Horcruxes for Everyone – A Framework for Key-Loss Recovery by Splitting Trust

Felix Hörandner
Graz University of Technology, Austria
Email: felix.hoerandner@iaik.tugraz.at

Christof Rabensteiner
Graz University of Technology, Austria
Email: christof.rabensteiner@iaik.tugraz.at

***Abstract*—Many modern applications require users to manage keys on their own devices, which, in case of device loss or failure, may lead to serious consequences, e.g., losing access to their Bitcoin wallet. These applications need a secure and user-friendly strategy that protects users from losing keys while preserving the keys' confidentiality. Fortunately, password-protected secret sharing (PPSS) can be used to design such a key-loss recovery strategy: It enables users to split their keys into shares, to distribute these shares across third parties and, if necessary, to recover keys via password authentication. However, deploying PPSS in a key-loss recovery strategy leaves the following practical questions unanswered: Which third parties should a user pick to diversify the trust? How can these third parties be recruited? And: How can other applications benefit from such a strategy?**

In this paper, we develop a framework for key-loss recovery, which allows users to distribute shares in a hierarchy that is aligned with relevant trust factors. As part of the framework, we propose a management app that supports users in building and managing hierarchical trust policies, and that offers its service to other applications. To convince organizations to operate servers, we implement our framework with a focus on server-side cost-efficiency. We extend a PPSS scheme with hierarchical trust policies, add efficient prevention of online guessing, and measure the performance of the overall system at-scale on AWS. The cost projection shows that deploying our framework is inexpensive: 40 organizations, each operating server resources for less than $20, support 50 million users when splitting and recovering their keys.

***Index Terms*—password-protected secret sharing; key-loss recovery; trust policy**

I. Introduction

Storing keys on the user's device is a common requirement for security-related applications. Managing Cryptocurrency wallets, end-to-end confidential data sharing in the cloud, challenge-response authentication, and national electronic signature solutions are a few example applications that require client-side key management. All these applications face the same challenge: What if the device storing the user's keys breaks, is lost, or gets stolen? Such events are very likely to occur in large user bases and can lead to disastrous consequences: Users irretrievably lose their Bitcoins, their encrypted personal data, or their ability to authenticate, etc. These consequences need to be prevented with a proper key-loss recovery strategy.

Traditional Approaches. As keys are too large to remember, traditional recovery approaches rely on a secure location, e.g., backing up the key material to a flash drive, or printing it as QR code on a sheet of paper. However, will the user keep the flash drive or QR code both secure and available for a long time until recovery is finally needed? These security and availability requirements for storage locations can be reduced by adding a knowledge factor, e.g., by password-encrypting the keys before storing them in the cloud. As human-memorizable passwords have limited entropy, the confidentiality of encrypted keys becomes questionable after a powerful cloud provider had an extended period of time for offline guessing attacks.

Secret Sharing, introduced by Shamir [1], can be used to split trust between third parties when recovering from key-loss: The user splits her key into shares, distributes these shares to trusted entities (e.g., family members), and, once needed, obtains a sufficient number of shares to reconstruct her key. The key's confidentiality is protected as long as a sufficient number of distinct parties stay honest. Password-Protected Secret Sharing (PPSS) [2], [3] extends secret sharing with build-in password-based authentication: The user supplies a password to the key splitting process, and the recovery process succeeds only with the same password. Trusted parties do not learn the password and cannot impersonate the user. These properties make PPSS a promising basis for key-loss recovery.

PPSS resembles the concept of Horcruxes from the Harry Potter universe: Horcruxes are parts (shares) of a person's soul (key) that are stored in objects (third parties). After death (device loss/failure), a Horcrux enables the resurrection of the person (recovery) via a spell (password). Since Horcruxes are considered the darkest of all magic, they are reserved for the elites and not open to the public.

Challenges. Analogous to Horcruxes, PPSS is simply not accessible for a wide audience; to change this, the following gaps need to be addressed: A number of trustworthy organizations need to be convinced to operate PPSS servers, so that users have sufficient options to choose a subset they trust. Reaching a trust decision is also not trivial: Users need to understand their decision and implications of future changes in the servers' environment (e.g., new laws, company mergers, corruptions) for a possibly large set of servers operated by various organizations. Moreover, it is hard to convince users to go through such a decision process again for each application they are using. In the long term, users also need to be able to change who they trust (e.g., when a party broke their trust) or their password (e.g., because it leaked).

Our Contribution. In this paper, we propose a user-friendly system for key-loss recovery that contributes to addressing the above-mentioned challenges when deploying PPSS on a large-scale in practice. Firstly, our *generic framework* supports users in building and maintaining a trust policy. 1) We propose to organize the trusted parties in a hierarchy along trust-relevant factors to make the implications of changes more understandable (e.g., when the law changes impacting multiple parties). 2) We reduce the burden on users by introducing a recommender system that suggests hierarchies of trustworthy parties, which users review and adapt to their liking. 3) A management app acts as the central point for users to maintain their trust policies, while various other applications can be connected to this management app to re-use its key recovery capabilities. 4) For long-term usage, the framework enables users to change their trust policy, password, or keys of connected applications.

Secondly, in the *implementation*, we focus on addressing the challenge of convincing organizations to operate a PPSS server by instantiating our framework to showcase its cost-efficiency. We 1) introduce the idea of hierarchical PPSS by integrating compartmented secret sharing [4] with the PPSS scheme by Abdalla et al. [5], 2) evaluate the performance of this cryptographic mechanism, 3) describe a mechanism to prevent online guessing with low input and storage requirements, and 4) measure the costs of deploying the overall system. A projection of the costs, based on our measurements, shows it is very inexpensive to operate servers on a large scale: approx. \$700 is sufficient to deploy a 40-server setup that is capable of handling 100 million split or recover operations, resulting in less than \$20 per participating organization.

