Threshold Key Storage via Fuzzy Extractors With Applications

Ciarán Mullan

Adva Network Security GmbH

Abstract. We propose a novel threshold key storage scheme that relies on biometric fuzzy extractors for the derivation of user keys. This approach builds upon the existing framework of password-protected secret sharing constructions, offering a potential improvement in security and user convenience. Usually in such schemes, users are required to generate and remember passwords, which in practice can introduce vulnerabilities and usability issues. By removing this reliance on traditional passwords, our scheme may enhance the overall security of threshold key storage solutions. Moreover, in situations where password recovery is not a viable option, our mechanism provides a dependable solution for online private key storage, ensuring users can access their keys securely and reliably.

Keywords: Key management, secret sharing, threshold cryptography, fuzzy extractors.

1 Introduction

Secret sharing is a cryptographic technique that splits a secret into several shares and distributes them to a group of semi-trusted parties. The parties learn nothing about the secret, which can be reconstructed at a later stage only if a certain threshold number of shares are recombined. One primary application of secret sharing is distributed key storage. Instead of storing a cryptographic key in a single place where it may be lost or stolen, it can be secret shared and distributed to a set of storage servers. In this way, the key does not exist in any single location and by the properties of the secret sharing scheme, both secrecy and availability of the key is ensured even if a limited number of shares are compromised. When performing a cryptographic operation, it suffices to gather enough consistent shares in one place to reconstruct the original key[1].

One standout attribute of secret sharing schemes is password-protection, whereby knowledge of a password is required to reconstruct the secret. This is significant because in a regular secret sharing scheme, any threshold number of parties can join forces to compute the secret. Unless this might be considered useful for the application, there is no reason to permit it. A number of works has been considered in this line [1], [2], [3], with perhaps the most efficient one to date being Jarecki et al [4]. Despite their appeal, the drawback to password-protection schemes is that the user must create and remember their credentials. Forgetting a username or password here is a disaster, as without any recovery mechanism (only the user ever knows the password), the secret is lost forever.

In this work, we demonstrate that the desirable security features that password-protection adds to distributed key storage applications can be preserved without requiring the user to remember any information. For this, we appeal to fuzzy extractors and biometric readings for key derivation. User keys are recomputed on the fly so there is no need to store them or any private biometric data.

We begin in Section 2 with a discussion of the relevant building blocks. In Section 3 we include a description of the most competitive password-protected secret sharing scheme to

[1] We are not concerned here with more sophisticated cryptographic techniques that involve parties performing computations over their shares, such as in secure multiparty computation.

David C. Wyld et al. (Eds): BIOM, CRBL, EDUPT, SIP, COMIT -2023
pp. 15-23, 2023. CS & IT - CSCP 2023 DOI: 10.5121/csit.2023.132102

date [4], which we use as a basis for our work. In Section 4 we describe our scheme and in Section 5 we consider two possible applications of the construction. In Section 6 we close with suggestions for future research.

2 Building Blocks

The cryptographic building blocks we use are Shamir's secret sharing scheme, threshold oblivious pseudorandom functions, and fuzzy extractors.

Shamir's Secret Sharing Scheme

In Shamir's (t, n) secret sharing scheme [5], let $\mathbb{F}_q$ be a finite field of prime power order q. To share a secret $k \in \mathbb{F}_q$, a user first selects values for a threshold parameter $t > 1$ and the number n of shares to generate, with $n > t$. The user then picks random elements $a_1, \ldots, a_{t-1} \in \mathbb{F}_q$ and constructs the polynomial $P(x) = k + \sum_{i=1}^{t-1} a_i x^i$. For $i = 1, 2, \ldots, n$, the user generates shares $s_i := (i, P(i))$ and distributes s_i to server S_i over secure channels. To later recover k, the user asks t of the servers to return their shares. Then k can be constructed using polynomial interpolation as

$$k := P(0) = \sum_{j=0}^{t-1} y_j \prod_{m=0, m \neq j}^{t-1} \frac{x_m}{x_m - x_j}.$$

Since knowledge of any $t-1$ shares leaves k undetermined, Shamir's scheme is both robust against corruption of up to $t-1$ servers, and it ensures availability as only t of the n servers are needed to recreate k.

