Reminisce for Securing Private-Keys in Public

Shlomi Dolev[1], Komal Kumari[2], Sharad Mehrotra[3], Baruch Schieber[2], and Shantanu Sharma[2(✉)]

[1] Ben-Gurion University of the Negev, Be'er Sheva, Israel
[2] New Jersey Institute of Technology, Newark, USA
shantanu.sharma@njit.edu
[3] University of California, Irvine, USA

Abstract. The private key associated with a blockchain is the sole means of linking a cryptocurrency asset to its owner, and any loss or compromise of this key could result in significant consequences. Generally, crypto-wallets suggest a private key created from a string of words, which can be stored in a private record such as a piece of paper. The fact that the private record holds the secret words and can leak the private key poses security concerns. In addition, private records are susceptible to the risk of getting lost or destroyed, resulting in loss of assets. Furthermore, clients lack complete control over generating their private key, as the wallet itself creates it. Our approach enables clients to securely generate and manage their own private keys, minimizing the risk of key loss. We developed an open-source technique allowing clients to employ memorized secrets to store and retrieve their private keys. We utilize Bloom filters using hash functions, such as SHA-256, to store and retrieve the private key from the Bloom filter.

Keywords: Crypto-wallets · crypto-currency · bitcoin · blockchain · Bloom filter · public cloud · memorized secrets.

1 Introduction

The private keys of cryptocurrency systems, *e.g.*, Bitcoin [1] and Ethereum [2], are the only way to associate the ownership of a client/user to their digital money. Loss/compromise of the private key can lead to severe implications, such as loss of money [13,14,22]. To avoid the risk of losing the private key, a client uses a crypto(graphic) wallet to store their private key. A crypto wallet (see, e.g., [20] for a relevant survey), may be either a 'cold/offline wallet', *e.g.*, a piece

This work was supported by the BGU-NJIT Institute for Future Technologies (seed grant), the Israeli Science Foundation (Grant No. 465/22), the Rita Altura trust chair in computer science, and by the Lynne and William Frankel Center for Computer Science. The work of S. Mehrotra is supported by NSF grants 2420846,2245372, 2133391, 2008993, and 1952247. The work of S. Sharma is supported by NSF grant 2245374.

S. Dolev et al. (Eds.): CSCML 2024, LNCS 15349, pp. 168–177, 2025.
https://doi.org/10.1007/978-3-031-76934-4_11

of paper or a flash drive, or a 'hot/online wallet' [3], as offered by Coinbase [4], Binance [5], and MetaMask [6]. Cold wallets, being offline, are harder to access by an adversary; however, they run the risk of being lost or destroyed, disabling the client from accessing their digital money.

In contrast, existing online wallets offer secure and easy access to digital assets. These online wallets, for every client, create a private key (a deterministic sequence of 256 bits [7]) that is a function of a string of secret words, called the *secret recovery phrase*. The online wallet generates the secret recovery phrase consisting of 12, 18, or 24 words, selected from a list of 2048 words [19,21]. The secret recovery phrase serves as an alternative to memorizing the 256-bit private key and is required to be securely stored at the client. However, in this process, the client neither generates the private key nor selects the secret recovery phrase.To access the wallet, clients need to present the secret recovery phrase to authenticate themselves. The use of these online wallets is problematic in two significant ways:

- *No control to the client.* This major issue arises because the wallet generates both the private key and the secret recovery phrase. For instance, in Coinbase wallet [8], clients are presented with an automatically generated 12-word secret recovery phrase, which represents the private key used by the clients to access the wallet and perform transactions. Hence, the clients lack complete control over the generation of the private key and the secret recovery phrase. Moreover, the wallet shares only the secret recovery phrase and not the private key with the client; hence, the private key is unknown to the clients.
- *Need to remember the secret recovery phrase.* The scheme's security comes with the client's ability to remember the secret recovery phrase presented by the wallet. If clients fail to remember the secret recovery phrase, they lose complete access to their crypto assets. Studies have indicated that humans struggle to remember or store such a combination of words ffectively [24]. They often tend to store these on a private record, *e.g.*, a personal computing device, a piece of paper, or a cloud [23]. However, such private records run the risk of getting misplaced, damaged, or compromised, resulting in the loss of assets. The brain wallet approach [15] in which a user sets a phrase to be remembered to serve as a key, yields choices of phrases that can be guessed with high probability (as shown in [25], where researchers discovered 884 brain wallets having 1,806 bitcoins). Moreover, brain wallets suffer from limited human memory, resulting in loss of bitcoins [22].

