the PAHD implementation from [31]. In this implementation, if Lookup returns $(x, \vec{v})$, checking the returned result involves $|\vec{x}| + 1$ public key operations on the verifier side. However, [31] has an optimization for verifier side caching that reduces this cost to 1 public key operation. We leverage his optimization in our instantiation too. Note that, for AcsesorCore, we need to invoke Lookup both on the clients (user devices) and on the guardians. While caching makes sense for clients, it is undesirable for the guardians to keep state per user. In order to use the caching optimization of [31] for guardians, we do the following. We let the guardians authenticate the latest state they see per user using a *Message Authentication Code (MAC)*. They can send this MAC tag and state to the server that can subsequently send it back to the guardians as part of a new logging proof. A malicious server could send a stale state and signature, but in the worst case this will make the guardian redo work. In other words, this does not affect the security or confidentiality properties of AcsesorCore.

Since, we use the instantiation of [31], we also inherit its leakage. In particular, the leakages are the following, where $\mathcal{L}$ is a stateful function:
$\underline{\mathcal{L}(\mathsf{Store}, x, v)}$: The first time $\mathcal{L}(\mathsf{Store}, \cdot, \cdot)$ is invoked, the function initializes a counter $ctr = 1$ and stores it in its state. On each subsequent call, the counter value in incremented by 1.

This function outputs x if this was the first update on x since $\mathsf{GetLookupHandle}(x)$ was called. It also stores (x, v, ctr) in its state.

$\underline{\mathcal{L}(\mathsf{GetLookupHandle}, x, D(x))}$: Say $D(x) = \vec{v}$ and $\vec{c}$ is the vector of the corresponding counter values (from the state). This function outputs $\vec{c}$.

With Nimble. Nimble [2] realizes the ledger functionality with Trusted Execution Environment (TEE). For Nimble, if Lookup returns $(x, \vec{v})$, checking the returned result involves $|\vec{x}|$ hashes and a constant number of public key operations on the verifier side (3, in the Nimble experiments). For this implementation of the ledger functionality both $\mathcal{L}(\mathsf{Store}, x, v)$ and $\mathcal{L}(\mathsf{GetLookupHandle}, x, D(x))$ output $\perp$.

4 Performance

Here we discuss the performance of Acsesor from the point of view of the different parties. We consider two ledger implementations: the OPTIKS PAHD [31] based on the oZKS library[1], and the a ledger built from Nimble [2] based on the similarly named library.[2]

Users' workload. To store a secret, a user needs two public key encryptions, a symmetric key encryption, and compute two commitments (implemented using hashing with nonce as in [7]). The total cost of this rarely executed operation is measured in microseconds on a commodity machine. The symmetric key operations are orders of magnitude cheaper than the public key encryption. As an example, we timed the the sealed box encryption in `libsodium`, and found it to take only about 143 μs on commodity hardware.

After storing a secret, the user needs to check that it was added correctly in the ledger by performing a ledger lookup. For both instantiations of a ledger (Section 3.2), the lookup validation is limited by a few public key operations. For example, with OPTIKS PAHD lookup verification takes just a bit over 100 μs. With Nimble, the user needs to verify three much slower ECDSA signatures, but still we measured this to take only 4.9 ms. These number are also listed in Table 1.

The cost of a recovery request involves public key generation, two public key decryptions, one symmetric key decryption, two hashes (for verifying the commitments), and one lookup to verify that the request was properly logged. Again, this is overall very cheap, with the public key decryptions being the dominating cost, but still only measured in microseconds. The user also needs to check the ledger to verify that the recovery request was logged correctly; again, this takes a few milliseconds at most.

The main cost for the user is in monitoring its recovery requests periodically, so it needs to repeatedly lookup its state from the ledger. However, by keeping track of its history, it just needs to check the latest state of its state in the ledger to see that there are no new or unexpected requests. The cost of this is again measured in milliseconds.

Guardians' workload. For a recovery request, the guardians need to do one public key decryption, one public key encryption, and at most four MAC operations (including the optimizations mentioned in Section 3.2 and Section 5). We measured the sealed box decryption in `libsodium`, and found it to take about 101μs on commodity hardware. Thus, both public key operations take between 100–200μs. The guardians do need to check the ledger as well, which may take up to milliseconds depending on the implementation, as was explained above. Nevertheless, these hardly constitute bottlenecks for the guardians even if they are serving a vast number of users.

Server's workload. By far the most interesting case is the server's workload, where running the ledger is by far dominating the complexity. Luckily, both [31] and [2] provide performance numbers for full-system benchmarks that include a database-based storage, which in a real-world implementation is crucial for resilience.

The recovery request and lookup throughput numbers for both types of ledgers are summarized in Table 1.

A few clarifying comments are in order. The state in OPTIKS PAHD-based ledger proceeds in epochs. This means that updates are made in batches of a customizable size. For example, the numbers in Table 1 use a batch size of 1024, resulting in short epoch times with an interquartile range of 1–5 seconds ([31, Figure 3a]). In contrast, Nimble proceeds in real time and can process a single operation at a time.

It is important to use the caching technique men-

[1]`https://GitHub.com/Microsoft/oZKS`

[2]`https://GitHub.com/Microsoft/Nimble`

Ledger	Recovery requests (req/s)	Lookups (req/s)	Lookup verification
OPTIKS PAHD	1,050–250	4,400–2,250	103–105 μs
Nimble	2,600	50,000	4.9 ms

Table 1: Performance results from OPTIKS [31] and Nimble [2] adapted for use in Acsesor to implement the ledger functionality. The results include the overhead of database-backed storage for resilience. For OPTIKS PAHD we show the ranges when the total number of recovery requests (entries in the ledger) grows from 1M to 64M. The Nimble lookup time comes from our measurement of 3 ECDSA verifications (from `Crypto++`) on a commodity machine, which is what Nimble requires.

tioned in Section 3.2. Otherwise OPTIKS PAHD will require to perform a full "history query", whose cost (computation and communication) scales with the number of recovery request in the user's state. The numbers in Table 1 assume the use of this optimization.

Another important note is that OPTIKS PAHD uses an internal caching technique to avoid expensive public key operations. For the Acsesor ledger to be able to utilize this, it needs to be able to store a precomputed hash for each active user in memory, which seems realistic even with a large number of users. The numbers in Table 1 assume this to be the case.

Nimble can support both Intel SGX and AMD SEV-SNP based enclaves. In [2] the authors find the SGX-based implementation to be significantly slower. Indeed, their benchmarks with database backed storage are done only with SEV-SNP enclaves, and these are the numbers we list in Table 1.

5 Policies and Extensions

Here we describe some policies we believe may be most practical to address real-world problems. We also describe technical extensions of the Acsesor system, such as extending to use more guardians, and using an extra layer of authentication with the guardians.

Adding more guardians and thresholding. While AcsesorCore uses two guardians for simplicity, the framework can be easily extended to support $n > 2$ guardians. We can extend all the security and confidentiality guarantees, as long as some fraction t of the guardians are honest (our basic construction can be viewed as $n = t = 2$). In the construction, we would use t-out-of-n secret sharing scheme to split the key sk_S into n shares. This can be beneficial, if availability or security of the guardians is questionable. By using threshold secret sharing, the secret key can be recovered, even if only a fraction of the guardians are available.

Threshold signatures and decryption. In AcsesorCore, the user recovers her secret using UserCompleteRecovery. Usually, the recovered secret will be used in some other cryptographic scheme, such as signing or decryption. The Acsesor framework can be extended such that, each guardian performs a partial signature/decryption using their key share in a privacy-preserving way (using a threshold signing/decryption scheme). Then, the pieces can be put together to construct the signature/decrypt using UserCompleteRecovery. Note that this will not require additional communication between any parties if the threshold signing/decryption scheme is non-interactive (e,g., threshold BLS signatures [6]). Finally, to ensure security in the case where the server is honest but both guardians are compromised (the signature equivalent of the Confidentiality property defined in Section 2.2), we could have the server operate an additional guardian node itself, and set the threshold structure to require that that node participate in every signature generation.

Storing many secrets. For simplicity, our AcsesorCore framework lets a user store one secret per username. The framework can be extended to support multiple stored secrets per user (under the same username) by letting the user attach a *storagecontext* string to its *storedblob* and encrypt it in ct_i for guardian i, along with the respective keyshare, username and policy and the commitment randomness. The *storagecontext* string could contain information about context in which the secret is stored. For example, the *storagecontext* string could be name of a website or wallet app. While a completely different application, such storagecontext strings were used in a similar fashion in logging in [8].

Out-of-band confirmation. An important policy could be adding a second layer of out-of-band confirmation for recovery, without changing the authentication flow of AcsesorCore. This could be implemented as follows. The user could encrypt their phone number or email id in the ciphertext ct_i for guardian i when generating *storedblob*. The policy could say that the guardian is supposed to send a notification to the encoded phone number or email id and should proceed with the recovery process only if a confirmation is received through the out-of-band channel (SMS/email).

Slow and fast recovery. Another important policy is a user-configurable wait time that the guardians are expected to wait before releasing their share of the secret.

To enable emergency recovery, one could set a long wait time, such as a week or a month, before the secret is released. If the slow recovery is maliciously initiated by someone who has compromised the server's system, or by the server itself, the long wait time will provide the real user enough time to notice that a recovery process has been initiated, or at least learn that the secrets are about to be compromised if a malicious server blocks the user's legitimate access attempts. In this case, the user can secure their account with the server and potentially change their secrets (passwords, keys) before they get revealed to the attacker.

Generally, a long wait time policy can allow a low-entropy secret (password to the user's account with the server) to be the sole secret protecting stronger secrets. Fast recovery with a short (or no) wait time can make sense when access to the secrets is needed immediately, and the main concern is to be able to reliably monitor recovery requests.

Minimizing storage of guardians for slow recovery As described above, if the policy requires a delayed release, naively, the guardian would have to hold on to the share until the time period has elapsed and release it only after that. This would increase the storage at the guardian. Instead, the guardians could create a MAC tag on the time they expect to release its share and send it back to the server along with the delay time. Once the delay period has elapsed, the server can send the MAC tag and payload back: the guardians can check the time and then prepare *decblob*.

Second factor. To provide stronger protection against person-in-the-middle attacks, especially with fast recovery policies, the Acsesor framework can be extended to support a second layer of authentication between the user and the guardians.

Authentication between users and guardians can be done through a few low-overhead mechanisms: a password or PIN, a signature-based hardware token, or a one-time email or SMS-based code. These options are in addition to any (possibly multi-factor) authentication performed by the server. For example, if the user intends to use a PIN as a second factor with a particular guardian, they can include this PIN when they create ct_i for that guardian, and similarly include an encryption of the PIN as part of their request.

