

# An Air-Gapped 2-Factor Authentication for Smart-Contract Wallets

Ivan Homoliak<sup>\*‡</sup>  
ivan\_homoliak@sutd.edu.sg

Dominik Breitenbacher<sup>†</sup>  
dominik@sutd.edu.sg

Alexander Binder<sup>†</sup>  
alexander\_binder@sutd.edu.sg

Pawel Szalachowski<sup>‡</sup>  
pawel@sutd.edu.sg

<sup>\*</sup>STE-SUTD Cyber Security Laboratory

<sup>†</sup>Singapore University of  
Technology and Design

<sup>‡</sup>Faculty of Information Technology,  
Brno University of Technology

**Abstract**—With the recent rise of cryptocurrencies, the security and management of crypto-tokens have become critical. We have witnessed many attacks on users, their software, or their providers, which have resulted in significant financial losses. To remedy these issues, many wallet solutions have been proposed to store users’ crypto-tokens. However, these solutions lack either essential security features, or usability, or do not allow users to express their spending rules.

In this paper, we propose a smart-contract cryptocurrency wallet framework that gives a flexible, usable, and secure way of managing crypto-tokens in a self-sovereign fashion. The proposed framework consists of three components (i.e., a decentralized client, a smart contract, and an authenticator) and provides 2-factor authentication performed in two stages of interaction with the blockchain. Our framework utilizes one-time passwords (OTPs) aggregated by a Merkle tree and hash chains in such a way that for every authentication only a 16B-long OTP is transferred from the authenticator to the client. Such a novel setting enables us to make a fully air-gapped authenticator, similar to Google Authenticator. We implemented our approach basing on the Ethereum platform. We have performed a cost analysis of the implementation and showed that the average cost of a transfer operation is comparable to existing 2-factor authentication solutions using smart contracts with multi-signatures.

## I. INTRODUCTION

Cryptocurrencies are successful beyond all expectations. Their amazing rise in the last decade has resulted in various open and decentralized platforms that allow users to conduct monetary transfers, write smart contracts used as financial agreements, or participate in predictive markets. If successful, cryptocurrencies promise to revolutionize many fields and businesses. Cryptocurrencies introduce their own crypto-tokens, which can be transferred in transactions authenticated by private keys that belong to crypto-token owners. These private keys are managed by wallet software that gives users an interface to interact with the cryptocurrency. There are many cases of stolen keys that were secured by various means [1], [2], [3]. Such cases have brought the attention of the research community to the security issues related to key management in cryptocurrencies [4], [5], [6]. According to the previous work [4], [6], there are a few categories of key management approaches in Bitcoin, also applicable to other cryptocurrencies.

Password-protected wallets encrypt private keys with user-selected passwords. Unfortunately, users often choose weak passwords that can be brute-forced if stolen by cryptocurrency-stealing malware [7]; optionally, such malware may use a

keylogger for capturing a passphrase [6], [8]. Another similar option is to use password-derived wallets that generate keys based on a provided password. However, they also suffer from the possibility of weak passwords [2]. Hardware wallets is a category that promises the provision of better security by introducing devices that enable only the signing of transactions, without revealing the private keys stored on a device. However, these wallets do not provide protection from an attacker with full access to a device [4], and more importantly, as presented by recent research [9], some of them may be exploited by sophisticated malware targeting inter-process communication mechanisms. Multi-factor (or multi-step) authentication is provided by wallets from the split control category, which enables spending crypto-tokens only when  $n$  of  $m$  secrets are used together. This can be achieved by threshold cryptography wallets [5], [10], multi-signature wallets [11], [12], [13], [14], and state-aware smart-contract wallets [15], [16], [17]. This last class of wallets is interesting, as spending rules are encoded in a smart contract; hence, it can be adjusted almost arbitrarily to the user’s needs.

A popular option for storing private keys is to deposit (or even generate) them at server-side hosted wallets and currency-exchange services [18], [19], [20]. In contrast to the previous categories, server-side hosted wallets imply trust in a third party, which is a potential risk of such a solution. Due to infamous cases of compromising server-side wallets [21], [22], [23], [24] or fraudulent currency-exchange operators [25], client-side hosted wallets have started to proliferate. In such wallets, the main functionality, including the storage of private keys, has moved to the user’s browser or a local application [26], [27], [28], [29], [30]. Alternatively, this category contains wallets that store only encrypted versions of keys by a third party [31], [32]. Hence, trust in the third party has been partially reduced, but the users still depend on the third party’s infrastructure.

**Proposed Approach.** In this paper, we propose a framework for smart-contract cryptocurrency wallets. Unlike traditional one-size-fits-all solutions, smart contract wallets allow users to customize their wallets to a high extent, similarly to the way they customize their bank accounts or credit cards.

For increased security of smart contract wallets, our framework utilizes 2-factor authentication of a user against data stored at the blockchain of a cryptocurrency, and it is accomplished by: 1) a decentralized client, 2) a smart-contract

wallet, and 3) an air-gapped authenticator. Our solution utilizes hash-based cryptographic constructs, namely a pseudo-random function, a Merkle tree, and hash chains. We propose a novel combination of these elements that minimizes the amount of data transferred from the authenticator device, which enables us to implement the authenticator in a fully air-gapped setting, not requiring a USB connection. Our proposed solution belongs to the category of state-aware smart contract wallets (see Section VIII). Although there exist several smart-contract wallets that are using multi-factor authentication against the blockchain [15], [17], to the best of our knowledge, none of them provide an air-gapped authentication in the form of short OTPs similar to Google Authenticator, which is the focus of our work.

**Contributions.** Our main contributions are as follows:

- We show that standard 2-factor authentication methods against the blockchain do not meet either the security or usability requirements for an air-gapped setting.
- We propose a smart-contract wallet framework protected by 2-factor authentication against the blockchain. Our approach utilizes OTPs that are managed in a novel way, enabling us to make an authenticator device fully air-gapped. We achieve this by delegating a Merkle proof construction at the client that stores non-confidential data of the Merkle tree.
- To increase the number of OTPs, we resolve the time-space trade-off at the client by combining hash chains with Merkle trees in a novel setting.
- We implement and evaluate our approach, and we provide the source code of our solution. The provided smart contract is self-contained, but its operation set can be extended easily by the community.
- We introduce and formalize the notion of k-factor authentication against the blockchain and k-factor authentication against the authentication factors themselves. Using these notions, we propose a classification of authentication schemes, and we apply it to related work.

## II. BACKGROUND AND PRELIMINARIES

We assume a generic cryptocurrency of which the blocks of records are stored in an ever-growing public distributed ledger called a *blockchain*, which is resistant by design against modifications. In a blockchain, blocks are linked using a cryptographic hash function, and each new block has to be agreed upon by participants running a consensus protocol – these participants are called *miners*. Each block may contain orders transferring crypto-tokens, application codes written in a platform-supported language, and the execution orders of such applications. These application codes are referred to as *smart contracts* and can encode arbitrary processing logic (e.g., agreements) written in a supported language. Interactions between clients and the cryptocurrency system are based on messages called *transactions*, which can contain either orders transferring crypto-tokens or calls of smart contract functions. All transactions sent to a blockchain are validated by miners who maintain a replicated state of the blockchain.

**Merkle Tree.** A Merkle tree is a data structure based on the binary tree in which each leaf node contains a hash of a single data block, while each non-leaf node contains a hash of its concatenated children. A Merkle tree enables efficient verification as to whether some data are associated with a leaf node by comparing the expected root hash of a tree with the one computed from a hash of the data in the query and the remaining nodes required to reconstruct the root hash (i.e., *proof* or *authentication path*). The reconstruction of the root hash has the logarithmic time complexity, which makes the Merkle tree an efficient scheme for membership verification.

**User Authentication.** Authentication is a method of verifying the claimed identity of a user who requests access to a resource or a service. If an authentication method requires the user to present a combination of two or more distinct factors proving knowledge, possession, or inherent biometric features, we refer to multi-factor authentication (MFA). In practice, 2-factor authentication (2FA) is commonly used; this is a variant of MFA that requires proving two of mentioned factors – usually knowledge and possession. In order to break particular factors of MFA, the attacker must extract a knowledge factor (e.g., password), physically steal a device possessed by the user, and forge the user’s biometric features, respectively. Multi-step authentication (MSA) is an authentication method that requires the user to present a combination of two or more instances of a single factor – proving either knowledge, or possession, or inherent biometric features. The most common variant of MSA is 2-step authentication (2SA), which requires proving two instances of a knowledge factor (e.g., password, software-based OTP). In contrast to MFA, breaking MSA requires the attacker to execute only a single type of malicious action, but multiple times in more contexts – for example, extracting a password (e.g., by a keylogger) and extracting a private seed of an application generating OTPs (e.g., by malware). Since terms MFA and MSA differ only in the realization of a particular step’s/factor’s data source (e.g., sealed hardware VS a software in the case of 2FA/2SA), we will further use only term MFA (or 2FA) for simplification of the terminology.

### A. Notation

By the term *operation* we refer to an action with a smart-contract wallet, which may involve, for instance, a transfer of crypto-tokens or a change of daily spending limits. Then, we use the term *transfer* for the indication of transferring crypto-tokens. By  $\{msg\}_U$  we denote the message *msg* digitally signed by *U*;  $\mathcal{RO}$  is the random oracle;  $h(\cdot)$  stands for a cryptographic hash function;  $h^i(\cdot)$  substitutes *i*-times chained function  $h(\cdot)$ , e.g.,  $h^2(\cdot) \equiv h(h(\cdot))$ ;  $\parallel$  is the string concatenation;  $F_k(\cdot) \equiv h(k \parallel \cdot)$  denotes a pseudo-random function that is parametrized by a secret seed *k*;  $\%$  represents modulo operation over integers; and  $SK_U$  is the private key of *U*, under the signature scheme of the public blockchain.

## III. PROBLEM DEFINITION AND DESIGN SPACE

The main goal of this work is to propose a cryptocurrency wallet framework that gives a secure and usable way of

managing crypto-tokens. In particular, we aim to achieve:

**Self-Sovereignty:** users can customize spending rules that are enforced by the cryptocurrency system deployed. In cryptocurrencies, self-sovereignty is a critical property and the main argument for self-sovereign wallets is the natural difference in user needs. Cryptocurrency platforms offer native pre-defined security features that are available for all users. Unfortunately, these solutions lack expressiveness, as they have to fit standard users (who, e.g., store crypto-tokens as a hobby) as well as large cryptocurrency exchange providers, which, due to the value stored, require much better protection.

**Security:** as we have seen in the past, a poor security level of cryptocurrency wallets has caused significant financial losses for individuals and companies [33], [1], [2], [3]. We argue that wallets should be designed with security in mind, and, in particular, we point out 2FA solutions, which have successfully contributed to the security of other environments [34], [35]. In the context of cryptocurrencies, we emphasize that implementing 2FA against the blockchain may be challenging, as all blockchain data are public. We see significant security benefits of an air-gapped setting, as it prevents from multiple attack vectors related to connected devices [9].

#### A. Threat Model

For a generic cryptocurrency described in Section II, we assume an attacker whose goal is to conduct unauthorized operations (e.g., transfers) on the user’s behalf or render a wallet unusable. The attacker is able to eavesdrop on the network traffic as well as to participate in the underlying consensus protocol. However, the attacker is unable to take over the cryptocurrency platform or to break the used cryptographic primitives. We assume that the attacker has obtained the user’s private key of a cryptocurrency account. Then we assume that the attacker is able to intercept and “override” the user’s transactions, e.g., by launching a man-in-the-middle (MITM) attack or by creating a conflicting malicious transaction with a higher fee, which will incentivize miners to discard the user’s transaction. Next, we assume that the legitimate user correctly executes the proposed protocols and  $h(\cdot)$  is an instantiation of  $\mathcal{RO}$ .

#### B. Design Space

As we present in Section VIII, there are many types of wallets with different properties. In our context, to achieve self-sovereignty we identify smart-contract wallets as a promising category. These wallets manage crypto-tokens by the functionality of smart contracts, enabling users to have customized control over their wallets. The advantages of these solutions are that spending rules can be defined by users and then enforced by the cryptocurrency platform itself – it is impossible to bypass them, as they are verified by miners. Therefore, using this approach, it is possible to build a flexible wallet with features such as daily spending limits or transfer limits.

With spending rules encoded within a smart contract, it is feasible to design and implement custom security features into a smart contract, such as an air-gapped 2FA, which is the goal of our work. In such a setting, an air-gapped authenticator produces one-time passwords (OTPs) to authenticate transactions at the smart contract.

We define the necessary security requirements for 2FA realized by OTPs using the smart contract as follows:

- 1) **Authenticity:** each OTP must be associated only with a single authenticator instance, and thus must allow the verification of the authenticity of an OTP.
- 2) **Linkage:** each OTP  $otp_i$  must be linked with exactly a single operation  $O_i$ , ensuring that  $otp_i$  cannot be used for the authentication of  $O_j, i \neq j$ .
- 3) **Independence:** the OTP of the operation  $O_i$  cannot be derived from another OTP  $otp_j$  of an operation  $O_j, i \neq j$  or an arbitrary set of such OTPs.

We argue that not all solutions are feasible for such a purpose. Asymmetric cryptography primitives such as digital signatures or zero-knowledge proofs are inadequate in this setting, despite the fact that they meet all OTP requirements. State-of-the-art signature schemes [36], [37], [38] with a short signature size produce a 32B-64B long output. However, transferring even 32B in a fully air-gapped environment by transcribing mnemonic words [39] would lack usability for regular users – transcription of 24 mnemonic words [39] takes 28s on average (considering data from the study [40]). Another drawback of asymmetric cryptography relates to its resource demands that increase the operational costs, both on authenticators and on smart contracts (smart contract platforms put a high execution cost for asymmetric cryptography). Finally, most of the currently deployed asymmetric constructions are vulnerable to quantum computing [41].