This paper tries to address the above-mentioned security concerns of the private key of crypto-wallets in terms of the creation and maintenance of the private key and asks the following question:

Is it possible to develop a mechanism that empowers the clients to create their own totally random and never-revealed private keys and store them securely without the risk of being lost?

1.1 Our Contribution

We develop a technique, entitled R2R (Reminisces to Rescue), that addresses our question. ***The key advantage is that, unlike crypto wallets, our technique leverages the client to create and manage their own private key without the risk of losing it.*** R2R uses memorized (possibly very long) secrets, which are different from the private key, and Bloom filters to store/retrieve the client's private key.

Humans showcase a great ability to recall ***memorized facts/reminisces/secrets that are unique and known only to the individual***, in contrast to remembering random strings of keywords, such as those generated by existing crypto wallets, as explained above. [9,24] showed that humans hardly remember random keywords compared to their own secrets.

Examples of memorized and/or owner-retrievable (typically long) secrets could be the first stanza of your favorite song, the fourth paragraph of the third chapter of your favorite book, or a dialogue from your favorite movie or TV show. Unlike the traditional keyword-based secrets, *e.g.*, the first name of your favorite teacher, the name of your first pet, or the last four digits of SSNs, these long-memorized secrets are insusceptible to dictionary attacks. R2R uses a Bloom Filter to associate "any" true random private key to private memorized secrets.

Specifically, R2R enables clients to create their own random bits as a private key and use memorized secrets to store the private key. To do so, R2R appends the private key at the end of the memorized secrets and stores the result, bit-by-bit, in a Bloom filter. This results in a pseudorandom sequence of zeros and ones. Such random bits hold no value unless the private memorized secrets of a client are known to the adversary. A client can publish or store replicas of the Bloom filter publicly in newspaper ads, clouds/Dropbox/local files, avoiding the risk of losing this sensitive information. To extract their private keys, the client performs lookup operations over the Bloom filter. We will discuss that even if selecting memorized secrets from a relatively smaller domain, say 50, then the adversary cannot learn the private key unless performing exponential operations.

1.2 Code and Demo Video of R2R Technique

https://tinyurl.com/R2R-Code provides code and demo video of R2R.

2 R2R Technique

This section develops R2R technique and explains it using an example. Pseudocode of R2R is given in Algorithms 1 and 2. We start by describing what the client and adversary know.

Client and Adversarial View. A client generates its private key and knows security questions provided by R2R and memorized secret answers to the questions. The client executes Algorithm 1 on its private key, resulting in a Bloom

filter, which is placed in the public domain, and executes Algorithm 2 over the Bloom filter to retrieve the private key. Note that remembering these security questions by the client implies the risk of the client forgetting the relevant questions. Thus, the questions and their order used by a client are also public (as long as the answers are private). Sect. 2.4 develops a method to conceal questions also from an adversary.

An adversary knows the security questions used by the client, their order, and the Bloom filter. We call this as *adversarial view*. Based on the adversarial view, an adversary wishes to learn the private key of the client. However, as will become clear soon, the Bloom filter merely appears as a pseudorandom sequence of zeros and ones to the adversary, with no meaningful information, unless all private memorized secrets are known to the adversary.

Assumptions: R2R technique assumes that:

1. A client always remembers the memorized secrets.
2. The Bloom filter resides in a public domain, mitigating the risk of it getting lost.
3. The security questions and their order of occurrence are publicly available.
4. A private key authentication mechanism exists within the online wallet to authenticate client's private key.

Algorithm 1: Storing a private key in a Bloom filter.

Inputs: q: # security questions, $[a_i]_{1 \leq i \leq q}$: a list of memorized secret answers, $|\mathbb{B}|$: the length of a Bloom filter, $\mathbb{K}$: a private key, and $|\mathbb{K}|$: the length of private key.

Function:

1. $append(*)$: Insert each of the $\mathbb{K}$ bits of the private key one-by-one, e.g., $answer1$, $answer10$, $answer101$, where 101 is the private key.
2. InsertIntoBloom$(*)$: Execute hash function to insert elements in Bloom filter.