The user would need to set up an independent PIN with every guardian to avoid a malicious guardian leaking the PIN. Note this is very different from the out-of-band policy described above. This second factor authentication does not require any out-of-band channel, but does require the user to set up a different PIN per guardian and provide them for each recovery request.

Finally, the user could choose a policy instructing the guardian to rate-limit requests (and hence PIN guesses); as the guardian can see the user's entire request history, this is straightforward to enforce.

6 Secret Management with Acsesor

In this section we describe how to cast many common secret management scenarios into the Acsesor framework.

6.1 Password Management

Existing solutions. Popular web browsers including Chrome [20], Firefox [17], Safari [3], and Edge [44] offer built-in password managers that allow users to save credentials and back them up in the web browser's cloud service. Most of these are custodial by default, with some exceptions (e.g., [19]). These custodial managers synchronize the users' secrets on all her devices and can use any standard methods for securing the account, such as 2FA.

Some other password managers, such as 1Password [1] and LastPass [28], are non-custodial. 1Password uses a combination of a user-stored cryptographic secret key, as well as a master password, to protect access to the secrets. LastPass derives an encryption key from a user's master password and uses it to encrypt their secrets. Neither of these can restore a user's access to their secrets in case they forget their password or lose access to their secret key.

Acsesor-based password management. One simple approach would be to have Acsesor store a cryptographic secret, and then use that to derive individual passwords locally on the user's device. Alternatively, one could use Acsesor to store each individual password - this would give more fine grained access control and auditability, but the resulting monitoring might be more complex for the average user. There are many options for which policies to choose, which allow for different tradeoffs in security and availability.

At one extreme, we could replicate the availability of custodial password managers with a policy that requires no second factors or delay. The user's account with the server would allow them to log in from different devices and the server can continue using standard methods to secure the account, such as 2FA. Configured with a such a recovery policy it would allow immediate synchronization of the secrets to a new device, but unlike existing solutions, the user would have a cryptographically secured log of this happening. Thus, no-one – not even the server itself – would be able to access the user's secrets without being detected. Thus, this would still provide a stronger guarantee than in the existing custodial solutions. In 2022 LastPass became a victim of a data breach [29], where some of their customers' data was stolen. With the Acsesor based solution it would be immediately clear exactly what was stolen.

At the other extreme, to achieve similar security to non-custodial schemes we could allow short time recovery with a password/PIN as a second factor. If the second factor involves a cryptographic secret key, then the security guarantee would be similar to that of 1Password, but with the additional benefit of transparency for access attempts to the password manager's cloud storage. In fact, we can do better with Acsesor. The reason the non-custodial factor has to be of cryptographic strength, instead of a password or PIN, is to mitigate the fact that otherwise the password manager's servers would create a single point of failure and, if compromised, could expose all users' secrets to brute force attacks. With Acsesor the second factor can be weaker, because the guardians can limit guessing attacks through rate limiting. The server holds only data encrypted with high-entropy secrets and every guess at the user's second factor requires a separate entry in the recovery log, as well as interaction with the guardians.

We believe one approach that might provide a good balance would be to allow short time recovery with a password/PIN second factor, and then to allow longer time recovery with no second factor. The short term recovery would have similar security to non-custodial schemes as discussed above, while the long term recovery would provide availability even if the user loses/forgets their password/PIN. This does leave open the possibility that an adversarial custodian might introduce a fake long term recovery request in an attempt to regain the secret. But in this case, this request would be detected immediately by the user, and they would have time to react. In the case of passwords it is often possible for a user to recover from a breach with enough warning (e.g. by moving to another password manager and changing all of their individual passwords using each individual account's forgotten-password-reset mechanism). If this is not the case, then we could instead secure the long term recovery with an out-of-band confirmation as discussed in the wallet setting below.

6.2 Cryptocurrency Wallets

Existing solutions One of the primary non-custodial secret storage scenarios today is cryptocurrency wallets. These wallets require that the user somehow store a cryptographic secret, whether that is on a hardware device, or on paper, etc. (For example the BIP-39 protocol used by the [14, 13, 35, 10] wallets requires that users remember/record a 12-word recovery passphrase.) This secret can then be used to derive or encrypt signing keys for each of their crypto coins. This has major availability issues, and many users have permanently lost access to their funds as a result.

An Acsesor-based wallet. A natural approach would be to use Acsesor to store the cryptographic secret (e.g. the BIP-39 passphrase) and then generate all of the individual coin private keys locally on the user's device. Again there is the alternative to store the individual coin private keys separately in Acsesor for more fine grained access control and auditing at the cost of more complex monitoring.

To achieve similar security/availability to existing schemes, we could use a policy which allows short term recovery with knowledge of a high entropy secret (e.g. a 12 word passphrase as in BIP-39). This would be strictly stronger than existing schemes in that it would guarantee that any accesses are logged, and the user would learn immediately if their private keys had been compromised.

However, because of the high cost of losing access to one's cryptocurrency secrets, we might also want a policy that allows for long term recovery. At the same time, allowing an adversarial server to access the secret is very dangerous; the server in that case could steal the funds, and if the user is not storing their secret locally (e.g. in the case where they are trying to perform a recovery), they would have no recourse. This means we need a fairly strong 2nd factor for the long term recovery as well. At the same time, this factor needs to be something that the user is unlikely to lose permanently.

Our proposal is for a policy which allows for short term recovery with a cryptographic secret/high entropy passphrase, and a long delay policy under which the guardian will first perform an out-of-band confirmation, and log the success along with the request, then wait the prescribed time, and then perform a second out-of-band confirmation before revealing the secret. Access to an out-of-band confirmation method like email or SMS seems like something a user is much less likely to lose permanently than a secret stored in hardware or in human memory. On the other hand, it maybe vulnerable to compromise (e.g. if the attacker can hijack the user's email account or phone number). This is why we include 2 rounds of out-of-band confirmation in our proposed policy - if the user sees that there has been a successful first out-of-band confirmation for their account, then they will have time to reclaim and strengthen security on their out-of-band confirmation method before the second confirmation occurs.

Finally, to match the functionality of threshold cryptography based wallets [46, 15, 27] (see Section 7 for more discussion on this), the user could specify a policy where the guardians do not actually return the key fragments, but instead use them to (threshold-)sign a transaction provided as a part of the recovery request as described in Section 5. This works particularly well with the Ethereum blockchain, as its BLS threshold signatures requires no interaction between the guardians.

7 Related Work

Several works have attempted to reduce the burden of trust on custodians using specialized hardware and cryptographic mechanism (such as threshold cryptography). Acsesor also aims to reduce the trust on custodians, but without compromising on usability. Acsesor has two main usability goals – not changing the authentication flow of traditional custodial systems (so that the user still needs to interact with a single entity: the server), and not assuming the user can store long term secrets. Acsesor also provides logging of recovery requests. This is particularly important since a malicious server can act on behalf of the user. We focus on the most relevant works from the literature and do not cover more general topics such as threshold cryptographic techniques for authentication since they typically do not provide logging.

Recently [37] proposed a cloud based secret recovery system where the goal is to let the users retrieve their secret from the cloud even when they lose all their authentication credentials. Their system is implemented using a public blockchain and TEEs. Their idea is the following: the user has to remember a public facing id. They post this id along with public key pk_u and the servers they will use for recovery, in a blockchain transaction. Each recovery request has to be posted on the blockchain and a certain amount of time will have to elapse before the servers release the secret. The key insight here is that, if the legitimate user did not lose their device, they will still have the sk_u corresponding to the pk_u. If they notice a recovery request on the blockchain that they did not make, and still has access to sk_u, they can use it to sign and post a signature to deny the request and stop the release of the secret. To keep the secret private from the cloud server, the authors propose using TEEs hosted by the cloud.

While the functionality has some similarity with Acsesor, the realization of it using blockchain is much more expensive. The user's monitoring cost will be significantly more compared to Acsesor, since they have to keep scanning blockchain transactions to perform monitoring (the authors do not report any performance numbers). As presented, this scheme will not be able to achieve the anonymity property of Acsesor since all the transactions are posted on the blockchain. Finally, trusting TEE is an additional assumption: there could be vulnerabilities in TEE itself which could compromise the confidentiality of the user's secret completely, which the authors also acknowledge. On the other hand, their system allows users to deny a recovery request in progress, which our system currently does not. It would be interesting future work to see whether their approach can be integrated into our system.

LARCH [12] is another recent work that provides auditability in an authentication framework by adding a third party logging service. While this system is not specifically designed to store and recover secrets, the goal of logging authentication attempts is similar to the goal of Acsesor of logging all recovery attempts to detect fraudulent requests. The key idea in [12] is to split the user's authentication credential between the client and the log service. Thus, each authentication attempt has to involve both the client and the log service, which enforces that each attempt is logged. But the entries are encrypted under the user's key (which the users need to store), so the log service cannot decrypt it. This is where the system significantly deviates from Acsesor's usability goal of not requiring the user to store any long term cryptographic secret.

CALYPSO [26] presents an entirely decentralized secret document management system, based on blockchains and skipchains [36]. The CALYPSO system stores secrets among a committee of trustees (similar to guardians in Acsesor) and provides flexible access control and a private-but-auditable log on the blockchain. However, CALYPSO does not satisfy either of our main two usability requirements: CALYPSO users must store several long term public/secret key pairs, and they must directly interact with the trustees. Furthermore, since all operations go through the blockchain it is inherently more heavyweight.

Next, we discuss cryptocurrency wallets. Fireblocks [15] uses threshold signatures within an Intel SGX enclave. ZenGo [46] uses distributed key generation to create and store shares on ZenGo servers and the user's device. They utilize biometrics and user's third-party cloud storage for emergency recovery. ZeroWallet [27] threshold-shares the secret into three parts. One part is derived from the user's password. Another is cached by the user's device, and a third one can be generated from another user password and a server-managed secret in an interactive protocol (using OPRF). This approach is practical for everyday use of a wallet without having to cache the secret itself: the first password would be required to reconstruct and use the secret. By instead using more servers, the ZeroWallet system can allow recovery even if the user loses access to their share and some of their passwords. However, due to the lack of a single central server, the user must remember the identities of their chosen recovery server – or possibly multiple servers – along with passwords for each of these. This is a major difference with Acsesor. Moreover, none of these wallets [15, 46, 27] provide any auditability guarantee. unlike Acsesor.

CanDID [33] is a decentralized identity system which addresses the problem of secret recovery. The system is significantly different from Acsesor as CanDID is built on a decentralized network of nodes instead of a centralized server. The CanDID committee stores secret-shares of the user's secret and when presented with sufficient authentication evidence, the committee can release the shares to the user. CanDID also does not have logging support.