II. Related Work

Password-Protected Secret Sharing, introduced by Bagherzandi et al. [2], extends secret sharing by introducing a password to authenticate the user without revealing this password to the servers. Their scheme and initial work from Camenisch et al. [3] rely on PKI not only during the split but also recovery phase. In follow-up work, Camenisch et al. remove the need for a PKI during the recovery phase [6], while Yi et al. [7] improve performance. Jarecki et al. [8] introduce the first *robust* PPSS, which enables a user to validate shares, and therefore exclude corrupted shares. In their follow-up work, Jarecki et al. [9], [10] further improve the efficiency of their schemes at the cost of dropping robustness. Abdalla et al. [5] improve upon [8] by replacing zero knowledge proofs and reducing the computation effort to verify shares.

Recovery with Trust-Splitting. Brookner et al. [11] propose key recovery by splitting the key into segments and storing each segment at a server. Their approach requires that all servers remain trusted and available, as losing one segment means losing the entire key. Huang et al. [12] use Shamir's secret sharing and thereby only require a threshold of servers for recovery. Souza et al. [13] additionally enable users to detect corrupted shares via publicly verifiable secret sharing. However, neither [12] nor [13] offer any means for authentication during recovery, which is crucial to ensure that only authorized parties obtain shares.

Recovery based on Password-Encryption. A user may derive a wrapping-key from her password (e.g., PKDF2 [14], scrypt [15]), use this key to wrap her actual key, and store the wrapped key in the cloud. As increasing costs for the key derivation propagate linearly to attackers, this approach only offers weak protection against cloud attackers who can perform long-term offline guessing attacks and have several orders of magnitude more resources.

Recovery based on Biometry. Users may use biometric cryptosystems (BCSs), such as fuzzy extractors [16] or biohashing [17], to protect keys with their biometric templates (e.g., fingerprints). However, BCSs rely on helper data to generate stable, high-entropy keys [18]. Such helper data cannot be memorized, must be kept confidential to prevent information leakage, and need to remain available for recovery, which brings us again to our initial problem.

III. Framework to Recover from Key-Loss

This section present our framework based on trust-splitting: First, we discuss the factors on which the trust depends that the user puts into a party and propose to organize the user's trust hierarchically along those factors. We then introduce a recommender system that helps users with creating trust policies and discuss incentives for organizations to offer PPSS servers. Eventually, we present the framework's architecture, its data flow, and considerations for long-term usage.

A. Trust Factors

Our framework relies on splitting a recovery key between many parties. The goal of the framework is to keep the recovery key available, even if some shares are lost, and to protect the key's confidentiality, even if some parties turn against the user by colluding. In this section, we discuss how internal and external trust factors influence a party's ability to keep shares confidential and available. We distinguish parties by location into *local* shares (stored by the user), *social* shares (handed to friends/family), and *remote* shares (managed by organizations).

Internal Factors. A party's trustworthiness to keep shares available and confidential depends on the party's motivation and competence. For local shares, we expect users to be motivated for their own benefit. Social share-holders are also expected to be motivated in order to maintain the existing trust relationship. However, as questioned in the introduction, both local and social shares could get lost due to lack of competence or dedication. For remote shares, we recommend users to derive an organization's motivation and competence from how the organization handles its core business (e.g., the trust in banks to handle our money). This relation also works in the other way: By losing or leaking shares, an organization damages the trust that was built for its core business. Therefore, users should select organizations that rely on being trustworthy.

External Factors. Besides motivation and competence, organizations also depend on external factors influenced by the environment in which they operate: Firstly, the *law* may force organizations to behave against their motivation, e.g., to disclose sensitive information (c.f. PATRIOT-ACT [19]). To reduce this risk, organizations from different legal frameworks should be chosen. Secondly, within an *industry sector*, organizations

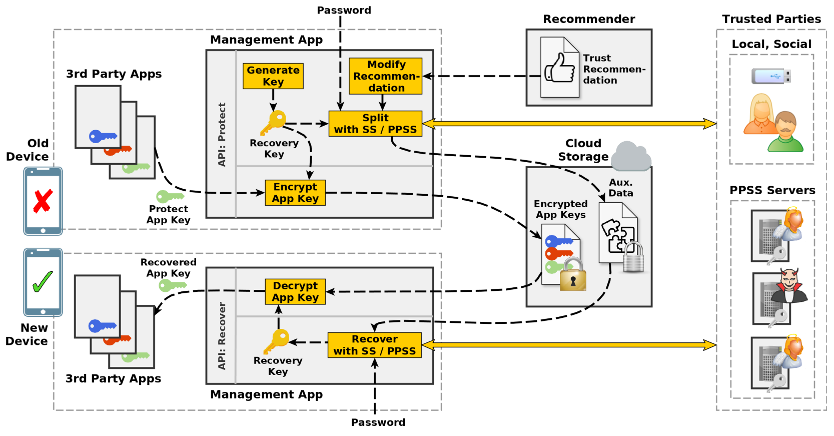

Fig. 1: Architecture and Data Flows

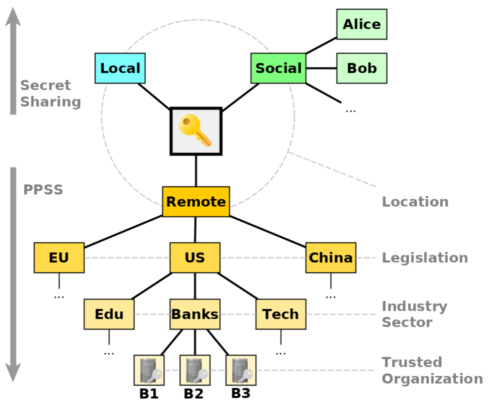

Fig. 2: Example Trust Hierarchy

might be compelled to use the same industry-specific software, procedures, or standards, which might affect multiple servers. For example, bugs in a widely-used banking software might allow an attacker to crash or even corrupt multiple servers. Also, categorizing organizations by industry sectors helps users to reason about trust aspects in the context of competition or mergers.

