SmartOTPs: An Air-Gapped 2-Factor Authentication for Smart-Contract Wallets

Ivan Homoliak
FIT, Brno University of Technology
Singapore University of Technology and Design
ihomoliak@fit.vutbr.cz

Dominik Breitenbacher
FIT, Brno University of Technology
dbreitenbacher@gmail.com

Ondrej Hujnak
FIT, Brno University of Technology
ihujnak@fit.vutbr.cz

Pieter Hartel
University of Twente
pieter.hartel@utwente.nl

Alexander Binder
Singapore University of Technology and Design
alexander_binder@sutd.edu.sg

Pawel Szalachowski
Singapore University of Technology and Design
pawel@sutd.edu.sg

ABSTRACT

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

In this paper, we propose SmartOTPs, a smart-contract wallet framework that gives a flexible, usable, and secure way of managing crypto-tokens in a self-sovereign fashion. The proposed framework consists of four components (i.e., an authenticator, a client, a hardware wallet, and a smart contract), and it provides 2-factor authentication (2FA) performed in two stages of interaction with the blockchain. To the best of our knowledge, our framework is the first one that utilizes one-time passwords (OTPs) in the setting of the public blockchain. In SmartOTPs, the OTPs are aggregated by a Merkle tree and hash chains whereby for each authentication only a short OTP (e.g., 16B-long) is transferred from the authenticator to the client. Such a novel setting enables us to make a fully air-gapped authenticator by utilizing small QR codes or a few mnemonic words, while additionally offering resilience against quantum cryptanalysis. We have made a proof-of-concept based on the Ethereum platform. Our cost analysis shows that the average cost of a transfer operation is comparable to existing 2FA solutions using smart contracts with multi-signatures.

ACM Reference Format:
Ivan Homoliak, Dominik Breitenbacher, Ondrej Hujnak, Pieter Hartel, Alexander Binder, and Pawel Szalachowski. 2020. SmartOTPs: An Air-Gapped 2-Factor Authentication for Smart-Contract Wallets. In *2nd ACM Conference on Advances in Financial Technologies (AFT '20), October 21–23, 2020, New York, NY, USA.* ACM, New York, NY, USA, 18 pages. https://doi.org/10.1145/3419614.3423257

1 INTRODUCTION

The success of cryptocurrencies has surpassed all expectations resulting in various open and decentralized platforms that allow users to conduct monetary transfers, write smart contracts, and participate in predictive markets. 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 a 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 [9, 16, 18, 26]. Such cases have brought the attention of the research community to the security issues related to key management in cryptocurrencies [14, 32, 34]. According to the previous work [14, 32], there are a few categories of key management approaches.

In password-protected wallets, private keys are encrypted with selected passwords. Unfortunately, users often choose weak passwords that can be brute-forced if stolen by malware [1]; optionally, such malware may use a keylogger for capturing a passphrase [14, 65]. Another similar option is to use password-derived wallets that generate keys based on the provided password. However, they also suffer from the possibility of weak passwords [26]. Hardware wallets are 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 the device. However, these wallets do not provide protection from an attacker with full access to the device [29, 44, 45], and more importantly, wallets that do not have a secure channel for informing the user about the details of a transaction being signed (e.g., [48]) may be exploited by malware targeting IPC mechanisms [15].

A popular option for storing private keys is to deposit them into server-side hosted (i.e., custodial) wallets and currency-exchange services [10, 20, 50, 61, 62, 66]. In contrast to the previous categories, server-side wallets imply trust in a provider, which is a potential risk of this category. Due to many cases of compromising server-side wallets [2, 9, 53, 68, 77] or fraudulent currency-exchange operators [76], 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 side [17, 19, 21, 40, 57]; hence, trust in the provider is reduced but the users still depend on the provider's infrastructure.

To increase security of former wallet categories, multi-factor authentication (MFA) is often used, which enables spending crypto-tokens only when a number of secrets are used together. However, we emphasize that different security implications stem from the multi-factor authentication made *against a centralized party* (e.g., using Google Authenticator) and *against the blockchain* itself. In the former, the authentication factor is only as secure as the centralized party, while the latter provides stronger security that depends on the assumption of an honest majority of decentralized consensus nodes (i.e., miners) and security of cryptographic primitives used. Wallets from a split control category [32] provide MFA against the blockchain. This can be achieved by threshold cryptography wallets [34, 56], multi-signature wallets [6, 25, 30, 74], and state-aware smart-contract wallets [22, 72, 75]. The last class of wallets is of our concern, as spending rules and security features can be encoded in a smart contract.