SafetyPin [11] uses Hardware Security Modules (HSMs) to create a system for mobile device backups. Access to the backups are protected with weak secrets (PINs) and the HSMs protect the PIN against guessing attacks. SafetyPin decentralizes the trust assumption by relying on a large network of HSMs; their security model provides data confidentiality as long as a large enough fraction of the HSMs are uncorrupted. They also add some auditability guarantees. A downside of this approach is the reliance of special hardware that can be expensive and whose operation is not transparent.

Tutamen [39] builds a secret-storage system with fine-grained access control on untrusted hardware by using a decentralized architecture of *access control* servers and *storage* servers that use threshold tokens and secret-sharing to distribute trust. Clients in the Tutamen system hold a long-lived private key and certificate that they use for authentication. This is a major difference from Acsesor where the users are not required to maintain any cryptographic state. Tutamen proposes as future work a publicly auditable log of attempts resembling certificate transparency [30].

The PAD Protocol [38] (*Privacy-Preserving Accountable Decryption*) presents an access delegation system based on a decentralized set of *trustee* and *validator* nodes. It uses a transparent log to record access attempts. While delegation is out of scope for Acsesor, this system is similar to Acsesor in its goal of logging access attempts. However, unlike Acsesor, this system also requires users to store cryptographic keys. Finally, we note that transparency logs themselves have a long history, with numerous constructions and use-cases including key transparency [34, 7, 5, 21, 24, 42, 41, 9, 31], binary transparency [18, 16, 23], credential transparency [8] etc.

8 Conclusion

We have introduced a new framework (Acsesor) for secret management, where a centralized server takes the role of a custodian. Instead of having to trust the server, the users will be given cryptographic proofs of its correct behavior. To eliminate single points of trust, Acsesor distributes the recovery process across a set of guardians the user can choose. But, the user is never required to interact directly with the guardians, which allows us to retain the high usability of centralized custodial solutions. As long as a large enough fraction of the guardians behave correctly, the user can be guaranteed to learn whether their secrets are being accessed by a malicious party, including the server. By allowing the guardians to implement flexible response policies, Acsesor can address a broad range of problem scenarios in classical secret management solutions: we have already outlined some promising applications.

References

[1] 1Password. [n. d.] https://1password.com. Accessed on 10/15/2023. ().

[2] Sebastian Angel, Aditya Basu, Weidong Cui, Trent Jaeger, Stella Lau, Srinath Setty, and Sudheesh Singanamalla. 2023. Nimble: rollback protection for confidential cloud services. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, 193–208.

[3] Apple. 2022. ICloud Data Security Overview. https://support.apple.com/en-us/HT202303. Accessed on 10/15/2023. (2022).

[4] Avast. [n. d.] 83% of Americans are Using Weak Passwords. https://press.avast.com/83-of-americans-are-using-weak-passwords. Accessed on 10/15/2023. ().

[5] Josh Blum et al. 2023. E2e encryption for zoom meetings. White Paper – Github Repository zoom/zoom-e2e-whitepaper, Version 4.2, https://github.com/zoom/zoom-e2e-whitepaper/tree/v4.2. (2023).

[6] Dan Boneh, Manu Drijvers, and Gregory Neven. 2018. Compact multi-signatures for smaller blockchains. In *International Conference on the Theory and Application of Cryptology and Information Security*. Springer, 435–464.

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

[8] Melissa Chase, Georg Fuchsbauer, Esha Ghosh, and Antoine Plouviez. 2022. Credential transparency system. In *Security and Cryptography for Networks*. Clemente Galdi and Stanislaw Jarecki, (Eds.) Springer International Publishing, Cham, 313–335. ISBN: 978-3-031-14791-3.

[9] Brian Chen, Yevgeniy Dodis, Esha Ghosh, Eli Goldin, Balachandar Kesavan, Antonio Marcedone, and Merry Ember Mou. 2022. Rotatable zero knowledge sets: post compromise secure auditable dictionaries with application to key transparency. In *Advances in Cryptology - ASIACRYPT 2022*. Full version: `https://eprint.iacr.org/2022/1264`. Springer International Publishing, Cham.

[10] Coinbase. [n. d.] Coinbase Wallet. `https://www.coinbase.com/wallet`. Accessed on 10/15/2023. ().

[11] Emma Dauterman, Henry Corrigan-Gibbs, and David Mazières. 2020. {Safetypin}: encrypted backups with {human-memorable} secrets. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, 1121–1138.

[12] Emma Dauterman, Danny Lin, Henry Corrigan-Gibbs, and David Mazières. 2023. Accountable authentication with privacy protection: The Larch system for universal login. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*. USENIX Association, Boston, MA, (July 2023), 81–98. ISBN: 978-1-939133-34-2. `https://www.usenix.org/conference/osdi23/presentation/dauterman`.

[13] Electrum. [n. d.] `https://electrum.org/`. Accessed on 10/15/2023. ().

[14] Exodus. [n. d.] `https://www.exodus.com/`. Accessed on 10/15/2023. ().

[15] Fireblocks. [n. d.] `https://fireblocks.com/`. Accessed on 10/15/2023. ().

[16] Mozilla Foundation. [n. d.] Security/binary transparency. `https://wiki.mozilla.org/Security/Binary_Transparency`. Accessed on 10/15/2023. ().

[17] Mozilla Foundation. 2022. Where are my logins stored? `https://support.mozilla.org/en-US/kb/where-are-my-logins-stored`. Accessed on 10/15/2023. (2022).

[18] Google. [n. d.] Binary transparency. `https://developers.google.com/android/binary_transparency/overview`. Accessed on 10/15/2023. ().

[19] Google. 2022. Get your bookmarks, passwords & more on all your devices. `https://support.google.com/chrome/answer/165139`. Accessed on 10/15/2023. (2022).

[20] Google. 2022. How Chrome protects your passwords. `https://support.google.com/chrome/answer/10311524`. Accessed on 10/15/2023. (2022).

[21] Google. [n. d.] Key transparency overview. `https://github.com/google/keytransparency/blob/master/docs/overview.md`. Accessed on 10/15/2023. ().

[22] Google. [n. d.] Online Security Survey: Google / Harris Poll. `https://services.google.com/fh/files/blogs/google_security_infographic.pdf`. Accessed on 10/15/2023. ().

[23] Richard Hansen and Vicente Silveira. 2022. Code verify: an open source browser extension for verifying code authenticity on the web. `https://engineering.fb.com/2022/03/10/security/code-verify/`. Accessed on 10/15/2023. (2022).

[24] Yuncong Hu, Kian Hooshmand, Harika Kalidhindi, Seung Jin Yang, and Raluca A. Popa. 2021. Merkle2: a low-latency transparency log system. In 285–303.

[25] IETF. [n. d.] Supply chain integrity, transparency, and trust (scitt). `https://datatracker.ietf.org/wg/scitt/about/`. Accessed on 10/15/2023. ().

[26] Eleftherios Kokoris Kogias, Enis Ceyhun Alp, Linus Gasser, Philipp Svetolik Jovanovic, Ewa Syta, and Bryan Alexander Ford. 2021. Calypso: private data management for decentralized ledgers. *Proceedings of the VLDB Endowment*, 14, CONF, 586–599.

[27] Aman Ladia. [n. d.] `http://amanladia.com/wp/zerowallet/`. ().

[28] LastPass. [n. d.] `https://lastpass.com`. Accessed on 11/16/2022. ().

[29] LastPass. [n. d.] Lastpass security incident. `https://blog.lastpass.com/2022/11/notice-of-recent-security-incident/`. Accessed on 10/15/2023. ().

[30] Ben Laurie, Adam Langley, and Emilia Kasper. 2013. Certificate Transparency. RFC 6962. (June 2013). DOI: `10.17487/RFC6962`.

[31] Julia Len, Melissa Chase, Esha Ghosh, Kim Laine, and Radames Cruz Moreno. 2023. OPTIKS: An Optimized Key Transparency System. Cryptology ePrint Archive, Paper 2023/1515. `https://eprint.iacr.org/2023/1515`. (2023). `https://eprint.iacr.org/2023/1515`.

[32] Harjasleen Malvai, Lefteris Kokoris-Kogias, Alberto Sonnino, Esha Ghosh, Ercan Oztürk, Kevin Lewi, and Sean Lawlor. 2023. Parakeet: practical key transparency for end-to-end encrypted messaging. *Cryptology ePrint Archive*.

[33] Deepak Maram et al. 2021. Candid: can-do decentralized identity with legacy compatibility, sybil-resistance, and accountability. In *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1348–1366.

[34] Marcela S. Melara, Aaron Blankstein, Joseph Bonneau, Edward W. Felten, and Michael J. Freedman. 2015. CONIKS: bringing key transparency to end users. In *24th USENIX Security Symposium, USENIX Security 2015*. USENIX Association, Washington, D.C., (Aug. 2015), 383–398. ISBN: 978-1-939133-11-3. `https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/melara`.

[35] MetaMask. [n. d.] `https://metamask.io/`. Accessed on 10/15/2023. ().

[36] Kirill Nikitin, Eleftherios Kokoris-Kogias, Philipp Jovanovic, Nicolas Gailly, Linus Gasser, Ismail Khoffi, Justin Cappos, and Bryan Ford. 2017. {Chainiac}: proactive {software-update} transparency via collectively signed skipchains and verified builds. In *26th USENIX Security Symposium (USENIX Security 17)*, 1271–1287.

[37] Chris Orsini, Alessandra Scafuro, and Tanner Verber. 2023. How to recover a cryptographic secret from the cloud. Cryptology ePrint Archive, Paper 2023/1308. `https://eprint.iacr.org/2023/1308`. (2023). `https://eprint.iacr.org/2023/1308`.

[38] PAD Protocol. [n. d.] `https://www.padprotocol.org/`. Accessed on 10/15/2023. ().

[39] Andy Sayler, Taylor Andrews, Matt Monaco, and Dirk Grunwald. 2016. Tutamen: a next-generation secret-storage platform. In *Proceedings of the Seventh ACM Symposium on Cloud Computing* (SoCC '16). Association for Computing Machinery, Santa Clara, CA, USA, 251–264. ISBN: 9781450345255. DOI: `10.1145/2987550.2987581`.

[40] Leona Tam, Myron Glassman, and Mark Vandenwauver. 2010. The psychology of password management: a tradeoff between security and convenience. *Behaviour & IT*, 29, (May 2010), 233–244. DOI: `10.1080/01449290903121386`.

[41] Nirvan Tyagi, Ben Fisch, Andrew Zitek, Joseph Bonneau, and Stefano Tessaro. 2022. VeRSA: verifiable registries with efficient client audits from RSA authenticated dictionaries. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*. ACM.

[42] Ioanna Tzialla, Abhiram Kothapalli, Bryan Parno, and Srinath Setty. 2022. Transparency dictionaries with succinct proofs of correct operation. In *Proceedings of the ISOC Network and Distributed System Security Symposium (NDSS)*. (Feb. 2022).