Outputs: $\mathbb{B}$: Bloom filter.

Generate $\mathbb{K}$ as a sequence of random bits of length $|\mathbb{K}|$
Initialize $\mathbb{B}$ of size $|\mathbb{B}|$ filled with all zeros
$answer \leftarrow$ Concatenate $a_1||a_2||\ldots||a_q$
for $j \in (1, |\mathbb{K}|)$ **do**
    $\mathbb{B} \leftarrow$ InsertIntoBloom$(answer.append(\mathbb{K}_j))$
return $\mathbb{B}$ and place it into a public domain

2.1 Storing Client's Private Key—Algorithm 1

Our idea is to use public storage and still benefit from the state-of-the-art pseudorandomness implied by the cryptographic hash function, such as the Secure Hash Algorithm (SHA) [16,18]. Algorithm 1 encodes the client's private key using memorized secrets and stores the private key in a Bloom filter, which can be published in a public domain. Let $\mathbb{K}$ be a private key of a client. Let q be

the number of security questions, and let a_i be the memorized secret answers to the i^{th} question. Let $\mathbb{B}$ be a Bloom filter using hash function $\mathbb{H}$. The client first concatenates all q memorized secret answers as: $a_1||a_2||\ldots||a_q$, resulting in *answer* (Line 3). Then, each bit of $\mathbb{K}$ is appended at the end of *answer*, one by one, and the resultant sequences are inserted into $\mathbb{B}$ using hash function $\mathbb{H}$ (similar to Bloom filter-based lookup table BFLUT [17]) (Lines 4–5). Finally, $\mathbb{B}$ is placed in the public domain.

Example of Algorithm 1. Suppose, Lisa is a client who wants to store private key 110, selects two questions, and the memorized secrets answers: (*i*) the first stanza of your favorite song, *e.g.*, "You are somebody..." from the song "You Need to Calm Down" by Taylor Swift, and (*ii*) the fourth paragraph of the third chapter of the favorite book, *e.g.*, "Thorndike tracked the behavior..." from the book "Atomic Habits". For simplicity, we are providing a few words from each memorized secret; however, in practice, these memorized secrets will comprise a complete stanza or paragraph. Lisa concatenates the two memorized secret answers as "You...Thorndike...". Then, Lisa creates a Bloom filter $\mathbb{B}$, as: $\mathbb{H}$("You...Thorndike...1"), $\mathbb{H}$("You...Thorndike...11"), $\mathbb{H}$("You...Thorndike...110") by setting one at the corresponding indices. Finally, $\mathbb{B}$ is placed in a public domain. Note that the adversary knows only the two security questions used by Lisa, but not the corresponding secret answers.

Aside. An alternative is just to use the output of hash digest, say SHA(*answer*), as the private key; however, the result may not be a valid private key for public/private key systems, which is used in the current crypto-wallets, where *answer* is the concatenated sequence of memorized secrets; see Line 2 of Algorithm 1. Another alternative to the Bloom filter-based solution could use SHA(*answer*) to generate a key for AES512, which in turn is used to encrypt and decrypt the signing private key—a private key that is coupled with a paired public key. The Bloom filter solution is more memory efficient when several private keys (say, one for each of the cryptocurrencies) have to be supported.

2.2 Retrieving Client's Private Key—Algorithm 2

The client uses Algorithm 2 to retrieve their private key by downloading $\mathbb{B}$ from the public domain for performing lookup operations over $\mathbb{B}$. Similar to Algorithm 1, the client first concatenates the q memorized secret answers as: $a_1||a_2||\ldots||a_q$, resulting in *answer* (Line 2). To the *answer*, the client appends bit zero and then bit one and performs a lookup in $\mathbb{B}$ for the appended sequence (Lines 7–8)—note that bits zero and one can be appended and checked in any order. For each successful lookup ($\mathbb{B}$ outputs as one), the client further appends bit zero, and then bit one and performs a lookup for the updated sequence (Line 9). In case of an unsuccessful lookup ($\mathbb{B}$ outputs zero), the process is terminated for the corresponding sequence. The process continues until the number of appended bits equals $|\mathbb{K}|$, to produce $\mathbb{K}$ (Lines 5–6).