We remark that it is possible to extend Shamir's efficient scheme in various directions, such as generalized access structures, dynamic addition of new shares or polynomial (keeping k fixed), changing the threshold value t, and so on. However, we won't use any of these options in our scheme.

Fuzzy Extractors

Fuzzy extractors, as introduced by Dodis et al. [6] provide a mechanism that given noisy, non-uniform input, reliably outputs a (nearly) random string s. The extraction process takes care of small errors, so that even if the input is changed slightly, the extractor still outputs the same string s. When used with user biometric input such as a fingerprint or iris scan, fuzzy extractors can be used to derive user-specific cryptographic keys from s. We present a formal definition of fuzzy extractors formulated by Boyen [7], as follows.

Let $\mathcal{M}$ be a metric space with distance function d, $\mathcal{H}_\infty$ be the min-entropy function, $D[A_1, A_2]$ be the statistical distance between two discrete probability distributions over a common domain, and let U_l be the uniform distribution over bitstrings of length l. An $(\mathcal{M}, m, l, t, \epsilon)$-*fuzzy extractor* is a pair of algorithms ($\mathcal{E}$-*Gen*, $\mathcal{E}$-*Reg*), where:

- On input $w \in \mathcal{M}$, the randomized algorithm $\mathcal{E}$-*Gen* extracts a private string $s \in \{0,1\}^l$ and a public string P such that for all random variables W over $\mathcal{M}$ such that $H_\infty[W] \geq m$ and dependent variables $(s, P) \leftarrow \mathcal{E}\text{-}Gen[W]$ it holds that $D[(s, P), (U_l, P)] \leq \epsilon$.
- On input $w' \in \mathcal{M}$ and public string P, $\mathcal{E}$-*Reg* outputs a string $s' \in \{0,1\}^l$ such that for any $w, w' \in \mathcal{M}$ with $d(w, w') \leq t$ and any pair $(s, P) \leftarrow \mathcal{E}\text{-}Gen[w]$, it holds that $s = \mathcal{E}\text{-}Reg(w', P)$.

Threshold Oblivious Pseudorandom Functions

Informally, a family of functions $f_k : \{0,1\}^a \to \{0,1\}^b$ with key k is pseudorandom (PRF) if $f_k(x)$ is efficiently computable given k, x, else without knowledge of k, x, indistinguishable from random. An *oblivious* PRF (OPRF) is then a protocol between a user and a server such that given a PRF family f_k, where the user holds x and the server holds k, at the end of the protocol the user learns $f_k(x)$ and the server learns nothing.

OPRFs are still a relatively new cryptographic notion, but due to their numerous applications are receiving more attention [8]. The OPRF of interest in this work is 2HashDH [2]. Let G be a cyclic group of prime order m, and let H, H' be hash functions with ranges $\{0,1\}^l$ and G respectively. For a random key $k \in \mathbb{Z}_m$ 2HashDH is defined as

$$f_k(x) := H(x, (H'(x)^k).$$

This PRF may be evaluated obliviously by a user holding x and server holding k as follows. The user generates a random $r \in \mathbb{Z}_m$ and sends $a = H'(x)^r \in G$. The server replies with $b = a^k \in G$ and the user computes $f_k(x) = H(x, b^{1/r})$.

Subsequently, Jarecki et al. [4] extend 2HashDH to a threshold setting ('2HashTDH') by letting k be (t, n)-Shamir secret shared. For a threshold oblivious evaluation, user sends $a = (H'(x))^r$ to t servers who reply with $b_i = a^{k_i}$. The user computes $b = a^k = \prod b_i^{\lambda_i}$, where λ_i are the Lagrange coefficients. The user can proceed to compute $f_k(x) = H(x, b^{1/r})$. The constructions are shown secure under One-More Diffie-Hellman style assumptions in the random oracle model.

TOPPSS: A Concrete Instantiation Using 2HashTDH, Jarecki et al. [4] build a password-protected secret sharing scheme called TOPPSS, whereby in their concrete instantiation, the user sets x to be their password pw and the secret key is derived from $H(pw, H'(pw)^k)$. As our scheme uses this instantiation as a basis, we include a description in Figure 1 which is reprinted from [4]. The details of the scheme will become clearer in the next section. For now, note that the quantity sid is taken to be the username, $\mathcal{SI}$ is the initial set of shareholders of size n, $\mathcal{SR} \subset \mathcal{SI}$ is the set of servers[2] selected for secret reconstruction, and $ssid$ is a unique session value.