[43] Trust Wallet. [n. d.] `https://trustwallet.com/`. Accessed on 10/15/2023. ().

[44] Dan Wesley, Saisang Cai, Andrea Courtright, and Andrea Barr. 2022. Microsoft edge password manager security. `https://learn.microsoft.com/en-us/deployedge/microsoft-edge-security-password-manager-security`. Accessed on 10/15/2023. (2022).

[45] Shalanda Young. 2022. Moving the U.S. government toward zero trust cybersecurity principles. `https://www.whitehouse.gov/wp-content/uploads/2022/01/M-22-09.pdf`. Accessed on 10/15/2023. (2022).

[46] ZenGo. [n. d.] `https://zengo.com/security-in-depth/`. Accessed on 10/15/2023. ().

APPENDIX

A Detailed Construction

Our construction is based on a CCA-secure public key encryption scheme, CCA-secure symmetric key encryption scheme and a binding and hiding commitment scheme. In more details, the algorithms are instantiated as follows:

- $\mathsf{GuardianKeyGen}(1^k)$: Generate and output a public key pair.
- $\mathsf{ServerInit}(1^k)$: The server calls $\mathsf{Init}()$ for the ledger functionality, gets back *params* and returns *params*.
- $\mathsf{UserStoreSecret}(s, u, policy, gpk_1, gpk_2)$: Generate a symmetric key sk_s, and use it to encrypt the secret s, forming ct_s. Secret share sk_s into $\mathsf{sk}_1, \mathsf{sk}_2$, *i.e.*, such that $\mathsf{sk}_1 \oplus \mathsf{sk}_2 = \mathsf{sk}_s$. Encrypt the two shares for the guardians, *i.e.*, $(\mathsf{sk}_1, u, policy, \mathsf{comrand}_1)$ under gpk_1 to get ct_1 and $(\mathsf{sk}_2, u, policy, \mathsf{comrand}_2)$ under gpk_2 to get ct_2, where $\mathsf{comrand}_i$ is randomness picked for a binding and hiding commitment scheme. Compute $h_1 = \mathsf{com}(\mathsf{sk}_1, \mathsf{comrand}_1), h_2 = \mathsf{com}(\mathsf{sk}_2, \mathsf{comrand}_2)$. Set $blob_i = ct_i||h_i$. Return $st = (u, storedblob), storedblob = (blob_1||blob_2||ct_s)$
- $\mathsf{ServerStoreandLog}(st_s, (u, storedblob))$: The server saves $(u, storedblob)$ in st_u. It first calls the ledger functionality with $\mathsf{Store}(u||\mathtt{storage}, storedblob)$. Then it calls the ledger functionality with $\mathsf{GetLookupHandle}(u||\mathtt{storage})$ and gets back handle. It sets $receipt = \mathsf{handle}$.
- $\mathsf{UserVerifyStorage}(st, receipt)$: The user parses st as $(u, storedblob)$ and $receipt$ as handle. Then, it

calls into ledger with command $\mathsf{Lookup}(\mathsf{handle})$. If it gets back $(x, \vec{v}) \neq \perp$, it does the following checks: (1) $x = u||\texttt{storage}$ (2) $|\vec{v}| = 1$ and (3) $\vec{v}[0] = \mathit{storedblob}$. If all the checks pass, return $\mathsf{success}$. Else, return $\mathsf{failure}$.

- $\mathsf{UserRequestRecovery}(u, \mathit{context})$: Generate a public key pair $(\mathsf{pk}_u, \mathsf{sk}_u)$. Output $st_u = (u, \mathsf{sk}_u, \mathsf{pk}_u, \mathit{context})$ as the state to store and $\mathit{msg} = (\mathsf{pk}_u, \mathit{context})$ as the request message to send to the service provider.

- $\mathsf{ServerProcessRequest}(st_s, (u, \mathit{msg}))$: It first calls the ledger functionality with $\mathsf{Store}(u||\texttt{recovery}, \mathit{msg})$. Then it calls the ledger functionality with $\mathsf{GetLookupHandle}(u||\texttt{recovery})$ to get $\mathsf{handle}_{\mathsf{recovery}}$ and with $\mathsf{GetLookupHandle}(u||\texttt{storage})$ to get $\mathsf{handle}_{\mathsf{storage}}$. Finally it looks up the $\mathit{storedblob}$ associated with u and parses $\mathit{storedblob} = (\mathit{blob}_1||\mathit{blob}_2||ct_s)$. It outputs $st'_s = st_s$, $\mathsf{servermsg}_1 = (\mathit{blob}_1||\mathsf{handle}_{\mathsf{recovery}})$, $\mathsf{servermsg}_1 = (\mathit{blob}_2||\mathsf{handle}_{\mathsf{recovery}})$, and $\mathit{receipt} = (\mathit{storedblob}||\mathsf{handle}_{\mathsf{recovery}}||\mathsf{handle}_{\mathsf{storage}})$.

- $\mathsf{GuardianResponse}(\mathit{gsk}, u, \mathit{msg}, \mathsf{servermsg})$: The guardian parses $\mathsf{servermsg}$ as $\mathit{blob}||\mathsf{handle}$ and parses $\mathit{blob} = ct||h$. It then decrypts ct using the guardian's secret key gsk to get $(\mathsf{sk}, u', \mathit{policy}, \mathsf{comrand})$. If $u' \neq u$ it outputs $\perp, \perp$. It then calls into the ledger functionality with command $\mathsf{Lookup}(\mathsf{handle})$. If this returns $(x, \vec{v}) \neq \perp$, check that $x = u||\texttt{recovery}$ and $\mathit{msg} \in \vec{v}$. If the checks do not verify, return $\perp, \perp$ and stop. Otherwise it encrypts $(\mathsf{sk}, \mathsf{comrand})$ under the public key pk_u from the request message msg to get ct_u. It sets $\mathit{decblob} = ct_u$ and outputs $\mathit{policy}, \mathit{decblob}$.

- $\mathsf{UserCompleteRecovery}(st_u, \mathit{decblob}_1, \mathit{decblob}_2, \mathit{receipt})$: The user parses st_u as $u, \mathsf{sk}_u, \mathsf{pk}_u$ and $\mathit{receipt}$ as $\mathit{storedblob}, \mathsf{handle}_{\mathsf{recovery}}, \mathsf{handle}_{\mathsf{storage}}$. It calls the ledger functionality with $\mathsf{Lookup}(\mathsf{handle}_{\mathsf{storage}})$. If it gets back $x, \vec{v}$, it checks if $x = u||\texttt{storage}$ and $\vec{v}[0] = \mathit{storedblob}$. Then it calls the ledger functionality with $\mathsf{Lookup}(\mathsf{handle}_{\mathsf{recovery}})$. If it gets back $x, \vec{v}$, it checks if $x = u||\texttt{recovery}$ and $(\mathit{context}, \mathsf{pk}_u) \in \vec{v}$.

  If all the checks pass, the user proceeds to decrypt $\mathit{decblob}_i$ using sk_u to obtain $(\mathsf{sk}_i, \mathsf{comrand}_i)$. It parses $\mathit{storedblob}$ as $\mathit{blob}_1||\mathit{blob}_2||ct_s$, and blob_i as $ct_i||h_i$. The user checks if $h_i = \mathit{com}(\mathsf{sk}_i, \mathsf{comrand}_i)$. If all the checks pass, the user computes $\mathsf{sk} \leftarrow \mathsf{sk}_1 \oplus \mathsf{sk}_2$ and uses sk to decrypt ct_s to get s back.

- $\mathsf{GetUserRecoveryHistory}(st_s, u)$: The server calls the ledger API with the command $\mathsf{GetLookupHandle}(u||\texttt{recovery})$. If it gets back $(\mathsf{handle}, \vec{v})$ from the functionality, it sets $\mathit{contexts} := \mathsf{contexts}(\vec{v})$ and $\mathit{receipt} := \mathsf{handle}$, where $\mathsf{contexts}$ is the function which outputs a vector containing the context field in each tuple in v.

- $\mathsf{VerifyRecoveryHistory}(u, \vec{\mathit{msg}}, \mathit{receipt})$:

  The user parses $\mathit{receipt}$ as handle and calls the ledger functionality with $\mathsf{Lookup}(\mathsf{handle})$. If it gets back $x, \vec{v}$, it checks that $x = u||\texttt{recovery}$ and $\mathit{contexts} = \mathsf{contexts}(\vec{v})$ where $\mathsf{contexts}$ is as described above.

B Security Proof Sketches

We give sketches for the security proofs here and defer the formal proofs to the next section.

User Privacy. Follows directly from the ledger abstraction. We note that in both ledger instantiations we consider, the leakage and other information that the adversary sees depends only on the adversarial user's actions . Thus, the adversary cannot learn anything about specifically which actions honest users are requesting, or which honest users are making requests.

Confidentiality. Follows directly from the ledger abstraction as well. To see this, note that the user's secret is only used to form ct_s, which is stored in the ledger (as part of $\mathit{storedblob}$), but not passed to the guardian. It is then returned to the user, who performs a bunch of checks, then decrypts ct_s and outputs the resulting secret. This means that the only way that an adversarial user or guardian could get information about the secret is if something about ct_s were to leak through the ledger functionality. However, in both our ledger instantiations, the leakage is such that the adversary will learn nothing about the values stored on the ledger unless it is given the handle to lookup the corresponding label; this never happens in our construction (as long as the server is honest) so confidentiality of the secret is preserved.

Transparency. This is based on CCA security of the PKE, CCA security of the SKE, security (hiding and binding) of the commitment scheme and by the definition of the ledger functionality. Very roughly, the ledger functionality guarantees that if an adversarial recovery request for the honest user's secret is given to the guardian and the guardian is able to verify that it has been logged, then that request will appear in the list of requests when the user later does a VerifyRecoveryHistory call. (Again by the ledger functionality, since the user checks in UserCompleteRecovery that their requests is correctly logged, we are guaranteed that all of the user's requests also appear in the log - this means that any adversarial request will appear as additional.) CCA security of the SKE and PKE are used to argue that the ciphertexts in the honest user's *storedblob* and in the *decblob* produced by the honest guardian hide their contents, even when the adversary can potentially pass in malicious values for decryption. The binding property of the commitment combined with the fact that *storedblob* is recorded on the ledger guarantees that the user can't be convinced to use an incorrect key share (this is important as a decrypting with an incorrect key share would correspond to a related key attack), and hiding guarantees that adding the commitment doesn't reveal anything about the key share.