The problem of long signatures exists also in hash-based signature constructs [42], [43], [44]. Lamport-Diffie one-time signatures (LD-OTS) [42] produce an output of length  $2|h(\cdot)|^2$ , which, for example in the case of  $|h(\cdot)| = 16B$  yields  $4kB$ -long signatures. The signature size of LD-OTS can be reduced by using one string of one-time key for simultaneous signing of several bits in the message digest (i.e., Winternitz one-time signatures (W-OTS) [43]) but at the expense of exponentially increased number of hash computations (in the number of encoded bits) during a signature generation and verification. The extreme case minimizing the size of W-OTS to  $|h(\cdot)|$  (for simplicity omitting checksum) would require  $2^{|h(\cdot)|}$  hash computations for signature generation, which is unfeasible.

Approaches based on symmetric cryptography primitives produce much shorter outputs, but it is challenging to implement them with smart-contract wallets. Widely used one-time passwords like HOTP [45] or TOTP [46] require the user to share a secret key with the authentication server. Then, with each authentication request the user proves that he has the key by returning the output of a pseudo-random function computed with a nonce (i.e., HOTP) or the current timestamp (i.e., TOTP). This approach is insecure in our setting, as the user would have to share his secret key with a smart-contract

wallet, making the key publicly visible.

One solution that does not need to publicly disclose secret information, and, at the same time, provides short enough OTPs, can be implemented by Lamport's hash chains [47]. A hash chain enables the production of many OTPs by the successive execution of a hash function, starting from  $k$  that represents a secret key of the authenticator. Upon the initialization, a smart contract is preloaded with the last generated value  $h^n(k)$ . When the user wants to authenticate the  $i$ th operation, he sends the  $h^{n-i}(k)$  to the smart contract. The smart contract computes the hash function consecutively  $i$  times and checks to ascertain whether the obtained value equals the stored value. However, the main disadvantage of this solution is that each OTP can be trivially derived from any previous one. Therefore, selective authentication is impossible, as revealing the  $i$ th OTP allows the attacker to derive any  $j$ th OTP, where  $j < i$ . In short, this scheme does not meet the security requirement on the independence of OTPs.

#### IV. PROPOSED APPROACH

For a generic cryptocurrency described in Section II, we propose a 2FA against the blockchain (see Section VIII-A), which consists of three components: 1) a decentralized client, 2) a smart-contract wallet, and 3) an air-gapped authenticator (see Figure 1). We first explain the key idea of our approach, which enables us to realize the authenticator as a fully air-gapped device. Then, we present the base protocol of our approach, and finally, we incrementally show modifications that improve the efficiency and usability of our approach.

##### A. Design of an Air-Gapped Authenticator

We propose an approach where OTPs are generated by a pseudo-random function  $F_k(\cdot)$  and then aggregated by a Merkle tree, providing a single value – the root hash. In contrast to digital signatures (such as RSA, DSA, or Merkle signatures), the root hash serves only for proving the non-repudiation of a particular OTP, thus an arbitrary message cannot be protected. To overcome this limitation, we propose a two-stage approach, where an operation is first submitted to the blockchain and then, in the second stage, it is executed upon authentication by the OTP that corresponds to the unique identifier of this operation.

Another challenge is the OTP size. Using the naïve version of our scheme, a 2FA requires the authenticator to provide an OTP and its proof. However, in such a naïve version, the user has to transfer  $O + S \times H$  bytes from the authenticator each time he confirms an operation, where  $O$  represents the size of an OTP,  $S$  represents the output size of a hash function, and  $H$  represents the height of a Merkle tree with  $N$  leaves; hence  $H = \log_2(N)$ . For example, if  $O = S = 32B$  and  $H = 10$ , then the user would have to transfer 352B each time he confirms an operation, which has a very low usability in an air-gapped setting. Even further reduction of  $O$  and  $S$  to 16B would not help to resolve this usability issue, as the amount of user transferred data would be equal to 176B.

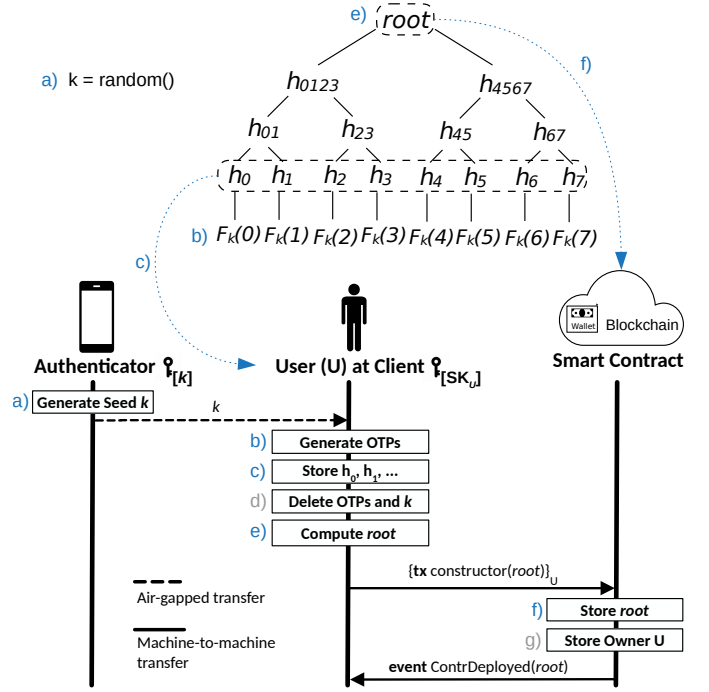


Fig. 1: Bootstrapping of the framework (in secure environment).

We make the observation that it is possible to decouple providing OTPs from providing their authentication paths (i.e., proofs). In this setting, the only data that need to be kept secret are OTPs, while any node of a Merkle tree may potentially be disclosed – no OTP can be derived from these nodes. Therefore, we propose providing OTPs by the authenticator, while their proofs can be constructed at the client from stored hashes of OTPs. This improvement enables us to fetch the nodes of the proof from the client's storage, while the user has to transfer only the OTP itself from the authenticator (i.e.,  $O = 16B$  by default) when confirming an operation.

##### B. Base Version

**Bootstrapping of the Framework.** As common in other schemes and protocols, our scheme assumes a one-time bootstrapping in a secure environment (depicted in Figure 1). First, a secret seed  $k$  is generated at the authenticator. Next, the user transfers  $k$  from the authenticator to the client in an air-gapped manner (i.e., rewriting a few mnemonic words). Then, the client generates OTPs, computing  $F_k(i) \mid i \in \{0, 1, \dots, N-1\}$ , where  $N$  is the number of leaves (i.e., equal to number of OTPs in the base version). Next, these OTPs are used for computing the root hash. After this step,  $k$  and the OTPs are deleted from the client, and the client stores only the leaves of the tree – i.e., the hashes of the OTPs, which does not contain any confidential data. Finally, the client deploys a smart contract (see Algorithm 1) on the blockchain, and the root hash is passed as an argument to the constructor of the smart contract.<sup>1</sup> In the constructor, the root hash is stored and the sender of the message is saved as the owner of the

<sup>1</sup>We emphasize that the client has a provided template of a smart contract and the deployment process is unnoticeable for the users.

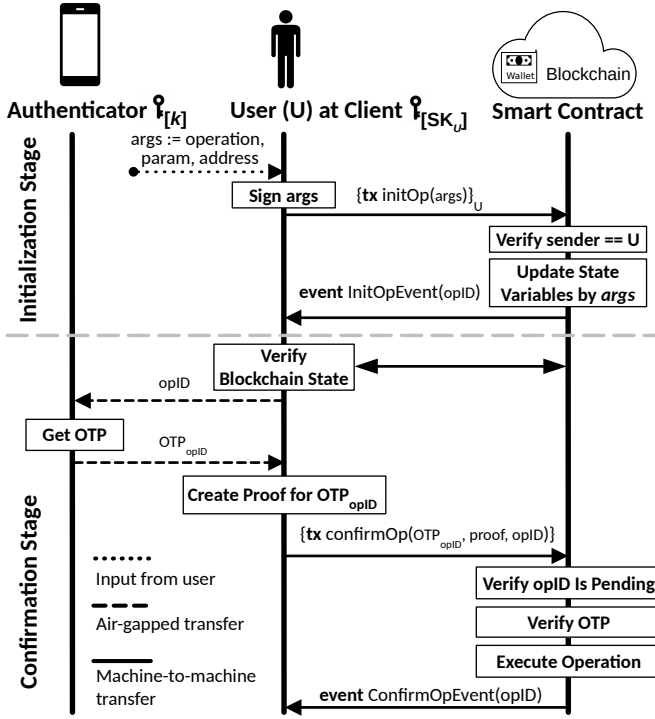


Fig. 2: Execution of an operation.

smart contract. The address of the owner enables the smart contract to verify whether a transaction was signed by the user who created the contract, while the root hash enables verifying whether an OTP was produced by the user's authenticator.

**Bootstrapping in an Insecure Environment.** The main advantage of the secure bootstrapping described above is its high usability. However, we propose an alternative bootstrapping of our framework in an insecure environment – assuming that secret seed  $k$  can be obtained by an adversary, who, additionally to our threat model, can eavesdrop  $k$  at the device where the client is running. To accomplish a bootstrapping in an insecure environment, instead of transferring secret seed  $k$ , the user performs a one-time transfer of all leaves of the Merkle tree from the authenticator to the client, which can be realized with a microSD card.

**Operation Execution.** When the wallet framework is initialized, it is ready for executing operations. To provide 2-factor authentication capability, we split the protocol of an operation execution into two stages, as depicted in Figure 2:

- 1) **Initialization Stage.** When the user decides to execute an operation with his smart-contract wallet, he enters the details of the operation into the client that creates a transaction calling  $initOp()$ , which is provided with operation-specific parameters – the type of operation (e.g., transfer), a numerical parameter (e.g., amount or daily limit), and an address parameter (e.g., recipient). Then, the client signs this transaction with the user's private key  $SK_U$  and sends it to the blockchain. In the function  $initOp()$ , the smart contract verifies whether the signature was created by the owner of the wallet (the

#### Algorithm 1: Smart contract wallet with 2FA

▷ VARIABLES AND FUNCTIONS OF ENVIRONMENT:  
*sender*: an address of the user who signed the current message,  
*balance*: the current balance of a contract,  
*transfer*( $r, v$ ): transfer  $v$  crypto-tokens from a smart contract to  $r$ ,  
*revert*( $m$ ): revert an effect of transaction, yielding message  $m$ ,

▷ DECLARATION OF TYPES:  
**Operation** { *addr*, *param*, *pending*, *type*  $\in \{TRANSFER, \dots\}$  }

▷ DECLARATION OF FUNCTIONS:  
**function** *constructor*( $r$ ) **public**  
      $operations \leftarrow []$ ;  
      $owner \leftarrow sender$ ;  
      $rootHash \leftarrow r$ ;  
      $nextOpID \leftarrow 0$ ; ▷ Top of operations

**function** *initOp*( $a, p, type$ ) **public**  
     **assert**  $sender = owner$ ; ▷ First factor of 2FA  
      $operations[nextOpID++] \leftarrow new\ Operation(a, p, true, type)$ ;

**function** *confirmOp*( $otp, \pi, opID$ ) **public**  
     **assert**  $true = operations[opID].pending$ ;  
      $verifyOTP(otp, \pi, opID)$ ; ▷ Second factor of 2FA  
      $execOp(operations[opID])$ ;  
      $operations[opID].pending \leftarrow false$ ;

**function** *verifyOTP*( $otp, \pi, opID$ ) **private**  
     **assert**  $deriveRootHash(otp, \pi, opID) = rootHash$ ;

**function** *execOp*( $oper$ ) **private**  
     **if**  $TRANSFER = oper.type$  **then**  
         **assert**  $oper.param \leq balance$ ;  
          $transfer(oper.addr, oper.param)$ ;  
     **else**  
          $revert("Unknown\ type\ of\ requested\ operation.")$ ;

first step of authentication), stores the parameters of the operation, and then assigns a sequential ID (i.e.,  $opID$ ) to the initiated operation. In the response from the smart contract, the client is provided with an  $opID$ .

- 2) **Confirmation Stage.** After the initiated transaction is persisted on the blockchain, the user proceeds to the second stage of the protocol. Additionally to our threat model, we propose a protection of our scheme against the attacker that tampers with the client (i.e., displaying different details of the operation from the blockchain's data), it is important to verify the details of the operation by other means than the client. For example, the user may use the HTTPS blockchain explorer, where the transaction details are displayed. Since the client is a decentralized application, another option is to use a watch-only instance of the client on a smartphone or any other device. After verification of details, the user passes the  $opID$  to the authenticator, which, in turn, computes and displays the OTP as  $F_k(opID)$ . Storing hashes computed from OTPs at the client enables the user to transfer only the displayed OTP from the authenticator to the client, which can be accomplished in an air-gapped manner. Considering a mnemonic implementation [39], this means an air-gapped transfer of 12 words in the case of  $O = 16B$ . Then, the client computes and appends the corresponding proof to the OTP. The proof of the OTP is computed from stored hashes of all OTPs in the client's storage. Next, the client sends a transaction with the OTP and its proof to the blockchain, calling the function  $confirmOp()$  of the smart contract. This function verifies the authenticity of the

OTP (i.e., the first requirement on OTPs) and whether it is associated with the requested operation (i.e., the second requirement on OTPs), which together implies the correctness of the OTP. In detail, upon calling the *confirmOp()* function with an operation ID and OTP with its proof as the arguments, the smart contract computes the root hash from the provided arguments by the function *deriveRootHash()* that is presented in Section B.<sup>2</sup> In this function, first a check of the correct length of the proof is made in order to avoid a second pre-image attack on the root hash. Second, an OTP index is derived from the passed proof,<sup>3</sup> and then the derived index is compared with the index of an operation that the user confirms (i.e., *opID*). If this check is successful, then the root hash is computed and compared with the expected root hash in the calling function *verifyOTP()*. In the positive case, the operation is executed (e.g., crypto-tokens are transferred).