Although there are several smart-contract wallets using MFA against the blockchain [22, 75], to the best of our knowledge, none of them provide an air-gapped authentication in the form of short OTPs similar to Google Authenticator.

Proposed Approach. In this paper, we propose SmartOTPs, a framework for smart-contract cryptocurrency wallets, which provides 2FA against data stored on the blockchain. The first factor is represented by the user's private key and the second factor by OTPs. To produce OTPs, the authenticator device of SmartOTPs 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, which enables us to implement the authenticator in a fully air-gapped setting, not requiring any USB or another connection. SmartOTPs belongs to the category of state-aware smart contract wallets, and it provides protection against the attacker that possesses the user's private key *or* the user's authenticator *or* the attacker that tampers with the client.

Contributions. Our main contributions are as follows:

- We show that standard 2FA methods against the blockchain do not meet either the security or usability requirements for an air-gapped setting (see Section 3.2).
- We propose SmartOTPs, a smart-contract wallet framework that provides 2FA against the blockchain while using short OTPs serving as the second factor (see Section 4). OTPs are managed in a novel way, enabling us to make an authenticator device fully air-gapped.
- 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 way (see Section 4.4).
- We implement and evaluate our approach (including hardware version of the authenticator), and we provide the source code of our solution (see Section 6).

2 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 by design resistant to 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 (i.e., *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). 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 replicate the state of the blockchain.

Merkle Tree. A Merkle tree is a data structure based on the binary tree in which every leaf node contains a hash of a single data block, while every 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 logarithmic time complexity, which makes the Merkle tree an efficient scheme for membership verification.

2.1 Notation

By the term *operation* we refer to an action with a smart-contract wallet using SmartOTPs, 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 $\{msg\}_{\mathbb{U}}$ we denote the message msg digitally signed by $\mathbb{U}$, and by $msg.\sigma$ we refer to the signature; $\mathcal{RO}$ is the random oracle; $h(.)$: stands for a cryptographic hash function; $h^i(.)$ substitutes i-times chained function $h(.)$, e.g., $h^2(.) \equiv h(h(.))$; $\|$ is the string concatenation; $h^i_{\mathcal{D}}(.)$ substitutes i-times chained function $h(.)$ with embedded domain separation, e.g., $h^2_{\mathcal{D}}(.) = h(2 \parallel h(1 \parallel .))$; $F_k(.) \equiv h(k \parallel .)$ denotes a pseudo-random function that is parametrized by a secret seed k; % represents modulo operation over integers; $\Sigma.\{KeyGen, Verify, Sign\}$ represents a signature scheme of the blockchain platform; $SK_{\mathbb{U}}, PK_{\mathbb{U}}$ is the private/public key-pair of $\mathbb{U}$, under Σ, and $a \mid b$ represents bitwise OR of arguments a and b.

3 PROBLEM DEFINITION

The main goal of this work is to propose a cryptocurrency wallet framework that provides a secure and usable way of managing crypto-tokens. In particular, we aim to achieve:

Self-Sovereignty: ensures that the user does not depend on the 3rd party's infrastructure, and the user does not share his secrets with anybody. Self-sovereign (i.e., non-custodial) wallets do not pose a single point of failure in contrast to server-side (i.e., custodial) wallets, which when compromised, resulted in huge financial loses [2, 9, 53, 68, 77].

Security: the insufficient security level of some self-sovereign wallets has caused significant financial losses for individuals and companies [16, 18, 26, 60]. 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 [3, 69]. Our motivation is to provide a cheap security extension of the hardware wallets (i.e., the first factor) by using OTPs as the second factor in a fashion similar to Google Authenticator.

3.1 Threat Model

For a generic cryptocurrency described in Section 2, we assume an adversary $\mathcal{A}$ whose goal is to conduct unauthorized operations on the user's behalf or render the user's wallet unusable. $\mathcal{A}$ is able to eavesdrop on the network traffic as well as to participate in the underlying consensus protocol. However, $\mathcal{A}$ is unable to take over the cryptocurrency platform nor to break the used cryptographic primitives. We further assume that $\mathcal{A}$ 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 include $\mathcal{A}$'s transaction and discard the user's one; this attack is also referred to as *transaction front-running*. We assume three types of exclusively occurring attackers, each targeting one of the three components of our framework: (1) $\mathcal{A}$ with access to the user's private key hardware wallet, (2) $\mathcal{A}$ that tampers with the client, and for completeness we also assume (3) $\mathcal{A}$ with access to the authenticator. Next, we assume that the legitimate user correctly executes the proposed protocols and $h(.)$ is an instantiation of $\mathcal{RO}$.