Algorithm 2: Retrieving a private key from a Bloom filter.

Inputs: $\mathbb{B}$: Bloom filter, q: # security questions, $[a_i]_{1 \leq i \leq q}$: a list of memorized secret answers, $|\mathbb{K}|$: the length of private key.
Function:
append$(*)$: Insert each of the $\mathbb{K}$ bits of the private key one-by-one, e.g., *answer*1, *answer*10, *answer*101, where 101 is the private key.
BloomLookup$(*)$: Execute hash to check the presence of elements in Bloom filter.
Outputs: $\mathbb{K}$: Private key/s.

```
1 Download 𝔹 from the public domain
2 answer ← Concatenate a_1||a_2||...||a_q
3 𝕂 ← [], bits ← ϕ
4 Function RetrievePrivateKey(answer, bits) begin
5     if |bits| == |𝕂| then
6         𝕂.append(bits) and return 𝕂
7     for j∈[0,1] do
8         if BloomLookup(answer.append(j)) then
9             RetrievePrivateKey(answer.append(j), bits.append(j))
```

Example of Algorithm 2. Consider, Lisa wants to retrieve the private key 110 from $\mathbb{B}$, using the same memorized secret answers, mentioned in the example of Algorithm 1. Lisa performs lookups: $\mathbb{H}$("You...Thorndike...0"), $\mathbb{H}$("You...Thorndike...1"). Suppose, $\mathbb{H}$("You...Thorndike...1") results in one. Then, Lisa appends zero and one to perform lookup for $\mathbb{H}$("You... Thorndike...10"), $\mathbb{H}$("You...Thorndike...11"). Note, since the lookup of $\mathbb{H}$("You...Thorndike...0") outputs zero, Lisa discontinues the append/lookup process for these sequences. Suppose, $\mathbb{H}$("You...Thorndike...11") results in one. The append/lookup continues until finally Lisa gets the output of $\mathbb{H}$("You...Thorndike...110") as one. Thus, Lisa retrieves $\mathbb{K}$ as 110.

2.3 Making False Positives to Zero

Bloom filter lookup comes with false positives with some probability. To avoid false positives, we can append a long sequence of ones or concatenate the memorized secret answers at the end of the private key. In particular, apart from performing insertions depicted in Algorithm 1, we append either a sequence of ones or *answer* at the end of *answer*$||\mathbb{K}$ that results in either *answer*$||\mathbb{K}||111\ldots$ or *answer*$||\mathbb{K}||$*answer*. The sequence of ones is inserted bit-by-bit into $\mathbb{B}$, while *answer* is inserted word-by-word. Note that this insertion will increase the size of $\mathbb{B}$. During retrieval, once we extract all the candidate private keys, say *candidate*, using Algorithm 2. We check all *candidate* appended with ones or *answer* bit by bit, and discard the *candidate* whose lookup operation results zero. The process continues until the client is left with a single *candidate* that is the private key of the client.

2.4 Hiding Questions

The client might consider not disclosing the questions and obfuscating these questions using reminisces in the questions too; e.g., using nicknames or polysemous words. For instance, a question "What is best in Israel" could have multiple answers such as a city (Haifa), food (Shakshuka), beach (Beit Yanai), actress (Gal Gadot), or TV series (Fauda). The client remembers only one thing that they really like. Suppose, for the client, the best in Israel is Haifa. Based on the keyword "Haifa," the client selects the publicly-known questions $\langle 8, 1, 9, 6, 1 \rangle$ provided by the R2R technique. This method enhances the concealing of the public questions used by clients, hence yielding a practically impossible search for the right answers.

2.5 Security Analysis

The adversarial view constitutes the security questions and the order used by the client; however, it does not include the memorized secret answers. Thus, the adversary will have to try all possible combinations for stanzas of all the songs and the fourth paragraph from the third chapter of every book. Note that using only one question, such as the first stanza of your favorite song, may make the technique less secure, since an adversary can focus and invest efforts to find this information and learn the private key. In contrast, using more than one question, enhances the security of the technique, as the adversary needs to learn all the correct answers to learn the private key. In particular, there are over 100M songs on Spotify [10] and over 5M English novels [11]. The adversary needs to try all possible $100\text{M} \times 5\text{M} \approx 2^{25}$ combinations to retrieve client's private key. As the client uses q security questions, such that each question has at least a domain of size of 1M, the complexity to learn the memorized secret answers will be at least $(1\text{M})^q$ or 2^{20q}.