Trust Hierarchy. To minimize exposure to a single trust factor and its accompanied risk, we propose to diversify the risk by selecting parties with diverse trust factor characteristics. As shown in the template of Figure 2, users organize the trust hierarchically along the described factors, which helps to understand initial trust relationships and to comprehend the impact of changes on the chosen trust policy. E.g., if a new law is passed or an industry-specific software vulnerability is discovered, a hierarchy visualizes which subtree is affected and how much of the user's trust policy remains (supposedly) trustworthy. Of course, template for hierarchical trust splitting may be simplified or extended to fit the users' needs.

B. *Supporting Users in Building Trust Policies*

Requiring users to define complex trust hierarchies from scratch represents a daunting task. To reduce the complexity of this task, we propose recommender systems that offer trust policies. Users may adapt these policies to their liking instead of defining them from the ground up. Of course, recommenders need to be trusted to some degree: They should not try to manipulate their users with recommendations. As trust decisions are very user-specific, users need to be able to specify which recommender they want to use.

Recommender Types. Recommendations might be based on 1) random choices within a predefined shape of the hierarchy, 2) decisions of peers, e.g., what friends and family selected, 3) decisions of the whole user base, or 4) the knowledge of a group of experts. As experts with a financial motivation might experience a conflict of interests when organizations aim for a good placement, voluntary experts without an agenda pose less risk.

Community-Driven Recommendations. We propose to adopt recommendations by a community of voluntary experts. These experts may discuss and agree on recommendations which contain: A hierarchical categorization of organizations, trust scores for the individual organizations, and arguments underpinning these suggestions to facilitate understanding for users, e.g., based on the organizations' technical descriptions, certifications, and track record. As laws change, companies merge, or news of breaches become public, etc., trust recommendations change over time, which may prompt users to change the distribution of their shares.

C. *Server Recruitment*

To enable users to split their data and trust among a set of chosen servers, a sufficiently large pool of servers needs to be available. Therefore, an important aspect is convincing organizations to operate a server, which boils down to a trade-off between effort and benefit.

Universality. To reduce complexities for servers as well as users, we aim for a universal framework that can be applied to a variety of use cases without application-specific changes. On the one hand, organizations should only need to deploy a single server without the need for application-specific configurations and deployments, which would hinder adoption and increase server-side effort. On the other hand, a universal system also reduces the burden on users, requiring them to select trustworthy services only once instead of for each application individually.

Incentives. While the common good may serve as sufficient incentive for national institutions or non-profit organizations, companies might require additional incentives, such as non-intrusive advertising: Firstly, being able to market altruistic motivations by helping users in recovering their lost key material sheds a positive light on companies. Secondly, by showing logos of the users' trusted organizations during a recovery process, users might feel grateful to these organizations that helped to save them from disastrous data loss.

Effort. Of course, in contrast to the incentives, the effort needs to be comparably low. Translated to our framework, this means that the costs for the servers in terms of processing time, as well as storage and traffic sizes, need to be as low as possible. We refer to Section IV-D to underline that our framework can be implemented very cost-efficiently for the server-side.

D. Management App and Architecture

The architecture, shown in Figure 1, combines the previously described aspects into the following main actors:

Management App. This client in the users' domain enables users to split and recover their keys via secret sharing techniques. A PPSS scheme is used for remote servers, while Shamir's secret sharing with out-of-band authentication is more convenient for local backups and the users' social circle. To facilitate adoption, the management app offers its key recovery capabilities through an API to other third-party apps. Therefore, users only need to interact with a single app to manage their trust decisions. The management app also integrates with a user-selected recommender to initially simplify the process of defining a trust hierarchy and to subsequently get notified of changes regarding the users' trust decisions. Furthermore, the management app also integrates with a user-specified cloud storage to store protected data.

PPSS Server. The PPSS server exposes the PPSS functionality via a network interface, while employing mechanisms to prevent online attacks. A large number of these servers should be deployed by a diverse set of trustworthy organizations.

Recommender. The recommender offers recommendations on the trustworthiness of PPSS servers and organizes these servers in a hierarchy. The community behind this service substantiates their recommendations with arguments. Over time, these recommendations change as new information becomes available, e.g., regarding breaches of individual services or new laws. The recommender pro-actively notifies the management app of changes within the recommended hierarchy, so that users can change their composition of shares according to the new situation. Multiple communities may operate their own recommenders to enable users to choose a community they trust.

Cloud Storage. A cloud storage is integrated into the management app to store auxiliary data (aux) independently of the users' devices, so that this data is available even if a device is lost. The stored data include information needed to perform the recovery process, such as the users' hierarchy of trusted parties, as well as encrypted keys. The resources for this cloud storage are provisioned by the user and do not need to be fully trusted, as the data stored on the cloud storage is not sufficient to recover the users' keys.