3.2 Design Space

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 cryptotokens 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 explicitly specified and then enforced by the cryptocurrency platform itself. Therefore, using this approach, it is possible to build a flexible wallet with features such as daily spending limits or transfer limits.

General OTPs. With spending rules encoded in a smart contract, it is feasible to design custom security features, such as OTP-based authentication serving as the second factor. In such a setting, the authenticator produces OTPs to authenticate transactions in the smart contract. However, in contrast to digital signatures, OTPs do not provide non-repudiation of data present in a transaction with an OTP; moreover, they can be intercepted and misused by the front-running or the MITM attacks. To overcome this limitation, we argue that a two-stage protocol $\Pi_O^{<G>}$ must be employed, enabling secure utilization of general OTPs in the context of blockchains. In the first stage of $\Pi_O^{<G>}$, an operation O, signed by the user $\mathbb{U}$, is submitted to the blockchain, where it obtains an identifier i. Then, in the second stage, O_i is executed on the blockchain upon the submission of OTP_i that is unambiguously associated with the operation initiated in the first stage.

Requirements of General and Air-Gapped OTPs. Based on the above, we define the necessary security requirements of general OTPs used in the blockchain as follows:

(1) **Authenticity:** each OTP must be associated only with a unique authenticator instance.
(2) **Linkage:** each OTP_i must be linked with exactly a single operation O_i, ensuring that OTP_i cannot be misused for the authentication of $O_j, i \neq j$.
(3) **Independence:** OTP_i linked with the operation O_i cannot be derived from OTP_j of an operation O_j, where $i \neq j$, or an arbitrary set of other OTPs.

Nevertheless, in the air-gapped setting (important for a high usability and security), one more requirement comes into play: **the short length of OTPs**. Short OTPs allow the users to use a relatively small number of mnemonic words or a small QR code to transfer an OTP in an air-gapped fashion. This requirement is of high importance especially in the case when the authenticator is implemented as a resource-constrained embedded device with a small display (e.g., credit-card-shaped wallet, such as CoolBitX [24]).

Analysis of Existing Solutions. We argue that not all solutions meet the requirements of air-gapped OTPs. Asymmetric cryptography primitives such as digital signatures or zero-knowledge proofs are inadequate in this setting, despite meeting all general OTP requirements. State-of-the-art signature schemes with reasonable performance overhead [8, 41] and short signature size produce a 48B-64B long output. The BLS signatures [12] go even beyond the previous constructs and might produce signatures of size 32B. Nevertheless, BLS signatures are unattractive in the setting of the smart contract platforms that put high execution costs for BLS signature verification, which is ~33 times more expensive than in the case of ECDSA with the equivalent security level [13]. Hence, we assume 48B as the minimal feasible OTP size for assymetric cryptography.

However, transferring even 48B in a fully air-gapped environment by transcription of mnemonic words [59] would lack usability for regular users – considering study from Dhakal et al. [27], transcription of 36 English words takes 42s on average, which is much longer than users are willing to "sacrifice." We note that the situation is better with QR code, but on the other hand it has two limitations: (1) when the authenticator is implemented as a simple embedded device, its display might be unable to fit a requested QR code with sufficient scanning properties (to preserve the maximal scanning distance of QR code, the "denser" QR code must be displayed in a larger image [67]) and (2) occasionally, the users might not have a camera in their devices, thus, they can proceed only with a fallback method that uses mnemonics. Finally, most of the currently deployed asymmetric constructions are vulnerable to quantum computing [7].

The problem of long signatures also exists in hash-based signature constructs [28, 46, 51]. Lamport-Diffie one-time signatures (LD-OTS) [46] produce an output of length $2|h(.)|^2$, which, for example in the case of $|h(.)| = 16B$ 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) [28]), 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 $|h(.)|$ (for simplicity omitting checksum) would require $2^{|h(.)|}$ 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 [54] or TOTP [55] require the user to share a secret key k with the authentication server. Then, with each authentication request the user proves that he possesses k by returning the output of an $F_k(.)$ computed with a nonce (i.e., HOTP) or the current timestamp (i.e., TOTP). This approach is insecure in the setting of the blockchain, as the user would have to share the secret k with a smart-contract wallet, making k publicly visible.