As Jarecki et al. mention, share distribution must go over secure channels (TLS, for example). And while it is possible to perform share reconstruction over insecure channels, to avoid certain man in-the-middle attacks, it is preferable also to use secure channels for share reconstruction. Furthermore the user is required to remember their username and password, something which experience tells is not a reliable assumption for a lot of humans.

In the next section we'll see how to overcome these issues whereby the user need not remember any keys or passwords, whilst maintaining the strong security features that the concrete instantiation of TOPPSS provides.

3 Threshold Key Storage Via Fuzzy Extractors

We now describe the components of our threshold key storage scheme which comes with strong security properties yet avoids having users remember credentials such as usernames and passwords. Since they are derived from a recomputable fuzzy extractor algorithm, there are no memory or storage requirements placed on the user. Moreover, it is reasonable to expect y_u to contain more entropy than a typical human memorable password. This is

[2] Note: in TOPPSS $t + 1$ shares are required for reconstruction.

Parameters: Security parameter ℓ, threshold parameters t, n, cyclic group $\langle g \rangle$ of prime order m, hash functions H_1, H_2, H_3 with ranges $\langle g \rangle$, $\{0,1\}^{2\ell}$ and $\{0,1\}^{2\ell}$, respectively.
Communication Setting: Communication between U and S_i's in INIT goes over secure channels, e.g. TLS, communication in REC not necessarily (see text).

INIT for user U on input $(sid, \mathcal{SI}, \mathsf{pw})$ where $\mathcal{SI} = \{S_1, ..., S_n\}$:

1. Pick $k \leftarrow_R \mathbb{Z}_m$, generate $(k_1, \ldots, k_n)$ as a (t, n)-Shamir's secret sharing of k over $\mathbb{Z}_m$, and send $(sid, \mathcal{SI}, i, k_i)$ to each $S_i \in \mathcal{SI}$.
2. After receiving ACK's from all servers in $\mathcal{SI}$, compute $v := H_2(\mathsf{pw}, H_1(\mathsf{pw})^k)$, parse $H_3(v)$ as $[C|K]$, send (sid, C) to all $S_i \in \mathcal{SI}$ and output K.

INIT for server S_i:

1. On $(sid, \mathcal{SI}, i, k_i)$ from U, abort if sid not unique or $S_i \notin \mathcal{SI}$, otherwise record $(sid, \mathcal{SI}, i, k_i)$ and send (sid, ACK) to U.
2. On (sid, C) from U, append C to tuple $(sid, \mathcal{SI}, i, k_i)$.

REC for user U on input $(sid, ssid, \mathcal{SR}, \mathsf{pw})$ where $\mathcal{SR} = \{S'_1, ..., S'_{t+1}\}$:

1. Pick $r \leftarrow_R \mathbb{Z}_m$, set $a := H_1(x)^r$, send $((sid, ssid), (\mathcal{SR}, a))$ to all $S'_i \in \mathcal{SR}$.
2. After receiving message $((sid, ssid), b_i, C)$ from all servers in $\mathcal{SR}$, abort if not all C's are the same. Otherwise compute $b := \prod_{S'_i \in \mathcal{SR}} b_i$ and $v := H_2(x, b^{1/r})$.
3. Parse $H_3(v)$ as $[C'|K']$. If $C' = C$ then output K', otherwise output FAIL.

REC for server $S_i \in \mathcal{SR}$:

1. On message $((sid, ssid), (\mathcal{SR}, a))$ from U, recover stored tuple $(sid, \mathcal{SI}, i, k_i, C)$. (Abort if there is no such tuple or if $a \notin \langle g \rangle$.) Compute $b_i := a^{\lambda_i \cdot k_i}$ where λ_i is an interpolation coefficient for index i and the subset of $t+1$ indices defined by $\mathcal{SR} \subseteq \mathcal{SI}$, and send $((sid, ssid), b_i, C)$ to U.