Policy Enforcement. This is based on CCA security of the PKE, CCA security of the SKE, security (hiding and binding) of the commitment scheme and by the definition of the ledger functionality. As in Transparency, CCA security of the SKE and PKE are used to argue that the ciphertexts in the honest user's *storedblob* and in the *decblob* produced by the honest guardian hide their contents, even when the adversary can potentially pass in malicious values for decryption. Here we also use CCA security of the PKE to guarantee that either the guardian gets the correct policy, or he gets a completely unrelated key share. The binding property of the commitment combined with the fact that *storedblob* is recorded on the ledger guarantees that the user can't be convinced to use an incorrect key share (this is important as a decrypting with an incorrect key share would correspond to a related key attack), and hiding guarantees that adding the commitment doesn't reveal anything about the key share.

History consistency. This follows directly from the Ledger functionality. VerifyRecoveryHistory(u, *contexts*, *receipt*) only accepts *contexts* if it is consistent with the latest $\vec{v}$ stored in the ledger under $u||$`recovery`. According to the ledger, the $\vec{v}$ can only change by having new elements added to the end . This means that *contexts* accepted in earlier queries must be a prefix of the contexts accepted in later queries. History consistency follows directly.

Robustness. Follows from binding property of the commitment scheme, correctness of the SKE encryption, and the definition of the ledger functionality. The ledger guarantees that if UserVerifyStorage sees that *storedblob* has been correctly stored, then UserCompleteRecovery will see the same *storedblob* (or output an error). The binding property of the commitment guarantees that the user will always use the correct key to decrypt the ct_s in *storedblob* (or output an error), and the correctness of the encryption guarantees that this will produce the correct result.

C Formal Security Definitions and Proofs

C.1 Security Definitions

Here we formally define the security properties of Acsesor.

C.1.1 User Privacy

User privacy considers the case where both the server and the guardians are honest, and aims to guarantee that the malicious users learn nothing about the honest users' actions.[3] To capture this, we give the adversary access to an oracle to which he can submit pairs of users and actions (either new secret registrations or secret recovery requests on behalf of users of his choice), and depending on a hidden bit, the game with either take the first or the second user/action in every pair. To prevent trivial attacks the actions are structured so that the adversary can never produce an invalid sequence of actions. Finally, we also provide oracles to allow the adversary to interact with the service on behalf of malicious users. The scheme provides anonymity if the adversary has negligible advantage in producing the hidden bit.

- Pick a bit $b \leftarrow \{0,1\}$. Initialize empty tables T_1, T_2
- $\mathsf{O}_{\mathsf{GuardianKeyGen}}()$:
  - If $gpk_1, gpk_2 \in T_1$ return $\perp$.
  - Otherwise, run $\mathsf{GuardianKeyGen}(1^\lambda) \rightarrow (gsk_1, gpk_1)$ and $\mathsf{GuardianKeyGen}(1^\lambda) \rightarrow (gsk_2, gpk_2)$.
  - Store $(gsk_1, gpk_1), (gsk_2, gpk_2) \in T_1$.
  - Return (gpk_1, gpk_2)
- $\mathsf{O}_{\mathsf{ServerInit}}()$:
  - If $params \in T_1$, return $\perp$.
  - Otherwise, run $\mathsf{ServerInit}(1^\lambda) \rightarrow params$. Save $st_S, params$ in T_1 and return $params$.
- $\mathsf{O}_{\mathsf{UserStoreSecret}OR\mathsf{UserRequestRecovery}}((u_0, s_0, policy_0), (u_1, s_1, policy_1))$
  - Set $u := u_b, s := s_b, policy := policy_b$.
  - If $u \notin T_2$, add u to T_2 and parse $\mathsf{payload}$ as $s, policy$.
  - Run $\mathsf{UserStoreSecret}(u, s, policy, gpk_1, gpk_2) \rightarrow (st_u, storedblob)$.
  - Run $\mathsf{ServerStoreandLog}(st_S, (u, storedblob)) \rightarrow (st'_S, receipt)$.
  - Run $\mathsf{UserVerifyStorage}(st_u, receipt)$. Let it return $\mathsf{answer} = \mathsf{success/failure}$.
  - Else, if $u \in T_2$:
  - Run $\mathsf{UserRequestRecovery}(u) \rightarrow (st_u, msg)$
  - Run $\mathsf{ServerProcessRequest}(st_S, (u, msg)) \rightarrow (receipt, \mathsf{servermsg}_1, \mathsf{servermsg}_2)$
  - Run $\mathsf{GuardianResponse}(gsk_i, u, msg, \mathsf{servermsg}_i) \rightarrow (policy, decblob_i)$ for $i \in \{1,2\}$
  - Run $\mathsf{UserCompleteRecovery}(st_u, decblob_1, decblob_2, receipt)$. Let $\mathsf{answer} = 1$ if the algorithm returns s' and $\mathsf{answer} = 0$ if the algorithm returns $\perp$.
  - Return answer.
- $\mathsf{O}_{\mathsf{ServerStoreandLog}}(u, stroredblob)$ If $u \in T_2$ return $\perp$. Otherwise, run $\mathsf{ServerStoreandLog}(st_S, (u, storedblob) \rightarrow (st'_S, receipt)$. Return $receipt$.
- $\mathsf{O}_{\mathsf{ServerProcessRequest}}(u, msg)$ If $u \in T_2$ return $\perp$. Otherwise, run $\mathsf{ServerProcessRequest}(st_S, (u, msg)) \rightarrow (receipt, \mathsf{servermsg}_1, \mathsf{servermsg}_2)$. Then, run $\mathsf{GuardianResponse}(gsk_i, u, msg, \mathsf{servermsg}_i) \rightarrow (policy, decblob_i)$ for $i \in \{1,2\}$. Return $(receipt, decblob_1, decblob_2)$.
- $\mathsf{O}_{\mathsf{GetUserRecoveryHistory}}(u)$ If $u \in T_2$ return $\perp$. Otherwise, run $\mathsf{GetUserRecoveryHistory}(st_S, u) \rightarrow (\vec{msg}, receipt)$. Return $(\vec{msg}, receipt)$.
- $\mathsf{O}_{\mathsf{Finalize}}(b')$: If $b' = b$, the game outputs 1 and stops. Otherwise it outputs 0 and stops.

[3]They learn only that an action has occurred, not what action it was nor who requested it.

C.1.2 Confidentiality

In this game, we model the confidentiality of the user's secret when the server is honest and the server's access control can prevent adversarial recovery requests but where both the guardians are malicious. Informally, we want to capture the fact, that, in the case where the service provider is honest, nobody (including other malicious users) should be able to learn a secret stored by an honest user. In the following game, the adversary has access to the following oracles. Its goal is to guess the bit b which either encrypts the honest user's (u^*) real secret, or encrypts some random string. We say that the adversary wins the game if it can guess the bit with non-negligible advantage over 1/2.

$\mathsf{O}_{\mathsf{Init}}()$ (Can be invoked only once):

- Run $\mathsf{ServerInit}(1^k) \to params, st_s$
- $\mathsf{recoveryst} = \perp$
- Pick $b \leftarrow \{0,1\}$
- Output $params$

$\mathsf{O}_{\mathsf{HonestUserStoreSecret}}(s, u^*, policy, gpk_1, gpk_2)$ (this can only be run once):

- Store u^*
- If $b = 0$: $\mathsf{UserStoreSecret}(s, u^*, policy, gpk_1, gpk_2) \to (storedblob^*, st)$
- Else if $b = 1$: $r \leftarrow \{0,1\}^*$, $\mathsf{UserStoreSecret}(r, u^*, policy, gpk_1, gpk_2) \to (storedblob^*, st)$
- Run $\mathsf{ServerStoreandLog}(st_s, (u^*, storedblob^*)) \to (st'_s, receipt)$ and set $st_s = st'_s$
- Return $\mathsf{UserVerifyStorage}(st, receipt)$.

$\mathsf{O}_{\mathsf{MaliciousUserStoreSecret}}(u, storedblob)$:

- If $u = u^*$: return $\perp$
- Run $\mathsf{ServerStoreandLog}(st_s, (u, storedblob)) \to (st'_s, receipt)$ and set $st_s = st'_s$
- Return $receipt$.

$\mathsf{O}_{\mathsf{HonestUserRequestRecovery}}()$:

- Run $\mathsf{UserRequestRecovery}(u^*) \to (st_{u^*}, msg)$
- Run $\mathsf{ServerProcessRequest}(st_s, (u^*, msg)) \to (st'_s, \mathsf{servermsg}_1, \mathsf{servermsg}_2, receipt)$ and set $st_s = st'_s$
- Store $\mathsf{recoveryst} = (receipt, st_{u^*})$
- Output $msg, \mathsf{servermsg}_1, \mathsf{servermsg}_2$

$\mathsf{O}_{\mathsf{HonestUserCompleteRecovery}}(decblob_1, decblob_2)$:

- If $\mathsf{recoveryst} = \perp$, return $\perp$. Otherwise, set $(receipt, st_{u^*}) = \mathsf{recoveryst}$. Set $\mathsf{recoveryst} = \perp$
- Run $\mathsf{UserCompleteRecovery}(st_u, decblob_1, decblob_2, receipt) \to s'$
- Output $s' == \perp$

$\mathsf{O}_{\mathsf{MaliciousUserRequestRecovery}}(u, msg, gpk_1, gpk_2)$:

- Abort if $u = u^*$
- Run $\mathsf{ServerProcessRequest}(st_s, (u, msg)) \to (st'_s, \mathsf{servermsg}_1, \mathsf{servermsg}_2, receipt)$ and set $st'_s = st_s$
- Return $\mathsf{servermsg}_1, \mathsf{servermsg}_2, receipt$

C.1.3 Transparency

Next, we want to investigate the security and confidentiality guarantees we gain when the server is compromised. Clearly, if both guardians are compromised as well, we cannot hope to achieve any security or confidentiality. Note however, that, as long as a threshold number of the guardians are honest (*i.e.*, at least 1 out of the 2), we can ensure that the adversary cannot get the secret unless the malicious request is logged.