A solution that does not publicly disclose secret information and, at the same time, provides short enough OTPs (e.g., $16B \simeq 12$ mnemonic words $\simeq$ QR code v1), can be implemented by Lamport's hash chains [47] or other single hash-chain-based constructs, such as T/Key [43]. A hash chain enables the production of many OTPs by the consecutive 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 ith operation, he sends the $h^{n-i}(k)$ to the smart contract in the second stage of $\Pi_O^{<G>}$. The smart contract then computes $h(.)$ consecutively i times and checks to ascertain whether the obtained value equals the stored value. However, the main drawback of this solution is that *each OTP can be trivially derived from any previous one*, and thereby this scheme does not meet the requirement of OTPs on independence. To detail an attack misusing this flaw, assume the MITM attacker possessing $SK_{\mathbb{U}}$ (i.e., the first factor) is able to initiate operations in the first stage of $\Pi_O^{<G>}$. The attacker $\mathcal{A}$ initiates operation O_i and waits for $\mathbb{U}$ to initiate and confirm an arbitrary follow-up operation $O_j, j > i$. When $\mathbb{U}$ sends OTP_j in the second stage of $\Pi_O^{<G>}$, $\mathcal{A}$ intercepts and "front-runs" the user's transaction by a malicious transaction with OTP_i computed as $h^{j-i}(OTP_j)$. Although one may argue that this scheme can be hardened by a modification denying to confirm older operations than the last initiated one, it would bring a race condition issue in which $\mathcal{A}$ might keep initiating operations in the first stage of $\Pi_O^{<G>}$ each time he intercepts a confirmation transaction from $\mathbb{U}$, causing the DoS attack on the wallet.

4 PROPOSED APPROACH

For a cryptocurrency described in Section 2, we propose SmartOTPs, a 2FA against the blockchain, which consists of: (1) a client $\mathbb{C}$, (2) a private key hardware wallet $\mathbb{W}$ equipped with a display, (3) a smart-contract $\mathbb{S}$, and (4) an air-gapped authenticator $\mathbb{A}$ that might be implemented as an embedded device with limited resources. First, we explain the key idea of our approach, which enables us to construct $\mathbb{A}$ as a fully air-gapped device. Then, we present the base version of SmartOTPs, and finally, we describe modifications.

4.1 Design of an Air-Gapped Authenticator

In our approach, OTPs are generated by a pseudo-random function $F_k(.)$ and then aggregated by a Merkle tree, providing a single value, the root hash ($\mathcal{R}$). $\mathcal{R}$ is stored at $\mathbb{S}$ and serves as a PK for OTPs. Assuming the two stage protocol $\Pi_O^{<G>}$ (further denoted as Π_O), the user $\mathbb{U}$ might confirm the initiated operation O_{opID} by a corresponding OTP_{opID} (provided by $\mathbb{A}$) in the second stage of Π_O, whereby $\mathbb{S}$ verifies the correctness of OTP_{opID} with use of $\mathcal{R}$. A challenge of such an approach is the size of an OTP.

Algorithm 1: Smart contract $\mathbb{S}$ with 2FA

```
▷ VARIABLES AND FUNCTIONS OF ENVIRONMENT:
  tx: a current transaction processed by $\mathbb{S}$,
  balance: the current balance of a contract,
  transfer(r, v): transfer v crypto-tokens from a smart contract to r,
▷ DECLARATION OF TYPES:
  Operation { addr, param, pending, type ∈ {TRANSFER, . . .} }
▷ DECLARATION OF FUNCTIONS:
function constructor(root, pk) public
    operations ← [];                                   ▷ An append-only list
    $PK_{\mathbb{U}}$ ← pk,  $\mathcal{R}$ ← root,  nextOpID ← 0;
    return $\mathbb{S}^{ID}$;                  ▷ Computed by a blockchain platform.
function initOp(a, p, type) public
    assert $\Sigma.verify(tx.\sigma, PK_{\mathbb{U}})$;                 ▷ 1st factor of 2FA
    opID ← nextOpID++;
    operations[opID] ← new Operation(a, p, true, type);
function confirmOp(otp, $\pi$, opID) public
    assert operations[opID].pending;
    verifyOTP(otp, $\pi$, opID);                          ▷ 2nd factor of 2FA
    execOp(operations[opID]);
    operations[opID].pending ← false;
function verifyOTP(otp, $\pi_{opID}$, opID) private
    assert deriveRootHash(otp, $\pi_{opID}$, opID) = $\mathcal{R}$;
function execOp(oper) private
    if TRANSFER = oper.type then
        assert oper.param ≤ balance;
        transfer(oper.addr, oper.param);
```