Data Flow. Protocol 1 gives details on the data flow. The management app generates a recovery key (rk), splits this key, and shares it with the user's hierarchy of trusted parties, i.e. local backup (loc), social circle (soc) and remote servers (rem). If a user wants to protect an app key (ak) of an app (referred by aid), she imports the app key into the management app, encrypts it with the recovery key and uploads the encrypted app key to the cloud storage. If recovery becomes necessary, the user reconstructs the recovery key from her local backup, social circle and trusted PPSS servers with her password, and uses this recovery key to decrypt the app keys.

```
API
Protect(aid, ak) → b:
  if no recovery key was set up then
      rk ← Setup Recovery Key()
  if no recovery key is cached locally then
      rk ← Recover Recovery Key()
  return status bit b ← Protect App Key(aid, ak, rk)
Recover(aid) → ak:
  if no recovery key is cached locally then
      rk ← Recover Recovery Key()
  return ak ← Recover App Key(aid, rk)

Internal Functions
Setup Recovery Key() → rk:
  Obtain password pw via user input
  Obtain trust recommendation rec from recommender
  Ask user to modify/accept rec, resulting in trust policy pol
  Generate recovery key rk ← SYM.KeyGen($1^\kappa$)
  Generate random user id uid as identifier in requests
  Split key to domains: (loc, soc, rem) ← SS.Split(rk, pol)
  Split remote share rem across remote servers:
      aux ← PPSS.Split(rem, pw, pol)
  Store loc locally, send soc to social circle
  Upload aux with (pol, uid) to user-selected cloud storage
  return rk
Recover Recovery Key() → rk:
  Obtain password pw via user input
  Download aux from user-selected cloud storage
  Parse uid from aux as identifier in requests
  Extract the user's policy pol from aux
  Recover the rem ← PPSS.Recover(pw, pol)
  Obtain local shares loc and social shares soc
  return rk ← SS.Recover(loc, soc, rem)
Protect App Key(aid, ak, rk) → b:
  Encrypt app key as eak ← SYM.Enc(rk, ak)
  Upload (aid, eak) to user-selected cloud storage
  return 1 if process was successful, and 0 otherwise
Recover App Key(aid, rk) → ak:
  Download (aid, eak) from user-selected cloud storage
  Decrypt and return app key ak ← SYM.Dec(rk, eak)
```

Protocol 1: Functions of the Management App

E. Long-Term Usage

Apart from wisely-chosen security parameters for the employed cryptography, the proposed system's security also depends on how multiple actors handle their security-related assets, such as server keys, app keys, the users' passwords, and recovery keys. In the long-term, it is reasonable to assume that a PPSS server, a third party app, or even the user's device becomes corrupted or loses the user's trust in another way.

Reasons to Change. 1) *Server Composition:* Various events might prompt the user to change her policy on which PPSS servers she deems trustworthy. For example, the user might learn about legal changes, company mergers or data breaches that might have a negative impact on her trusted servers or the relation between those servers. Ideally, the recommender notifies the user's management app of relevant news, which filters these notifications to only displays those, that are also relevant to the user's policy. With this information, the user

```
Change Recovery Key():
    Obtain old rk from local cache or Recover Recovery Key()
    Create new recovery key rk' but reuse pw and pol with:
        $\mathsf{rk}' \leftarrow$ Setup Recovery Key()
    Download all protected app keys $\mathsf{eak}_{\mathsf{aid}}$ from the cloud storage
    Re-Encrypt keys: $\mathsf{eak}'_{\mathsf{aid}} \leftarrow \mathsf{SYM.Enc}(\mathsf{rk}', \mathsf{SYM.Dec}(\mathsf{rk}, \mathsf{eak}_{\mathsf{aid}}))$
    Reupload the updated app keys $\mathsf{eak}'_{\mathsf{aid}}$
Change Trust Policy(pol'):
    Run Change Recovery Key(), but when setting up the new rk',
    use the new pol' instead of the previous pol
Change Password(pw'):
    Run Change Recovery Key(), but when setting up the new rk',
    use the new pw' instead of the previous pw
Change App Key(aid, ak'):
    Obtain old rk from local cache or Recover Recovery Key()
    Run Protect App Key(aid, ak', rk), replacing the old eak with a
    new eak' at the cloud storage
```

Protocol 2: Maintenance via the Management App

may adjust her trust policy. 2) *Password:* Users might want to change their password, for example, if the password was stolen. 3) *App Keys:* The keys of external apps might also become corrupted, and therefore need to be replaced.

Operations. Protocol 2 details algorithms to change the trust policy (e.g. upon notification from the recommender), password, app keys and recovery key, to limit the consequences of breaches. The design enables to pro-actively replace keys, which reduces the risk of yet unknown breaches by limiting the time-frame for attackers.

Additional External Means. Our proposed framework still allows external applications to employ various other mechanisms that help in minimizing the consequences of stolen keys. For example, these external applications may make use of forward secrecy [20], [21] to protect the confidentiality of previous messages even if the key is stolen, or updateable encryption [22] to allow a semi-trusted entity to re-key existing ciphetexts and therefore limit the time-frame for an attack.

IV. Implementation

This section discusses implementation details for our proposed framework. While the framework uses Shamir's secret sharing for shares in the local and social domain, this section focuses on integrating remote servers through a PPSS scheme. As the guiding principle, we aim to minimize costs for the servers, even if that requires moving computing and storage efforts to the client-side. We argue that this is a reasonable trade-off, as split and recovery processes occur infrequently for individual users, who would rather be able to choose from a multitude of trusted servers than conserve an insignificant amount of resources. In the remainder of this section, we explain how to share secrets with hierarchical policies and integrate these policies into an adapted version of Abdalla et al.'s PPSS scheme [5]. We further present a low-cost protection mechanism that mitigates online guessing attacks. Eventually, we evaluate the performance of our hierarchical PPSS and measure deployment costs on Amazon Web Services (AWS).

