Fuzzy Encryption for Secret Recovery

Esha Ghosh Daniel Buchner Jonathan Lee Melissa Chase Kirk Olynyk
Piali Choudhury Rahee Peshawaria

April 2020

1 Introduction

The introduction of Bitcoin led to a renewed interest in decentralized systems backed by users-controlled, user-custodied cryptographic keys. All current implementations of cryptocurrencies, 'smart contract' systems, and decentralized identifier protocols require users to maintain secure control over cryptographically random secrets composed of long strings (e.g. 32 byte private keys) that humans are generally unable to remember.

The decentralized systems and applied cryptography communities have generally employed two mechanisms for aiding in the recovery of these unmemorable secrets: 1) randomly selected mnemonic phrases, and 2) Shamir-based secret sharing schemes:

Mnemonic phrase schemes (e.g., BIP 39) generate a set of 12-24 randomly selected words from a corpus that forms a secret seed. While these mnemonic schemes map an unmemorable secret to words, affording some degree of memorability and human error correction (e.g., illegibly written words can be deduced), successful regeneration of the secret requires the user to produce all the words in their exact form and order. These requirements make mnemonic schemes rather unwieldy in practice.

Shamir-based secret sharing schemes use polynomial interpolation as a basis for dividing a secret into N shares, wherein some subset threshold of T shares must be recombined in any order to regenerate the secret. The user can distribute the resulting shares in different ways, for example, storing shares on different devices, hiding paper printouts of shares in different physical locations, or sending shares to a set of collaborators for future retrieval. Typically, this scheme is used for what's known as 'social recovery' models, where you use an app to distribute shares to selected friends or contacts. While this common secret sharing scheme affords the user some flexibility by only requiring them to reproduce a subset of shares, in any order, the shares themselves are long, cryptographically random strings, which makes human recollection of shares impossible.

Both schemes provide various advantages over simply attempting to remember or store long, unmemorable secret strings of random numbers and letters, but the approaches come with almost diametrically opposed trade-offs. With mnemonic phrases, inputs are words, which increases memorability, but users must correctly reproduce all words in the exact order in which they were generated. Secret sharing schemes, on the other hand, only require the user to reproduce a threshold subset of shares, in any order. Unlike mnemonic phrases, however, the shares are long, random strings of letters and numbers, not words or other human-friendly inputs.

The goal of this scheme is to deliver the desirable features of both the aforementioned schemes in one mechanism that allows a user to encrypt a secret with a set of N stringifiable inputs (words, images, etc.), wherein only a T threshold subset of inputs must be recombined in any order to decrypt the secret. The level of security afforded by the scheme is based on a number of factors, including size of the input corpus, entropy of the input selection, and the tolerance of the threshold. The input corpuses, approaches for input selection, and recovery user experiences that produce the best outcomes are still being investigated.

2 Envisioned Technology

We are envisioning a technology that will enable harnessing the entropy in the human-memorizable sets and generate strong cryptographic keys from it, while tolerating a small number of errors. The desirable/required features of our technology are:

1. *No cryptographic secret to memorize*: We will like our scheme to be usable without any need to remember or protect cryptographic secrets. This means, a user is expected to remember/securely protect her pass-phrase only. Any state information generated by the scheme in order to generate the cryptographic key material from the pass-phrase can be stored in any public repository.

2. *Secure recovery*: Only a correct pass-phrase (tolerating a small number of errors) will be able to recover the cryptographic key. The error tolerance threshold is a system parameter. Any pass-phrase that has more than that many errors should fail to recover the cryptographic key or any partial information about it.

3. *Reusability of pass-phrases*: A user should be able to reuse her pass-phrase to generate many cryptographic keys. For example, if a user generates a cryptographic key using a pass-phrase and the key gets compromised, she needs to generate a new cryptographic key. We want her to be able to re-use her pass-phrase to generate a fresh cryptographic key.