In the following, we present extensions of our framework that improve its efficiency and usability, and introduce new features.

### C. Increasing the Number of OTPs

A small number of OTPs can have usability and security consequences. First, users executing many transactions<sup>4</sup> would need to create new OTPs often, and thus change their addresses. Second, an attacker possessing *SK<sub>U</sub>* can flood the smart-contract wallet with initialized operations, rendering all the OTPs unusable. Therefore, we need to increase the number of OTPs in order to make an attack unfeasible. However, increasing the number of OTPs linearly increases the amount of data that the client needs to preserve in its storage (i.e., leaves of the tree). For example, if the number of OTPs is  $2^{20}$ , then the client has to store *16.7MB* of data (considering  $S = 16B$ ), which is feasible even on storage-limited devices. However, e.g., for  $2^{35}$  OTPs, the client has to store *549.8GB* of data, which may be inconvenient even on PCs.

To solve this issue, we propose modifying the base approach by applying a time-space trade-off method [48] for OTPs. Namely, we introduce hash chains of which last items are aggregated by a Merkle tree. With such a construction, OTPs can be encoded as elements of a chain and revealed layer by layer in the reverse direction to the direction of creating the chains. This allows multiplication of the number of OTPs by the chain length without increasing the client's storage overhead, but imposing a larger number of hash computations on the smart contract and the authenticator.

An illustration of this construction is presented in the bottom left part of Figure 3.<sup>5</sup> A hash chain of length  $P$  is built from each OTP assumed so far. Then, the last items of all hash chains are used as the first iteration layer that

provides  $N/P$  OTPs.<sup>6</sup> Similarly, the penultimate items of all the hash chains are used as the second iteration layer, etc., until the last iteration layer consisting of the first items of hash chains has been reached (i.e., outputs of  $F_k(\cdot)$ ). We emphasize that introducing hash chains may cause a violation of the requirement on the independence of OTPs if implemented naively – i.e., OTPs from lower iteration layers can be used to derive OTPs in upper layers. Therefore, to enforce this requirement, we invalidate all the OTPs associated with the previous iteration layers by a sliding window at the smart contract.

The authenticator has to be updated to provide OTPs by

$$\text{getOTP}(i) = h^{\alpha(i)} \left( F_k \left( \beta(i) \right) \right), \quad (1)$$

where  $i$  is an operation ID,  $\alpha(i)$  determines an index in a hash chain, and  $\beta(i)$  determines an index in the last iteration layer of OTPs. We provide concrete expressions for  $\alpha(i)$  and  $\beta(i)$  in Equation 3, which involves all proposed improvements and optimizations. A derivation of the root hash from the OTP in the smart contract needs to be updated as well (see Algorithm 6 of Appendix).<sup>7</sup> In detail, the smart contract needs to execute  $P - \alpha(i) - 1 = \left\lfloor \frac{iP}{N} \right\rfloor$  hash computations, which is a complementary number to a number of hash computations at the authenticator with regard to the length of hash chain  $P$ . Also, the client needs to be modified, requiring a computation of a proof to use a leaf index relative to a current iteration layer of OTPs, computed as  $i \% \frac{N}{P}$ .

With this improvement, given the number of leaves equal to  $2^{20}$  and  $P = 2^{10}$ , the client stores only *16.7MB* of data,<sup>8</sup> the client can use  $2^{30}$  OTPs within a single tree. On the other hand, this approach requires, on average, the execution of additional  $P/2$  hash computations at the smart contract, imposing additional costs. However, our experiments prove the benefits of this approach, as shown in Section V-A.

### D. Depletion of OTPs

Even with the previous modification, the number of OTPs remains bounded, and thus they may be depleted. We propose handling depleted OTPs by a special operation that replaces the current tree with a new one. In order to introduce a new tree securely, we propose updating the root hash value while using the last OTP of the current tree for confirmation. Nevertheless, for this purpose we cannot use our protocol consisting of two stages, as the attacker possessing *SK<sub>U</sub>* could be “faster” than the user and might initialize the last operation and thus block all the funds in the wallet. If we were to allow repeated initialization of this operation, then we would create a race condition issue.

To avoid this race condition issue, we propose a dedicated protocol that replaces the root hash during three stages of

<sup>2</sup>Note that this algorithm contains, not yet described, improvements.

<sup>3</sup>Note that each item of the proof dedicates one bit to indicate the left-/right position of this item in the Merkle tree, enabling us to derive the *opID*.

<sup>4</sup>For example, in the Ethereum, a few smart contracts have over  $2^{20}$  transactions executed.

<sup>5</sup>Note that this figure contains further, not yet described, improvements.

<sup>6</sup>We assume for simplicity that the greatest common divisor of  $N, P$  is  $P$ .

<sup>7</sup>Note that algorithms contain further, not yet described, improvements.

<sup>8</sup>Hashes built from the first iteration layer of OTPs.



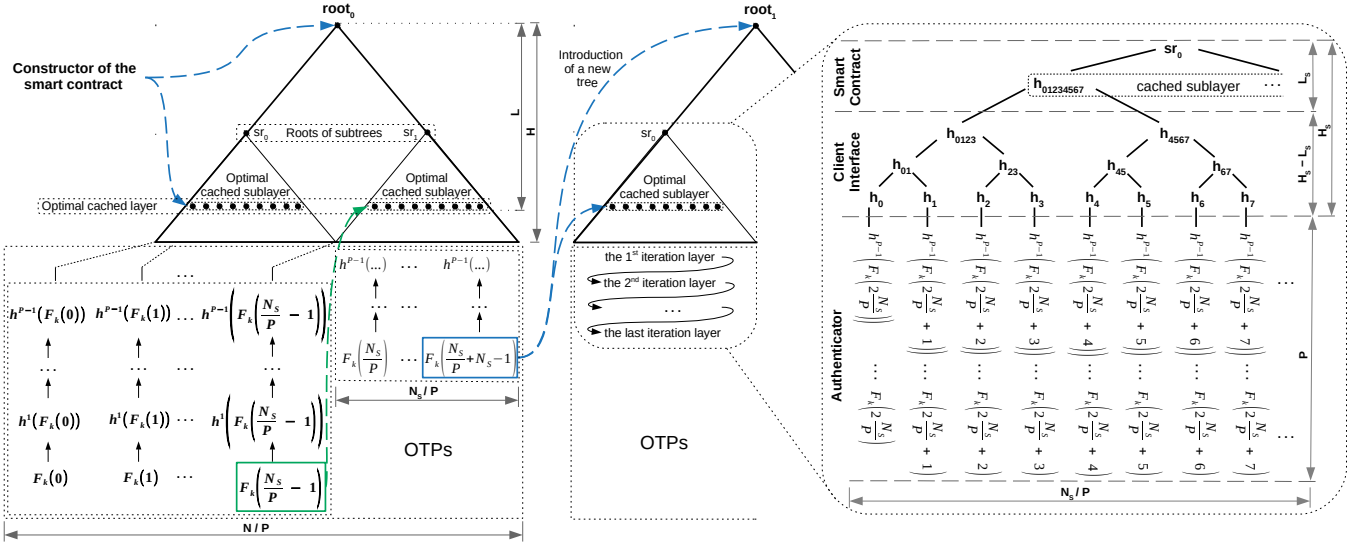


Fig. 3: An overview of our approach and its improvements. In the left part of the figure, we depict the partitioning of a tree into subtrees, while for each subtree we append hash chains that balance the size of the client's storage and the number of hash computations at the smart contract. In the middle part of the figure, we depict the adjustment of the next parent tree. In the right part of the figure, we depict the details of a subtree, which contains a split control in the provision/verification of OTPs (and their proofs) across the components of our framework. In detail, OTPs are provided by the authenticator, while proofs are constructed from the  $H$ th layer of the parent Merkle tree stored at the client. A proof contains only  $H_S - L_S$  nodes, as the remaining  $L_S$  nodes are under the control of the smart contract.

interaction with the blockchain, which require two append-only lists  $L_1$  and  $L_2$  (see Algorithm 2):

- 1) the user enters  $OTP_{N-1}$  to the client. The client sends  $h(OTP_{N-1} \parallel root^{new})$  to the smart contract, which appends it to  $L_1$ .
  - 2) The client sends  $root^{new}$  to the smart contract, which appends it to  $L_2$ .
  - 3) The client passes  $OTP_{N-1}$  with  $\pi_{N-1}$  to the smart contract, where the first matching entries of  $L_1$  and  $L_2$  are located in order to perform the introduction of  $root^{new}$ .
- Finally,  $L_1$  and  $L_2$  are cleared for future updates.

Locating the first entries in the lists relies on the append-only feature of lists, and hence no attacker can make the first valid pair of entries in the lists. With this improvement, the authenticator needs to be adapted to support an unlimited number of operations (see Equation 3).<sup>9</sup> In order to adapt this improvement at the client, the client needs to compute and store leaves associated with a new tree. Therefore, a user provides the client with  $k$  again. Alternatively, the new leaves can be transferred by microSD card, if insecure environment is considered.

#### E. Cost Optimizations

**Caching at the Smart Contract.** With a high Merkle tree, the derivation of the root hash from a leaf node may be costly. Although the number of hash computations coming from the Merkle tree is logarithmic in the number of leaves, the cost imposed on the blockchain platform may be significant for higher trees. We propose to reduce this cost by caching an arbitrary tree layer of depth  $L$  at the smart contract, and do proof verifications against a cached layer. Hence, each call of

#### Algorithm 2: Introduction of a new root hash at the smart contract

```

 $L_1 \leftarrow [];$   $\triangleright$  Items have form  $\langle h(\text{new root hash} \parallel OTP) \rangle$ 
 $L_2 \leftarrow [];$   $\triangleright$  Items have form  $\langle \text{new root hash} \rangle$ 

function 1_newRootHash(hRootAndOTP) public
  assert sender = owner;
  assert nextOpId %  $N = N - 1$ ;  $\triangleright$  The last oper. of tree
   $L_1.append(hRootAndOTP);$ 

function 2_newRootHash(rootnew) public
  assert sender = owner;
  assert nextOpId %  $N = N - 1$ ;  $\triangleright$  The last oper. of tree
   $L_2.append(root^{new});$ 

function 3_newRootHash(otp,  $\pi$ ) public
  assert nextOpId %  $N = N - 1$ ;  $\triangleright$  The last oper. of tree
  verifyOTP(otp,  $\pi$ , nextOpId);
  for  $\{i \leftarrow 0; i < L_2.len; i++\}$  do
    for  $\{j \leftarrow 0; j < L_1.len; j++\}$  do
      if  $h(L_2[i] \parallel otp) = L_1[j]$  then
        rootHash  $\leftarrow L_2[i]$ ;
         $L_1, L_2 \leftarrow [], []$ ;
        nextOpId++;

```

*deriveRootHash()* will execute  $L$  fewer hash computations in contrast to the version that derives the root hash, while the client will transfer by  $L$  fewer elements in the proof.

The minimal operational cost can be achieved by directly caching leaves of the tree, which accounts only for hash computations coming from hash chains, not a Merkle tree. However, storing such an amount of cached data on the blockchain is too expensive. Therefore, this cost optimization must be considered as a trade-off between the depth  $L$  of the cached layer and the price required for the storage of such a cached layer at the blockchain (see Section V-A, where an optimal cached layer is determined).

We depict this optimization in the left part of Figure 3, and we show that an optimal caching layer can be further

<sup>9</sup>Note that equation contains further, not yet introduced, optimizations.

partitioned into caching sublayers of subtrees (we introduce subtrees later). The right part of the figure shows how we combine Merkle tree and hash chains. To enable this optimization, the cached layer of the Merkle tree must be stored by the constructor of the smart contract, instead of the root hash. Since that moment onward, the cached layer replaces the functionality of the root hash, reducing the size of proofs. During the confirmation of operation, an OTP and its proof are used for the derivation of a particular node in the cached layer, instead of the root hash. Then the derived value is compared with an expected node present in the cached layer. The index of an expected node in the cached layer is computed as

$$idxInCache(i) = \left\lfloor \left( i \% \frac{N}{P} \right) / 2^{H-L} \right\rfloor, \quad (2)$$

where  $i$  is the ID of an operation.

**Partitioning to Subtrees.** The caching of the optimal layer minimizes the combined (i.e., deployment and operational) costs of the wallet, but on the other hand, it requires prepayment for storing the cache on the blockchain. If the cached layer were to contain a high number of nodes, then the initial deployment cost could be prohibitively high, and moreover, the user might not deplete all the prepaid OTPs.

To overcome this usability issue, we propose partitioning an optimal cached layer to groups having the same size, forming sublayers that belong to subtrees (see the left part of Figure 3). Starting with the deployment of the smart contract, the cached sublayer of the first subtree and the “parent” root hash are passed to the constructor; the cached sublayer is stored on the blockchain and its consistency against the parent root hash is verified. Then during the operational stage, when confirmation of operation is performed, the passed OTP is verified against a particular node in the cached sublayer of the current subtree (saving costs for not doing verification against the parent root hash).