4.1.1 ***From Straw-Man to the Base Version.***

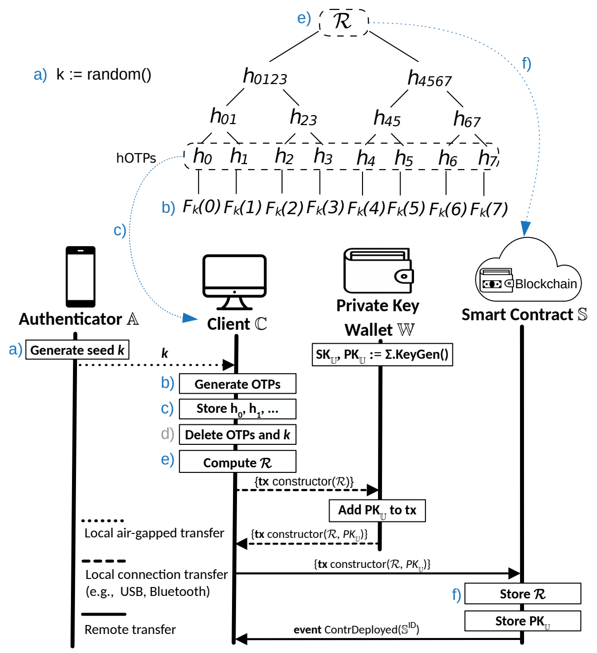
Using the straw-man version, a 2FA requires $\mathbb{A}$ to provide an OTP and its proof. However, in such a straw-man version, the user $\mathbb{U}$ has to transfer $\frac{(S+S\times H)}{8}$ bytes from $\mathbb{A}$ each time he confirms an operation, where S represents the bit-length of an OTP as well as the output of $h(.)$, and H represents the height of a Merkle tree with N leaves; hence $H = log_2(N)$. For example, if $S = 256$ and $H = 10$, then $\mathbb{U}$ would have to transfer 352B each time he confirms an operation, which has very low usability in an air-gapped setting utilizing transcription of mnemonic words [59] (i.e., 264 words) or scanning of several QR codes (e.g., 21 QR codes v1) displayed on an embedded device with a small display. Even further reduction of S to 128 bits would not help to resolve this issue, as the amount of user transferred data would be equal to $176B \simeq 132$ mnemonic words $\simeq 11$ QR codes v1.

We make the observation that it is possible to decouple providing OTPs from providing their proofs. 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 $\mathbb{A}$, while their proofs can be constructed at $\mathbb{C}$ from stored hashes of OTPs. This modification enables us to fetch the nodes of the proof from the storage of $\mathbb{C}$, while $\mathbb{U}$ has to transfer only the OTP itself from $\mathbb{A}$ when confirming an operation (i.e, $S = 128 \simeq 12$ mnemonic words by default).

4.2 Base Version

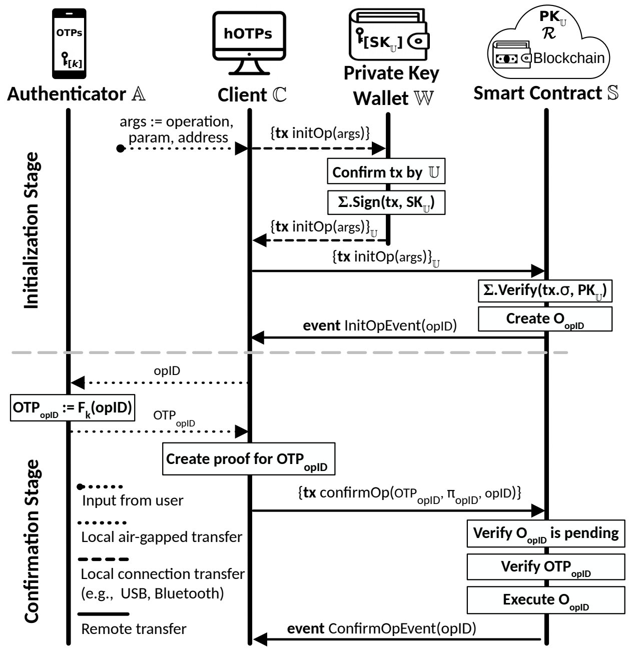
4.2.1 ***Secure Bootstrapping.***

As common in other schemes and protocols, by default, we assume a secure environment for bootstrapping protocol Π_B^S (see Figure 1 and Appendix A.5), which means that $\mathbb{C}$ is trusted and cannot be compromised during execution of Π_B^S. First, $\mathbb{A}$ generates a secret seed k, which is stored as a recovery phrase by $\mathbb{U}$. $\mathbb{W}$ generates a key-pair $SK_{\mathbb{U}}, PK_{\mathbb{U}} \leftarrow \Sigma.KeyGen()$. Next, $\mathbb{U}$ transfers k from $\mathbb{A}$ to $\mathbb{C}$ in an air-gapped manner (i.e., transcribing a few mnemonic words or scanning a QR code).