A. Secret Sharing for Hierarchical Policies

To model the hierarchical structure of users' trust policies, we adapt the idea of compartmented secret sharing [4], so that a sufficient number of shares from different hierarchically organized compartments (i.e., groups of servers) need to be recursively combined to reconstruct the secret. As described in Section III-A, by mapping trust factors to groups, it becomes easier for users to understand the implications when these factors change (e.g. law change makes subtree untrustworthy).

Hierarchical Secret Sharing. For hierarchical secret sharing, we apply Shamir's work [1] recursively. The Split and Recover algorithms are defined as follows:

The Split algorithm takes the description of a tree in the form of threshold-size pairs (t_i, n_i) (that can be derived from a trust policy) to build a corresponding tree of shares. Starting from the root, the secret is split into multiple shares. These individual shares are then further split into sub-shares to develop the various subtrees according to the tree of threshold-size pairs. The algorithm returns the sub-share values for leaf nodes.

To Recover, sub-share values of nodes with the same direct parent according to the tree description are combined to reconstruct the parent's share. This process is performed repeatedly from the leaves up until the root is reached and therefore the initially shared secret value has been recovered.

In the remainder of this paper, we consider hierarchical secret sharing only for trees where each node on the i^{th} level has n_i children with a threshold t_i. We denote secret sharing on such a tree as $({}^{t_1}\!/_{n_1}, ..., {}^{t_l}\!/_{n_l})$ hierarchical secret sharing.

Thresholds. Such $({}^{t_1}\!/_{n_1},...,{}^{t_l}\!/_{n_l})$ hierarchical secret sharing does not have a single fixed threshold but rather a range, as the ability to reconstruct depends on which parts of the tree are available (or corrupted when considering the adversary). This threshold lies between $t_{min}{=}\prod_{i\geq 1} t_i$ and $t_{max}{=}S(1){+}1$, where $S(i)$ is the maximum number of servers that collude but are still not able to reconstruct the subtree's value on level i:

$$S(i) = \begin{cases} t_i{-}1 & \text{on leaf level} \\ (t_i{-}1)\cdot(\prod_{j\geq i} n_j) \ + \ (n_i{-}t_i{+}1)\cdot S(i{+}1) & \text{else} \end{cases}$$

Such a threshold range favors users if diverse servers were organized along trust factors as described in Section III-A: The lower threshold enables users to reconstruct even if only t_{min} servers are available. While an adversary may corrupt servers randomly until reaching the threshold, for hierarchical secret sharing it is significant which parts of the tree are corrupted. Either the adversary tries to recursively corrupt a minimally required set of servers or whole subtrees. If the user diversified both servers and subtrees, corrupting enough servers to obtain the secret becomes an expensive undertaking for any adversary. The probability of an adversary to recover a key is 0 for $t_{min}{-}1$ corrupted servers and grows until 1 at t_{max} corrupted servers.

B. Hierarchical PPSS

Next, we integrate hierarchical secret sharing with password protection.

Basis. The modular construction by Abdalla et al. [5] serves as basis for our hierarchical PPSS. Their scheme requires little computation, communication, and storage costs, especially on the server-side. On a high-level, their scheme uses an oblivious pseudo-random function (OPRF) to generate a pseudo-random value in a protocol between the user, applying her password, and a server, employing its secret key, without revealing the password or key material to the other side. Such pseudo-random values from different servers are then used to encrypt the individual shares. Considering efficiency, in Abdalla et al.'s construction, each server only needs to evaluate a single OPRF per split and recovery operation, which is a single exponentiation when employing their One-More-Gap-Diffie-Hellman-based PRF (OMGDH). Servers only need to store the OPRF key pair, as the user encrypts the shares locally with the OPRF output and uploads the ciphertexts to her cloud storage.

Robustness. Additionally, Abdalla et al. developed a robust gap threshold secret sharing scheme (RGTSSS) and integrated it into their PPSS to detect incorrect responses in the recovery process, i.e. provide robustness. Unfortunately, applying this robustness property directly to the individual hierarchy levels gives attackers additional information which facilitates offline password guessing. Corrupt servers who obtained the user's auxiliary data may use the robustness property to verify their password guess: They check if the password guess resulted in an OPRF output that decrypts the encrypted share to a correct share.

Our framework does not rely on robustness as we assume that a large number of users quickly report servers returning incorrect responses, which are then flagged by the recommender. Likely, only very few not-flagged servers return incorrect responses, which allows users to identify an honest set of servers within a limited number of tries.

Hierarchical PPSS. In Abdalla et al.'s PPSS scheme, we replace the calls to RGTSSS' Split and Recover with their non-robust but hierarchical counterparts from Section IV-A. The hierarchical shares are then encrypted/decrypted with the OPRF results. The tree description in the form of threshold-size pairs is needed to correctly combine the sub-shares, which we also store within the data that is uploaded to the cloud storage. We denote $({}^{t_1}/_{n_1}, ..., {}^{t_l}/_{n_l})$ hierarchical PPSS analogously to hierarchical secret sharing, and only consider likewise shaped trees in the remainder of this paper.

