Acsesor: A New Framework for Auditable Custodial Secret Storage and Recovery (Preliminary Version)

Melissa Chase
Microsoft Research
melissac@microsoft.com

Hannah Davis[§]
University of California, San Diego
h3davis@eng.ucsd.edu

Esha Ghosh
Microsoft Research
esha.ghosh@microsoft.com

Kim Laine
Microsoft Research
kim.laine@microsoft.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, among other dangers.**

Several password managers and cryptocurrency wallets today utilize non-custodial solutions, where their 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. One can argue that these solutions have a stronger security model, as the service provider no longer constitutes a single point of trust. However, the obvious downside is that it is very difficult for people to store cryptographic secrets reliably, making emergency recovery a serious problem.

We present Acsesor: a new 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. 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. Additionally, Acsesor retains the strong resilience guarantees that custodial systems provide against fraud attacks.

Finally, 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. For example, a slow recovery policy, where the guardians wait for a predefined time until responding, can replace the cumbersome passphrases many cryptocurrency wallets implement today for emergency recovery.

We also instantiate the Acsesor framework with a base protocol built of standard primitives: standard encryption schemes and privacy-preserving transparency ledgers. Our construction requires no persistent storage from its users and supports an expansive array of configuration options and extensions.

§. Work done at Microsoft Research

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 destroy 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 service providers are compromised or their data leaked, users may be left in the dark while their credentials are 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. For example, papers and storage devices may go missing or get stolen, people travelling or experiencing homelessness 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], [33] and vast amounts of wealth in cryptocurrency wallets [13], [41], [58], [15], [16] are protected by self-managed passwords, passphrases, and keys. Unfortunately, given the option, 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 [5]. 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 [24]. A study in Behavior and Information Technology [53] 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 example, very recently, in a security breach in LastPass, an unauthorized party was able to gain access to some of their customers' information. [34]. For some services, such as cryptocurrency wallets, custodial secret management is problematic due to the liability risks and anti-money laundering regulations that financial service providers 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 *service provider*, 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 2.4 and various policies in Section 4. They then use this information to encode their secret, and store the resulting blob with the service provider. When the user wants to recover their secret, they contact the service provider with their request; the service provider authenticates the user and then passes the request to the appropriate guardians who aid in recovery.

To protect against a service provider that tries to initiate a recovery attempt without the user's knowledge or an attacker that compromises the service provider's authentication, Acsesor requires the service provider to log all recovery requests in a transparent append-only log. The guardians are responsible for ensuring that request they are responding to is logged before responding to it. This approach is inspired by the recent advances in transparency technology (e.g., key transparency, binary transparency, software transparency, credential transparency) in the: industries [8], [23], [46], [22], [19], governments [60], standardization bodies [28] and academic literature [40], [10], [56], [57], [11].

Finally, Acsesor asks its users to periodically monitor their accounts (using their own devices) to detect any fraudulent recovery requests in the log. 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.

Benefits of Acsesor. 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 service provider.
  The service provider can utilize any standard security and fraud detection mechanisms to protect their users' accounts. This means they can use weak secrets like passwords or pins, allow account recovery via SMS or a live customer service call. The service provider 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. Secret management and recovery systems have many applications, and different users may have vastly different needs. Tolerance for latency varies substantially between applications, as does the degree of trust in the service provider and the availability and identities of trusted third parties. Users may want to allow different types of second factors, depending on the strength of their security/availability concerns and the availability of things like trusted hardware or devices; in the enterprise setting organizations might want to include their own requirements or tie into their own established identity systems. We designed Acsesor to accommodate all of this variation.
- *Security:* We can guarantee that even if the service provider acts maliciously, as long as an appropriate fraction of the guardians honest, the user's policy will be enforced, and all access requests will be logged.

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 service provider, so that it may be compatible with any existing identity systems.

In the registration phase, the user registers with the Acsesor service provider. 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 service provider for storage, along with the encrypted secret.

To recover their secrets, the user authenticates with the service provider and requests to initiate a recovery, at which time the service provider posts a receipt of the request in the transparent append-only log, along with a desired one-time public key the user has chosen. When the guardians see this request, they download the encrypted key pieces from the service provider, decrypt them, re-encrypt under the user's posted public key, and route them back to the user through the service provider. The user downloads the encrypted secret from the service provider, as well as the encrypted shares of their encryption key, reconstructs the key, and decrypts the secret.

The user will continuously monitor the transparent append-only log for potential fraudulent recovery attempts done on their behalf.

Acsesor policies To ensure no-one can post and complete a fraudulent recovery attempt while the actual secret owner is temporarily unavailable (*e.g.*, sleeping), Acsesor supports a user-configurable wait time policy. Upon registering their secrets with Acsesor, the user chooses a recovery policy that specifies a wait time the guardians must wait from the time the message was posted on the log to when they should proceed with their part of the recovery process. We believe realistic wait times could be anything from an hour or two to several months, depending on the scenario. A policy may also specify times when recovery may not proceed (e.g., the middle of the night in the user's time zone).

The idea of user-specified policies can be taken much further to provide flexibility to Acsesor. For example, most secret accesses would probably not be emergency recovery scenarios, in which case long wait times are impractical. Instead, Acsesor supports policies that require the user to provide additional evidence – a second layer of authentication – when initiating recovery. For example, a policy could specify a very short wait time *if* the user additionally proves the knowledge of a weak secret, such as a PIN or a password. The policy can further specify a rate limit (enforced by the guardians) on the recovery attempts with a weak secret to prevent guessing attacks. To protect against a scenario where they forget/lose access to the weak secret, they can additionally register a policy with a long wait time when no extra secret is provided or known.

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. For additional security, the policies can be executed within trusted execution environments (TEEs), such as Intel SGX.

1.2. Related Work

Several works have attempted to reduce the burden of trust on custodians using specialized hardware and threshold cryptography. For example, password-protected secret sharing (PPSS) [6], [29], [30] allows a secret to be shared across multiple custodians, removing any single points of failure. Access to the shares is protected with a password the user must memorize or store. The benefit is that the user can protect a high-entropy secret, such as a cryptographic secret key, with a low-entry secret (a password). If a sufficient threshold of these custodians become compromised, an offline password-guessing attack may be possible, but under normal conditions the custodians can enact standard security measures, such as two-factor authentication and rate limiting, to mitigate online password-guessing attacks.

PPSS as a primitive has a few drawbacks for our applications. Current PPSS schemes do not support logging of recovery attempts, and by design they do not offer recovery options if a user forgets their password. Another downside is that most PPSS schemes require the user to interact and authenticate directly with all their custodians. This property complicates 2FA, increases client communication costs, and makes fraud detection mechanisms more costly. A partial exception is "fully dynamic PPSS" [47], whose users do not need to know which of their guardians are online during recovery. In fully dynamic PPSS, after initial registration, a majority of the guardians can deregister, rotate, and enroll other guardians without user participation. Acsesor can also be "fully dynamic" by this definition.

Some cryptocurrency wallets use various flavors of secure multi-party computation to provide better security guarantees to their users. Fireblocks [18] uses distributed key generation to create secret key shares across multiple servers and threshold signatures to avoid ever having to bring together the users' secret key shares. For additional protection against malicious admins, it performs the threshold signatures computations within an Intel SGX enclave. ZenGo [62] similarly uses distributed key generation to create and store shares on ZenGo servers and the user's device. They utilize biometrics (face scan) and user's third-party cloud storage for emergency recovery. Similar to PPSS, these schemes also require users to authenticate to each server and do not have support for logging.