$\mathsf{O}_{\mathsf{Init}}()$ (Can be invoked only once):

- Run $\mathsf{GuardianKeyGen}(1^k) \rightarrow (gpk_1, gsk_1)$
- Initialize empty list $\mathcal{L}$
- $b \leftarrow \{0,1\}$
- Initialize $caught = 0$, $\mathsf{storagesucceeded} = 0$, and $\mathsf{recoveryst}$
- Output gpk_1

$\mathsf{O}_{\mathsf{HonestUserStoreSecret}}(s, u^*, policy, gpk_2)$ (can only be invoked once):

- Store u^*
- If $b = 0$: $\mathsf{UserStoreSecret}(s, u^*, policy, gpk_1, gpk_2) \rightarrow (storedblob^*, st^*)$
- Else if $b = 1$: $r \leftarrow \{0,1\}^*$, $\mathsf{UserStoreSecret}(r, u^*, policy, gpk_1, gpk_2) \rightarrow (storedblob^*, st^*)$
- Store st^* and output $storedblob^*$

$\mathsf{O}_{\mathsf{UserVerifyStorage}}(receipt)$ (can only be invoked once, and must be after $\mathsf{O}_{\mathsf{HonestUserStoreSecret}}$ is invoked)

- Set $\mathsf{storagesucceeded} = \mathsf{UserVerifyStorage}(st^*, receipt)$.
- Return $\mathsf{storagesucceeded}$

$\mathsf{O}_{\mathsf{HonestUserRequestRecovery}}(context)$:

- If $\mathsf{storagesucceeded} = 0$, return $\perp$
- Run $\mathsf{UserRequestRecovery}(u^*, context) \rightarrow (st_{u^*}, msg)$
- Record $\mathsf{recoveryst} = st_{u^*}$
- Record $context$ in $\mathcal{L}$
- Output msg

$\mathsf{O}_{\mathsf{HonestGuardianResponse}}(u, msg, \mathsf{servermsg})$:

- Run $\mathsf{GuardianResponse}(gsk_1, u, msg, \mathsf{servermsg}) \rightarrow policy, decblob$
- Output $policy, decblob$

$\mathsf{O}_{\mathsf{HonestUserCompleteRecovery}}(decblob_1, decblob_2, receipt)$:

- If $\mathsf{storagesucceeded} = 0$ or $\mathsf{recoveryst} = \perp$, return $\perp$.
- Set $st = \mathsf{recoveryst}$ and $\mathsf{recoveryst} = \perp$.
- Run $\mathsf{UserCompleteRecovery}(st, decblob_1, decblob_2, receipt) \rightarrow s'$
- Output $s' == \perp$

$\mathsf{O}_{\mathsf{HonestVerifyRecoveryHistory}}(contexts, \pi)$ (must be run exactly once, and must occur after the last call to $\mathsf{O}_{\mathsf{HonestGuardianProcessRequest}}$)

- If $contexts \not\subseteq \mathcal{L}$ or $\mathsf{recoveryst} \neq \perp$, set $caught = 1$
- If $\mathsf{VerifyRecoveryHistory}(u^*, contexts, \pi) = 0$ set $caught = 1$

Game output: The game runs the adversary with the above oracles to obtain bit b'. If $caught = 1$, the game outputs a random bit, otherwise it outputs b'.

The scheme provides Transparency if the advantage of this game in producing the bit b is negligible.

C.1.4 Policy Enforcement

This is also a property that applies when at least one guardian is honest, but the server may be malicious. This property captures that the adversary cannot learn an honest user's secret unless there is an adversarial request for which the honest guardian determines that the user's policy has been satisfied.

$\mathsf{O}_{\mathsf{Init}}()$ (Can be invoked only once):

- Run $\mathsf{GuardianKeyGen}(1^k) \rightarrow (gpk_1, gsk_1)$

- Initialize empty list $\mathcal{L}$, empty vector $\mathsf{Decblobs}$, counter $j = 0$, state $\mathsf{recoveryst} = \perp$, and bit $\mathsf{storagesucceeded} = 0$
- $b \leftarrow \{0, 1\}$
- Output gpk_1

$\mathsf{O}_{\mathsf{HonestUserStoreSecret}}(s, u^*, policy, gpk_2)$ (this can only be run once):

- Store u^*
- If $b = 0$: $\mathsf{UserStoreSecret}(s, u^*, policy, gpk_1, gpk_2) \to (storedblob^*, st^*)$
- Else if $b = 1$: $r \leftarrow \{0, 1\}^*$, $\mathsf{UserStoreSecret}(r, u^*, policy, gpk_1, gpk_2) \to (storedblob^*, st^*)$
- Store st^* and output $storedblob^*$

$\mathsf{O}_{\mathsf{UserVerifyStorage}}(receipt)$ (can only be invoked once, and must be after $\mathsf{O}_{\mathsf{HonestUserStoreSecret}}$ is invoked)

- Set $\mathsf{storagesucceeded} = \mathsf{UserVerifyStorage}(st^*, receipt)$.
- Return $\mathsf{storagesucceeded}$

$\mathsf{O}_{\mathsf{HonestUserRequestRecovery}}()$:

- If $\mathsf{storagesucceeded} = 0$, return $\perp$
- Run $\mathsf{UserRequestRecovery}(u^*) \to (st_{u^*}, msg)$
- Add msg to $\mathcal{L}$ and store $\mathsf{recoveryst} = st_{u^*}$
- Output msg

$\mathsf{O}_{\mathsf{HonestGuardianExtractPolicy}}(u, msg, \mathsf{servermsg})$:

- Run $\mathsf{GuardianResponse}(gsk_1, u, msg, \mathsf{servermsg}) \to policy, decblob$
- Store $\mathsf{Decblobs}[j] = (msg, policy, u, decblob)$, set $j = j + 1$, and output $policy, j$

$\mathsf{O}_{\mathsf{HonestGuardianReleaseResponse}}(j)$ (this captures the case when the guardian determines the policy in the jth request has been satisfied):

- If $\mathsf{Decblobs}[j] = \perp$, return $\perp$. Otherwise, parse $\mathsf{Decblobs}[j] = (msg, policy, u, decblob)$
- If $msg \notin \mathcal{L}$ and $policy = policy^*$ and $u = u^*$, return $\perp$. (If there is an adversarial request for the honest user's secret, for which the honest guardian determines that the honest user's policy is satisfied, then we cannot guarantee any hiding for the secret. In fact, the adversary would in the be able to trivially win the game just based on the desired functionality. Thus, we will not allow such queries.)
- Else, return $\mathsf{Decblobs}[j]$

$\mathsf{O}_{\mathsf{HonestUserCompleteRecovery}}(decblob_1, decblob_2, msg, receipt)$:

- If $\mathsf{storagesucceeded} = 0$ or if $\mathsf{recoveryst} = \perp$, return $\perp$
- Set $st = \mathsf{recoveryst}$ and set $\mathsf{recoveryst} = \perp$
- Run $\mathsf{UserCompleteRecovery}(st, decblob_1, decblob_2, receipt) \to s'$
- Output $s' == \perp$

C.1.5 Robustness

Robustness ensures that, even when the server and one or both the guardians are malicious, a user will either be able to get their secret back, or detect that the secret was tampered with.

- Initialize empty tables T_1, T_2, T_3
- $\mathsf{O}_{\mathsf{HonestUserStoreSecret}}(s, u, policy, gpk_1, gpk_2)$:
  - If $u \in T_1$: return $\perp$. Else, proceed.

- Run $\mathsf{UserStoreSecret}(s, u, policy, gpk_1, gpk_2) \rightarrow (st_u, storedblob)$.
- Save $(u, s, st_u, storedblob)$ in T_1
- Return *storedblob*

- $\mathsf{O}_{\mathsf{UserVerifyStorage}}(u, receipt)$:
  - If $u \in T_1$: lookup the corresponding st_u. Otherwise, return $\perp$.
  - Run $\mathsf{UserVerifyStorage}(st_u, receipt)$. If it returns success, insert $(u, s, st_u, storedblob, \mathsf{success})$ to T_2. Return success.
  - Otherwise, return failure.

- $\mathsf{O}_{\mathsf{UserRequestRecovery}}(u)$
  - If $u \notin T_2$ or $\exists(u, st_u, msg) \in T_3$: return $\perp$.
  - Otherwise, run $\mathsf{UserRequestRecovery}(u) \rightarrow (st_u, msg)$. Store (u, st_u, msg) in T_3.
  - Return *msg*.

- $\mathsf{O}_{\mathsf{UserCompleteRecovery}}(u, decblob_1, decblob_2, receipt)$
  - If $u \notin T_3$: return $\perp$ and stop.
  - Otherwise, lookup (u, st_u, msg) from T_3.
  - run $\mathsf{UserCompleteRecovery}(st_u, decblob_1, decblob_2, receipt)$.
  - Delete (u, st_u, msg) from T_3.
  - If UserCompleteRecovery returns $\perp$, output $\perp$.
  - If the above algorithm returns s', lookup s corresponding to u from T_2. If $s \neq s'$, output 1 and stop.

C.1.6 History consistency

History consistency says that if a user u checks their recovery at two different times, the list of contexts given the first time must be a prefix of the list given the second time.

- Initialize empty table T, $error = 0$
- $\mathsf{O}_{\mathsf{HonestVerifyRecoveryHistory}}(u, contexts, receipt)$:
  - Run $\mathsf{VerifyRecoveryHistory}(u, contexts, receipt)$. If the output is failure, return $\perp$.
  - Else, if $T[u] = \perp$, set $T[u] = contexts$ and return 0
  - Else, if $T[u]$ is not a prefix of *contexts*, set $error = 1$ and return 1.
  - Else, set $T[u] = contexts$ and return 0.

The adversary wins if they can cause *error* to be set to 1.

C.2 Security Proofs

User Privacy Calls to $\mathsf{O}_{\mathsf{UserStoreSecret} OR \mathsf{UserRequestRecovery}}$ only respond with answer, which is 1 bit of information (whether the request succeeded or failed). For the other oracles, the adversary calls them with adversarially chosen users. The responses from those oracles could potentially leak information about the honest user updates, used in $\mathsf{O}_{\mathsf{UserStoreSecret} OR \mathsf{UserRequestRecovery}}$, which could help the adversary win the game. We first show that for our protocol this is not the case.

Recall that the *receipt* returned by these oracles, are, in the real protocol, the handles returned by the ledger functionality. The ledger functionality could return some additional leakage to the adversary. The other output from the oracles (*decblob*s and $\vec{msg}$) do not leak any information about the honest user updates.

For the Nimble based ledger instantiation, the functionality does not have any leakage. For the PAHD based ledger instantiation, the functionality does have additional leakage. However, the leakage is completely simulatable from the adversarially chosen users alone (see Section 3.2) except the update notifications. However, in the game, already knows the total number of updates, since it equals the number of calls to $\mathsf{O}_{\mathsf{UserStoreSecret}OR\mathsf{UserRequestRecovery}}$). So, this leakage does not reveal any information about the honest user updates.

Finally, we need to argue that the adversary cannot construct a sequence of queries to $\mathsf{O}_{\mathsf{UserStoreSecret}OR\mathsf{UserRequestRecovery}}$ such that the answer trivially distinguishes between u_0 and u_1. Note that, the adversary always has to call the oracle with $s, policy$. If this was the first query for the honest userhandle, then the oracle performs the flow for $\mathsf{UserStoreSecret}$. Otherwise it ignores the input $s, policy$ and treats it as a recovery request. Hence, by correctness of our scheme the adversary can never cause the oracle to output $\mathsf{answer} \neq 1$. This, the adversary cannot craft a sequence of queries to make the oracles behave differently for u_0 and u_1. This concludes out proof.