If the last OTP of the current subtree is reached, then no operation other than the introduction of the next subtree can be initialized (see the green dashed arrow in Figure 3). Namely, the client introduces the next subtree in a single step by calling the smart contract’s function *setNextSubtree()* with the arguments containing:

- 1) the last OTP of the current subtree  $OTP_{(N_S-1)+\delta N_S}$ ,  $\delta \in \{1, \dots, N/N_S - 1\}$  with its proof  $\pi_{otp}$ , and
- 2) the cached sublayer of the next subtree with  $\pi_{cr}$  (the proof of the next subtree’s root), both computed by the client from its storage.

The pseudo-code of introducing the next subtree (i.e., its sublayer) is shown in Algorithm 3. The current subtree’s cached sublayer is replaced by a new one, which is verified by the function *subtreeConsistency()* against the parent root hash with the use of the passed proof  $\pi_{cr}$  of the new subtree’s root. Notice that introducing a new subtree invalidates all initialized and not yet confirmed operations of the previous subtree. At the authenticator, this improvement requires accommodating the iteration over layers of hash chains in shorter periods.

---

**Algorithm 3:** Introduction of the next subtree at the smart contract

---

```
currentSubLayer[]; ▷ Adjusted in the constructor

function setNextSubtree(nextSubLayer, otp,  $\pi_{otp}$ ,  $\pi_{cr}$ ) public
  assert sender = owner;
  assert nextOperId %  $N \neq N - 1$ ; ▷ Not the last op. of parent
  assert nextOperId %  $N_S = N_S - 1$ ; ▷ The last op. of subtree
  assert currentSubLayer.len = nextSubLayer.len;
  assert deriveRootHash(otp,  $\pi_{otp}$ , nextOperId) = rootHash;
  currentSubLayer ← nextSubLayer;
  croot ← reduceMT(currentSubLayer, currentSubLayer.len);
  assert subtreeConsistency(croot,  $\pi_{cr}$ , rootHash);
  nextOperId++; ▷ Accounts for this introduction of a subtree
```

---

The authenticator provides OTPs by using the formula from Equation 1 with the following functions:

$$\begin{aligned} \alpha(i) &= P - \left\lfloor \frac{(i \% N_S)P}{N_S} \right\rfloor - 1, \\ \beta(i) &= \left\lfloor \frac{i}{N_S} \right\rfloor \frac{N_S}{P} + \left( i \% \frac{N_S}{P} \right), \end{aligned} \quad (3)$$

where  $i$  is an operation ID and  $N_S$  is the number of OTPs provided by a single subtree. We remark, that due to this optimization, the update of a new parent root hash requires, additionally to Algorithm 2, the introduction of a cached sublayer of the first subtree.

#### F. Functionality Extension of the Wallet

So far, we have considered only the crypto-token transfer operation. However, our proposed protocol enables us to extend the set of operations; for example, to give users more flexibility and security in managing their crypto-tokens. In order to increase the security features of our wallet, we extended the operation set by supporting daily spending limits and last resort information.

**Daily Limit.** Adjusting a daily spending limit is a functionality that contributes primarily to the user’s self-monitoring of expenses, but at the same time, it avoids typos in transfers that exceeds a daily limit and are irreversible. This operation has the only argument representing an amount that can be spent in a single calendar day.

**Last Resort Address and Timeout.** As users may lose their secrets, leading to an unrecoverable state, we propose an extension that deals with such a situation based on the last resort address and timeout options. This sort of a functionality needs two dedicated operations of our protocol, one for the adjustment of the last resort address and another one for the adjustment of the timeout. If the timeout has elapsed, then anyone may call a dedicated function that transfers all the crypto-tokens of the wallet to the last resort address and destroys the contract. Note that the last resort address is enforced to be different than the address of the owner of the smart contract in order to avoid transferring all funds of the wallet to the owner’s address (i.e., that might be under control of the adversary) when the user loses all secrets. Also note that refreshment of the most recent activity is made only in the second stage of our protocol, requiring an OTP.



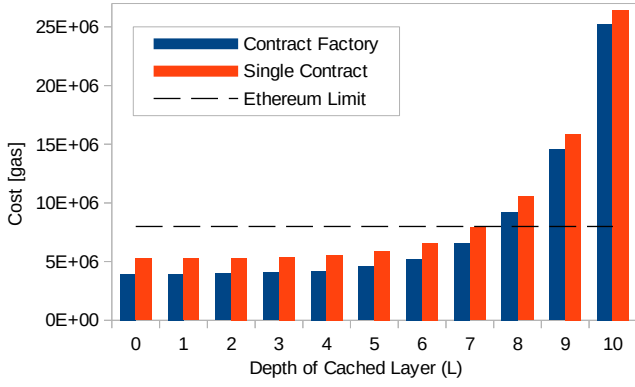


Fig. 4: Deployment costs ( $H = H_S$ ).

## V. REALIZATION IN PRACTICE

We have selected the Ethereum cryptocurrency and the Solidity language for the implementation of the smart contract, HTML with JS for DAPP of the client, and Java (Android) for smartphone App of the authenticator.<sup>10</sup> The smart contract and the client are cryptocurrency-specific components, while the authenticator is cryptocurrency-agnostic and thus can be directly reused in any other cryptocurrencies with similar capabilities. In our implementation, we selected the size of OTPs equal to  $16B$ , which has practical advantages for an air-gapped authenticator, producing OTPs that are 12 mnemonic words long,<sup>11</sup> and at the same time, providing a sufficient security level for most use cases. Next, we used SHA-3 (with truncated output to  $16B$ ) as  $h(\cdot)$  and as an instantiation of  $F_k(\cdot)$ . We selected size of the seed  $k$  equal to  $16B$ , enabling the user to rewrite 12 mnemonic words to backup/restore the authenticator. The source code of our full implementation contains 5316 lines of code (see details in Table I of Appendix) and will be made available upon publication.

### A. Analysis of the Costs

Executing smart contracts over blockchain, i.e., performing computations and storing data, has its costs. In Ethereum Virtual Machine (EVM), these costs are expressed by the level of execution complexity of particular instructions, referred to as *gas*. One unit of gas has its market price in GWEI. In this section, we analyze the costs of our approach using the same value for the hash size  $S$  as well as the OTP size  $O$ , namely,  $16B$ . The  $S$  significantly influences the gas consumption for storing the cached layer at the blockchain, while the effect of  $O$  is negligible. We remark that measured costs can also be influenced by EVM internals (e.g., 32B-long words and corresponding alignment). In the following, we perform a series of gas measurements experiments investigating the parameters of the Merkle tree and hash chains.

#### 1) Costs Related to the Merkle Tree:

**Deployment Cost.** The cost of smart contract deployment is driven mainly by the depth  $L_S$  of the cached sublayer of the first subtree and the size  $S$ . A less significant factor is

<sup>10</sup>Note that the App does not require any permissions.

<sup>11</sup>Considering a dictionary size of 2048 words [39].

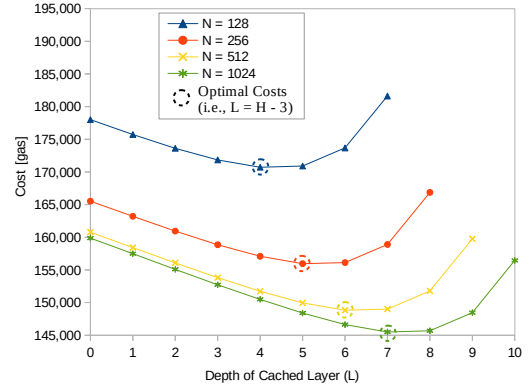


Fig. 5: Average total cost per transfer ( $H = H_S$ ).

the consistency check of a Merkle tree, which is driven by  $L_S$ : the higher  $L_S$  is, the more layers need to be reduced. Similarly, the greater  $H - H_S$  is, the more steps need to be done in proof verification. On the other hand, deployment costs are independent of the length  $P$  of a hash chain; therefore, we omit the hash chain in this experiment and set  $P = 1$ . Further, we abstract from the concept of subtrees in order to analyze a single tree (i.e.,  $H = H_S$ ). The deployment costs of our scheme with respect to depth  $L$  of the cached layer are presented in Figure 4. The figure differentiates two cases – one uses a single wallet contract and the second assumes a factory contract that creates instances of the wallet contract. Thanks to the factory contract, we managed to save approximately  $1.3M$  of gas, which we assume in all the following. Since  $8M$  is the maximum gas limit at the Ethereum main network, we can build a caching layer with  $L_S = 7$  maximally. As we will see later, the maximum  $H_S$  that can be used for the optimal caching layer of a subtree is  $H_S = 10$ , yielding  $2^{10}$  leaves and thus  $2^{10}P$  OTPs per subtree.

**Cost of a Transfer.** Although the cost of each operation supported by our two stage protocol is similar, for this experiment we selected the transfer of crypto-tokens  $O^t$ . We measured the average total cost per  $O^t$  as follows:

$$\begin{aligned}
 O^t\_cost(L, N, P) &= \overline{cost}(O^t(L, N, P)) + \frac{cost(O^d(L))}{N}, \\
 \overline{cost}(O^t(L, N, P)) &= \frac{1}{N} \sum_{i=1}^N cost(O_i^t(L, N, P)), \\
 cost(O_i^t(L, N, P)) &= cost(O_i^{t.init}()) \\
 &\quad + cost(O_i^{t.confirm}(L, N, P)),
 \end{aligned}$$

where the function  $cost()$  measures the cost of an operation in gas units, and  $O_d$  represents the deployment operation. As the purpose of the cached layer is to reduce the number of hash computations in  $confirmOp()$ , the size of an optimal cached layer is subject to a trade-off between the cost of storing the cached layer at the blockchain and the savings benefit of the caching. To explore the properties of the only Merkle tree, we adjusted  $H = H_S$  and  $P = 1$ . As each execution of  $O^t$  (i.e.,  $O_i^t$ ) may have a slightly different gas

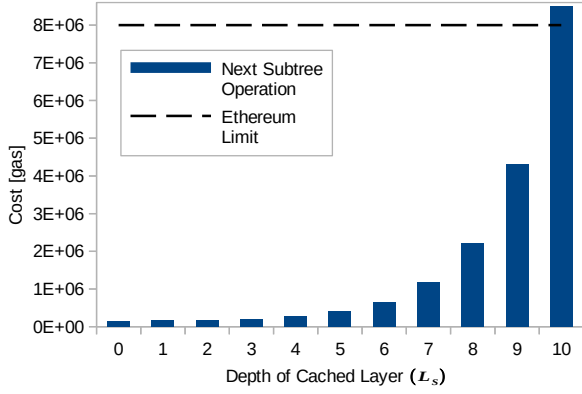


Fig. 6: Cost of introducing the next subtree ( $H = 20$ ,  $H_S = 10$ ).

cost, we measured the average cost of a transaction (i.e.,  $\overline{\text{cost}}(O^t(L, N, P))$  for both stages of our protocol.<sup>12</sup> For completeness, we present the transaction costs of all proposed operations in Appendix C. In Figure 5, we can see that the total average cost per transfer decreases with the increasing number of OTPs, as the deployment cost is spread across more OTPs. The optimal point depicted in the figure minimizes  $O^t$  by balancing  $\text{cost}(O^d(L))$  and  $\overline{\text{cost}}(O^t(L, N, P))$ . An important observation is that the depth of the cached layer for such an optimal point equals  $H - 3$ . In contrast to the version without caching, this optimization has brought a cost reduction of 3.87%, 5.61%, 7.32%, and 8.92%, for 128, 256, 512, and 1,024 leaves (i.e., OTPs here), respectively.

Another question we explored is the number of transfer operations to be executed until a profit of the caching optimization has begun (see Figure 7). We computed a rolling average cost per  $O^t$ , while distinguishing between an optimal caching layer and disabled caching – a profit from caching begins after 53, 90, and 156 transfers, respectively.

**Costs with Subtrees.** We measured the cost of introducing the next subtree within a parent tree depending on  $L_S$ , while we set  $H = 20$  and  $H_S = 10$  (see Figure 6). We found out that when subtrees (and their caching sublayers) are introduced within a dedicated operation, it is significantly cheaper in contrast to introducing subtrees during the deployment.

### 2) Costs Related to Hash Chains:

Since each iteration layer of hash chains contributes to an average cost of  $\text{confirmOp}()$  with around the same value, we measured this value on a few trees with  $P$  up to 512. Next, using this value and the deployment cost, we calculated the average total cost per transfer by adding layers of hash chains to a tree with  $H = H_S$ , and thus increasing  $N$  by a factor of  $P$  until the minimum cost was found. As a result, the optimal caching layer shifted to the leaves of the tree (see Figure 8a), which would, however, exceed the gas limit of Ethereum. To respect the gas limit, we adjusted  $L = 7$ , as depicted in Figure 8b. In contrast to the configurations with  $L = 0$  and  $P = 1$  (from Figure 5), we achieved savings of

<sup>12</sup>Note that the cost of  $\text{initOp}()$  is equal to  $\sim 70k$  of gas for all the operations.

27.80%, 19.61%, 14.95%, and 12.51% for trees with  $H$  equal to 7, 8, 9, and 10, respectively. For the sake of completeness, we calculated costs for  $L = 0$  as well (see Figure 8c). Note that for  $L = 0$  and  $L = 7$ , smaller trees are less expensive, as they require less operations related to proof verification in contrast to bigger trees; these operations consume substantially more gas than operations related to hash chains. Although in this experiment we minimized the total cost per transfer by finding an optimal  $P$ , we highlight that increasing  $P$  contributes to the cost only minimally, but on the other hand, it increases the variance of the cost.

### 3) Costs in Fiat Money:

We assume the average exchange rate of ETH/USD equal to 122 and the “standard” gas price 5 GWEI<sup>13</sup> as of February 13, 2019. For example in the case of  $N = 2^{27}$  (i.e.,  $H = 20$ ,  $H_S = 10$ ,  $P = 2^7$ ,  $L_S = 7$ ), expenses per transfer operation are \$0.10, while expenses for deployment and introduction of a new subtree are \$5.02 and \$0.71, respectively.