CALYPSO [32] presents an entirely decentralized secret document management system, based on blockchains and skipchains [44] as compared to the hybrid model of Acsesor (and inherits the challenges of a fully decentralized system as discussed above). It enables auditable and fair (atomic) release of valuable documents from one owner to another, after some kind of condition has been met. A blockchain logs the transactions (writes and reads) and a skipchain provides dynamic identity and access control for the participants. A committee of *trustees* holds shares of the decryption key for

the secret document and validates whether conditions for its release are met.

CanDID [38] is a decentralized identity system. In addition to multiple other issues with existing decentralized identity proposals, it addresses the problem of secret recovery. The system is built upon a decentralized network of nodes – the *CanDID committee* – which is also used for emergency recovery. Namely, the CanDID committee stores secret-shares of the user's secret and when presented with sufficient authentication evidence from legacy web services (such as the user's email provider), the committee can release the shares to the user. But CanDID does not have logging support.

SafetyPin [14] 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 [52] 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. Tutamen logs attempts and their associated certificates without assurance, but it also proposes as future work a publicly auditable log resembling certificate transparency [35].

The PAD Protocol [49] ("PAD" stands for *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. Access delegation is done by the secret owner sharing a cryptographic access token to the delegatee. Both the secret owner and the delegatee in this model must store long-lived keypairs for authentication. If the secret owner wants to track access to their secret, they must also store a mapping from access tokens to delegatees.

Transparency logs themselves have a long history, with numerous constructions and use-cases including key transparency [40], [10], [8], [23], [27], [57], [56], [12], binary transparency [22], [19], [50], credential transparency [11] and transparency in US courts' requests for access to tech companies' customer data [21]. We use the *Append-Only Zero-Knowledge Set* (aZKS) construction of [10] to build our transparency log, as it provides strong privacy properties.

2. The Acsesor Framework

We begin by describing the most basic version of our framework. Here we consider a user who will store their secret using two guardians. In this basic scheme, the user can only store one secret. We will discuss extensions in Section 4.

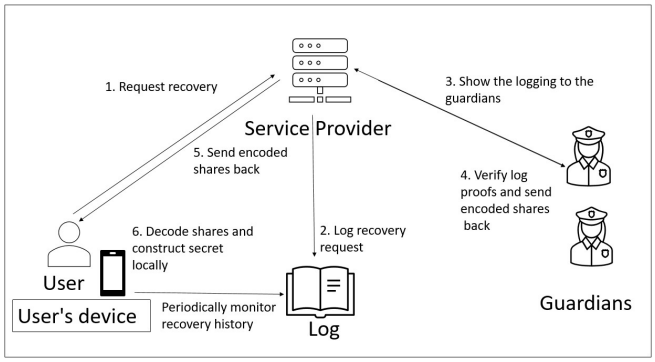

Figure 1. Recovery using Acsesor

The system operates as follows. 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 an initial commitment to an empty log that will be posted on the bulletin board. Each guardian will run GuardianKeyGen to generate it's own key pair. (See Section 2.4 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. While we assume that they must somehow authenticate to the service provider, this authentication may not be cryptographic, 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 collect a batch of such recovery requests from different users and add them to the log using ServerUpdate to produce a new commitment, which is posted to the bulletin board (Section 2.5).

As part of this process, the service provider also produces a proof that each request was included in the log. It will then forward each request message to the appropriate guardians along with the user's stored blob and a proof that the request has been added in the log. The guardians will retrieve the latest commitment from the bulletin board and run GuardianResponse to verify the proof and 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.

Monitoring the log. 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 proof generated using

GetUserRecoveryHistory that this list is complete with respect to the latest commitment posted on the bulletin board. The user will verify this proof using VerifyRecoveryHistory.

Auditing the log. The approach we take, where the users and guardians only need to retrieve the latest commitment from the bulletin board, requires some entity to verify that the commitments posted at different times are consistent with one another, *i.e.*, that the corresponding logs only grow and no entries are ever deleted. This is done by running AuditServerUpdate to verify the service provider's proof (generated in ServerUpdate) on each consecutive pair of commitments. This could be performed by users, the guardians, or a third party.

2.1. The Algorithms

In more detail, our system requires the following algorithms:

- $\mathsf{GuardianKeyGen}(1^k) \rightarrow (\mathsf{pk}, \mathsf{sk})$: 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.
- $\mathsf{ServerInit}(1^k) \rightarrow (st_s^0, com_0)$: The service provider will run this before the system starts to initialize its state. st_s^0 denotes the service provider's initial state and com_0 the first commitment that will be posted to the bulletin board.
- $\mathsf{UserStoreSecret}(s, u, policy, gpk_1, gpk_2) \rightarrow storedblob$: The user will run this algorithm to generate an 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 a required presentation of a second factor. These are other policies are discussed in Section 4.
- $\mathsf{UserRequestRecovery}(u) \rightarrow (st_u, msg)$: The user will run this algorithm to prepare a recovery request message msg. The only information required is their username u; st_u is a state that the user will later use to extract the secret from the response to their request.
- $\mathsf{ServerUpdate}(st_s, \Delta) \rightarrow (st'_s, com, \pi_{\mathsf{append}}, \Pi)$: The service provider runs this algorithm to update the recovery log to include the additional entries contained in Δ. Entries are of the form (u, msg_u) corresponding to each user's recovery request. The service provider's initial state is st_s, and its state after the update is st'_s. com is the new commitment to the log, and π is a proof that the new log extends the old one (append-only property). Finally, it generates a proof that each request in Δ has been included in the new log; the collection of all of these proofs is Π.
- $\mathsf{GuardianResponse}(\mathsf{sk}, u, i, msg, storedblob, com, \pi) \rightarrow (decblob, policy)$: 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, an index i representing whether this guardian corresponds to the user's gpk_1 or gpk_2, the recovery request message msg, the ciphertext ct generated when the user stored their secret, the latest commitment to the log com, and a proof π that the request (u, msg) was included in the log. It outputs a partially decrypted ciphertext $decblob$ and the policy $policy$ that the guardian should apply before releasing it.
- $\mathsf{UserCompleteRecovery}(st_u, decblob_1, decblob_2) \rightarrow s$ 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 and the partially decrypted ciphertexts produced by the two guardians. It produces either the secret s, or $\perp$ indicating that some error has occurred.
- $\mathsf{GetUserRecoveryHistory}(st_s, u) \rightarrow (\vec{msg}, \pi)$: The service provider 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 $\vec{msg}$ of request messages and a proof π that it is complete as of the latest state of the log.
- $\mathsf{VerifyRecoveryHistory}(u, \vec{msg}, com, \pi) \rightarrow 0$ 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, $\vec{msg}$ is the claimed list of requests, com is the commitment to the latest log, and π is the service provider's proof that this list is complete.
- $\mathsf{AuditServerUpdate}(com, com', \pi) \rightarrow 0$ or 1: This is run by an auditor (who could be user, a guardian, or a third party) to verify that a pair of commitments correspond to correctly extended logs. com commits to the log at the present epoch, com' commits to the log for the subsequent epoch, and π is the append proof proving that nothing was deleted in the update.