Confidentiality

G_0 Let G_0 be the honest server confidentiality game as described above for bit $b = 0$ instantiated with our protocol as described in Appendix A.

G_1 As in G_0, but in $\mathsf{HonestUserCompleteRecovery}$, omit the final step which decrypts ct_s to get s and instead output 1 iff all the checks pass.

Since decryption can never produce $\perp$, and the only thing we do with the decrypted value is to return it and then check whether it is $\perp$, this is identical.

G_2 As in G_1, but form ct_s as the encryption of a random r instead of s.

This is identical if we instantiate the ledger with either of the two solutions we discuss in Section 3.2. Note that, outside of the leakage, the ledger functionality never reveals anything about the values stored in $\vec{v}$ associated with an x except on queries for that x. That means that any calls to the ledger for adversarial users will not reveal information about the honest user's storedblob. The Nimble instantiation has no leakage, so the equivalence follows directly. As discussed above, the leakage on queries for the adversarial users can be simulated given only those queries, their response, and the update notifications sent by the ledger. In particular, it is independent of the values that are stored for the honest user.

G_3 Let G_0 be the honest server confidentiality game as described above for bit $b = 1$ instantiated with our protocol as described in Appendix A.

This is again identical for the same reason as G_0, G_1 above.

Transparency

G_0 Let G_0 be the transparency game as described above for bit $b = 0$ instantiated with our protocol as described in Appendix A.

G_1 As in G_0, but store a list $\mathcal{L}_{msg}$ of all of the messages msg generated in $\mathsf{HonestUserRequestRecovery}$. End the game and set $caught = 1$ if the adversary ever submits a message $msg \notin \mathcal{L}_{msg}$ for $u = u^*$ to $\mathsf{HonestGuardianResponse}$ and the same msg is later generated by oracle $\mathsf{HonestUserRequestRecovery}$, or if the same msg is generated in two different $\mathsf{HonestUserRequestRecovery}$ calls.

This is indistinguishable by CPA security of the PKE, which implies that the public key output by KeyGen *must have enough entropy, so this case will occur with only negligible probability.*

G_2 As in G_1, but if there is a message msg generated in $\mathsf{HonestUserRequestRecovery}$ call for which the adversary never makes a $\mathsf{Store}(u^*||\mathtt{recovery}, msg)$ call, set $caught = 1$.

We argue that this is identical to the previous game by definition of the ledger. Suppose that there is a message msg generated in $\mathsf{HonestUserRequestRecovery}$ *call for which the adversary never makes a*

$\mathsf{Store}(u^*||\mathtt{recovery}, msg)$ *call. In that case either the subsequent* HonestUserRequestRecovery *is never called (in which case* $\mathsf{recoveryst} \neq \perp$ *at the end of the game and caught is set to* 1*), or the* HonestUserRequestRecovery *call will retrieve* $(x, \vec{v})$ *where either* $x \neq u||\mathtt{recovery}$ *or* $msg \notin \vec{v}$ *in which case* UserRequestRecovery *will output* $\perp$ *and caught will be set to* 1.

G_3 As in G_2, but if the adversary ever submits a message $msg \notin \mathcal{L}$ for $u = u^*$ to HonestGuardianResponse and GuardianResponse does not output $\perp$, set a flag *advMsg*. If at the end of the game, $advMsg = 1$, then set $caught = 1$.

We argue that this is identical to the previous game, because if at the end of the game $advMsg = 1$*, then it must be the case in Game* G_1 *that* $caught = 1$ *as well.*
By the change in G_2*, we can focus on the case where for every message msg generated in* HonestUserRequestRecovery*, there is a corresponding* $\mathsf{Store}(u^*||\mathtt{recovery}, msg)$ *call. Since* GuardianResponse *does not output* $\perp$ *on the* $msg \notin \mathcal{L}$*, this means it retrieves* $(u||\mathtt{recovery}, \vec{v})$ *from the ledger where* $msg \in \vec{v}$*. Since the ledger vectors are append only by definition, and by the change in* G_1 *we can assume that msg will never by added to* $\mathcal{L}_{msg}$*; this means that the* HonestVerifyRecoveryHistory *call at the end of the game will retrieve* $(x, \vec{v})$ *where either* $x \neq u||\mathtt{recovery}$ *or* $(msg||\mathcal{L}_{msg}) \subseteq \vec{v}$*. In the first case* VerifyRecoveryHistory *will output* $\perp$ *and caught will be set to 1. In the second case, if* $\mathsf{contexts}(\vec{v}) \neq contexts$*, again* VerifyRecoveryHistory *will output* $\perp$ *and caught will be set to 1. The final case is if* $\mathsf{contexts}(msg||\mathcal{L}_{msg}) \subseteq \mathsf{contexts}(\vec{v}) = contexts$*. This means* $|\mathcal{L}| = |\mathcal{L}_{msg}| = |contexts| - 1$*. This means contexts* $\not\subseteq \mathcal{L}$ *and caught is again set to* 1.

G_4 As in G_3, but if the adversary ever submits a message $msg \notin \mathcal{L}$ for $u = u^*$ to HonestGuardianResponse and GuardianResponse does not output $\perp$, then set $caught = 1$ and end the game before generating ct_u, outputting a random bit.

Since, by the change in G_3*, if the adversary ever submits a message* $msg \notin \mathcal{L}$ *for* $u = u^*$ *to* HonestGuardianResponse *and* GuardianResponse *does not output* $\perp$*, then* $caught = 1$ *and the adversary's output is ignored. That means that it is equivalent if as soon as* $caught = 1$ *we stop running the adversary and end the game.*

G_5 As in G_4 but store $(\mathsf{sk}_1, u, policy, \mathsf{comrand}_1, ct_1)$ used in UserStoreSecret in $\mathsf{O}_{\mathsf{HonestUserStoreSecret}}$ as $(\mathsf{sk}_1^*, u^*, policy^*, \mathsf{comrand}_1^*, ct_1^*)$. (Note that $u^*, policy^*$ match those already stored by the game.) If ct_1^* appears as part of servermsg passed to $\mathsf{O}_{\mathsf{HonestGuardianResponse}}$, do not decrypt, but instead just return use the stored $(\mathsf{sk}_1^*, u^*, policy^*, \mathsf{comrand}_1^*)$.

This is indistinguishable by correctness of the symmetric encryption scheme.

G_6 As in G_5 but replace the ct_1 generated as part of UserStoreSecret in $\mathsf{O}_{\mathsf{HonestUserStoreSecret}}$ with an encryption of a random string.

This is indistinguishable by CCA security of the symmetric encryption scheme.

G_7 As in G_6, but if ct_1^* appears in the servermsg passed to $\mathsf{O}_{\mathsf{HonestGuardianResponse}}$ and the *msg* used in that call is in $\mathcal{L}$, store the resulting ct_u as ct_u^*. If ct_u^* appears in a *decblob* passed to $\mathsf{O}_{\mathsf{HonestUserCompleteRecovery}}$, do not decrypt, just use the stored $(\mathsf{sk}_1^*, \mathsf{comrand}_1^*)$.

This is indistinguishable by correctness of the PKE.

G_8 As in G_7, but if ct_1^* appears in the servermsg passed to $\mathsf{O}_{\mathsf{HonestGuardianResponse}}$ and the *msg* used in that call is in $\mathcal{L}$, generate the resulting ct_u by encrypting a random message and store it as ct_u^*.

Since $msg \in \mathcal{L}$ *means this is an encryption under an honestly generated public key, this is indistinguishable by CCA security of the PKE. (This technically requires hybrids or multi-user security, but is straightforward.)*

G_9 As in G_7, but if ct_1^* appears in the $\mathsf{servermsg}$ passed to $\mathsf{O}_{\mathsf{HonestGuardianResponse}}$, generate the resulting ct_u by encrypting a random message and store it as ct_u^*. (Omit the $msg \in \mathcal{L}$ check.)

Note that the guardian will check whether the u provided as input matches the stored u^*, which in turn matches the u^* in the game. If $\mathsf{GuardianResponse}$ *does not output $\perp$, but gets as far as generating ct_u, then this check succeeds, which means that $u = u^*$, and by the change we made in game G_4, if $msg \notin \mathcal{L}$ we will end the game before generating ct_u. Thus, if we get as far as generating ct_u, it must be the case that $msg \in \mathcal{L}$ and so removing the condition changes nothing.*

G_{10} As in G_9, but if the *storedblob* given as part of the input *receipt* in $\mathsf{HonestUserCompleteRecovery}$ is different from that generated in $\mathsf{HonestUserStoreSecret}$, immediately return $\perp$.

This is identical by the ledger functionality.

G_{11} As in G_{10} but in $\mathsf{HonestUserCompleteRecovery}$, instead of checking $h_1 = \mathsf{com}(\mathsf{sk}_1, \mathsf{randcom}_1)$, check if $ct_u = ct_u^*$.

These games are identical. Recall that if $ct_u = ct_u^*$ we use the stored values $\mathsf{sk}_1^*, \mathsf{randcom}_1^*$ *instead of decrypting, and by the change in game G_9, if we reach this check we know $h_1 = h_1^*$. So in game G_8, we check $h_1^* = \mathsf{com}(\mathsf{sk}_1^*, \mathsf{randcom}_1^*)$ which is true by construction. Thus, removing this check makes no diffence.*

G_{12} As in G_{11}, but in $\mathsf{HonestUserStoreSecret}$, replace h_1 with a $\mathsf{com}(0, \mathsf{randcom}')$ for freshly chosen $\mathsf{randcom}'$.

Note that $\mathsf{randcom}_1$ *is no longer used anywhere in game G_{11}, so this follows from hiding of the commitment*

G_{13} As in G_{12}, but store sk_2, h_2 generated in $\mathsf{HonestUserStoreSecret}$ as sk_2^*, h_2^*. If the checks pass in $\mathsf{HonestUserCompleteRecovery}$, but the resulting $\mathsf{sk}_2 \neq \mathsf{sk}_2^*$, abort.

Note that we have already argued that if the checks pass, storedblob = storedblob*, and hence $h_2 = h_2^*$ so this is indistinguishable by binding of the commitment scheme.

G_{14} As in G_{13} but store sk generated in $\mathsf{HonestUserStoreSecret}$ as sk^*. In $\mathsf{HonestUserCompleteRecovery}$, instead of decrypting with $\mathsf{sk}_1^* \oplus \mathsf{sk}_2$, decrypt with the stored sk^*.

Since by the change in G_{13} we have $\mathsf{sk}_2 = \mathsf{sk}_2^*$, *and by construction of* $\mathsf{UserStoreSecret}$ *we have* $\mathsf{sk}^* = \mathsf{sk}_1^* \oplus \mathsf{sk}_2^*$, *this is identical.*

G_{15} As in G_{14} but instead of generating $\mathsf{sk}_1^*, \mathsf{sk}_2^*$ as shares of sk^*, simply sample random sk_2^*.