## VI. SECURITY ANALYSIS

In this section, we elaborate the security properties of our scheme and its resilience to the attacker model under consideration (see Section III-A), assuming the random oracle model.

**Security Claim 1.** *The attacker with access to  $SK_U$  is able to initiate operations but is unable to confirm them.*

*Justification.* Security of our protocol is achieved by meeting all requirements on OTPs (see Section III-B). In detail, the requirement on the *independence* of two different OTPs is satisfied by the definition of  $F_k(\cdot) \equiv h(k \parallel \cdot)$ , where  $h(\cdot)$  is instantiated by  $\mathcal{RO}$ . This is applicable when  $P = 1$ . However, if  $P > 1$ , then items in previous iteration layers of OTPs can be computed from the next ones. Therefore, to enforce this requirement, we employ an explicit invalidation of all OTPs belonging to all previous iteration layers by a sliding window at the smart contract (see Section IV-C). The requirement on the *linkage* of each  $OTP_i$  with operation  $O_i$  is satisfied thanks to 1)  $\mathcal{RO}$  used for instantiation of  $h(\cdot)$  and 2) by the definition of the Merkle tree, preserving the order of its aggregated leaves. By meeting these requirements, the attacker is able to initiate an operation  $O_j$  in the first stage of our protocol but is unable to use an  $OTP_i$  intercepted in the second stage of the protocol to confirm  $O_j$ , where  $j \neq i$ . Finally, the requirement on the *authenticity* of OTPs is ensured by a random generation of  $k$  and by anchoring the root hash associated with  $k$  at the constructor of the smart contract.  $\square$

**Security Claim 2.** *Assuming  $\delta \in \{0, \dots, \frac{N}{N_S} - 2\}$ : the attacker with access to  $SK_U$  is unable to deplete all OTPs or misuse a stolen OTP that introduces the  $(\delta + 1)$ th subtree.*

*Justification.* When all but one OTPs of the  $\delta$ th subtree are depleted, the last remaining operation  $O_{(N_S-1)+\delta N_S}$ ,  $\delta \in$

<sup>13</sup><https://ethgasstation.info/>.

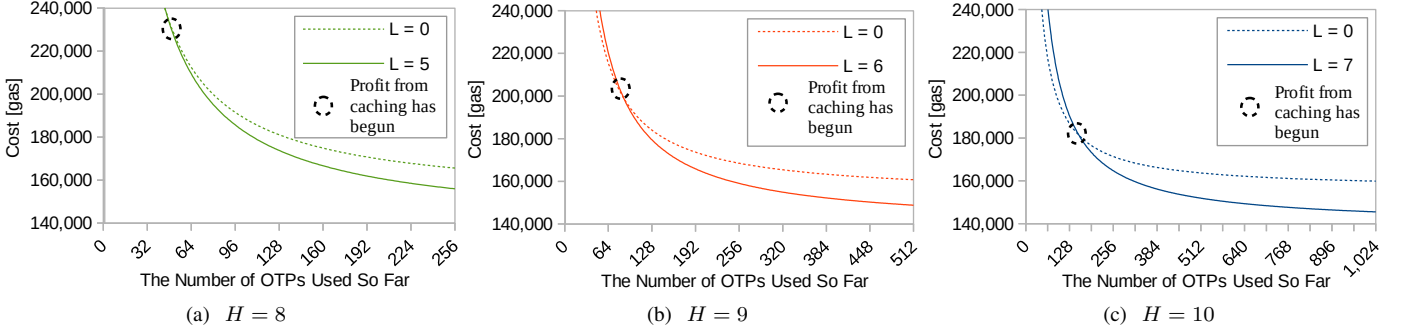


Fig. 7: Rolling average cost per transfer ( $H = H_S$ ).

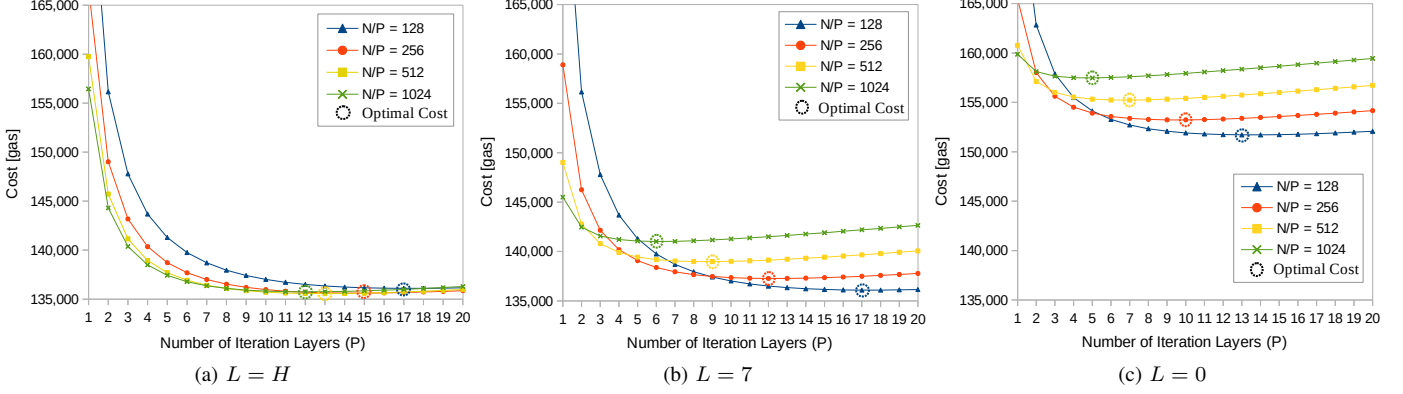


Fig. 8: Average total cost per transfer with regards to the length  $P$  of hash chains.

$\{0, \dots, \frac{N}{N_S} - 2\}$  is enforced by the smart contract to be the introduction of the next subtree. This operation is executed in a single transaction calling the function `setNextSubtree()` of the smart contract (see Algorithm 3) requiring the corresponding  $OTP_{(N_S-1)+\delta N_S}$  that is under control of the user; hence the attacker cannot execute the function to proceed with a further depletion of OTPs in  $(\delta + 1)th$  subtree. If the attacker were to intercept  $OTP_{(N_S-1)+\delta N_S}$  during the introduction of the next subtree, he could use the intercepted OTP only for the introduction of the next valid subtree since the function `setNextSubtree()` also checks a valid cached sublayer of the  $(\delta + 1)th$  subtree against the parent root hash.  $\square$

**Security Claim 3.** Assuming  $\delta = \frac{N}{N_S} - 1$ : the attacker with access to  $SK_U$  is neither able to deplete all OTPs nor introduce a new parent tree nor render the wallet unusable.

*Justification.* In contrast to the adjustment of the next subtree, the situation here is more difficult to handle, since the new parent tree cannot be verified at the smart contract against any paramount field. If we were to use the protocol for execution of an operation (see Section IV) while constraining to the last initialized operation  $O_{(N-1)+\eta N}$ ,  $\eta \in \{0, 1, \dots\}$  of the parent tree, then the attacker could render the smart contract wallet unusable by submitting an arbitrary root hash in `initOp()` and thus block all the funds of the user. If we were to allow repeated initialization of this operation, then we would create a race condition issue. Therefore, this operation needs to be

handled outside of the two stage protocol, using two unlimited append-only lists  $L_1$  and  $L_2$  that are manipulated in three stages of interaction with the blockchain (see Section IV-D). In the first stage,  $h(\text{root}^{new} \parallel OTP_{(N-1)+\eta N})$  is appended to  $L_1$ , hence the attacker cannot extract the value of OTP. In the second stage,  $\text{root}^{new}$  is appended to  $L_2$ , and finally, in the third stage, the user reveals the OTP for confirmation of the first matching entries in both lists. Although the attacker might use an intercepted OTP from the third stage for appending malicious arguments into  $L_1$  and  $L_2$ , when he proceeds to the third stage and submits the intercepted OTP to the smart contract, the user's entries will match as the first ones, and thus a benign parent root hash will be introduced.  $\square$

#### A. Further Properties and Implications

**Requirement on Block Confirmations.** Most cryptocurrencies suffer from long time to finality, potentially enabling the accidental forks, which create parallel inconsistent blockchain views. On the other hand, this issue is not present at blockchain platforms with fast finality, such as Algorand [49], HoneyBadgerBFT [50], StrongChain [51], enabling the direct application of our approach. In the context of blockchains with long time to finality, overly fast confirmation of an operation may be dangerous, as, if an operation were initiated in an “incorrect” view, an attacker holding  $SK_U$  would hijack the OTP and reuse it for a malicious operation settled in the “correct” view. To prevent this threat, the recommendation

is to wait for several block confirmations to ensure that an accidental fork has not happened. For example, in Ethereum, the recommended number of block confirmations to wait is 12 (i.e.,  $\sim 3$  minutes).

**Attacks with a Post Quantum Computer.** Although a resilience to quantum computing (QC) is not the focus of this paper, it is worth to note that our scheme inherits a resilience to QC from the hash-based cryptography. The resilience of our scheme to QC is dependent on the output size of the functions  $F_k(\cdot)$  and  $h(\cdot)$  used for the generation of OTPs and their aggregation, respectively. A generic QC attack against these functions is Grover's algorithm [52], providing a quadratic speedup in searching for the input of the black box function. As indicated by recent research [53] using this algorithm (under realistic assumptions), the security of SHA-3 is reduced from 256 to 166 bits; in our case 98 bits, which provides near term security (i.e., 10 years).

## VII. DISCUSSION

**Passwords.** For simplicity, in the description of our approach we abstracted from password protection at the authenticator and the client. However, to achieve  $(2 + 1/1)$ -factor authentication (see Section VIII-A), it is necessary to set them up. This authentication scheme represents a signature made by  $SK_U$  and OTP from the authenticator + a password to access  $SK_U$  and password to access the authenticator.

**Usability.** Our approach inherits the common usability characteristics of 2FA schemes, such as an extra device to carry, effort for securely storing recovery phrase  $k$ , effort for recalling/entering passwords, and effort for an air-gapped transfer of OTPs, which, according to data from Dhakal et al. [40], takes 14s on average. Note that these usability implications are the same as in the case of existing smart contract wallets with 2FA [17], [15]. In addition to the previous, our approach requires the user to introduce a new subtree/parent tree once in a while. Nevertheless, we envision this effort to be related only to large businesses rather than regular users; considering the example from Section V-A3, the user has to introduce the next subtree after using 132,072 OTPs, while 134,217,728 OTPs are available to use before re-initialization of a new parent tree.

**Costs.** With consumption of up to  $\sim 140k$  gas units per operation, our approach is comparable to equivalent 2FA solutions based on smart contracts: Ethereum MultiSigWallet [17] requires  $\sim 275k$  gas units<sup>14</sup> and TrezorMultisig2of3 [15] requires  $\sim 95k$  gas units<sup>15</sup> per operation.

**General Purpose Applicability.** In this work, we focused on smart contract wallets. However, we note that the protocol of our scheme can be utilized in any smart contract based application for the purpose of 2-factor authentication.

<sup>14</sup><https://etherscan.io/tx/0xdb6e938...> and <https://etherscan.io/tx/0x328a7cc...>

<sup>15</sup><https://etherscan.io/tx/0xfc7bb...> (two signatures in a single transaction).

## VIII. RELATED WORK

In this section, we first compare our approach with other hash-based cryptography approaches, then we propose a classification of blockchain-related authentication schemes, and finally we extend the previous work of Eskandari et al. [4] and Bonneau et al. [6], by categorizing and reviewing examples of cryptocurrency wallet solutions. For an overview of these solutions, we refer the reader to Appendix D.

**Related Hash-Based Approaches.** Although Merkle signatures [44] utilize Merkle trees for aggregation of several one-time verification keys (e.g., [42]), the size of these keys and signatures is substantially larger than in the case of our approach. Even further optimization of the signature size (i.e., Winternitz OTS [43]) does not make signatures so short as in our approach. Next, we highlight that we utilize hash chains for multiplication of OTPs, which is different than their application in Winternitz OTS [43] that utilize them for the purpose of reducing the size of a single Lamport-Diffie OTS [42] by encoding multiple bits of a message digest into the number of recurrent hash computations.

### A. Classification of Authentication Schemes

In the context of the blockchain, we distinguish between  $k$ -factor authentication *against the blockchain* and  $k$ -factor authentication *against the authentication factors* themselves. For example, an authentication method may require a user to perform 2-of-2 multi-signature in order to execute a transfer, while the user may keep each private key stored in a dedicated device – each requiring a different password. In this case, 2FA is performed against the blockchain, since both signatures are verified by all miners of the blockchain. Additionally, a one-factor authentication is performed once in each device of the user by entering a password in each of them. For clarity we classify authentication schemes by the following notation:

$$\left( Z + X_1 / \dots / X_Z \right),$$

where  $Z \in \{0, 1, \dots\}$  represents the number of authentication factors against the blockchain and  $X_i \in \{0, 1, \dots\} \mid i \in [1, \dots, Z]$  represents the number of authentication factors against the  $i$ -th factor of  $Z$ . With this in mind, we remark that the previous example provides  $(2 + 1/1)$ -factor authentication: twice against the blockchain (i.e., two signatures), once for accessing the first device (i.e., the first password), and once for accessing the second device (i.e., the second password). Since the previous notation is insufficient for authentication schemes that use secret sharing [54], we extend it as follows:

$$\left( Z^{(W_1, \dots, W_Z)} + \left( X_1^1, \dots, X_1^{W_1} \right) / \dots / \left( X_Z^1, \dots, X_Z^{W_Z} \right) \right),$$

where  $Z$  has the same meaning as in the previous case,  $W_i \in \{0, 1, \dots\} \mid i \in [1, \dots, Z]$  denotes the minimum number of secret shares required to use the complete  $i$ -th secret  $X_i$ . With this in mind, we remark that the aforementioned example provides  $\left( 2^{(1,1)} + (1)/(1) \right)$ -factor authentication: twice against

the blockchain (i.e., two signatures), once for accessing the first device (i.e., the first password), and once for accessing the second device (i.e., the second password). We consider an implicit value of  $W_i = 1$ ; hence, the classification  $(2 + 1/1)$  represents the same as the previous one (the first notation suffices). If one of the private keys were additionally split into two shares, each encrypted by a password, then such an approach would provide  $\left(2^{(2,1)} + (1,1)/(1)\right)$ -factor authentication.