2.2. Security and Confidentiality Guarantees

For a user registering their secrets in an Acsesor system, the system guarantees certain properties as long as not all 3 parties (the service provider and the two guardians) are compromised:

Confidentiality of honest user's secrets. *(honest service provider)* If the service provider is honest and at least one of the guardians is honest then, even if the attacker controls the other guardian and the other users, as long as the attacker is

not able to compromise the service provider's authentication, the attacker will not be able to learn the honest user's secret.

(malicious service provider) If the service provider is malicious (or if the service provider's authentication is compromised), and at least one of the guardians is honest, then the attacker cannot compromise the honest user's secret until the following happens: the honest guardian releases a recovery response for an adversarially generated request, which it will only do when the user's release policy is satisfied.

Correctness of policy. If at least one of the guardians is honest, then if it is asked to recover and given the user's $storedblob$, it will correctly extract the user's intended policy.

Logging of recovery attempts. If at least one of the guardians is honest, every recovery attempt on an honest user's account will be logged before the user's secret is released. In other words, even if the attacker controls the server, the other guardian, and other users, any recovery attempt on an honest user's account will be logged and must be included the next time the user runs GetUserRecoveryHistory; otherwise the user will detect that the history is incomplete.

Private recovery history. Only the service provider can reveal the presence of any attempt on the log, and as long as it behaves honestly, only authenticated users and their designated guardians receive this information. This means, for example, that the commitments posted on the bulletin board and the interactions between the user and the service provider, and between and the guardians and the service provider, do not reveal any information about other users' usernames or recovery request patterns.

2.3. Discussion

Our approach does have a few drawbacks, but we believe they are outweighed by the advantages of Acsesor. Overcoming these challenges is something we leave open for future work.

- *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 seems worthwhile in exchange for the usability advantages discussed above.
  A more subtle point is that the service provider also gates access to the log: if the service provider does not respond, then the user will be unable to monitor the log. In this case, the user must consider their account as potentially compromised.
- *Transferring between devices:* Acsesor requires users to regularly monitor the log and report fraudulent recovery attempts. If users stay logged in on a device, this process can be automated, but if the device is lost, broken, or logs out, the user must proactively monitor from another device. While many users are likely to struggle with this behavioral shift, even if they fail to monitor the log, they are no worse off than with current custodial solutions, where compromise detection happens at the service provider's discretion. Furthermore, concerned users can optionally sacrifice convenience for security by adding a second factor to prevent fraudulent recovery in the absence of regular monitoring.
- *Notification fatigue.* Acsesor allows the user's device to notify them of any suspicious access requests, but it assumes that the user will pay attention to these notifications. Our hope is that these notifications will be rare enough, and the secrets important enough, that this is not too much of a concern.
- *Responding to possible compromise.* If the user does detect a recovery request they did not initiate and this detection happens before the delay period has passed, the user may have the opportunity to respond before the secret is actually released. The extent to which this is possible depends on the application. For example, users of a password manager built from Acsesor could change their passwords if they detect a fraudulent recovery attempt, or if Acsesor were used to support a cryptocurrency wallet the user could transfer their funds to a different address.
  In other applications, or if the user does not have a local copy of the secret available, then this may not be possible. However, even if there is no effective way to respond to secret compromise, we view our approach as valuable because it makes this compromise visible to the user. This also gives the service provider a reputational incentive not to compromise user accounts.
- *Compromise detection vs prevention* As a custodial system, Acsesor has some inherent vulnerability to a malicious service provider attempting a person-in-the-middle attack. This is one of the major obstacles to adoption for many custodial secret-management systems [48], and we address it in two ways. Our first countermeasure is the transparent log and customizable delay policy, which gives the user advance notification of any malicious activity on their account and the time to respond. The second countermeasure is an optional second factor that bypasses the service provider and authenticates the user directly to the guardians, thus preventing person-in-the-middle attacks. We discuss how this second factor can be used to customize the balance between recoverability and security in more detail in Section 4.

2.4. 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 3.4 for more details).

The guardians' operations are limited to decrypting and re-encrypting key shares and verifying the presence of recovery attempts in the service provider's log against the commitments published on the bulletin board. 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 more than $n - t$ guardians went offline in a burglary, fire, or flood, the secrets would be destroyed. 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.

Trusted execution environments. HSMs are probably not sufficiently flexible to provide the functionality of guardians, but Trusted Execution Environments (TEEs), such as Intel SGX enclaves, could. As with any secure system deployment relying on TEEs, the users would need to trust the implementation, which would likely mean at least open-sourcing it.

One benefit is that the service provider could run the TEEs on machines it controls. This would then provide a way to make the service fully custodial, in that it would not require the involvement of user devices or third parties, while also enforcing compromise detection.

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. In the TEE case, the service provider or TEE operator would need to post the public key of the TEE along with an attestation that it is running the appropriate code.

2.5. Bulletin Board

The bulletin board is a place for the service provider to commit to its state in a public and irreversible way. We make no prescriptions about its implementation, as long as it preserves the append-only property: once entries are added they can never be deleted. For a robust, failure-resistant instantiation, the bulletin board could be hosted on a blockchain. A simpler version would be a straightforward web server run by an independent third party, optionally mirrored by others. It is security-critical that the user and the guardians they choose maintain a consistent view of the bulletin board; this should be considered by any implementation. This is a standard assumption [8], [23], [46], [22], [19], [50], [31], [51], [40], [61], [45], [3], [10], [7], [54], [39], [27], [37], [55], [2], [56], [11], [57]

3. Our construction

We begin by giving an overview of our construction, then provide a more detailed description.

3.1. Construction overview

Roughly, our construction proceeds as follows:

Storing Secrets. To store a secret s, the user will pick a random key sk_s which they use to encrypt the secret s to form ct_s. They then secret shares that key (sk_s) into two, and encrypts one share to each guardian along with their username and the policy under which it should be released, forming ct_1, ct_2. The three ciphertexts (ct_1, ct_2, ct_s) are then stored at the server in the user's account.

Secret Recovery. When the user requests a recovery, it chooses a random key pair $(\mathsf{pk}_u, \mathsf{sk}_u)$, and sends the public key pk_u as the recovery request. The service provider logs this request, then forwards this public key along with the user's ciphertexts to the guardians. 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 and the policy under which the response should be released. It checks that this policy has been satisfied and that the request has been properly logged, before releasing the re-encrypted ciphertext. Finally, the user decrypts the ciphertexts in each of the guardian's responses, uses the results to reconstruct sk_s, and then uses that to decrypt and extract the stored secret.

Logging. What remains is to provide more information on how the logging occurs. The main challenge is to ensure that the server can efficiently prove to the user that they are seeing *all* of the recovery attempts, and to do that without revealing information about other users as part of the proof.

To do this, we build on Append-Only Zero-Knowledge Sets (aZKS) [10]. aZKS provides a way for a server to commit to a dictionary and prove membership of $(\mathsf{label}, \mathsf{value})$ pairs in that dictionary, or output $(\mathsf{label}, \perp)$ if the label does not appear in the dictionary. The server can also provide "append" proofs that given two commitments, the dictionary corresponding to one commitment is a subset of the other. Finally, these proofs are zero-knowledge in the sense that membership and non-membership proofs reveal nothing about the other entries in the database, and append proofs

reveal only the number of new entries that were added. An aZKS can be implemented using a sparse Merkle Tree and a Verifiable Random Function (VRF) [42].