Security. We need to ensure that combining hierarchical secret sharing with password-protection does not assist attackers in offline password guessing. Attackers must not be able to check password guesses on subtrees-level by a group of less than t_{min} colluding participants. Instead, we require that a password guess can only be verified be recovering shares along the whole tree, requiring at least t_{min} correct shares. More precisely, an adversary must only succeed with negligible probability at distinguishing between a set of $t_{min}-1$ random values (incorrectly decrypted shares) and a set of correct shares.

We sketch why an adversary with less than t_{min} values (e.g. from colluding servers) cannot exploit the hierarchical structure for offline guessing attacks based on arguments from Shamir [1]. Firstly, the adversary must not be able to distinguish between a single correct share and a same-size random value. Shares are

Server Device	KeyGen	Split	Recover
AWS c5.xlarge	1.19	1.18	1.18
Google Pixel 2	5.51	5.45	5.44

TABLE I: Server Execution Times (in ms), independent of t, n, or hierarchy

Client Device	Abdalla's Robust PPSS			Our Hierarchical PPSS		
	t/n	Split	Recover	t_i/n_i	Split	Recover
AWS c5.xlarge	2/3	54	33	2/3	49	33
	3/4	71	50	3/4	66	49
	6/9	286	101	⅔, ⅔	148	99
	29/36	3586	490	⅔, ⅔, ¾	591	476
Google Pixel 2	2/3	284	183	2/3	243	163
	3/4	366	266	3/4	323	243
	6/9	1334	541	⅔, ⅔	720	481
	29/36	16965	2544	⅔, ⅔, ¾	2862	2307

TABLE II: Client Execution Times (in ms)

generated by Shamir's Split operation, where the coefficients for a polynomial $q(x)$ are chosen randomly before evaluating $q(x_i){=}y_i$ as shares. Further assuming that only y_i without any further formatting is returned, a correct share cannot be distinguished from a random value.

Secondly, considering the set of correct shares or random values together, the adversary must not be able to distinguish the sets by using the Recover operation. 1) Inputs and outputs are from the same finite field. 2) There are no error cases. Given any set of points with y_i from the shares and distinct x_i, one and only one polynomial $q(x)$ of degree t_i-1 exists that interpolates these points (i.e., where $q(x_i){=}y_i$). Therefore, for any set of inputs to Recover, there is a polynomial q with an output value $q(0)$ with no error cases. 3) The distribution of recovered outputs is uniform. Given t_i-1 points, for each candidate value D', an attacker can construct a polynomial q' that interpolates all points and has $q'(0){=}D'$ (i.e., recovers to any value) with equal probability.

Performance. We implemented our hierarchical PPSS based on Abdalla et al.'s scheme [5] with their OMGDH-OPRF for the Java platform with parameters chosen according to NIST's recommendation [23] for 128bit security. Our implementation builds on the IAIK JCE[1] and Archistar's library [24]. Tables I and II present our performance measurements, where the client shares a 128bit AES key with n servers and reconstructs the key on a subset of t servers. Note that the single-threaded implementation only makes use of one core during the benchmark. This evaluation has been run on an AWS instance (c5.xlarge) as well as a mobile phone (Google Pixel 2, Android 8). The presented times are an average of 100 runs. As each server only evaluates a single OPRF during Split and Recover, their computation time is independent of the users' choice of n and t. To give an intuition on the performance, we also include measurements of our implementation of Abdalla's scheme, where we contrast t in Abdalla's robust scheme with t_{max} in our non-robust but hierarchical scheme.

[1]https://jce.iaik.tugraz.at/

C. Protection Against Online Guessing

PPSS schemes rely on mechanisms that make online guessing attacks on the users' passwords infeasible. We tackle the challenge to implement a protection mechanism, that operates on the little information available to the server-side, and has low computation and storage requirements.

Per-User Keys. To limit online guessing attacks per user, the server needs to be able to distinguish for which user any given request is intended. User-binding can be achieved by requiring an individual server-side private key for each user. To reduce the needed storage resources, the server may derive per-user private keys sk from a master key mk and a user-provided identifier uid. For example, in the OMGDH-OPRF from Abdalla et al. [5], the server derives the per-user private key $\mathsf{sk}_{\mathsf{uid}} \leftarrow \mathsf{mk} + \mathsf{uid}$, while users derive the respective public keys $\mathsf{pk}_{\mathsf{uid}} \leftarrow \mathsf{pk} \cdot g^{\mathsf{uid}}$ on their own. Users include their uid in requests to enable the servers to identify which keys to use.

Windowed Request Limit. As the server does not learn if the requester used the correct password in Abdalla et al.'s construction, the server needs to solely rely on the user uid sent in the requests and the time of receiving to prevent online guessing. We employ a simple but effective mechanism: Only a limited number of requests are accepted for an uid within a server-side time window. Requests for an uid exceeding this limit are blocked. The server maintains a table for the current time window, associating user uids with a request counter. Assuming 128bit for uid to prevent guessing and 8bit for the counter, the server needs to store 132bit or 17byte per uid requested in the time frame. The server resets the table after the epoch to reset blocked uids and to free storage.

Denial of Service. Of course, request limits represent a Denial of Service (DoS) threat for legitimate users, if attackers are able to spend the users' tries. To place a request on behalf of a user, the attacker would need access to the respective uid, which is randomly chosen by the user and stored at the cloud storage. Therefore, the cloud storage may mount DoS attacks on the PPSS servers, which is, however, unlikely as the cloud storage could simply delete the user's auxiliary data to prevent her from reconstructing the recovery key. Other attackers would need to guess the uid, which is only successful with negligible probability if chosen from a sufficiently large space (128bit in our case).