### B. Wallet Types

**Keys in Local Storage.** In this category of wallets, the private keys are stored in plaintext form on the local storage of a machine, thus providing  $(1 + 0)$ -factor authentication. Examples that enable the use of unencrypted private key files are Bitcoin Core [55] or MyEtherWallet [56] wallets.

**Password-Protected Wallets.** These wallets require a user-specified password to encrypt a private key stored on the local storage, thus providing  $(1 + 1)$ -factor authentication. Examples that support this functionality are Armory Secure Wallet [11], Electrum Wallet [12], MyEtherWallet [56], Bitcoin Core [55], and Bitcoin Wallet [57]. This category addresses physical theft, yet enables the brute force of passwords and digital theft (e.g., keylogger).

**Password-Derived Wallets.** Password-derived wallets [58] (a.k.a., brain wallets or hierarchical deterministic wallets) can compute a sequence of private keys from only a single mnemonic string and/or password. This approach takes the advantage of the key creation in the ECDSA signature scheme that is used by many blockchain platforms. Examples of password-derived wallets are Electrum [12], Armory Secure Wallet [11], and Daedalus Wallet [59]. The wallets in this category provide  $(1 + X_1)$ -factor authentication (usually  $X_1 = 1$ ) and also suffer from weak passwords [2].

**Hardware Storage Wallets.** In general, wallets of this category include devices that can only sign transactions by private keys stored inside a sealed storage, while the keys never leave the device. To sign a transaction, the user connects a device to a machine and enters his passphrase. When signing a transaction, the device displays the transaction's data to the user, who may verify the details. Thus, wallets of this category usually provide  $(1 + 1)$ -factor authentication. Popular USB (or Bluetooth) hardware wallets containing displays are offered by Trezor [60], Ledger [61], KeepKey [62], and BitLox [63]. An example of a USB wallet that is not resistant against keyloggers is Ledger Nano [64] – it does not have a display, and after connecting to a machine, it requires a user to enter a PIN code through the client running on the machine.  $(1 + 0)$ -factor authentication is provided by a credit-card-shaped hardware wallet from CoolBitX [65]. A hybrid approach that relies on a server providing a relay for 2FA is offered by BitBox USB Wallet [66]. Although a BitBox device does not have a display, after connecting to a machine, it communicates with the client running on the machine and at the same time, it communicates with a smartphone app through BitBox's sever; each requested transaction is displayed and

confirmed by the user on the smartphone. One limitation of this solution is the lack of self-sovereignty, as 2FA requires a connection to a server.

**Split Control – Threshold Cryptography.** In threshold cryptography [54], [67], [68], [69], a key is split into several parties which enables the spending of crypto-tokens only when  $n$ -of- $m$  parties collaborate. Wallets based on threshold cryptography provide  $\left(1^{(W_1, \dots, W_n)} + (X_1, \dots, X_n)\right)$ -factor authentication, as only a single signature verification is made on a blockchain, but  $n$  verifications are made by parties that compute a signature. Therefore, all the computations for co-signing a transaction are performed off-chain, which provides anonymity of access control policies (i.e., a transaction has a single signature) in contrast to the multi-signature scheme that is publicly visible on a blockchain. An example of this category is presented by Goldfeder et al. [5]. One limitation of this solution is a computational overhead that is directly proportional to the number of involved parties  $m$  (e.g., for  $m = 2$  it takes 13.26s). Another example of this category is a USB dongle called Mycelium Entropy [10], which, when connected to a printer, generates triplets of paper wallets using 2-of-3 Shamir's secret sharing; providing  $(1^{(2)} + (0, 0))$ -factor authentication.

**Split Control – Multi-Signature Wallets.** In the case of multi-signature wallets,  $n$ -of- $m$  owners of the wallet must co-sign the transaction made from the multi-owned address. Thus, the wallets of this category provide  $(n + X_1 / \dots / X_n)$ -factor authentication. One example of a multi-owned address approach is Bitcoin's Pay to Script Hash (P2SH).<sup>16</sup> Examples supporting multi-owned addresses are Lockboxes of Armory Secure Wallet [11] and Electrum Wallet [12]. A property of multi-owned addresses is that each transaction with such an address requires off-chain communication. A hybrid instance of this category and client-side hosted wallets category is Trusted Coin's cosigning service [13], which provides a 2-of-3 multi-signature scheme – the user owns a primary and a backup key, while TrustedCoin owns the third key. Each transaction is signed first by user's primary key and then, based on the correctness of the OTP from Google Authenticator, by TrustedCoin's key. Another hybrid instance of this category and client-side hosted wallets is Copay Wallet [14]. With Copay, a user can create a multi-owned Copay wallet, where the user has all keys in his machines and each transaction is co-signed by  $n$ -of- $m$  keys. Transactions are resent across user's machines during multi-signing through Copay.

**Split-Control – State-Aware Smart Contracts.** State-aware smart contracts provide “rules” for how crypto-tokens of a contract can be spent by owners, while they keep the current setting of the rules at the blockchain. The most common example of state-aware smart contracts is the 2-of-3 multi-signature scheme that provides  $(2 + X_1/X_2)$ -factor authentication. An example of the 2-of-3 multi-signature approach that supports

<sup>16</sup>We refer to the term *multi-owned address of P2SH* for clarity, although it can be viewed as *Turing-incomplete smart contract*.

only Trezor hardware wallets is *TrezorMultisig2of3* from Unchained Capital [15]. One disadvantage of this solution is that the user has to own three Trezor devices, which may be an expensive solution that, moreover, relies only on a single vendor. Another example of this category, but using the  $n$ -of- $m$  multi-signature scheme, is Parity Wallet [16]. However, two recent critical bugs [70], [71] have caused the multi-signature scheme to be currently disabled. The  $n$ -of- $m$  multi-signature scheme is also used in *Ethereum MultiSigWallet* from ConsenSys [17].

**Hosted Wallets.** Common features of hosted wallets are that they provide an online interface for interaction with the blockchain, managing crypto-tokens, and viewing transaction history, while they also store private keys at the server side. If a hosted wallet has full control over private keys, it is referred to as a *server-side wallet*. A server-side wallet acts like a bank – the trust is centralized. Due to several cases of compromising such server-side wallets [21], [22], [23], [24], the hosted wallets that provide only an interface for interaction with the blockchain (or store only user-encrypted private keys) have started to proliferate. In such wallets, the functionality, including the storage of private keys, has moved to the user’s browser (i.e., client). We refer to these kinds of wallets as *client-side wallets* (a.k.a., hybrid wallets [4]).

**Server-Side Wallets.** Coinbase [18] is an example of a server-side hosted wallet, which also provides exchange services. Whenever a user logs in or performs an operation, he authenticates himself against Coinbase’s server using a password and obtains a code from Google Authenticator/Authy app/SMS. Other examples of server-side wallets having equivalent security level to Coinbase are Circle Pay Wallet [19] and Luno Wallet [20]. The wallets in this category provide  $(0 + 2)$ -factor authentication when 2FA is enabled.

**Client-Side Wallets.** An example of a client-side hosted wallet is Blockchain Wallet [32]. Blockchain Wallet is a password-derived wallet that provides 1-factor authentication against the server based on the knowledge of a password and additionally enables 2FA against the server through one of the options consisting of Google Authenticator, YubiKey, SMS, and email. When creating a transaction, the user can be authenticated by entering his secondary password. Equivalent functionality and security level as in Blockchain Wallet are offered by BTC Wallet [31]. In contrast to Blockchain Wallet, BTC wallet uses 2FA also during the confirmation of a transaction. Other examples of this category are password-derived wallets, like Mycelium Wallet [26], CarbonWallet [27], Citowise Wallet [28], Coinomi Wallet [29], and Infinito Wallet [30], which, in contrast to the previous examples, do not store backups of encrypted keys at the server. A 2FA feature is provided additionally to password-based authentication, in the case of CarbonWallet. In detail, the 2-of-2 multi-signature scheme uses the machine’s browser and the smartphone’s browser (or the app) to co-sign transactions.

## IX. CONCLUSION

In this paper, we have proposed a smart-contract wallet framework that provides a secure and usable method of managing crypto-tokens. The framework provides 2-factor authentication that is executed in two stages of interaction with the blockchain and protects against an attacker possessing a user’s private key. Our framework uses OTPs realized by pseudo-random function, Merkle trees, and hash chains. We combine these primitives in a novel way, which provides usability in an air-gapped setting that protects against a Man-in-the-Machine [9] attacker as well as quantum computer attacks. The protocol of our framework is general and can be utilized, beside the wallets, in any smart contract application for the purpose of 2-factor authentication. In future work, we plan to investigate 2-factor authentication schemes in the context of emerging privacy-preserving smart contract platforms [72], [73].

## REFERENCES

- [1] J.-P. Buntinx, “Brain wallets are not secure and ‘no one should use them,’ says study,” 2016. [Online]. Available: <https://news.bitcoin.com/brain-wallets-not-secure-no-one-use-says-study/>
- [2] N. Courtois, G. Song, and R. Castellucci, “Speed optimizations in bitcoin key recovery attacks,” *Tatra Mountains Mathematical Publications*, vol. 67, no. 1, pp. 55–68, 2016.
- [3] V. Chia, P. Hartel, Q. Hum, S. Ma, G. Piliouras, D. Reijnders, M. van Staalduinen, and P. Szalachowski, “Rethinking blockchain security: Position paper,” in *Proceedings of the IEEE International Conference on Blockchain (Blockchain)*, 2018.
- [4] S. Eskandari, J. Clark, D. Barrera, and E. Stobert, “A first look at the usability of bitcoin key management,” *preprint arXiv:1802.04351*, 2018.
- [5] S. Goldfeder, R. Gennaro, H. Kalodner, J. Bonneau, J. A. Kroll, E. W. Felten, and A. Narayanan, “Securing bitcoin wallets via a new dsa/ecdsa threshold signature scheme,” 2015.
- [6] J. Bonneau, A. Miller, J. Clark, A. Narayanan, J. A. Kroll, and E. W. Felten, “Sok: Research perspectives and challenges for bitcoin and cryptocurrencies,” in *Security and Privacy (SP), 2015 IEEE Symposium on*. IEEE, 2015, pp. 104–121.
- [7] “Cryptocurrency-stealing malware landscape,” Dell SecureWorks, 2015. [Online]. Available: <http://www.opensource.im/cryptocurrency/cryptocurrency-stealing-malware-landscape-dell-secureworks.php>
- [8] A. Peyton, “Cyren sounds siren over bitcoin siphon scam,” FinTech Futures, 2017. [Online]. Available: <https://www.bankingtech.com/2017/01/cyren-sounds-siren-over-bitcoin-siphon-scam/>
- [9] T. Bui, S. P. Rao, M. Antikainen, V. M. Bojan, and T. Aura, “Man-in-the-machine: exploiting ill-secured communication inside the computer,” in *27th USENIX Security Symposium (USENIX Security 18)*. USENIX Association, 2018, pp. 1511–1525.
- [10] Mycelium Holding LTD, “Mycelium Entropy,” 2018. [Online]. Available: <https://mycelium.com/mycelium-entropy.html>
- [11] Armory Technologies, Inc, “Bitcoin Armory,” 2016. [Online]. Available: <https://www.bitcoinarmory.com>
- [12] Electrum Technologies GmbH, “Electrum Bitcoin wallet,” 2018. [Online]. Available: <https://electrum.org/>
- [13] TrustedCoin, LLC, “TrustedCoin cosigning service,” 2018. [Online]. Available: <https://trustedcoin.com>
- [14] Copay, “The Secure, Shared Bitcoin Wallet,” 2018. [Online]. Available: <https://copay.io/>
- [15] Unchained Capital, “TrezorMultisig2of3: Ethereum Multisignature smart contract,” 2018. [Online]. Available: <https://github.com/unchained-capital/ethereum-multisig>
- [16] P. Technologies, “Parity Wallet,” 2018. [Online]. Available: <https://www.parity.io/>
- [17] Consensys, “Ethereum MultiSigWallet,” 2017. [Online]. Available: <https://github.com/ConsenSys/MultiSigWallet>
- [18] Coinbase, “Coinbase wallet,” 2018. [Online]. Available: <https://www.coinbase.com/>