What we need is somewhat more than a dictionary - we need to be able to map a username to a list of values (recovery requests), in such a way that we can later add to this list, and the user can know that they are seeing the entire list. To do this we adopt an approach from the key transparency literature [8], [23], [40], [10], [12], [56], [27], [57]: when the first recovery request for u is logged, we add $(u||1)$ to the dictionary, when the second recovery request for that username is added, we add $(u||2)$ etc. Then, to prove that the user is being shown the entire history, the service provider will prove membership for $(u||i)$ for each request, and then prove non-membership for the next one (i.e. if there have been n requests, the service provider will show non-membership for $(u||n+1)$). Similarly, when the guardian checks that the request is logged, the service provider will show it all of the requests to date - this prevents an attack where the service provider shows the user requests numbered 1 through n and a non-membership proof for $u||n+1$ but then shows the guardian a request logged under $u||n+2$. (See Section 3.4 for possible optimizations.)

3.2. Detailed Construction

Our construction is based on a CCA-secure public key encryption scheme, an s-RKA-secure symmetric key encryption scheme [17], and an aZKS [10]. In more details, the algorithms are instantiated as follows:

- $\mathsf{GuardianKeyGen}(1^k)$: Generate and output a public key pair.
- $\mathsf{ServerInit}(1^k)$: Initialize the aZKS and output a commitment to an empty dictionary.
- $\mathsf{UserStoreSecret}(s, u, policy, gpk_1, gpk_2)$: 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.*, $(\mathsf{sk}_1, u, policy)$ under gpk_1 to get ct_1 and $(\mathsf{sk}_2, u, policy)$ under gpk_2 to get ct_2. Output $storedblob = (ct_1, ct_2, ct_s)$.
- $\mathsf{UserRequestRecovery}(u)$: Generate a public key pair $(\mathsf{pk}_u, \mathsf{sk}_u)$. Output $st_u = \mathsf{sk}_u$ as the state to store and $msg = \mathsf{pk}_u$ as the request message to send to the service provider.
- $\mathsf{ServerUpdate}(st_s, \Delta)$:
  1) Collect the set of entries to be added to the dictionary as follows. Initialize an empty set S. For each $(u, msg) \in \Delta$, look up any previously logged requests for this username $msg_1, \ldots, msg_n$, and add $(u||n+1, msg)$ to S and to the dictionary.
  2) Compute a new commitment to the dictionary with S added.
  3) For each of the newly added entries $(u||n+1, msg)$, compute a membership proof π_j for each of the previous requests $\pi_1, \ldots, \pi_n$ for that username, and a membership proof for the new request π_{n+1}. Compute a non-membership proof π_{n+2} for $(u||n+2)$. Add all these, *i.e.*, $\pi_{\mathsf{inc}} = ((\pi_1, msg_1) \ldots, (\pi_n, msg_n), \pi_{n+1}, \pi_{n+2})$, to Π as the logging proof for (u, msg).
- $\mathsf{GuardianResponse}(\mathsf{sk}, u, i, msg, storedblob, com, \pi_{\mathsf{inc}})$: Let $msg_{n+1} = msg$ and verify each of the proofs in π_{inc}, returning $\perp$ if any verification fails. Recall that $storedblob$ consist of three ciphertexts (ct_1, ct_2, ct_s) and $msg = \mathsf{pk}_u$. Decrypt ct_i using the guardian's secret key sk to get $(\mathsf{sk}_i, u', policy)$. If $u' \neq u$, output $\perp$. Encrypt the resulting sk_i under the public key pk_u from the request message to get ct_u. Return a reponse consisting of the resulting ciphertext ct_u and the original ct_s from the user's $storedblob$.
- $\mathsf{UserCompleteRecovery}(st_u, decblob_1, decblob_2)$: Recall that st_u contains the decryption key sk_u, each $decblob_i$ consists of two ciphertexts (ct_{ui}, ct_{si}), and it should be the case that $ct_{s1} = ct_{s2}$. If not, then stop and output $\perp$. Use sk_u to decrypt the key shares $\mathsf{sk}_1, \mathsf{sk}_2$ from ct_{u1}, ct_{u2}, respectively. Finally, combine sk_1 and sk_2 to obtain sk and use it to decrypt ct_{s1} and obtain the original secret s.
- $\mathsf{GetUserRecoveryHistory}(st_s, u)$: Look up all logged requests for this username $msg_1, \ldots, msg_n$. For $j \in 1...n$ compute a membership proof π_j, for $(u||j, msg_j)$. Compute a non-membership proof π_{n+1} for $(u||n+1)$. Output the proof $\pi = (\pi_1, \ldots, \pi_{n+1})$ and the request history $\vec{msg} = (msg_1, \ldots, msg_n)$.
- $\mathsf{VerifyRecoveryHistory}(u, \vec{msg}, com, \pi)$: Recall that π is a list of proofs $(\pi_1, \ldots, \pi_{n+1})$ and $\vec{msg}$ is the list of request messages $(msg_1, \ldots, msg_n)$. Verify all the proofs and return 1 iff all verifications succeed.
- $\mathsf{AuditServerUpdate}(com, com', \pi)$: Run the aZKS VerifyUpd procedure on (com, com') to verify the proof π.

3.3. Security and Confidentiality

Confidentiality of honest user's secrets.

With an honest service provider: Since the public key encryption scheme used by the honest guardian is CCA-secure, the corresponding ct_i in the $storedblob$ does not reveal any information about that key share, which in turn, protects sk_S. The security of the symmetric key encryption scheme protects ct_S from leaking any information about the secret s. Note that the malicious guardian could potentially encrypt an incorrect sk_i, and as a result the user would decrypt with an sk'_s offset by an additive factor from the correct sk_s. However, due to the s-RKA security of the ske scheme, this decryption would always result in $\perp$, and thus it would not reveal any information about s.

With a malicious service provider: The differences from the honest service provider setting are that 1) the service provider can choose what $storedblob$ to send to the honest guardian, 2) we allow the service provider to see $decblob_1$ for the honest user's requests[1], 3) the service provider can choose what $decblob_1$ to give to the UserCompleteRecovery. 1) is not a problem, because CCA-security of public key encryption scheme means that any modified ciphertext will either fail decryption or encrypt a message unrelated to s. 2) does not reveal information, because the request is encrypted under pk_u. For 3), CCA-security of the public key encryption scheme means that if the adversary tries to maul the ct_u in $decblob_1$, it will again either fail decryption or encrypt an unrelated message. s-RKA security of the symmetric key encryption scheme means similarly that the adversary cannot learn anything by modifying ct_s.

Correctness of policy. This property follows from the correctness of the public key encryption scheme directly. In other words, if the user encrypts a given $policy$, then when the honest guardian decrypts it, it will decrypt to the same policy. If the service provider attempts to modify the ciphertext, then from the confidentiality with malicious service provider argument above, the service provider will not be able to learn anything about s.