D. Cost Measurements and Projections

To evaluate the costs of operating a PPSS server, we deployed our server-side implementation to an AWS instance and measured costs for computation, communication, and storage for a defined time window. By scaling the results obtained within that time window, we are able to make projections for an arbitrary number of sharing/recovering operations. We chose AWS to create a link between resource requirements and operational costs, but we recommend organizations to operate PPSS servers on their own infrastructure to avoid introducing a limited number of cloud service providers as risk for multiple servers.

	1 Server	40 Servers		
	Measured	Scaled	Price	Total
Operations	5.23 M	100.00 M		
Computation	1.00 h	31.89 days	0.194 $/h	$148.49
Traffic Out	8.55 GB	6.09 TB	0.090 $/GB	$548.47
Table Storage	88.84 MB	63.33 GB	0.100 $/GB	$6.33
Static Storage	1.00 GB	40.00 GB	0.100 $/GB	$4.00
				$707.28

TABLE III: Measurements on a Single AWS c5.xlarge Instance and Cost Projection for a 40-Server Setup

Test Setup. Our test setup consists of two AWS c5.xlarge instances running Amazon Linux AMI 2018.03.0 that are deployed in the same region. Firstly, the PPSS server exposes the implemented cryptography through a Rapidoid v5.5.5[2] web server with Java 1.8. Secondly, the load generator makes requests with wrk v4.1.0[3]. As client operations are considerably more costly than server operations, we avoid this bottleneck on the load generator by generating request values from an identical distribution with a Lua script. These instances communicate via TLS (tls_ecdhe_ecdsa_with_aes_128_gcm_sha256) using a self-signed certificate of a 256bit EC key.

Results. Table III presents the measurements for a single server over a one hour window as well as scaled costs for 100 million operations in a 40-server setup. As Split or Recover require the same resources, we summarize these terms as "operation". The main cost factors are outgoing traffic at \$548.47 and computation at \$148.86 when processing 100 million operations. Incoming traffic is free on AWS. Costs for times, when too powerful machines are idle, are outside the scope of our projection. Besides the constant storage size for OS and server application (assumed to be 1GB per server), a table for rate-limiting needs to be stored. Even if no table entries are cleared in a month-long period, the overall monthly storage costs are negligible. Apart from storage costs and idle times, our results are independent of the timespan. The sum of these operational costs, \$707.66, is split among all participating organizations. Assuming that each user splits and recovers her key once to/from a trust hierarchy of 40 servers, this setup is able to support 50 million users using 2 operations.

V. Discussion

Requirements for Recovery. In case a user needs to recover her keys, e.g., on a new device after losing the old one, the following is required: 1) The user needs to remember her password. 2) The user needs to have access to the chosen cloud storage. 3) A sufficient subset of trusted parties from the user's policy has to be available and honest. We assume that these requirements are satisfied at the time of recovery, as these requirements were the key aspects upon which users selected the involved cloud storage and PPSS servers.

Trust. The user trusts the cloud storage and share holders to not collude with each other. The cloud storage is trusted to store,

[2]https://www.rapidoid.org/
[3]https://github.com/wg/wrk

protect, hand out, and delete (if requested) the user's auxiliary information and encrypted keys. The user trusts that servers protect their keys, prevent online guessing, and do not deny service. Finally, the recommender is trusted not to manipulate users with its recommendations.

Threats. Next, we discuss threats made possible if actors violate the user's trust or become corrupted.

For *unauthorized key recovery*, an attacker needs to obtain a sufficient number of shares across the local, social, and remote domain: between t_{min} and t_{max} depending on which parts of the user's hierarchy are affected. To obtain remote shares, the attacker needs to mount a offline password guessing attack, which requires access to the user's encrypted shares on the cloud storage, as well as, t_{min} to t_{max} corrupted servers within the remote subtree.

For *denial of service*, an attacker has more options: The attacker could prevent the user from obtaining her auxiliary information (e.g., corrupting the user's cloud storage and deleting the data). Alternatively, attackers may try to guess or learn the user's uid, which allows them to lock the user out by triggering the servers' request limit. While guessing the uid might not be feasible, learning it could be achieved by corrupting the cloud storage or a PPSS server and reading the uid from the users' auxiliary data or incoming requests, respectively. Secure channels between user's client and actors (e.g., TLS and honest DNS) protect the uid from eavesdropping.

Usability. All concepts that are based on splitting the users' trust across multiple entities obviously face the challenge that users need to decide on whom to trust. In framework supports users by integrating user-selectable recommenders that suggest hierarchical trust policies with arguments underpinning their decisions. These recommendations may be adapted by users to their liking. Such a recommender also notifies users about changes that are relevant to their trust policies (e.g., legal changes), allowing them to revise the policy accordingly. Furthermore, we strive to design a universal system, where users only have to create their trust policy once, via the management app, while third-party apps may integrate with the management app's API to make use of the key recovery mechanisms.

VI. Conclusion

In this paper, we proposed a framework for key-loss recovery based on PPSS, which makes it possible to rely on human-memorizable passwords without being vulnerable to offline guessing attacks. Our framework contributes to addressing challenges when applying PPSS in practice, such as supporting users to reach trust decisions and convincing organizations that it is inexpensive to operate a PPSS server. In the framework, a management app splits a key into shares and distributes them locally (backup), socially (friends, family), and/or remotely (servers). This app allows the user, with the help of community-driven recommenders, to create and maintain hierarchically-organized trust policies. We implemented and extended Abdalla et al.'s PPSS scheme [5] to support hierarchical trust policies and prevent online guessing. Finally, we evaluated the costs of deploying our implementation on remote servers at large scale and came to the conclusion that serving 100 million requests (either split or recovery) would cost a consortium of 40 partners less than \$20 per partner.