Selecting memorized secrets from a large domain is *not* a restriction of R2R. A client can also select memorized secrets from a smaller-sized domain, say 50. In this case, the client needs to select multiple questions. Recall that since questions and their order are available in public, it does not pose a risk of forgetting them. For example, for memorized secrets, each with a domain of size 50, a client may select 20 questions. Here, the adversary needs to try 2^{215} combinations, which is computationally infeasible, to learn the memorized secret and then the key.

In general, the adversary, to learn the private key, has to exhaust all possibilities to produce the correct memorized secret (i.e., *answer*, see Line 2 of Algorithm 1) in a reasonable time.

2.6 Desiderata

We may avoid copying the Bloom filter to local memory by hiding the probe sequence (*i.e.*, the Bloom filter positions accessed by the lookup operation). We can (*i*) randomly select the next bit extension to lookup, deciding between querying the zero extension or the one extension, (*ii*) interleave queries with fake

queries, (*iii*) split the Bloom filter into several parts, and query the server that holds the part we need to query according to the addresses it maintains.

3 Experimental Evaluation

We conducted experiments on a Mac machine having 10 cores and 64GB RAM and implemented R2R in (≈500 lines of) Python. We leveraged Python's *hashlib* library (supporting SHA-256, SHA-512) to create Bloom filter using SHA-256. The Bloom filter, $\mathbb{B}$, parameters: the number of hash functions h and size of $\mathbb{B}$ (denoted by $|\mathbb{B}|$), are computed based on the formula given in [17], as follows: $|\mathbb{B}| = -(n\ln f)/(\ln 2)^2$ and $h = (|\mathbb{B}|/n)\ln 2$, for false positive rate f (set as 10^{-65}), where n is the number of values to be inserted into $\mathbb{B}$. For experiments, we used five questions, but R2R client can select at most twelve questions. Streamlit framework [12] is used to design our client interface.

Exp 1: Time Taken to Store/Retrieve $\mathbb{K}$ from $\mathbb{B}$. We present the time taken to store/retrieve a 256-bit key $\mathbb{K}$ to/from $\mathbb{B}$, using five memorized secret answers, denoted by $a_{i\in\{1,5\}}$. In this experiment, n is calculated as $|\mathbb{K}| + \sum_{i=1}^{q} WordCount(a_i)$, where $|\mathbb{K}|$ is the length of the key $\mathbb{K}$ and function *WordCount* calculates number of words in a_i. Based on this, we obtain n=256+438=694. Note that $\sum_{i=1}^{q} WordCount(a_i)$ denotes the number of words added at the end of $answer||\mathbb{K}$ to avoid false positives (we called them as ***pad*** in this and next experiments), as discussed above in Sect. 2.

For $n = 694$, R2R results in $\mathbb{B}$ of size ≈26.3KB and $h = 215$ to store $\mathbb{K}$. The time taken to store $\mathbb{K}$ in $\mathbb{B}$ is ≈271.3 milliseconds (ms) and the time taken to retrieve $\mathbb{K}$ from $\mathbb{B}$ is ≈139.6 ms. The time to store the $\mathbb{K}$ is higher compared to the retrieval of $\mathbb{K}$ from $\mathbb{B}$, since we are also storing $answer||\mathbb{K}$ appended with *pad* of size 438 to avoid false positives, while we terminate our retrieval algorithm when we get only one key without checking 438 positions in $\mathbb{B}$.

Exp 2: Variation in the Size of $\mathbb{B}$ as a Function of the Padding. We perform this experiment to find the size of $\mathbb{B}$ as a function of the padding. We increase padding from 100 to 438 and compute $|\mathbb{B}|$, the time to create $\mathbb{B}$, and the time to fetch the key; see table below:

Padding size	100	200	300	438
Size of $\mathbb{B}$	13.6 KB	17.3 KB	21.1 KB	26.3 KB
Time to create $\mathbb{B}$	141.5 ms	183 ms	214.8 ms	271.3 ms
Time to retrieve key from $\mathbb{B}$	138.6 ms	138 ms	139.7 ms	139.6 ms

4 Conclusion

We presented R2R technique that empowers the clients to create their own totally random and never-revealed private keys and store them securely without the risk of being lost. Allow the key generation by the client avoids using trusted entities in producing a private key, which hinder the main purpose and promise of a private key. R2R stores the client's private key into a Bloom filter with the help of memorized (possibly very long) secrets, which are different from the private key, and keeps the Bloom filter in public.