4. *Non-iterability*: A pass-phrase will consist of a set of elements, for example "I love to sail forbidden seas, and land on barbarous coasts" is a candidate pass-phrase which consists of a collection of words. We want to have the following property: an adversary who has the public state information will be able to validate its guess for a complete pass-phrase using the state information (by checking if the secret key recovery fails or not), but it should not be able to validate its guess for individual elements in the pass-phrase. For example, an adversary guesses just one word, say, forbidden. It should not be able to learn whether the guessed word belongs to the pass-phrase or not.

3 Our Scheme

We directly use the scheme used in [1] and extend it to satisfy all the features listed above. In particular, the scheme in [1] already satisfies properties 1,2 and 4 listed in Section 2. We extend it to support feature 3. Here we first describe the technical details of our scheme and then discuss how the different parameters contribute build a cryptographically strong system.

3.1 Technical Preliminaries

Each algorithm/party in our scheme, including the potential adversary, has access to the params:

- A mapping function map that maps every word in the input domain to a number between 1 and $\mathsf{corpus\_size}$. For the algorithms we assume the input words are already mapped to a number. Both $\mathsf{map}, \mathsf{map}^{-1}$ are publicly known.
- A pseudorandom function f (like HMAC)
- A memory-hard hash function H (like scrypt)

Input Set Let us denote the human-memorizable set if inputs as $\tilde{\mathsf{W}}$. Let $\mathsf{setsize}$ denote the size of $\tilde{\mathsf{W}}$. Note that the size of the universe for the input set can either be large (superpolynomial in $\mathsf{setsize}$) or small (polynomial in the size of $\mathsf{setsize}$). Our proposed scheme builds on the large universe scheme from [1] and is more general than constructions for small universe [1].

3.2 Scheme

Our proposed scheme has the following algorithms. The high level idea of our scheme is to combine the scheme described above with a universal hash function [2] based extractor [1] to first generate a master cryptographic secret key and then use a PRF to derive multiple cryptographic keys from this master secret key.

SetupParams(setsize, correct_threshold, corpus_size, p, usersalt, extractor) $\rightarrow$ params

This algorithm generates all the parameters of the scheme where the input is defined as follows:

- setsize: this is the number of words required for establishing the initial secret and for recovering the secret. This is specified by the user.
- correct_threshold: this is the minumum number of words that must be correctly guessed in order to successfully recover the secret. This must be greater than half of the number of words (setsize). This is specified by the user.
- corpus_size: This is the size of the set of allowed words. This means that both the original words and recovery words must be represented by integers in the range $[0 \ldots (\mathsf{corpus\_size} - 1)$. This is specified by the user.
- p: This a prime number such that $\mathsf{setsize} < \mathsf{p} < 2 \times \mathsf{setsize}$. The user does not set this number, instead it is derived from the corpus_size which is specified by the user. This prime defines a finite field F_{p} of prime order p
- usersalt: user specific salt to slowdown brute-force attack. Created by the constructor. This value is not specified by the user.
- extractor: bytes required of key generation. More specifically, $\mathsf{extractor} = (s_0, s_1, s_2, \ldots, s_n) \in F_p^n$ where each s_i is chosen uniformly at random from F_p^n. This is automatically generated by the constructor. This value is not specified by the user.

GenerateSecret(params, W, keynums) $\rightarrow$ (sk[keynums], state)

This algorithms works in the following steps:

1. Let $\mathsf{W}_{\mathsf{sorted}} = \{a_1, \ldots, a_n\}$
2. Call inputhash $\leftarrow \mathsf{H}(original\_words || \mathsf{W}_{\mathsf{sorted}})$
3. Call GenSketch(params, W). Let this return state
4. Append inputhash to state
5. Compute the following: $e = \prod_{i=1}^{n} s_i * a_i \ mod \ p$
6. Compute ek $\leftarrow \mathsf{H}(keys || e)$
7. For $i = [1, \mathsf{keynums}]$ do the following: $\mathsf{sk}[i] \leftarrow f_{\mathsf{ek}}(i)$
8. The algorithm outputs (sk[keynums], state)