Logging of recovery attempts. Note that in GuardianResponse the honest guardian will check that the corresponding msg is logged using the log inclusion proof π_{inc} from the service provider. Suppose that this proof includes membership proofs for $(u||1, msg_1), \ldots, (u||n, msg_n), (u||n+1, msg)$. Then, the next time the user requests a history proof, we have that by the append-only soundness property of the underlying aZKS, the adversarial service provider cannot provide non-membership proofs for any of $(u||1), \ldots, (u||n+1)$, or provide membership for incorrect values for any of these entries. Since the user history proof requires showing a list of membership proofs for consecutive request numbers starting with 1 and ending with a non-membership proof, this means that $(u||n+1, msg)$ must be included in the history shown to the user; otherwise VerifyRecoveryHistory will fail.

Private access history. The service provider logs all the recovery attempts in aZKS. By the privacy property of aZKS, neither the aZKS commitments, nor the proofs (both append proofs and membership/non-membership proofs) leak any extra information about the underlying dictionary. Thus, he privacy of access history directly follows from the privacy of aZKS.

3.4. Optimizations for our Construction

The construction we presented in Section 3.2 can be optimized in various ways. Here we describe those optimizations.

1. Recall that we do not provide confidentiality once the guardian returns $decblob_1$ for adversarial requests.

Caching recovery history proofs. Recall that a recovery history proof consists of the following: for a given username, u, if the logged requests are $msg_1, \ldots, msg_n$, then the recovery history proof consists of membership proofs $\pi_j, j \in [n]$, for $(u||j, msg_j)$ and a non-membership proof π_{n+1} for $(u||n+1)$. The user's device verifies these proofs in VerifyRecoveryHistory.

If the user is logged in from the same device and has cached the last version number j^* for which it verified the membership proof, then, next time the device gets online, it can start checking the proofs only from $u||(j^*+1)$. This means, if the user did not make any recovery attempt during the time this device was offline (and there was no fraudulent attempt on this user's account), then, ideally, the new proof will just be a single non-membership proof for $u||(j^*+1)$.

Removing ct_s from $decblob$. In our construction, $decblob_i$ consists of (ct_{ui}, ct_{si}) for $i \in \{1, 2\}$. We include ct_{si} in $decblob_i$ for notational simplicity. In practice, the service provider could directly return ct_s to the user and ct_{si} can be removed from $decblob_i$. Similarly, the guardian only needs access to ct_i, so the service provider could remove the other two ciphertexts from $storedblob$ before it is sent to the guardian.

Minimizing logging verification cost of guardians. In our construction, the guardian first checks all the membership proofs $\pi_1, \ldots, \pi_n$ for the given username u as part of the logging proof. If the guardian could cache the last membership proof they checked for each username, then they would not need to go back and check from $u||1$. However, this will require the guardians to maintain state for every user account for which they are one of the guardians. This is not ideal.

Instead, the guardians could sign the latest version number j^* for each user account for which they verified the membership proof. They can send this signature and payload to the service provider that can subsequently send it back to the guardians as part of a new logging proof. This way the guardians will entirely avoid verifying older membership proofs. A malicious service provider could send a stale version number 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 Acsesor.

Minimizing storage of guardians. Honest guardians are supposed to release their share of the secret (encrypted under pk_u) as per the policy of the user. 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, again, would increase the storage at the guardian. Instead, the guardians could use the same trick as before: they can sign the time they expect to release the share and send it back to the service provider along with the delay time. Once the delay period has elapsed, the service provider can send the signed payload back, and the guardians can check the time and then prepare and release $decblob$. In any case, this does not degrade the security or confidentiality of Acsesor.

4. Policies and Extensions

In this section 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 our basic Acsesor framework 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 in a straightforward way as well, 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.

Supporting threshold signatures and decryption. In our basic Acsesor framework, 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 in additional communication between any parties if the threshold signing/decryption scheme is non-interactive (e,g., threshold BLS signatures [9]).

Bulletin board and auditing. When registering their secrets, the user could specify a policy that requires the guardians to check a custom bulletin board for consistency. For example, users in an organization may want to have their organization mirror the bulletin board. This can prevent situations where bulletin board(s) commonly used by guardians are compromised by an external attacker.

Another security-related policy the user may want to specify is requesting guardians themselves to audit the log (append proofs) before releasing secrets. Acting as auditors in addition to acting as guardians could be a guarantee that some guardians provide and advertise openly to users when they register their secrets and select their desired guardians.

Storing many secrets. For simplicity, our Acsesor framework, as presented, lets a user store one secret per username. The framework can be easily extended to support multiple stored secrets per user (under the same username) by letting the user attach a "context" string to its $storedblob$ and encrypt it in ct_i for guardian i, along with the respective keyshare, username and policy. The context string could contain information about context in which the secret is stored/used.This context string can be appended to the msg before adding it to the aZKS. While a completely different application, context strings were used in a similar fashion in logging in [11].

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.

For example, 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 service provider's system, or by the service provider 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 service provider blocks the user's legitimate access attempts. In this case, the user can secure their account with the service provider 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 service provider) 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. For example, custodial password managers would benefit from increased trust by using an Acsesor back-end with fast recovery, as the log would reveal access attempts even by the service provider itself.

Two-layer authentication.

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 service provider.

For example, if the user intends to use a PIN as a second factor with a particular guardian, they can include this PIN when the form the encrypted share 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. Finally the user could choose a policy instructing the guardian to rate-limit requests (and hence PIN guesses); as the guardian sees the user's entire request history, this is straightforward to enforce.

Cancellation policy. If a user's account is hijacked by an attacker, they may be unable to recover it before the attacker is able to, *e.g.*, test every possible PIN for a second factor with a short wait time. This could be prevented by a policy that allows only a few attempts before requiring a stronger second factor.

5. Secret Management with Acsesor

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

5.1. Password Management

There are both custodial and non-custodial password managers. Popular web browsers including Chrome [26], Firefox [20], Safari [4], and Edge [59] offer built-in password managers that allow you to save your credentials and back them up in the web browser's cloud service. Most are custodial by default, although Safari's iCloud Keychain is non-custodial and Chrome supports a non-custodial option [25]. These custodial managers make users' secrets available on any device by simply logging in to your account, and save new credentials with a single tap. The account provider can use any standard security features for securing the account, such as 2FA.

Many other password managers, such as 1Password [1] and LastPass [33], have a non-custodial element to them. For example, 1Password uses a combination of a user-stored cryptographic secret key, as well as a master (account) password, to protect access to the secrets. The user needs to maintain access to the secret key, as well as memorize the password. LastPass derives an encryption key from a user's master password and uses it to encrypt their secrets. It optionally supports additional protection by multi-factor authentication. Neither password manager can restore a user's access to their secrets in case they forget their password or lose access to any additional cryptographic material protecting their secrets.

Acsesor can provide a strict improvement over the functionalities described above. A custodial password manager could improve its trust profile by storing the secrets with Acsesor. The user's account with the service provider would allow them to log in from different devices and the service provider can continue using standard methods to secure the account, such as 2FA. Configured with a fast 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 service provider itself – would be able to access the user's secrets without being detected after the compromise.

Functionality of non-custodial password managers can be replicated and improved using Acsesor as well. The password manager would again control the account, but this time Acsesor would be used to register the secrets with a fast recovery policy requiring a second factor, as was explained in Section 4. 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 service provider 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.