R2R offers security against alphabetically exhaustive search. Using long answers to publicly available question prevents alphabetically exhaustive search, preventing an adversary to learn the key. Further, clients do not need to remember the questions used, as they will become public, as well as, long answers (e.g., a book chapter). The only need is to remember which chapter and then the client can find the chapter. Making questions and Bloom filter public avoids the possibility of losing the private storage in keeping the key secure and avoid the use of a password manager, which could be lost or corrupted.

References

1. Bitcoin. https://bitcoin.org/en/
2. Ethereum. https://ethereum.org/
3. Hot Wallet vs. Cold Wallet. https://www.investopedia.com/hot-wallet-vs-cold-wallet-7098461
4. Coinbase. https://www.coinbase.com/
5. Binance. https://www.binance.com/en
6. Metamask. https://metamask.io/
7. How Do Crypto Wallets Work. https://medium.com/@hamilton_21385/how-do-crypto-wallets-work-3ca749464f87
8. Coinbase: Create a Coinbase Wallet. https://help.coinbase.com/en/wallet/getting-started/create-a-coinbase-wallet
9. How Chunking Pieces of Information Can Improve Memory. https://tinyurl.com/4aewnnb2
10. How Many Songs are There in the World? https://www.musicianwave.com/how-many-songs-are-there-in-the-world/
11. How many novels have been published in English? http://tinyurl.com/37dkf8k4
12. Streamlit. https://streamlit.io/
13. Compromised Private Keys: Primary Targets and Upcoming Solutions. https://tinyurl.com/yvxkpk95
14. $35 million stolen in attacks on Atomic Wallet cryptocurrency customers. https://therecord.media/millions-stolen-in-atomic-wallet-attack
15. Brainwallet. https://en.bitcoin.it/wiki/Brainwallet
16. Dang, Q.: Secure hash standard (2015). 04 Aug 2015
17. Dolev, S., Gudes, E., Segev, E., Ullman, J.D., Weintraub, G.: BFLUT bloom filter for private look up tables. In: Dolev, S., Katz, J., Meisels, A. (eds.) CSCML 2022. LNCS, vol. 13301, pp. 499–505. Springer, Cham (2022). https://doi.org/10.1007/978-3-031-07689-3_35

18. Dworkin, M.J.: Sha-3 standard: permutation-based hash and extendable-output functions, 2015. National Institute of Standards and Technology, NIST FIPS 202. https://doi.org/10.6028/NIST.FIPS.202
19. Everything you need to know about your 12-word secret recovery phrase. https://tinyurl.com/4v682pu
20. Houy, S., Schmid, P., Bartel, A.: Security aspects of cryptocurrency wallets - A systematicliterature review. ACM Comput. Surv. **56**(1), 4:1–4:31 (2024)
21. What is a 12 word seed phrase?. https://tinyurl.com/3btynyx9
22. This man owns $321M in bitcoin — but he can't access it because he lost his password. https://tinyurl.com/hzp8stav
23. What's in the Cloud? https://tinyurl.com/yux3y3yd
24. Ur, B., et al.: "i added '!' at the end to make it secure": observing password creation in the lab. In: Cranor, L.F., Biddle, R., Consolvo, S., (eds.), Eleventh Symposium On Usable Privacy and Security, SOUPS 2015, Ottawa, Canada, July 22-24, 2015, pp. 123–140. USENIX Association (2015)
25. Vasek, M., Bonneau, J., Castellucci, R., Keith, C., Moore, T.: The bitcoin brain drain: examining the use and abuse of bitcoin brain wallets. In: Grossklags, J., Preneel, B. (eds.) FC 2016. LNCS, vol. 9603, pp. 609–618. Springer, Heidelberg (2017). https://doi.org/10.1007/978-3-662-54970-4_36