RecoverSecret(params, state, W′, keynums) $\rightarrow$ (sk[keynums]/$\perp$)

1. Let $\mathsf{W}'_{\mathsf{sorted}}$ be the sorted set.
2. If $\mathsf{state.inputhash} = \mathsf{H}(original\_words || \mathsf{W}'_{\mathsf{sorted}})$, then Goto Step 6. Else go to the next step.
3. Call RecSet(params, state, W′). If this returns $\perp$, abort. If not, let this return W.
4. Let $\mathsf{W}_{\mathsf{sorted}} = \{a_1, \ldots, a_n\}$.
5. Check if $\mathsf{state.inputhash} = \mathsf{H}(original\_words || \mathsf{W}_{\mathsf{sorted}})$. If not, abort. Else go to the next step.
6. Compute the following: $e = \prod_{i=1}^{n} s_i * a_i \ mod \ p$
7. Compute ek $\leftarrow \mathsf{H}(keys || e)$
8. For $i = [1, \mathsf{keynums}]$ do the following: $\mathsf{sk}[i] \leftarrow f_{\mathsf{ek}}(i)$.
9. The algorithm outputs (sk[keynums]).

GenSketch(params, W) $\rightarrow$ state

Let $W = \{x_1, \ldots, x_s\}$. Construct polynomial p' with roots from W:

That is, let $p'(z) = \prod_{x_i \in w}(z - x_i)$

Let $p'(z) =$
$z^s + \sum_{i \in [s-1]} \alpha_i z^i$, then output the top t coefficients $(\alpha_{s-1}, \ldots, \alpha_{s-t})$.
(By expanding out $\prod_{x \in w}(z - x)$,

$\alpha_{s-1} = \sum_{x_i \in w} x_i$,
$\alpha_{s-2} = \sum_{x_i, x_j \in w, i \neq j} x_i x_j$,
...
$\alpha_{s-t} = \sum_{S \subseteq [s], |S| = t} \left(\prod_{i \in S} x_i\right)$

RecSet(params, W′, state) $\rightarrow$ W.
Let $W' = \{x_1, \ldots, x_s\}$.

Construct polynomial $p_{\text{high}}(\cdot)$ of degree s as follows:

p_{high} is the polynomial of degree s whose top coefficient is 1,the next t coefficients (that is, $(s-1, \ldots s-t)$) come from the state, and the remaining coefficients are 0

Compute $\{b_1, \ldots, b_s\}$ as :

$b_i = p_{\text{high}}(a_i), i \in [s]$

Find a polynomial p_{low} of degree $s - t - 1$ such that $p_{\text{low}}(a_i) = b_i$ for at leat $s - t/2$ of the a_i using Berlekamp-Welsh-Decoder:

$p_{\text{low}} \leftarrow$ Berlekamp-Welsch-Decoder $\left(\{(a_i, b_i)\}_{i \in [s]}, s, s - t, t/2, p\right)$
If no such polynomial exists

Output Fail

Else

Set $p_{\text{diff}} = p_{\text{high}} - p_{\text{low}}$
Check if p_{diff} has distinct roots, else abort

Check $z^p - z \equiv 0 \pmod{p_{\text{diff}}}$ (Note that $z^p - z = \prod_{\alpha \in \mathbb{F}_p}(z - \alpha)$ due to Fermat's little theorem. Thus $p_{\text{diff}} \mid z^p - z$ if and only if it does not have repeated roots.)