References

[1] A. Shamir, "How to Share a Secret," *Commun. ACM*, vol. 22, no. 11, pp. 612–613, 1979.

[2] A. Bagherzandi, S. Jarecki, N. Saxena, and Y. Lu, "Password-Protected Secret Sharing," in *ACM Conference on Computer and Communications Security*. ACM, 2011, pp. 433–444.

[3] J. Camenisch, A. Lysyanskaya, and G. Neven, "Practical yet Universally Composable Two-Server Password-Authenticated Secret Sharing," in *ACM Conference on Computer and Communications Security*. ACM, 2012, pp. 525–536.

[4] G. J. Simmons, "How to (Really) Share a Secret," in *CRYPTO*, ser. LNCS, vol. 403. Springer, 1988, pp. 390–448.

[5] M. Abdalla, M. Cornejo, A. Nitulescu, and D. Pointcheval, "Robust Password-Protected Secret Sharing," in *ESORICS (2)*, ser. LNCS, vol. 9879. Springer, 2016, pp. 61–79.

[6] J. Camenisch, A. Lehmann, A. Lysyanskaya, and G. Neven, "Memento: How to Reconstruct Your Secrets from a Single Password in a Hostile Environment," in *CRYPTO (2)*, ser. LNCS, vol. 8617. Springer, 2014, pp. 256–275.

[7] X. Yi, F. Hao, L. Chen, and J. K. Liu, "Practical Threshold Password-Authenticated Secret Sharing Protocol," in *ESORICS (1)*, ser. LNCS, vol. 9326. Springer, 2015, pp. 347–365.

[8] S. Jarecki, A. Kiayias, and H. Krawczyk, "Round-Optimal Password-Protected Secret Sharing and T-PAKE in the Password-Only Model," in *ASIACRYPT (2)*, ser. LNCS, vol. 8874. Springer, 2014, pp. 233–253.

[9] S. Jarecki, A. Kiayias, H. Krawczyk, and J. Xu, "Highly-Efficient and Composable Password-Protected Secret Sharing (Or: How to Protect Your Bitcoin Wallet Online)," in *EuroS&P*. IEEE, 2016, pp. 276–291.

[10] ——, "TOPPSS: Cost-Minimal Password-Protected Secret Sharing Based on Threshold OPRF," in *ACNS*, ser. LNCS, vol. 10355. Springer, 2017, pp. 39–58.

[11] G. Brookner and L. Frey, "Technique for Split Knowledge Backup and Recovery of a Cryptographic Key," Feb. 7 2008, US Patent App. 11/708,750.

[12] Z. Huang, Q. Li, D. Zheng, K. Chen, and X. Li, "YI Cloud: Improving User Privacy with Secret Key Recovery in Cloud Storage," in *SOSE*. IEEE Computer Society, 2011, pp. 268–272.

[13] R. D'Souza, D. Jao, I. Mironov, and O. Pandey, "Publicly Verifiable Secret Sharing for Cloud-Based Key Management," in *INDOCRYPT*, ser. LNCS, vol. 7107. Springer, 2011, pp. 290–309.

[14] B. Kaliski, "PKCS #5: Password-Based Cryptography Specification Version 2.0," *RFC2898*, pp. 1–34, 2000.

[15] C. Percival, "Stronger Key Derivation via Sequential Memory-Hard Functions," 2009, BSDCan09.

[16] Y. Dodis, L. Reyzin, and A. D. Smith, "Fuzzy Extractors: How to Generate Strong Keys from Biometrics and Other Noisy Data," in *EUROCRYPT*, ser. LNCS, vol. 3027. Springer, 2004, pp. 523–540.

[17] A. T. B. Jin, D. N. C. Ling, and A. Goh, "Biohashing: Two Factor Authentication Featuring Fingerprint Data and Tokenised Random Number," *Pattern Recognition*, vol. 37, no. 11, pp. 2245–2255, 2004.

[18] A. Nagar, K. Nandakumar, and A. K. Jain, "Biometric Template Transformation: A Security Analysis," in *Media Forensics and Security*, ser. SPIE Proceedings, vol. 7541. SPIE, 2010, p. 75410O.

[19] United States, "The USA PATRIOT Act: Preserving Life and Liberty: Uniting and Strengthening America by Providing Appropriate Tools Required to Intercept and Obstruct Terrorism," U.S. Dept. of Justice, Tech. Rep., 2001.

[20] A. Menezes, P. C. van Oorschot, and S. A. Vanstone, *Handbook of Applied Cryptography*. CRC Press, 1996.

[21] C. G. Günther, "An Identity-Based Key-Exchange Protocol," in *EUROCRYPT*, ser. LNCS, vol. 434. Springer, 1989, pp. 29–37.

[22] D. Boneh, K. Lewi, H. W. Montgomery, and A. Raghunathan, "Key Homomorphic PRFs and Their Applications," in *CRYPTO (1)*, ser. LNCS, vol. 8042. Springer, 2013, pp. 410–428.

[23] National Institute of Standards & Technology, "SP 800-57. Recommendation for Key Management, Part 1: General (Rev 4)," Tech. Rep., 2016.

[24] T. Lorünser, A. Happe, and D. Slamanig, "ARCHISTAR: Towards Secure and Robust Cloud Based Data Sharing," in *CloudCom*. IEEE CS, 2015, pp. 371–378.