Note that sk_1^* *is no longer used anywhere in game G_{14}, so this is identically distributed.*

G_{16} As in G_{15}, but in $\mathsf{HonestUserCompleteRecovery}$, if all checks pass up to the point of decrypting ct_s, simply return 0 (i.e. $s \neq \perp$)

This is indistinguishable by correctness of the SKE - decryption will never fail since the ciphertext was correctly encrypted under sk^*.

G_{17} As in G_{16}, but in $\mathsf{HonestUserStoreSecret}$, choose random r and compute ct_s as an encryption of r instead of s.

Note that sk *is now only used to generate ct_s, so the follows from CPA security of the SKE.*

G_{18} **to** G_{31} Repeat the above changes in reverse.

G_{32} The transparency game with $b = 1$.

Policy Enforcement

G_0 Let G_0 be the policy enforcement game as described above for bit $b = 0$ instantiated with our protocol as described in Appendix A.

G_1 As in G_0, but in $\mathsf{GuardianResponse}$, before computing ct_u, check $msg \notin \mathcal{L}$ and $policy = policy^*$ and $u = u^*$ and if so, set $\mathsf{Decblobs}[j] = \perp$, $j = j + 1$ and immediately return $policy, j$.

This is identical, since the game will in any case check this condition in $\mathsf{GuardianReleaseResponse}$ *before releasing* $\mathsf{Decblobs}[j]$ *and return* $\perp$ *if it is not true.*

G_2 As in G_1, but store the ct_1 formed in $\mathsf{HonestUserStoreSecret}$ as ct_1^*, and in $\mathsf{GuardianResponse}$, in addition to the checks added in G_1, also check whether $ct_1 = ct_1^*$.

This is indistinguishable by the correctness of the encryption scheme, since we already are checking whether the decrypted policy $= policy^*$ *and* ct_1^* *is an encryption of* $policy^*$.

G_3 As in G_2 but store $(\mathsf{sk}_1, u, policy, \mathsf{comrand}_1, ct_1)$ used in $\mathsf{UserStoreSecret}$ in $\mathsf{O}_{\mathsf{HonestUserStoreSecret}}$ as $(\mathsf{sk}_1^*, u^*, policy^*, \mathsf{comrand}_1^*, ct_1^*)$. (Note that $u^*, policy^*$ match those already stored by the game.) If ct_1^* appears as part of $\mathsf{servermsg}$ passed to $\mathsf{O}_{\mathsf{HonestGuardianResponse}}$, do not decrypt, but instead just return use the stored $(\mathsf{sk}_1^*, u^*, policy^*, \mathsf{comrand}_1^*)$.

This is indistinguishable by correctness of the symmetric encryption scheme.

G_4 As in G_3 but replace the ct_1 generated as part of $\mathsf{UserStoreSecret}$ in $\mathsf{O}_{\mathsf{HonestUserStoreSecret}}$ with an encryption of a random string.

This is indistinguishable by CCA security of the symmetric encryption scheme.

G_5 As in G_4, but if ct_1^* appears in the $\mathsf{servermsg}$ passed to $\mathsf{O}_{\mathsf{HonestGuardianResponse}}$, store the resulting ct_u as ct_u^*. If ct_u^* appears in a *decblob* passed to $\mathsf{O}_{\mathsf{HonestUserCompleteRecovery}}$, do not decrypt, just use the stored $(\mathsf{sk}_1^*, \mathsf{comrand}_1^*)$.

This is indistinguishable by correctness of the PKE.

G_6 As in G_5, but if ct_1^* appears in the $\mathsf{servermsg}$ passed to $\mathsf{O}_{\mathsf{HonestGuardianResponse}}$, generate the resulting ct_u by encrypting a random message and store it as ct_u^*.

Recall that, by the change in G_1, *before forming* ct_u *we check that* $msg \in \mathcal{L}$. *This means this is an encryption under an honestly generated public key, so* G_5 *and* G_6 *are indistinguishable by CCA security of the PKE. (This technically requires hybrids or multi-user security, but is straightforward.)*

G_7 As in G_6, but if the *storedblob* given as part of the input *receipt* in $\mathsf{HonestUserCompleteRecovery}$ is different from that generated in $\mathsf{HonestUserStoreSecret}$, immediately return $\perp$.

This is identical by the ledger functionality.

G_8 As in G_7 but in $\mathsf{HonestUserCompleteRecovery}$, instead of checking $h_1 = \mathsf{com}(\mathsf{sk}_1, \mathsf{randcom}_1)$, check if $ct_u = ct_u^*$.

These games are identical. Recall that if $ct_u = ct_u^*$ *we use the stored values* $\mathsf{sk}_1^*, \mathsf{randcom}_1^*$ *instead of decrypting, and by the change in game* G_7, *if we reach this check we know* $h_1 = h_1^*$. *So in game* G_8, *we check* $h_1^* = \mathsf{com}(\mathsf{sk}_1^*, \mathsf{randcom}_1^*)$ *which is true by construction. Thus, removing this check makes no diffence.*

G_9 As in G_8, but in $\mathsf{HonestUserStoreSecret}$, replace h_1 with a $\mathsf{com}(0, \mathsf{randcom}')$ for freshly chosen $\mathsf{randcom}'$.

Note that $\mathsf{randcom}_1$ *is no longer used anywhere in game* G_8, *so this follows from hiding of the commitment*

G_{10} As in G_9, but store sk_2, h_2 generated in $\mathsf{HonestUserStoreSecret}$ as sk_2^*, h_2^*. If the checks pass in $\mathsf{HonestUserCompleteRecovery}$, but the resulting $\mathsf{sk}_2 \neq \mathsf{sk}_2^*$, abort.

Note that we have already argued that if the checks pass, storedblob $=$ *storedblob**, *and hence* $h_2 = h_2^*$ *so this is indistinguishable by binding of the commitment scheme.*

G_{11} As in G_{10} but store sk generated in $\mathsf{HonestUserStoreSecret}$ as sk^*. In $\mathsf{HonestUserCompleteRecovery}$, instead of decrypting with $\mathsf{sk}_1^* \oplus \mathsf{sk}_2$, decrypt with the stored sk^*.

Since by the change in G_{10} *we have* $\mathsf{sk}_2 = \mathsf{sk}_2^*$, *and by construction of* $\mathsf{UserStoreSecret}$ *we have* $\mathsf{sk}^* = \mathsf{sk}_1^* \oplus \mathsf{sk}_2^*$, *this is identical.*

G_{12} As in G_{11} but instead of generating $\mathsf{sk}_1^*, \mathsf{sk}_2^*$ as shares of sk^*, simply sample random sk_2^*.

Note that sk_1^* *is no longer used anywhere in game* G_{11}*, so this is identically distributed.*

G_{13} As in G_{12}, but in $\mathsf{HonestUserCompleteRecovery}$, if all checks pass up to the point of decrypting ct_s, simply return 0 (i.e. $s \neq \perp$)

This is indistinguishable by correctness of the SKE - decryption will never fail since the ciphertext was correctly encrypted under sk^**.*

G_{14} As in G_{13}, but in $\mathsf{HonestUserStoreSecret}$, choose random r and compute ct_s as an encryption of r instead of s.

Note that sk *is now only used to generate* ct_s*, so the follows from CPA security of the SKE.*

G_{15} **to** G_{27} Repeat the above changes in reverse.

G_{28} The transparency game with $b = 1$.

Robustness

G_0 Same as the Robustness game, where all the algorithms are instantiated with our protocol implementation.

G_1 Same as before, except the following: In $\mathsf{UserCompleteRecovery}$, parse $receipt$ as $storedblob', \mathsf{handle}_{\mathsf{recovery}}, \mathsf{handle}_{\mathsf{storage}}$. Retrieve the row for u from T_2 to get $storedblob$. If $storedblob \neq storedblob'$, abort.

These two games are identical by the ledger $\mathcal{L}$ functionality.

G_2 Same as the previous game, except the following: in $\mathsf{UserCompleteRecovery}$, use $storedblob$ from T_2.

These two games are identically distributed.

G_3 Same as before, except the following: In $\mathsf{UserStoreSecret}$, when $\mathsf{sk}_1, \mathsf{sk}_2$ are generated, store them in T_1 and T_2 as well. More specifically, store $(u, s, \mathsf{sk}_1, \mathsf{sk}_2, st_u, storedblob)$ in T_1 and $(u, s, \mathsf{sk}_1, \mathsf{sk}_2, st_u, storedblob, \mathsf{success})$ in T_2.

These two games are identically distributed.

G_4 Same as before, except the following: In $\mathsf{UserCompleteRecovery}$, decrypt $decblob_i$ using sk_u to obtain $(\mathsf{sk}'_i, \mathsf{comrand}_i)$ for $i \in \{1, 2\}$. Check that $h_i = com(\mathsf{sk}_i, \mathsf{comrand}_i)$. In addition, check that $\mathsf{sk}'_1 = \mathsf{sk}_1, \mathsf{sk}'_2 = \mathsf{sk}_2$, where $\mathsf{sk}_1, \mathsf{sk}_2$ are stored in the row of u in T_2.

If an adversary can distinguish between G_3 and G_4, this means, we can use it to break the binding property of the commitment scheme.

G_5 Same as above, except the following: In $\mathsf{UserCompleteRecovery}$, compute $\mathsf{sk} \leftarrow \mathsf{sk}_1 \oplus \mathsf{sk}_2$ where $\mathsf{sk}_1, \mathsf{sk}_2$ are stored in the row of u in T_2. Use sk to decrypt ct_s.

These two games are identically distributed.

G_6 Same as above, except the following: instead of decrypting ct_s, directly output s where s is stored in the row of u in Table T_2.

These two games are identically distributed by the correctness of the symmetric encryption scheme SKE. This concludes our proof.

History Consistency This follows directly from the Ledger functionality. $\mathsf{VerifyRecoveryHistory}(u, contexts, receipt)$ only accepts $contexts$ if it is consistent with the latest $\vec{v}$ stored in the ledger under $u||\texttt{recovery}$. According to the ledger, the $\vec{v}$ can only change by having new elements added to the end . This means that $contexts$ accepted in earlier queries must be a prefix of the contexts accepted in later queries. History consistency follows directly.

Property	Server	Guardians
User Privacy	Honest	Honest
Confidentiality	Honest	can be Malicious
Policy-Enforcement	can be Malicious	≥ 1 Honest
Transparency	can be Malicious	≥ 1 Honest
Robustness	can be Malicious	can be Malicious
History Consistency	can be Malicious	can be Malicious

Table 2: This table captures the trust assumptions for each of the security properties. In all properties we assume some of the users in the system are malicious and can collude with the malicious parties.