a) k := random()
e) $\mathcal{R}$
h_{0123}
h_{4567}
h_{01}
h_{23}
h_{45}
h_{67}
hOTPs
h_0 h_1 h_2 h_3 h_4 h_5 h_6 h_7
b) $F_k(0)$ $F_k(1)$ $F_k(2)$ $F_k(3)$ $F_k(4)$ $F_k(5)$ $F_k(6)$ $F_k(7)$
c)
f)
Authenticator $\mathbb{A}$
Client $\mathbb{C}$
Private Key Wallet $\mathbb{W}$
Blockchain
Smart Contract $\mathbb{S}$
a) Generate seed k
k
$SK_{\mathbb{U}}, PK_{\mathbb{U}} := \Sigma$.KeyGen()
b) Generate OTPs
c) Store $h_0, h_1, \ldots$
d) Delete OTPs and k
e) Compute $\mathcal{R}$
{tx constructor($\mathcal{R}$)}
Add $PK_{\mathbb{U}}$ to tx
{tx constructor($\mathcal{R}$, $PK_{\mathbb{U}}$)}
{tx constructor($\mathcal{R}$, $PK_{\mathbb{U}}$)}
f) Store $\mathcal{R}$
Store $PK_{\mathbb{U}}$
event ContrDeployed($\mathbb{S}^{ID}$)
Local air-gapped transfer
Local connection transfer (e.g., USB, Bluetooth)
Remote transfer

Figure 1: Bootstrapping of SmartOTPs in a secure environment (Π_B^S).



OTPs
[k]
Authenticator $\mathbb{A}$
hOTPs
Client $\mathbb{C}$
[SK$_{\mathbb{U}}$]
Private Key Wallet $\mathbb{W}$
PK$_{\mathbb{U}}$
$\mathcal{R}$
Blockchain
Smart Contract $\mathbb{S}$
Initialization Stage
args := operation, param, address
{tx initOp(args)}
Confirm tx by $\mathbb{U}$
Σ.Sign(tx, $SK_{\mathbb{U}}$)
{tx initOp(args)}$_{\mathbb{U}}$
{tx initOp(args)}$_{\mathbb{U}}$
Σ.Verify(tx.σ, $PK_{\mathbb{U}}$)
Create O_{opID}
event InitOpEvent(opID)
opID
$OTP_{opID} := F_k(opID)$
OTP_{opID}
Create proof for OTP_{opID}
Confirmation Stage
Input from user
{tx confirmOp(OTP_{opID}, π_{opID}, opID)}
Local air-gapped transfer
Verify O_{opID} is pending
Verify OTP_{opID}
Local connection transfer (e.g., USB, Bluetooth)
Execute O_{opID}
Remote transfer
event ConfirmOpEvent(opID)

Figure 2: Execution of an operation (Π_O).

Then, $\mathbb{C}$ generates OTPs by computing $F_k(i) \mid i \in \{0, 1, \ldots, N-1\}$, where N is the number of leaves (equal to the number of OTPs in the base version). Next, $\mathbb{C}$ computes and stores the leaves of the tree – i.e., the hashes of the OTPs (i.e., $hOTPs$), which do not contain any confidential data.[1] After this step, k and the OTPs are deleted from $\mathbb{C}$, and $\mathbb{C}$ computes $\mathcal{R}$ from the stored hashes of the OTPs. Then, $\mathbb{C}$ creates a transaction containing constructor of $\mathbb{S}$ (see Algorithm 1) with $\mathcal{R}$ as the argument and passes it to $\mathbb{W}$ for appending $PK_{\mathbb{U}}$. Finally, $\mathbb{C}$ sends the transaction with the constructor to the blockchain where the deployment of $\mathbb{S}$ is made. In the constructor, $\mathcal{R}$ with $PK_{\mathbb{U}}$ are stored and ID of $\mathbb{S}$ (i.e., $\mathbb{S}^{ID}$) is assigned by a blockchain platform and returned in a response.[2] Storing $\mathcal{R}$ and $PK_{\mathbb{U}}$ binds an instance of $\mathbb{S}$ with the user's authenticator $\mathbb{A}$ and the user's private key wallet $\mathbb{W}$, respectively. In detail, $PK_{\mathbb{U}}$ enables $\mathbb{S}$ to verify whether an arbitrary transaction was signed by the user who created $\mathbb{S}$, while $\mathcal{R}$ enables the verification whether the given OTP was produced by the user's $\mathbb{A}$.