Fig. 1. Concrete Instantiation of TOPPSS based on 2HashTDH [4].

significant because if an adversary ever learns the OPRF key k, he may start to guess pw in order to learn K. With more entropy contained in y_u, this guessing strategy is much less likely to succeed.

We will give an example application using these components in the next section, where it will become clear which party perform which steps.The eight components of our scheme are: Parameter Selection, OPRF Key Generation, OPRF Key Share Generation, User Key Generation, Secret Key Generation, OPRF Key Share Distribution, User Key Regeneration, Secret Key Regeneration.

3.1 Parameter Selection

We begin by selecting values for the secret sharing parameters l, m, t, n, where l is the security parameter (typically $l = 128$ or $l = 256$), m a large prime ($m \approx 2^l$), t the threshold parameter, and n the number of shares/servers. Choices for t and n depend on

tradeoffs between secrecy, availability, and efficiency. But for example t may be a small positive integer and n may be a simple function of t, e.g. $n = 2t+1$ or $n = 3t$. A description $\mathcal{D} = \{l, m, t, n\}$ of the scheme, to which we will append other items, is output.

Algorithm 1 `Parameter Selection`
`Input:` $\emptyset$
`Output: Description` $\mathcal{D} = \{l, m, t, n\}$

1: Select values for l, m, t, n and set $\mathcal{D} = \{l, m, t, n\}$.

3.2 OPRF Key Generation

A random OPRF key k is generated that is to be secret-shared and distributed to the storage servers.

Algorithm 2 `OPRF Key Generation`
`Input:` $\mathcal{D} = \{l, m, t, n\}$
`Output: OPRF key` $k \in \mathbb{Z}_m$

1: Pick $k \in \mathbb{Z}_m$ uniformly at random.

3.3 OPRF Key Share Generation

This is standard (t, n)-Shamir secret sharing of the OPRF key k. Namely, pick a random degree $t-1$ polynomial f over $\mathbb{Z}_m$ subject to the condition that the constant term is equal to k. For convenience, let the n shares be generated as points $(i, f(i))$ for $i = 1 \to n$.

Algorithm 3 `OPRF Key Share Generation`
`Input:` $\mathcal{D}$, k
`Output: Shares` $\mathcal{K} = (k_1, \ldots, k_n)$ `of` k

1: Perform (t, n)-Shamir secret sharing of k over $\mathbb{Z}_m$.

3.4 User Key Generation

Here we use a fuzzy extractor scheme. The algorithm $\mathcal{E}$-*Gen* is applied on user biometric input w (e.g. a fingerprint reading) to produce a random user key $y_u \in \{0,1\}^l$ and helper string P. Quantity y_u is used as user input to the threshold OPRF protocol, and replaces the user password element pw of the TOPPSS scheme.

Algorithm 4 `User Key Generation`
`Input:` $\mathcal{D}$, `user biometric reading` w
`Output: User key` $y_u \in \{0,1\}^l$ `and helper string` P

1: User U computes $\mathcal{E}$-$Gen(w)$ on input reading w and extracts helper string P and user key $y_u \in \{0,1\}^l$.
2: U stores and backs up P. P may be considered public.

3.5 Secret Key Generation

In this step the actual secret key $K \in \{0,1\}^l$ is computed. This value is derived from both the random OPRF key k and the user key y_u. Check value C is also created, just as in TOPPSS, which serves to verify correctness of secret regeneration in a later step. We refer the reader to [4] for background details.

Algorithm 5 `Secret Key Generation`
`Input:` $\mathcal{D}$, k, y_u
`Output: Check value` C, `secret` K

1: Let $G = \langle g \rangle$ be a cyclic group of prime order m and let H_1, H_2, H_3 be hash functions with ranges G, $\{0,1\}^{2l}$, $\{0,1\}^{2l}$ respectively. Append to $\mathcal{D}$ a description of $\{G = \langle g \rangle, H_1, H_2, H_3\}$.
2: Compute $v := H_2(y_u, H_1(y_u)^k)$ and output $(C, K) := H_3(v)$.