- [19] Circle Internet Financial Limited, "Circle Pay," 2018. [Online]. Available: <https://www.circle.com/en/pay>
- [20] Luno, "Luno wallet," 2018. [Online]. Available: <https://www.luno.com/wallet/>
- [21] Wolfie Zhao, "Bithumb \$31 Million Crypto Exchange Hack: What We Know (And Don't)," 2018. [Online]. Available: <https://www.coindesk.com/bithumb-exchanges-31-million-hack-know-dont-know/>
- [22] Rachel Abrams and Nathaniel Popper, "Trading Site Failure Stirs Ire and Hope for Bitcoin," 2014. [Online]. Available: <https://dealbook.nytimes.com/2014/02/25/trading-site-failure-stirs-ire-and-hope-for-bitcoin/>
- [23] Reuters, "Bitcoin Worth \$72M Was Stolen in Bitfinex Exchange Hack in Hong Kong," 2016. [Online]. Available: <http://fortune.com/2016/08/03/bitcoin-stolen-bitfinex-hack-hong-kong/>
- [24] T. Moore and N. Christin, "Beware the middleman: Empirical analysis of bitcoin-exchange risk," in *International Conference on Financial Cryptography and Data Security*. Springer, 2013, pp. 25–33.
- [25] M. Vasek and T. Moore, "There's no free lunch, even using bitcoin: Tracking the popularity and profits of virtual currency scams," in *International conference on financial cryptography and data security*. Springer, 2015, pp. 44–61.
- [26] M. H. LTD, "Mycelium wallet," 2018. [Online]. Available: <https://wallet.mycelium.com/>
- [27] CarbonWallet.com, "Multi Signature Online Cryptocurrency Wallet," 2018. [Online]. Available: <https://carbonwallet.com/>
- [28] Citowise Developments, "Citowise wallet," 2018. [Online]. Available: <https://citowise.com/wallet>
- [29] Coinomi Ltd, "Coinomi Wallet," 2018. [Online]. Available: <https://coinomi.com/>
- [30] Infinity Blockchain Labs Europe, "Infinito wallet," 2018. [Online]. Available: <https://www.infinitowallet.io/>
- [31] BTC.com, "BTC wallet: Powerful Bitcoin and Bitcoin Cash wallet," 2018. [Online]. Available: <https://wallet.btc.com>
- [32] Blockchain Luxembourg S.A., "Blockchain wallet," 2018. [Online]. Available: <https://blockchain.info/wallet/>
- [33] Binance, "Binance Security Breach Update," 2019. [Online]. Available: <https://binance.zendesk.com/hc/en-us/articles/360028031711-Binance-Security-Breach-Update>
- [34] B. Schneier, "Two-factor authentication: too little, too late," *Communications of the ACM*, vol. 48, no. 4, p. 136, 2005.
- [35] F. Aloul, S. Zahidi, and W. El-Hajj, "Two factor authentication using mobile phones," in *Computer Systems and Applications, 2009. IEEE/ACS International Conference on*. IEEE, 2009, pp. 641–644.
- [36] D. J. Bernstein, N. Duif, T. Lange, P. Schwabe, and B.-Y. Yang, "High-speed high-security signatures," *Journal of Cryptographic Engineering*, vol. 2, no. 2, pp. 77–89, 2012.
- [37] D. Johnson, A. Menezes, and S. Vanstone, "The elliptic curve digital signature algorithm (ecdsa)," *International journal of information security*, vol. 1, no. 1, pp. 36–63, 2001.
- [38] D. Boneh, B. Lynn, and H. Shacham, "Short signatures from the weil pairing," in *International Conference on the Theory and Application of Cryptology and Information Security*. Springer, 2001, pp. 514–532.
- [39] M. Palatinus, P. Rusnak, A. Voisine, and S. Bowe, "BIP-39," 2013. [Online]. Available: <https://github.com/bitcoin/bips/blob/master/bip-0039.mediawiki>
- [40] V. Dhakal, A. M. Feit, P. O. Kristensson, and A. Oulasvirta, "Observations on typing from 136 million keystrokes," in *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*, ser. CHI '18. New York, NY, USA: ACM, 2018, pp. 646:1–646:12. [Online]. Available: <http://doi.acm.org/10.1145/3173574.3174220>
- [41] D. J. Bernstein, "Introduction to post-quantum cryptography," in *Post-quantum cryptography*. Springer, 2009, pp. 1–14.
- [42] L. Lamport, "Constructing digital signatures from a one-way function," Technical Report CSL-98, SRI International Palo Alto, Tech. Rep., 1979.
- [43] C. Dods, N. P. Smart, and M. Stam, "Hash based digital signature schemes," in *IMA International Conference on Cryptography and Coding*. Springer, 2005, pp. 96–115.
- [44] R. C. Merkle, "A certified digital signature," in *Conference on the Theory and Application of Cryptology*. Springer, 1989, pp. 218–238.
- [45] D. M'raihi, M. Bellare, F. Hoornaert, D. Naccache, and O. Ranen, "Hotp: An hmac-based one-time password algorithm," Tech. Rep., 2005.
- [46] D. M'raihi, S. Machani, M. Pei, and J. Rydell, "Totp: Time-based one-time password algorithm," Tech. Rep., 2011.
- [47] L. Lamport, "Password authentication with insecure communication," *Communications of the ACM*, vol. 24, no. 11, pp. 770–772, 1981.
- [48] M. Hellman, "A cryptanalytic time-memory trade-off," *IEEE transactions on Information Theory*, vol. 26, no. 4, pp. 401–406, 1980.
- [49] Y. Gilad, R. Hemo, S. Micali, G. Vlachos, and N. Zeldovich, "Algorand: Scaling byzantine agreements for cryptocurrencies," in *SOSP*, 2017.
- [50] A. Miller, Y. Xia, K. Croman, E. Shi, and D. Song, "The honey badger of bft protocols," in *ACM CCS*, 2016.
- [51] P. Szalachowski, D. Reijnders, I. Homoliak, and S. Sun, "Strongchain: Transparent and collaborative proof-of-work consensus," in *USENIX Security*, 2019.
- [52] L. K. Grover, "A fast quantum mechanical algorithm for database search," in *Proceedings of the twenty-eighth annual ACM symposium on Theory of computing*. ACM, 1996, pp. 212–219.
- [53] M. Amy, O. Di Matteo, V. Gheorghiu, M. Mosca, A. Parent, and J. Schanck, "Estimating the cost of generic quantum pre-image attacks on sha-2 and sha-3," in *International Conference on Selected Areas in Cryptography*. Springer, 2016, pp. 317–337.
- [54] A. Shamir, "How to share a secret," *Communications of the ACM*, vol. 22, no. 11, pp. 612–613, 1979.
- [55] Bitcoin Project, "Bitcoin Core," 2018. [Online]. Available: <https://bitcoin.org/en/download>
- [56] MyEtherWallet, Inc, "MyEtherWallet," 2018. [Online]. Available: <https://www.myetherwallet.com/>
- [57] Bitcoin Wallet developers, "Bitcoin Wallet," 2018. [Online]. Available: <https://github.com/bitcoin-wallet/bitcoin-wallet>
- [58] G. Maxwell, "Deterministic wallets," 2011. [Online]. Available: <https://bitcointalk.org/index.php?topic=19137>
- [59] Daedalus Team, "Daedalus Wallet," 2018. [Online]. Available: <https://daedaluswallet.io/>
- [60] Trezor, "Trezor," 2018. [Online]. Available: <https://trezor.io/>
- [61] Ledger, "Ledger Nano S," 2018. [Online]. Available: <https://www.ledger.com/products/ledger-nano-s>
- [62] KeepKey, "The Simple Cryptocurrency Hardware Wallet," 2018. [Online]. Available: <https://www.keepkey.com/>
- [63] BitLox, "BitLox wallet," 2018. [Online]. Available: <https://www.bitlox.com>
- [64] Ledger, "Ledger Nano," 2018. [Online]. Available: <https://www.ledgerwallet.com/products/1-ledger-nano>
- [65] CoolBitX, "The CoolWallet S," 2018. [Online]. Available: <https://coolwallet.io/>
- [66] SHIFT Cryptosecurity, "BitBox hardware wallet," 2018. [Online]. Available: <https://shiftcrypto.ch/>
- [67] P. MacKenzie and M. Reiter, "Two-party Generation of DSA Signatures," in *Annual International Cryptology Conference*. Springer, 2001, pp. 137–154.
- [68] R. Gennaro, S. Jarecki, H. Krawczyk, and T. Rabin, "Secure distributed key generation for discrete-log based cryptosystems," *Journal of Cryptology*, vol. 20, no. 1, pp. 51–83, 2007.
- [69] G. R. Blakley *et al.*, "Safeguarding cryptographic keys," in *Proceedings of the national computer conference*, vol. 48, 1979, pp. 313–317.
- [70] Parity Technologies, "The Multi-sig Hack: A Postmortem," 2017. [Online]. Available: <https://paritytech.io/the-multi-sig-hack-a-postmortem/>
- [71] —, "A Postmortem on the Parity Multi-Sig Library Self-Destruct," 2017. [Online]. Available: <https://paritytech.io/a-postmortem-on-the-parity-multi-sig-library-self-destruct/>
- [72] A. Kosba, A. Miller, E. Shi, Z. Wen, and C. Papamanthou, "Hawk: The blockchain model of cryptography and privacy-preserving smart contracts," in *2016 IEEE symposium on security and privacy (SP)*. IEEE, 2016, pp. 839–858.
- [73] R. Cheng, F. Zhang, J. Kos, W. He, N. Hynes, N. Johnson, A. Juels, A. Miller, and D. Song, "Ekiden: A platform for confidentiality-preserving, trustworthy, and performant smart contract execution," *arXiv preprint arXiv:1804.05141*, 2018.
- [74] Consensys, "Ethereum Multisignature Wallet," 2018. [Online]. Available: <https://github.com/Gnosis/MultiSigWallet>

## APPENDIX

### A. Notation

In the appendix, we use the following notation in addition to the notation used above: function  $LSB(.)$  extracts a value of the least significant bit (LSB);  $a \ll b$  represents the bitwise

left shift of argument  $a$  by  $b$  bits; and  $a \mid b$  represents bitwise OR of arguments  $a$  and  $b$ .

### B. Details of Algorithms and Implementation

When bootstrapping the client, OTPs of the last iteration layer of hash chains are generated by Algorithm 4 using current parent tree index  $pti$  and secret seed  $k$ . Generated OTPs are then processed by hash chains, obtaining the first iteration layer of OTPs; this layer is further aggregated into root hash by Algorithm 5, which contains recursive in-situ implementation. When an OTP is used for authentication of an operation, the root hash is derived from the OTP and its proof – first, by resolving hash chains and then proof itself (see Algorithm 6). The details of programming languages and technology used in our work are presented in Table I.

#### Algorithm 4: Generation of OTPs of the last it. layer

```

function generateOTPs( $k, N, \eta$ )
  LL_OTPs  $\leftarrow []$ ;
  for  $\{i \in [0, \dots, \frac{N}{P} - 1]\}$  do
    LL_OTPs.append( $F_k(\eta * \frac{N}{P} + i)$ );
  return LL_OTPs;

```

#### Algorithm 5: Aggregation of OTPs

```

function aggregateOTPs(OTPs)
  hOTPs  $\leftarrow []$ ;
  for  $\{i \in [0, \dots, OTPs.len - 1]\}$  do
    hOTPs[i]  $\leftarrow h^P(OTPs[i])$ ;  $\triangleright$  Leaves of parent tree
  return reduceMT(hOTPs, hOTPs.len);

function reduceMT(hashes, length)
  if  $l = length$  then
    return hashes[0];
  for  $\{i \leftarrow 0; i \leq length/2; i++\}$  do
    hashes[i]  $\leftarrow h(hashes[2i] \parallel hashes[2i + 1])$ ;
  return reduceMT(hashes, length / 2);

```

#### Algorithm 6: A reconstruction of the root hash from OTP and its proof

```

function deriveRootHash( $otp, \pi, opID$ )
  assert  $\pi.len = H$ ;  $\triangleright H = \log_2(\frac{N}{P})$ 
  assert  $opID \% (\frac{N}{P}) = \text{deriveIdx}(\pi)$ ;
  res  $\leftarrow h^{l(otpID \% N) * P / N}(otp)$ ;  $\triangleright$  Resolve hash chain
   $\triangleright$  Then resolve  $\pi$ 
  for  $\{i \leftarrow 0; i < \pi.len; i++\}$  do
    if  $l = LSB(\pi[i])$  then
      res  $\leftarrow h(res \parallel \pi[i])$ ;  $\triangleright$  A node of  $\pi[i]$  is on the right
    else
      res  $\leftarrow h(\pi[i] \parallel res)$ ;  $\triangleright$  A node of  $\pi[i]$  is on the left
  return res;

function deriveIdx( $\pi$ )
  idx  $\leftarrow 0$ ;
  for  $\{i \leftarrow 0; i < \pi.len; i++\}$  do
    if  $l = LSB(\pi[i])$  then
      idx  $\leftarrow idx \mid (1 \ll i)$ ;
  return idx;

```

| Component            | Language               | Technology   | LoC  |
|----------------------|------------------------|--------------|------|
| Smart Contract       | Solidity, Javascript   | Truffle      | 647  |
| Decentralized Client | Javascript, HTML, JSON | DAPP, jQuery | 1939 |
| Authenticator        | Java                   | Android      | 1564 |
| Tests                | Javascript             | Truffle      | 1166 |
| <b>Sum</b>           |                        |              | 5316 |

TABLE I: Development effort in Lines of Code (LoC).