4.2.2 *Operation Execution*.

When the wallet framework is initialized, it is ready for executing operations by a two-stage protocol Π_O (see Figure 2 and Appendix A.5):

(1) **Initialization Stage**. When $\mathbb{U}$ decides to execute an operation with SmartOTPs, he enters the details of the operation into $\mathbb{C}$ 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, $\mathbb{C}$ sends this transaction to $\mathbb{W}$, which displays the details of the transaction and prompts $\mathbb{U}$ to confirm signing by a hardware button. Upon confirmation, $\mathbb{W}$ signs the transaction by $SK_{\mathbb{U}}$ and sends it back to $\mathbb{C}$. $\mathbb{C}$ forwards the transaction to $\mathbb{S}$. In the function *initOp()*, $\mathbb{S}$ verifies whether the signature was created by $\mathbb{U}$ (the first factor), stores the parameters of the operation, and then assigns a sequential ID (i.e., *opID*) to the initiated operation. In the response from $\mathbb{S}$, $\mathbb{C}$ is provided with an *opID*.

(2) **Confirmation Stage**. After the transaction (that initiated the operation) is persisted on the blockchain, $\mathbb{U}$ proceeds to the second stage of Π_O. $\mathbb{U}$ enters $opID$ to $\mathbb{A}$, which, in turn, computes and displays OTP_{opID} as $F_k(opID)$. Storing $hOTPs$ computed from OTPs at $\mathbb{C}$ enables $\mathbb{U}$ to transfer only the displayed OTP from $\mathbb{A}$ to $\mathbb{C}$, which can be accomplished in an air-gapped manner. Considering the mnemonic implementation [59], this means an air-gapped transfer of 12 words in the case of $O = 16$B. Then, $\mathbb{C}$ computes and appends the corresponding proof π_{opID} to the OTP. The proof of the OTP is computed from stored $hOTPs$ in the $\mathbb{C}$'s storage (or directly fetched from the storage if $\mathbb{C}$ stores all nodes of the Merkle tree). Next, $\mathbb{C}$ sends a transaction with OTP_{opID} and its proof π_{opID} to the blockchain, calling the function *confirmOp()* of $\mathbb{S}$, which handles the second factor. This function verifies the authenticity of the OTP (i.e., the first requirement of OTPs) and its association with the requested operation (i.e., the second requirement of OTPs), which together implies the correctness of the provided OTP.[3] In detail, upon calling the *confirmOp()* function with $opID$, OTP_{opID}, and π_{opID} as the arguments, $\mathbb{S}$ reconstructs the root hash from the provided arguments by the function *deriveRootHash()* that is presented in Appendix A.2.[4] If the reconstructed value matches the stored value $\mathcal{R}$, the operation is executed (e.g., crypto-tokens are transferred).

[1]To improve performance during provisioning of proofs, $\mathbb{C}$ might additionally store non-leaf nodes, increasing the requirement on $\mathbb{C}$'s storage 2x.
[2]Note that $\mathbb{S}^{ID}$ represents a public identification of $\mathbb{S}$, which serves as a destination for sending crypto-tokens to $\mathbb{S}$ by any party.

[3]Note that SmartOTPs meet the third requirement of OTPs by the design.
[4]Note that this algorithm contains, not yet described, improvements.

In the following, we present extensions of SmartOTPs, improving its efficiency and usability, and introducing new features.

4.3 Bootstrapping in an Insecure Environment

The main advantage of Π_B^S described above is its high usability, requiring only an air-gapped transfer of k and connected $\mathbb{W}$. However, Π_B^S is not resistant against $\mathcal{A}$ tampering with $\mathbb{C}$; $\mathcal{A}$ might intercept k or forge $\mathcal{R}$ for $\mathcal{R}'$. Similarly, $\mathcal{A}$ might forge $PK_{\mathbb{U}}$ for $PK_{\mathcal{A}}$, while staying unnoticeable for $\mathbb{U}$ who expects that $\mathbb{S}^{ID}$ obtained is correct. Therefore, we propose an alternative bootstrapping protocol Π_B^I (see Appendix A.5), assuming that $\mathcal{A}$ can tamper with $\mathbb{C}$ during bootstrapping. In this protocol, first we protect SmartOTPs from the interception of k and then from forging $\mathcal{R}$ and $PK_{\mathbb{U}}$.