3.6 OPRF Key Share Distribution

In this step shares $\mathcal{K} = (k_1, \ldots, k_n)$ of the OPRF key k are distributed. Here we assume the existence of (at least) n storage servers S_i, $i = 1 \to n$, all of which are capable of establishing a secure (authenticated, encrypted) channel with whomever distributes the shares. (For some applications, various concerns may enter here. For example, a user may be concerned with which countries servers are located in, with which policy the n servers are chosen, which entities own the servers, etc.)

Algorithm 6 `OPRF Key Share Distribution`
`Input: check value` C, `OPRF key shares` $\mathcal{K} = (k_1, \ldots, k_n)$
`Output:` $\mathcal{D}$

1: Over a secure channel send (k_i, C) to S_i.

3.7 User Key Regeneration

If the user wishes to reconstruct their secret key, they first retrieve stored helper string P and take a new biometric reading; $\mathcal{E}$-$Reg(w', P)$ is called to reliably recompute user key y_u.

Algorithm 7 `User Key Regeneration`
`Input: User reading` w', `helper string` P
`Output:` $y_u \in \{0,1\}^l$

1: Retrieve stored string P.
2: Regenerate user key $y_u \in \{0,1\}^l$ by calling $\mathcal{E}$-$Reg(w', P)$ on new input sample w'.

3.8 Secret Key Regeneration

In this step a threshold oblivious computation is performed to reconstruct the secret K. About error handling, in some scenarios, the simple check $C' == C$ may not suffice as it does not identify where the error is located, i.e. which shareholder(s) became malicious or corrupted. If this is a concern, at some extra cost it is possible to include share verifiability

to identify corrupt OPRF key shareholders [2]. Alternatively, assuming the existence of t consistent shares, a general combinatorial method may be employed to detect corrupt shares.

Algorithm 8 Secret Key Regeneration
`Input: user input` y_u, t `server shares` k_i
`Output: user outputs` K

1: U picks a random $r \in \mathbb{Z}_m$ and sends $a := H_1(y_u)^r$ to t servers S_i, $i = 1 \to t$.
2: Server S_i replies with $b_i := a^{k_i}, C$.
3: U computes $b = H_1(y_u)^k$ via Lagrangian interpolation in the exponents amd computes $(C', K') = H_3(H_2(y_u, b))$.
4: If $C' == C$, U outputs K'. Else, `handle error`.

4 Application

In this section we describe an application that utilize the above components. The application is private key storage, which may be of interest in the cryptocurrency space.

Private Key Storage

In this scenario, consider a crypto-currency user wishing to protect their private key which unlocks their crypto-assets. Current popular methods for private key derivation use human language word sequences as seed phrases (BIP39 [9], SLIP39 [10]). The user needs to privately store this word seed, because if it is lost then the assets are irretrievable. In what is called *Shamir backup*, the seed is protected using Shamir's (t, n) secret sharing, thus preventing a single point of failure.

Utilizing the above components, we outline an alternative method for private key generation and backup. Let the user be denoted by U. We assume U holds a biometric reader $\mathcal{X}$ implementing $\mathcal{E}$-*Gen* and $\mathcal{E}$-*Reg*, and access to a small number of storage servers $S_1, \ldots, S_n$. As before, we use a group $G = \langle g \rangle$ of order m and hash functions H_1, H_2, H_3 with ranges G, $\{0,1\}^{2l}$, $\{0,1\}^{2l}$, respectively.

Algorithm 9 Private Key Restoration

1: Key Generation. U selects parameters $\{l, m, t, n\}$, takes a reading $(y_u, P) = \mathcal{E}\text{-}Gen(w)$, and generates a random OPRF key $k \in \mathbb{Z}_m$. U then computes $v := H_2(y_u, H_1(y_u)^k)$ and sets $(C, K) := H_3(v)$. The private key is derived from K. P is stored and backed up, and may be considered public. y_u is discarded. The OPRF key k may optionally be kept offline, under the mattress.
2: Share Distribution. U creates secret shares $(k_1, \ldots, k_n)$ of the OPRF key k and over a secure channel sends (C, k_i) to server S_i.
3: Key Regeneration. When it is time to use the private key, U retrieves P and recomputes $y_u = \mathcal{E}\text{-}Reg(w', P)$. They then pick a random $r \in \mathbb{Z}_m$ and send $a := H_1(y_u)^r$ to t of the n servers S_i. The S_i reply with $b_i := a^{k_i}, C$. U computes $b = H_1(y_u)^k$ via Lagrangian interpolation in the exponents and computes $(C', K') = H_3(H_2(y_u, b))$. If C' does not equal C, `handle error`. Else, U has computed K and hence the private key.