In an extreme case, malware may be able to access both a secret key held on a device, as well as read an account password as it is typed on a keyboard. In this case the attacker may be able to read and recover local secrets cached on the compromised device. They would also be able to recover passwords stored in the cloud (for example those generated on the user's other devices), but in this case the access would be detected by the user.

Finally, password managers usually store a cached copy of the secrets on local devices. This is necessary to ensure availability in any centralized secret management system, for example, when the user's account is hijacked or the service suffers a critical outage. Scenarios where cached access is impossible (stolen, lost, or destroyed device) and simultaneously remote access is blocked (hijacked account, service provider behaving maliciously) are outside the scope of all such solutions, including Acsesor.

5.2. Emergency Access and Digital Bequeathment

Password manager providers need to address the possibility of a forgotten password, lost credentials, or other reasons a legitimate owner of the account would need to retrieve their secrets but be unable to. In other cases, the legitimate owner may have passed away or become disabled to an extent that they cannot access their account anymore, and instead their family members or coworkers may have an immediate need to access the stored secrets through a digital bequeathment process. In fact, the new Personal Information Protection Law (PIPL) in China provides legal rights for relatives to get access to a deceased person's information [36], even if this data is stored in different countries.

In principle, custodial password managers do not have this problem. The user may be able to prove to the service provider that they truly are the rightful owner of the account, in which case the service provider may help them recover access. In practice, the proof of identity could require a lengthy customer service interaction, which may be difficult to arrange in an emergency situation. Non-custodial password manager service providers cannot assist with recovery at all, and instead attempt to make it clear to their users that they truly are in charge of managing their own secrets.

1Password recommends printing an *emergency kit* with all account details, including the account password and the cryptographic secret key. They recommend storing it in a safe place and giving a copy to a trusted contact, such as a family member. This approach has the risk of a lost or stolen emergency kit, and heightens the potential for technological abuse. Complicating this further, the account owner has no way of knowing whether the emergency kit has been used. Even if 1Password provided a notification of a log-in attempt from a new device, this behavior relies on 1Password behaving honestly.

Both 1Password and LastPass have notions of distributing access to groups of trusted contacts that are also users of the same service. For example, if a LastPass emergency contact attempts to use their delegated access to unlock an account, LastPass will notify the real owner of the account and grant access only after a configurable wait time. The wait time gives the owner a chance to act in case of a malicious emergency recovery attempt. However, if LastPass was itself behaving maliciously, it may be able to compromise the emergency recovery delegation workflow and disregard the desired wait times and not notify the user.

Acsesor's capability to support flexible recovery policies provides a framework suitable for implementing various emergency access and digital bequeathment capabilities. Upon registering their secrets with the service, the user can specify an emergency access policy that they communicate to the guardians. For example, a user may register the same secret under two different policies: one for immediate access, where the guardians expect a proof of a second factor as usual, and another one for emergency access, where the guardians will wait for some time before responding to the request. The emergency access policy may optionally require a second factor, but in some cases the user may want to set a policy with a particularly long wait time that requires no additional secrets. In all cases, the user, or their designated emergency contact, would still need to be able to authenticate with the service provider to initiate the recovery, but for that standard account security and recovery mechanisms (as in custodial solutions) can be used. If the user's device is periodically monitoring the recovery log, they should detect any fraudulent recovery attempts before the wait time expires. The concerns raised above regarding a user not knowing whether a 1Password emergency kit has been used, or whether LastPass has hijacked the emergency recovery delegation workflow, are addressed by the transparency properties of Acsesor's recovery log.

5.3. Cryptocurrency Wallets

One of the primary non-custodial secret storage scenarios today is cryptocurrency wallets. Almost all non-custodial wallets, such as Coinbase Wallet [13], Metamask [41], Exodus [16], Electrum [15], and many others, provide recovery from a 12-word BIP-39 recovery passphrase the user must memorize or store safely. To improve the reliability of recovery, all wallet providers make it extremely clear that the recovery passphrase controls access to the contents of the wallet and cannot be restored if forgotten or lost. So-called *MPC wallets* use secure multi-party computation techniques to provide extra security by distributing the secret key generation and signature operations across multiple nodes, never actually requiring the key to be assembled in one location; the idea is to prevent the secret keys from being stolen from a compromised device. However, this does require the user to authenticate to each of the MPC nodes.

The Coinbase Wallet is an example of a non-custodial wallet. For emergency recovery, it provides an option to back up the recovery passphrase to Google Drive or iCloud. Since all keys are generated from the passphrase, the wallet can be easily recreated on a new device from the cloud backup. On the other hand, if this storage account is compromised, the user's secrets may be compromised and they may be completely unaware.

ZenGo [62] and Fireblocks [18] are both non-custodial MPC wallets. For example, ZenGo uses distributed key generation to create secret key shares – one for the user's device and one for the ZenGo server. Next, the user's device chooses a symmetric encryption key to encrypt its share and stores this key in a cloud storage account owned by the user, for example, in Google Drive. The user's encrypted share is stored by ZenGo and is protected by biometric authentication: a face scan. When transactions need to be signed, the user authenticates with ZenGo using their face scan to retrieve their encrypted share, retrieves the share encryption key from their cloud storage, and decrypts their share. Finally, the transaction is signed with a threshold signature protocol. This design is meant to protect against single points of failure and to avoid having to memorize a master secret. For situations where the user's face changes substantially, for example, as a result of an accident, ZenGo supports adding alternative face scans for authentication.

The requirements for cryptocurrency wallets are slightly different from password managers, because the secrets protected by wallets are always cryptographic and unmemorable. If one somehow loses access to the wallet recovery in any sense is entirely impossible, whereas many of the accounts a password manager would protect are themselves custodial and may be recoverable through other means. This means that availability is particularly important for wallets. Furthermore, delayed compromise detection is useless for wallets, as stolen digital assets are impossible to recover, whereas for password managers delayed detection may still be very helpful.

Therefore, cryptocurrency wallets could use Acsesor as a back-end to store their users' secret keys, but fast recovery without a strong second factor would be a very dangerous policy to use. For policies with weak or no second factors, the wait times should always be substantial enough. To match the functionality of MPC wallets, 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. This works particularly well with the Ethereum blockchain, as its BLS signatures provide a particularly convenient threshold signature scheme with no interaction between the guardians.

5.4. Enterprise Key Management Services

Enterprise key management services, such as Azure Key Vault (Microsoft) and AWS Key Management Service (Amazon), offer a custodial way to handle key storage and access control of valuable keys. They aim at separating cryptographic keys from data using either software isolation or hardware security modules (HSMs). In both cases, access to the keys is gated using authentication and authorization through the service providers' identity services. Availability

of the keys is guaranteed in a number of ways; for example, by replicating the secrets across multiple geographically distributed HSMs.

Secret management systems intended for the highest value secrets, such as Microsoft Azure Key Vault's Managed HSM solution, require the user to download and store an encrypted backup of the HSM for emergency recovery purposes [43]. While this shifts the management and protection burden back to the user, there is nevertheless a security and usability gain: the encrypted backup is ideally much less frequently used than the API on the HSM and therefore should be easier to secure. However, the weight of this burden should not be understated. The encryption should be shared to at least three distinct RSA private keys and each share being stored on a separate device, such as an encrypted USB drive or an offline on-premise HSM, stored in separate geographic locations and in a lock box or a safe. No single person should have access to all of the keys.