$W \leftarrow$ Find-Roots(p_{diff})
(Find-Roots returns all the roots of a given polynomial)

Berlekamp-Welsch-Decoder $\left(\{(a_i, b_i)\}_{i\in[n]}, n, k, t, p\right)$
(fixes upto t-errors, assuming $t \leq \frac{n-k}{2}$)

Figure out the error locator polynomial E and the corrector polynomial Q:

- We want to find E with degree t (such that $E(a_i) = 0$ iff a_i is an error location, i.e., $b_i \neq x_i$ where x_i was the original set element that was encoded)
- Let $E = E_0 + E_1 x + \cdots + E_{t-1}x^{t-1} + x^t$
- Q has degree $k + t - 1$, such that $\forall i \in [s], Q(a_i) = E(a_i)x_i$ where x_i is the original set element.
  (Note that $\forall i \in [s], Q(a_i) = E(a_i)b_i$ by definition of error locator, since $E(a_i) \neq x_i,\ E(a_i) = 0$ and otherwise $E(a_i) = b_i = x_i$.)
- Let $Q = Q_0 + Q_1 x + \cdots + Q_{k+t-1}x^{k+t-1}$
- Solve the following linear equations over $\mathbb{F}_p$ to get Q, E
  - for each $a_i \in [s]$
    $$\sum_{j=0}^{k+t-1} Q_j \left(a_i\right)^j = b_i \sum_{j=0}^{t} E_j \left(a_i\right)^j$$
  - $E_t = 1$ (This forces the above LP to return non-trivial solutions. Without this constraint, note that, $Q = 0, E = 0$ is valid solution. Also note that LP with the added constraint $E_t = 1$ is feasible if the LP without it has a non-trivial solution. To see this, assume that Q', E' are non-zero polynomials satisfying every other constraint and the leading coefficient of E', E'_t is not 1. Then we can get Q, E satisfying $E_t = 1$ by dividing everything with E'_t. That is $Q = \frac{1}{E'_t}Q'$ and $E = \frac{1}{E'_t}E'$.)
- If no such solution exists, abort.
- Return $P = \frac{Q}{E}$ where the polynomial division is over $\mathbb{F}_p$ when the remainder is 0. Abort otherwise.

4 Correctness and Security Guarantees

Immunity against Brute-Force attack Suppose an adversary initially has probability at most 2^k of guessing the input set W from the universe corpus_size. Then if the adversary has access to state generated by GenSketch(params, W), the adversary's chance of guessing W correctly improves only by a little, specifically, to $2^{k'}$ where $k' \leq k - \log\binom{(\text{corpus\_size} - \text{setsize} + t)}{t}$ where t is the errorthershold $= 2 \times (\text{setsize} - \text{correct\_threshold})$. (This follows from the guarantees in (Theorem 6.1,[1]).) If the adversary gets to see e.g. a ciphertext encrypted with sk, that may allow him to confirm whether each guess is correct. But even then the adversary will have to compute the memory-hard hash to see whether his guess is correct; an adversary who computes q memory-hard hashes will find the correct sk with probability $\leq (q+1)/2^{k'}$.

Hashing Original Words We add this extra layer of hashing over the core scheme from [1] to handle the cases of small corpus and small sets. In this case, since the entropy of the input is very low, the algorithm does not guarantee security against brute-force attack by adversary. Adding the hash and checking against it at the time of recovery provides correctness for this case.

Immunity against DoS If the state is tampered with, then the scheme does not give any recovery guarantee. For example, if an adversary can tamper with state and change state.hash, it can essentially cause

the recovery to fail for correct input from user, causing a DoS attack on the user. For immunity against this, it is absolutely crucial to maintain the `state` tamper-free and available.

References

1. Yevgeniy Dodis, Rafail Ostrovsky, Leonid Reyzin. Adam Smith: Fuzzy Extractors: How to Generate Strong Keys from Biometrics and Other Noisy Data, EUROCRYPT 2004.
   URL: http://web.cs.ucla.edu/ rafail/PUBLIC/89.pdf

2. Owen Kaser, Daniel Lemire. Strongly universal string hashing is fast, Computer Journal (2014) 57 (11): 1624-1638
   URL: https://arxiv.org/abs/1202.4961