### C. Cost of All Operations

Operational costs of all implemented operations are shown in Table II. In the table, we do not account for deployment costs, and hence we measure only instant gas consumption of the function calls. Note that the first execution of an operation may be more expensive in contrast to the next ones due to an initialization of some variables (i.e., +15k or +30k of gas depending on the operation). Thus, we measured the gas consumption of the second and later executions of each type of operation. The cost measurements were obtained using configuration with optimal cost (i.e.,  $L_S = H_S - 3$ ),  $H = H_S$  and  $P = 1$ , and they are independent of the height  $H$ .

| Operation  | Stage                                  | Mean<br>[gas] | Standard<br>Deviation<br>[gas] | Sum<br>[gas] |
|--|--|---------------|--------------------------------|--------------|
| Transfer   | Init.                                  | 70,558        | 0                              | 139,098      |
|  | Confirm.                               | 68,540        | 129                            |              |
| Set Daily Limit                                  | Init.                                  | 69,342        | 0                              | 133,938      |
|  | Confirm.                               | 64,596        | 129                            |              |
| Set Last Resort<br>Timeout                       | Init.                                  | 69,342        | 0                              | 134,324      |
|  | Confirm.                               | 64,982        | 474                            |              |
| Set Last Resort<br>Address                       | Init.                                  | 70,366        | 0                              | 135,604      |
|  | Confirm.                               | 65,238        | 129                            |              |
| Introduction<br>of the Next<br>Parent Tree       | Stage 1                                | 34,223        | -                              | 1,165,691    |
|  | Stage 2                                | 49,459        | -                              |              |
|  | Stage 3                                | 1,082,009     | -                              |              |
| Introduction of<br>the Next Subtree              | Depends mainly on $L_S$ (see Figure 6) |               |                                |              |
| Send Crypto-Tokens to<br>the Last Resort Address | -                                      | 13,887        | -                              | 13,887       |

TABLE II: Costs of all operations ( $H = 10$ ,  $L_S = 7$ ,  $P = 1$ ).

### D. Classification and Properties of Wallets

We present a comparison of wallets and approaches from related work (see Section VIII) in Table III. In the table, we classify authentication schemes using our proposed classification, while we also survey a few selected security and usability properties of the wallets [4]. In the following, we briefly describe each property and explain criteria stating how we attributed the properties to particular wallets.

**Air-Gapped Property.** We attribute this property (Y) to approaches involving hardware device storing secret information, which do not need a connection to a machine in order to operate.

**Resilience to Tampering with the Client.** We attribute this property (Y) to all hardware wallets that sign transactions within a device, while they require the user to confirm transaction’s details at a device (based on displayed information). Then, we attribute this property to wallets containing multiple clients that collaborate in several steps to co-sign transactions (a chance that all of them are tampered with is small). We partially (P) attribute this property to approaches that do not implicitly provide the protection but instead require additional conditions to meet – e.g., checking the state of the blockchain.

**Post-Quantum Resilience.** We attribute this property (Y) to approaches that utilize hash-based cryptography that is known to be resilient against quantum computing attacks [53].

**No Need for Off-Chain Communication.** We attribute this property (Y) to approaches that do not require a former communication among parties (or devices) to build a (co-)signed transaction, before submitting it to a blockchain.

**Malware Resistance (e.g., Key-Loggers).** We attribute this property (Y) to approaches that either enable signing transactions inside of a device or split signing control over secrets across multiple devices.

**Secret(s) Kept Offline.** We attribute this property (Y) to approaches that keep secrets inside their sealed storage, while they expose only signing transaction functionality. Next, we attribute this property to paper wallets and fully air-gapped devices.

**Independence of Trusted Third Party.** We attribute this property (Y) to approaches that do not require trusted party for operation, while we do not attribute this property to all client-side and server-side hosted wallets. We partially (P) attribute this property to approaches requiring an external relay server for their operation.

**Resilience to Physical Theft.** We attribute this property (Y) to approaches that are protected by an encryption password or PIN. We partially (P) attribute this property to approaches that do not provide password and PIN protection but have a specific feature to enforce uniqueness of an environment in which they are used (e.g., bluetooth pairing).

**Resilience to Password Loss.** We attribute this property (Y) to approaches that provide means for recovery of secrets (e.g., a seed of hierarchical deterministic wallets).

| Authentication Scheme                       |  |   | Air-Gapped Property | Resilience to Tampering with the Client | Post-Quantum Resilience | No Need for Off-Chain Communication | Malware Resistance | Secret(s) Kept Offline | Independence of Trusted Third Party | Resilience to Physical Theft | Resilience to Password Loss | Comments  |
|---|--|---|---------------------|---|-------------------------|-------------------------------------|--------------------|------------------------|-------------------------------------|------------------------------|-----------------------------|---|
| Classification                              | Details  |   |                     |   |                         |                                     |                    |                        |                                     |                              |                             |   |
| Keys in Local Storage                       | 1 + (0)  | Private key   |                     |   |                         |                                     |                    |                        |                                     |                              |                             |   |
| Bitcoin Core [55]                           | 1 + (0)  | For one of the options                                    | N                   | N                                       | N                       | Y                                   | N                  | N                      | Y                                   | N                            | N/A                         |   |
| MyEtherWallet [56]                          | 1 + (0)  | For one of the options                                    | N                   | N                                       | N                       | Y                                   | N                  | N                      | Y                                   | N                            | N/A                         |   |
| Password-Protected Wallets                  | 1 + (1)  | Private key + encryption                                  |                     |   |                         |                                     |                    |                        |                                     |                              |                             |   |
| Armory Secure Wallet [11]                   | 1 + (1)  |   | N                   | N                                       | N                       | Y                                   | N                  | N                      | Y                                   | Y                            | N                           |   |
| Electrum Wallet [12]                        | 1 + (1)  |   | N                   | N                                       | N                       | Y                                   | N                  | N                      | Y                                   | Y                            | N                           |   |
| MyEtherWallet (Offline) [56]                | 1 + (1)  |   | N                   | N                                       | N                       | Y                                   | N                  | N                      | Y                                   | Y                            | N                           |   |
| Bitcoin Core [55]                           | 1 + (1)  |   | N                   | N                                       | N                       | Y                                   | N                  | N                      | Y                                   | Y                            | N                           |   |
| Bitcoin Wallet [57]                         | 1 + (1)  |   | N                   | N                                       | N                       | Y                                   | N                  | N                      | Y                                   | Y                            | N                           |   |
| Password-Derived Wallets                    | 1 + (X <sub>1</sub> )  |   |                     |   |                         |                                     |                    |                        |                                     |                              |                             |   |
| Armory Secure Wallet [11]                   | 1 + (1)  |   | N                   | N                                       | N                       | Y                                   | N                  | N                      | Y                                   | Y                            | Y                           |   |
| Electrum Wallet [12]                        | 1 + (1)  |   | N                   | N                                       | N                       | Y                                   | N                  | N                      | Y                                   | Y                            | Y                           |   |
| Daedalus Wallet [59]                        | 1 + (2)  | 2 passwords   | N                   | N                                       | N                       | Y                                   | N                  | N                      | Y                                   | Y                            | Y                           |   |
| Hardware Storage Wallets                    | 1 + (X <sub>1</sub> )  |   |                     |   |                         |                                     |                    |                        |                                     |                              |                             |   |
| Trezor [60]                                 | 1 + (1)  |   | N                   | Y                                       | N                       | Y                                   | Y                  | Y                      | Y                                   | Y                            | Y                           |   |
| Ledger [61]                                 | 1 + (1)  |   | N                   | Y                                       | N                       | Y                                   | Y                  | Y                      | Y                                   | Y                            | Y                           |   |
| KeepKey [62]                                | 1 + (1)  |   | N                   | Y                                       | N                       | Y                                   | Y                  | Y                      | Y                                   | Y                            | Y                           |   |
| BitLox [63]                                 | 1 + (2)  | 2 passwords*  | N                   | Y                                       | N                       | Y                                   | Y                  | Y                      | Y                                   | Y                            | Y                           | * Additionally, a protection against the evil maid attack                                       |
| CoolWallet S [65]                           | 1 + (0)  |   | N                   | Y                                       | N                       | Y                                   | Y                  | Y                      | Y                                   | P <sup>†</sup>               | N/A                         | † Depending on the mode   |
| Ledger Nano [64]                            | 1 + (2)  | Password + GRID card                                      | N                   | N                                       | N                       | Y                                   | N                  | Y                      | Y                                   | Y                            | Y                           |   |
| BitBox USB Wallet [66]                      | 1 + (2)  | 1 password and App  | N                   | Y                                       | N                       | Y                                   | Y                  | Y                      | P <sup>‡</sup>                      | Y                            | Y                           | ‡ Requires a relay server   |
| Split Control – Threshold Cryptography      | 1 <sup>(W<sub>1</sub>)</sup> + (X <sub>1</sub> <sup>1</sup> , ..., X <sub>1</sub> <sup>W<sub>1</sub></sup> ) |   |                     |   |                         |                                     |                    |                        |                                     |                              |                             |   |
| Goldfeder et al. [5]                        | 1 <sup>(2)</sup> + (1,1)   | Assuming 2 devices, each protected by a password          | N                   | Y                                       | N                       | N                                   | Y                  | N/A                    | N/A                                 | N/A                          | N/A                         |   |
| Mycelium Entropy [10]                       | 1 <sup>(2)</sup> + (0,0)   |   | N                   | Y                                       | N                       | N                                   | Y                  | Y                      | Y                                   | Y                            | N/A                         |   |
| Split Control – Multi-Signature Wallets     | Z + (X <sub>1</sub> / ... / X <sub>z</sub> )   |   |                     |   |                         |                                     |                    |                        |                                     |                              |                             |   |
| Lockboxes of Armory Secure Wallet [11]      | Z + (X <sub>1</sub> / ... / X <sub>z</sub> )   | Z up to 7, X <sub>i</sub> = 1                             | N                   | Y                                       | N                       | N                                   | Y                  | N                      | Y                                   | Y                            | N                           |   |
| Electrum Wallet [12]                        | Z + (X <sub>1</sub> / ... / X <sub>z</sub> )   | Z up to 15, X <sub>i</sub> = 1                            | N                   | Y                                       | N                       | N                                   | Y                  | N                      | Y                                   | Y                            | Y                           |   |
| Trusted Coin’s cosigning service [13]       | 2 + (1/2)  | 2 private keys + 2 passwords and Google Auth.             | N                   | Y                                       | N                       | N                                   | Y                  | N                      | N                                   | Y                            | Y                           | A hybrid client-side wallet   |
| Copay Wallet [14]                           | 2 + (1/1)  |   | N                   | P                                       | N                       | N                                   | Y                  | N                      | P                                   | Y                            | Y                           | A hybrid client-side wallet   |
| Split-Control – State-Aware Smart Contracts | Z + (X <sub>1</sub> / ... / X <sub>z</sub> )   |   |                     |   |                         |                                     |                    |                        |                                     |                              |                             |   |
| TrezorMultisig2of3 [15]                     | 2 + (1/1)  | Assuming that each device is protected by a password      | N                   | Y                                       | N                       | N                                   | Y                  | Y                      | Y                                   | Y                            | Y                           |   |
| Parity Wallet [16]                          | Z + (X <sub>1</sub> / ... / X <sub>z</sub> )   | Z is unlimited, X <sub>i</sub> = 1                        | N                   | Y                                       | N                       | Y                                   | Y                  | N                      | Y                                   | Y                            | Y                           |   |
| Gnosis project [74]                         | Z + (X <sub>1</sub> / ... / X <sub>z</sub> )   | Z up to 50, X <sub>i</sub> = 1                            | N                   | Y                                       | N                       | Y                                   | Y                  | N                      | Y                                   | N/A                          | Y                           |   |
| Our Approach                                | 2 + (1/1)  | Private key and OTPs + passwords                          | Y                   | P <sup>§</sup>                          | Y                       | Y                                   | Y                  | Y                      | Y                                   | Y                            | Y <sup>#</sup>              | §Thanks to a two-stage protocol<br>#Also resilient to lost of all secrets (last resort options) |
| Server-Side Wallets                         | 0 + (X <sub>1</sub> )  |   |                     |   |                         |                                     |                    |                        |                                     |                              |                             |   |
| Coinbase [18]                               | 0 + (2)  | Password, Google Auth./SMS                                | N                   | N                                       | N                       | Y                                   | N                  | N                      | N                                   | Y                            | Y                           |   |
| Circle Pay [19]                             | 0 + (2)  | —” —  | N                   | N                                       | N                       | Y                                   | N                  | N                      | N                                   | Y                            | Y                           |   |
| Luno Wallet [20]                            | 0 + (2)  | Password and Google Auth.                                 | N                   | N                                       | N                       | Y                                   | N                  | N                      | N                                   | Y                            | Y                           |   |
| Client-Side Wallets                         | Z + (X <sub>1</sub> )  |   |                     |   |                         |                                     |                    |                        |                                     |                              |                             |   |
| Blockchain Wallet [32]                      | 1 + (2)  | Password and one of: Google Auth., YubiKey, SMS, or email | N                   | N                                       | N                       | Y                                   | N                  | N                      | N                                   | Y                            | Y                           |   |
| BTC Wallet [31]                             | 1 + (2)  | —” —  | N                   | N                                       | N                       | Y                                   | N                  | N                      | N                                   | Y                            | Y                           |   |
| Mycelium Wallet [26]                        | 1 + (1)  |   | N                   | N                                       | N                       | Y                                   | N                  | N                      | N                                   | Y                            | Y                           |   |
| CarbonWallet [27]                           | 2 + (2)  | 2 private keys stored in browser and smartphone           | N                   | Y                                       | N                       | N                                   | N                  | N                      | N                                   | Y                            | Y                           |   |
| Citowise Wallet [28]                        | 1 + (2)  |   | N                   | P <sup>¶</sup>                          | N                       | Y                                   | N                  | P <sup>¶</sup>         | N                                   | Y                            | Y                           | ¶If combined with Trezor or Ledger  |
| Coinomi Wallet [29]                         | 1 + (1)  |   | N                   | N                                       | N                       | Y                                   | N                  | N                      | N                                   | Y                            | Y                           |   |
| Infinito Wallet [30]                        | 1 + (1)  |   | N                   | N                                       | N                       | Y                                   | N                  | N                      | N                                   | Y                            | Y                           |   |

TABLE III: Comparison of state-of-the-art cryptocurrency wallets.