The starkness of this picture belies a surprising piece of information: although the service provider is unable to access or extract keys from the managed HSMs, it does have access to the HSM API and can therefore request signatures or decryptions, if it is willing to accept that these operations will be logged.

This use case seems like a perfect match for Acsesor. It relieves the user of the need for HSMs, encrypted USB drives, and geographically separated lock boxes or safes. In addition, it allows for a flexible choice of the user's preferred guardians and enables logging of access attempts. Accessing the HSM secrets through the encrypted backup avoids logging entirely, whereas using Acsesor would ensure the company is alerted when secret recovery is initiated.

In some cases, it could be possible to shift the entire key management service to run in Acsesor. This may require the guardians to interact to run any necessary cryptographic multi-party protocols.

5.5. Disk Encryption

There are numerous products offering disk encryption functionalities, with different options for key management. Some use a combination of a password and a cryptographic key to derive the encryption/decryption key, whereas others only use a single cryptographic key. Some allow backing up the cryptographic keys to a cloud storage account or sharing them with their organization's administrator. Forgetting the password or losing access to the cryptographic key is catastrophic, as there is no way to recover these.

Acsesor can be used to secret-share a cryptographic disk encryption key to the guardians. In this case, instead of deriving the disk encryption key, it would simply be recovered from Acsesor. For normal use, the user can set a policy for fast recovery, requiring a password or a token device. For emergency recovery, they can set up another fast recovery option registered under an administrator's password, or even a slow recovery option. In business scenarios, at least some of the guardians could be hosted by the organization or a partner organization that is guaranteed to monitor every employee's account as a part of their normal IT operations.

6. Conclusion

We have introduced Acsesor, a new framework for secret management, where a centralized service provider takes the role of a custodian. Instead of having to trust the service provider, 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. However, 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 service provider. Finally, by allowing the guardians to implement flexible response policies, Acsesor can address a broad range of problem scenarios in classical secret management solutions. For example, a slow recovery policy, where the guardians wait for a predefined time until responding, can replace the cumbersome passphrases many cryptocurrency wallets implement today for emergency recovery.

To the best of our knowledge, this is the first framework for secret management that adds strong cryptographic auditability and confidentiality guarantees, and opens up several new and promising research directions. We have already outlined how Acsesor framework can improve the current secret management systems for a wide range of applications. It will be interesting to tailor and optimize Acsesor further for each of these applications, and discover new ones.

References

[1] 1Password. https://1password.com. Accessed on 11/16/2022.

[2] Shashank Agrawal and Srinivasan Raghuraman. Kvac: Key-value commitments for blockchains and beyond. In Shiho Moriai and Huaxiong Wang, editors, *Advances in Cryptology - ASIACRYPT 2020*, Lecture Notes in Computer Science. Springer, 12 2020.

[3] Mustafa Al-Bassam and Sarah Meiklejohn. Contour: A practical system for binary transparency. In Joaquín García-Alfaro, Jordi Herrera-Joancomartí, Giovanni Livraga, and Ruben Rios, editors, *International Workshop on Cryptocurrencies and Blockchain Technology (CBT), 2018*, Lecture Notes in Computer Science. Springer, 2018.

[4] Apple. icloud security overview. Apple Supporthttps://support.apple.com/en-us/HT202303, 2022. Accessed on 12/2/2022.

[5] Avast. 83% of Americans are Using Weak Passwords. https://press.avast.com/83-of-americans-are-using-weak-passwords. Accessed on 11/20/2022.

[6] Ali Bagherzandi, Stanislaw Jarecki, Nitesh Saxena, and Yanbin Lu. Password-protected secret sharing. In *Proceedings of the 18th ACM conference on Computer and Communications Security*, pages 433–444, 2011.

[7] David A. Basin, Cas Cremers, Tiffany Hyun-Jin Kim, Adrian Perrig, Ralf Sasse, and Pawel Szalachowski. ARPKI: attack resilient public-key infrastructure. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2014.

[8] Josh Blum, Simon Booth, Brian Chen, Oded Gal, Maxwell Krohn, Julia Len, Karan Lyons, Antonio Marcedone, Mike Maxim, Merry Ember Mou, Jack O'Connor, Surya Rien, Miles Steele, Matthew Green, Lea Kissner, and Alex Stamos. E2e encryption for zoom meetings. White Paper – Github Repository zoom/zoom-e2e-whitepaper, Version 3.2, https://github.com/zoom/zoom-e2e-whitepaper/blob/master/zoom_e2e.pdf, 2022.

[9] Dan Boneh, Manu Drijvers, and Gregory Neven. Compact multi-signatures for smaller blockchains. Cryptology ePrint Archive, Paper 2018/483, 2018. https://eprint.iacr.org/2018/483.

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

[11] Melissa Chase, Georg Fuchsbauer, Esha Ghosh, and Antoine Plouviez. Credential transparency system. In Clemente Galdi and Stanislaw Jarecki, editors, *Security and Cryptography for Networks*, pages 313–335, Cham, 2022. Springer International Publishing.

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

[13] Coinbase. Coinbase Wallet. https://www.coinbase.com/wallet. Accessed on 11/18/2022.

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

[15] Electrum. https://electrum.org/. Accessed on 11/18/2022.

[16] Exodus. https://www.exodus.com/. Accessed on 11/18/2022.

[17] Sebastian Faust, Juliane Krämer, Maximilian Orlt, and Patrick Struck. On the related-key attack security of authenticated encryption schemes. In Clemente Galdi and Stanislaw Jarecki, editors, *Security and Cryptography for Networks*, pages 362–386, Cham, 2022. Springer International Publishing.

[18] Fireblocks. https://fireblocks.com/. Accessed on 11/25/2022.

[19] Mozilla Foundation. Mozilla security/binary transparency. https://wiki.mozilla.org/Security/Binary_Transparency. Accessed on 12/2/2022.

[20] Mozilla Foundation. Where are my logins stored? Mozilla Supporthttps://support.mozilla.org/en-US/kb/where-are-my-logins-stored, 2022. Accessed on 12/2/2022.

[21] Jonathan Frankle, Sunoo Park, Daniel Shaar, Shafi Goldwasser, and Daniel Weitzner. Practical accountability of secret processes. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 657–674, 2018.

[22] Google. Google binary transparency. https://developers.google.com/android/binary_transparency. Accessed on 12/2/2022.

[23] Google. Key transparency overview. https://github.com/google/keytransparency/blob/master/docs/overview.md. Accessed on 12/2/2022.

[24] Google. Online Security Survey: Google / Harris Poll. https://services.google.com/fh/files/blogs/google_security_infographic.pdf. Accessed on 11/20/2022.

[25] Google. Get your bookmarks, passwords & more on all your devices. Google Help Centerhttps://support.google.com/chrome/answer/165139#passphrase&zippy=%2Ccreate-a-passphrase, 2022. Accessed on 12/2/2022.

[26] Google. How chrome protects your passwords. Google Help Centerhttps://support.google.com/chrome/answer/10311524?hl=en#zippy=%2Chow-password-protection-works%2Chow-we-protect-your-data%2Cyoure-in-control, 2022. Accessed on 12/2/2022.

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

[28] IETF. Supply chain integrity, transparency, and trust (scitt). https://datatracker.ietf.org/wg/scitt/about/. Accessed on 12/2/2022.