Discussion

First note that the user only needs to store P (well, along with the parameters of the scheme and locations of the shares). Since P is assumed to pose no security risk, it may

be stored and backed up in any reliable manner. Next note that even if an adversary learns a threshold number of shares of the OPRF key k, no information is leaked about the private key. Hence an adversary cannot steal the underlying crypto-assets by stealing the shares. Contrasted with the *Shamir backup* approach, this is an advantage, and the worst an adversary could do it to destroy the shares (in which case the private key is still recoverable, if k was safely stored offline). Another nice property is that a travelling user need only carry the reader $\mathcal{X}$, which by itself contains no information. So in theory, the private key may exist only in volatile memory when in use.

One shortcoming of the scheme is that the servers need to perform a computation, namely modular exponentiation. So a regular storage server will need adapting to implement this. There are also open questions surrounding how to implement a fuzzy extractor which confidently satisfies the theoretical definition. Practical instances of fuzzy extractors need to be carefully assessed in terms of their entropy output, both in terms of intra-user and inter-user aspects. An interesting approach worthy of further investigation is to explore the efficacy of machine learning when performing biological readings for cryptographic key generation [11].

Conclusion

In this work, we introduced fuzzy extractors as a mechanism for protecting secret shares in threshold key storage schemes. The solution removes requirements on users having to remember passwords, which is of particular importance when password recovery mechanisms are not available.

Acknowledgment

This work has been partially funded in the framework of the CELTIC-NEXT project AI-NET-PROTECT (Project ID C2019/3-4) by the German Federal Ministry of Education and Research (#16KIS1279K).

References

1. A. Bagherzandi, S. Jarecki, N. Saxena, Y. Lu, “Password-protected secret sharing.” In Proceedings of the 18th ACM conference on Computer and Communications Security, 2011, October, pp. 433-444.
2. S. Jarecki, A. Kiayias, H. Krawczyk, “Round-optimal password-protected secret sharing and T-PAKE in the password-only model,” In Advances in Cryptology–ASIACRYPT 2014: 20th International Conference on the Theory and Application of Cryptology and Information Security, Kaoshiung, Taiwan, ROC, December 7-11, 2014, Proceedings, Part II 20 2014, Springer Berlin Heidelberg, pp. 233-253.
3. 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 IEEE European Symposium on Security and Privacy – EuroS&P 2016, pp. 276–291.
4. S. Jarecki, A. Kiayias, H. Krawczyk, J. Xu, “TOPPSS: cost-minimal password-protected secret sharing based on threshold OPRF,” In Applied Cryptography and Network Security – ACNS 2017, Springer, 2017, pp. 39-58.
5. A. Shamir, “How to share a secret,” Communications of the ACM 22.11, 1979, pp. 612-613.
6. Y. Dodis, L. Reyzin, A. Smith, “Fuzzy extractors: How to generate strong keys from biometrics and other noisy data,” In Advances in Cryptology-EUROCRYPT 2004: International Conference on the Theory and Applications of Cryptographic Techniques, Interlaken, Switzerland, May 2-6, 2004. Proceedings 23, Springer Berlin Heidelberg, 2004, pp. 523-540.
7. X. Boyen, “Reusable cryptographic fuzzy extractors,”. In Proceedings of the 11th ACM conference on Computer and Communications Security 2004 Oct 25, pp. 82-91.
8. S. Casacuberta, J. Hesse, A. Lehmann, “SoK: Oblivious Pseudorandom Functions,” In 2022 IEEE 7th European Symposium on Security and Privacy (EuroS&P) 2022 Jun 6 pp. 625-646.
9. https://github.com/bitcoin/bips/blob/master/bip-0039.mediawiki

10. https://github.com/satoshilabs/slips/blob/master/slip-0039.md
11. Z. Wu, Z. Lv, J. Kang, W. Ding, J. Zhang, “Fingerprint bio-key generation based on a deep neural network,” International Journal of Intelligent Systems. 2022 Jul; 37(7), pp. 4329-58.