[29] Stanislaw Jarecki, Aggelos Kiayias, and Hugo Krawczyk. Round-optimal password-protected secret sharing and t-pake in the password-only model. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 233–253. Springer, 2014.

[30] Stanislaw Jarecki, Aggelos Kiayias, Hugo Krawczyk, and Jiayu Xu. Highly-efficient and composable password-protected secret sharing (or: How to protect your bitcoin wallet online). In *2016 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 276–291. IEEE, 2016.

[31] Tiffany Hyun-Jin Kim, Lin-Shung Huang, Adrian Perrig, Collin Jackson, and Virgil D. Gligor. Accountable key infrastructure (AKI): a proposal for a public-key validation infrastructure. In *22nd International World Wide Web Conference, WWW '13*. International World Wide Web Conferences Steering Committee / ACM, 2013.

[32] Eleftherios Kokoris-Kogias, Enis Ceyhun Alp, Linus Gasser, Philipp Jovanovic, Ewa Syta, and Bryan Ford. Calypso: Private data management for decentralized ledgers. *Cryptology ePrint Archive*, 2018.

[33] LastPass. https://lastpass.com. Accessed on 11/16/2022.

[34] LastPass. Lastpass security incident. https://blog.lastpass.com/2022/11/notice-of-recent-security-incident/. Accessed on 12/02/2022.

[35] Ben Laurie, Adam Langley, and Emilia Kasper. Certificate Transparency. RFC 6962, June 2013.

[36] Kennedys Law. PRC enacts the Personal Information Protection law (PIPL): A 10-Point cheat sheet. https://kennedyslaw.com/thought-leadership/article/prc-enacts-the-personal-information-protection-law-a-10-point-cheat-sheet/. Accessed on 11/23/2022.

[37] Derek Leung, Yossi Gilad, Sergey Gorbunov, Leonid Reyzin, and Nickolai Zeldovich. Aardvark: An asynchronous authenticated dictionary with applications to account-based cryptocurrencies. In *31st USENIX Security Symposium, USENIX Security 2022*. USENIX Association, 2022.

[38] Deepak Maram, Harjasleen Malvai, Fan Zhang, Nerla Jean-Louis, Alexander Frolov, Tyler Kell, Tyrone Lobban, Christine Moy, Ari Juels, and Andrew Miller. Candid: Can-do decentralized identity with legacy compatibility, sybil-resistance, and accountability. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 1348–1366. IEEE, 2021.

[39] Sarah Meiklejohn, Pavel Kalinnikov, Cindy S. Lin, Martin Hutchinson, Gary Belvin, Mariana Raykova, and Al Cutter. Think global, act local: Gossip and client audits in verifiable data structures. *CoRR*, abs/2011.04551, 2020.

[40] Marcela S. Melara, Aaron Blankstein, Joseph Bonneau, Edward W. Felten, and Michael J. Freedman. CONIKS: Bringing key transparency to end users. In *24th USENIX Security Symposium, USENIX Security 2015*, pages 383–398, Washington, D.C., August 2015. USENIX Association.

[41] MetaMask. https://metamask.io/. Accessed on 11/18/2022.

[42] Silvio Micali, Michael Rabin, and Salil Vadhan. Verifiable random functions. In *40th annual symposium on foundations of computer science (cat. No. 99CB37039)*, pages 120–130. IEEE, 1999.

[43] Microsoft. About the managed hsm security domain. https://learn.microsoft.com/en-us/azure/key-vault/managed-hsm/security-domain. Accessed on 12/2/2022.

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

[45] Kirill Nikitin, Eleftherios Kokoris-Kogias, Philipp Jovanovic, Nicolas Gailly, Linus Gasser, Ismail Khoffi, Justin Cappos, and Bryan Ford. CHAINIAC: proactive software-update transparency via collectively signed skipchains and verified builds. In *26th USENIX Security Symposium, USENIX Security 2017*. USENIX Association, 2017.

[46] Novi Financial. Auditable key directory. https://github.com/novifinancial/akd/, 2021. Accessed on 12/2/2022.

[47] Akif Patel and Moti Yung. Fully dynamic password protected secret sharing: Simplifying ppss operation and maintenance. In Shlomi Dolev, Oded Margalit, Benny Pinkas, and Alexander Schwarzmann, editors, *Cyber Security Cryptography and Machine Learning*, pages 379–396, Cham, 2021. Springer International Publishing.

[48] Sarah Pearman, Shikun Aerin Zhang, Lujo Bauer, Nicolas Christin, and Lorrie Faith Cranor. Why people (don't) use password managers effectively. In *Proceedings of the Fifteenth USENIX Conference on Usable Privacy and Security*, SOUPS'19, page 319–338, USA, 2019. USENIX Association.

[49] PAD Protocol. https://www.padprotocol.org/. Accessed on 11/20/2022.

[50] Vicente Silveira Richard Hansen. Code verify: An open source browser extension for verifying code authenticity on the web. Engineering at Meta Blog – March 10, 2022 https://engineering.fb.com/2022/03/10/security/code-verify/, 2022. Accessed on 12/2/2022.

[51] Mark Dermot Ryan. Enhanced certificate transparency and end-to-end encrypted mail. In *Proceedings of the ISOC Network and Distributed System Security Symposium (NDSS)*. The Internet Society, 2014.

[52] Andy Sayler, Taylor Andrews, Matt Monaco, and Dirk Grunwald. Tutamen: A next-generation secret-storage platform. In *Proceedings of the Seventh ACM Symposium on Cloud Computing*, SoCC '16, page 251–264, New York, NY, USA, 2016. Association for Computing Machinery.

[53] Leona Tam, Myron Glassman, and Mark Vandenwauver. The psychology of password management: A tradeoff between security and convenience. *Behaviour & IT*, 29:233–244, 05 2010.

[54] Alin Tomescu, Vivek Bhupatiraju, Dimitrios Papadopoulos, Charalampos Papamanthou, Nikos Triandopoulos, and Srinivas Devadas. Transparency logs via append-only authenticated dictionaries. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS 2019, London, UK, November 11-15, 2019*. ACM, 2019.

[55] Alin Tomescu, Yu Xia, and Zachary Newman. Authenticated dictionaries with cross-incremental proof (dis)aggregation. Cryptology ePrint Archive, Paper 2020/1239. https://eprint.iacr.org/2020/123, 2020.

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

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

[58] Trust Wallet. https://trustwallet.com/. Accessed on 11/18/2022.

[59] Dan Wesley, Saisang Cai, Andrea Courtright, and Andrea Barr. Microsoft edge password manager security. Microsoft Learnhttps://learn.microsoft.com/en-us/deployedge/microsoft-edge-security-password-manager-security, 2022. Accessed on 12/2/2022.

[60] Shalanda Young. Moving the U.S. government toward zero trust cybersecurity principles. https://www.whitehouse.gov/wp-content/uploads/2022/01/M-22-09.pdf, 2022. Accessed on 12/2/2022.

[61] Jiangshan Yu, Mark Ryan, and Cas Cremers. How to detect unauthorised usage of a key. Cryptology ePrint Archive, Paper 2015/486. http://eprint.iacr.org/2015/486, 2015.

[62] ZenGo. https://zengo.com/security-in-depth/. Accessed on 11/18/2022.