To avoid the interception of k, instead of transferring k, $\mathbb{U}$ performs a transfer of all leaves of the Merkle tree (i.e., *hOTPs*) from $\mathbb{A}$ to $\mathbb{C}$, which can be achieved with a microSD card. Note that the leaves are hashes of OTPs, hence they do not contain any confidential data. Next, to protect SmartOTPs from forging of $PK_{\mathbb{U}}$ and $\mathcal{R}$, we require a deterministic computation of $\mathbb{S}^{ID}$ by a blockchain platform using $PK_{\mathbb{U}}$ and $\mathcal{R}$, hence $\mathbb{S}^{ID}$ can be computed and displayed together with $\mathcal{R}$ in $\mathbb{W}$ before the deployment of $\mathbb{S}$. In detail, $\mathbb{S}^{ID}$ is computed as $h(PK_{\mathbb{U}} \parallel \mathcal{R})$, thus each pair consisting of a public key and a root hash maps to the only $\mathbb{S}^{ID}$. However, even with this modification, $\mathcal{R}$ can still be forged by $\mathbb{C}$. Therefore, when transaction with the constructor is sent to $\mathbb{W}$, $\mathbb{U}$ has to compare $\mathcal{R}$ displayed at $\mathbb{W}$ with the one computed and displayed by $\mathbb{A}$. In the case of equality, $\mathbb{U}$ records $\mathbb{S}^{ID}$ displayed in $\mathbb{W}$.

4.4 Increasing the Number of OTPs

A small number of OTPs can have negative usability and security implications. First, users executing many transactions[5] would need to create new OTPs often, and thus change their addresses. Second, an attacker possessing $SK_{\mathbb{U}}$ can flood $\mathbb{S}$ with initialized operations, rendering all the OTPs unusable. Therefore, we need to increase the number of OTPs to make the attack unfeasible. However, increasing the number of OTPs linearly increases the amount of data that $\mathbb{C}$ needs to preserve in its storage. For example, if the number of OTPs is 2^{20}, then $\mathbb{C}$ has to store $33.6MB$ of data (considering $S = 16B$ and $\mathbb{C}$ storing all leaves), which is feasible even on storage-limited devices. However, e.g., for 2^{32} OTPs, $\mathbb{C}$ needs to store $137.4GB$ of data, which might be infeasible even on PCs, especially when $\mathbb{C}$ handles multiple instances of SmartOTPs.

To resolve this issue, we modify the base approach by applying a time-space trade-off [38] for OTPs. Namely, we introduce hash chains of which last items are aggregated by the Merkle tree. With such a construction, OTPs can be encoded as elements of chains and revealed layer by layer in the reverse order of creating the chains. This allows multiplication of the number of OTPs by the chain length without increasing the $\mathbb{C}$'s storage but imposing a larger number of hash computations on $\mathbb{S}$ and $\mathbb{A}$. Nonetheless, smart contract platforms set only a low execution cost for $h(.)$.

An illustration of this construction is presented in the bottom left part of Figure 3.[6] 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, which provides $\frac{N}{P}$ OTPs.[7] 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 (i.e., outputs of $F_k(.)$) has been reached (see the middle part of Figure 3). We emphasize that introducing hash chains may cause a violation of the requirement on the independence of OTPs if implemented incorrectly; i.e., OTPs from upper iteration layers can be derived from lower layers. Therefore, to enforce this requirement, we invalidate all the OTPs of all the previous iteration layers by a sliding window at $\mathbb{S}$.

Furthermore, if a hash chain were to use the same hash function throughout the entire chain, it would be vulnerable to birthday attacks [39]. To harden a hash chain against a birthday attack, a *domain separation* proposed by Leighton and Micali [49] can be used: a different hash function is applied in each step of a hash chain. Note that without domain separation, inverting the ith iterate of $h(.)$ is i times easier than inverting a single hash function (see the proof in [37]). Therefore, we use a different hash function for all but the last iteration layer $1 \leq i < P$ as follows:

$$h_{\mathcal{D}[i]}(x) \quad = \quad h(P - i + 1 \parallel x), \tag{1}$$

where x represents the OTP from the next iteration layer.

Although domain separation hardens a single hash chain against the birthday attack, this attack is still possible within the current iteration layer, which is an inevitable consequence of using multiple hash chains. Therefore, the number of leaves $\mathcal{L}$ (i.e., N/P) is the parameter that must be considered when quantifying the security level of our scheme (see Section 5).

With this improvement, $\mathbb{A}$ is updated to provide OTPs by

$$getOTP(i) \quad = \quad h_{\mathcal{D}}^{\alpha(i)}\Bigg(F_k\Big(\beta(i)\Big)\Bigg), \tag{2}$$

where i is the operation ID, $\alpha(i)$ determines the index in a hash chain, and $\beta(i)$ determines the index in the last iteration layer of OTPs. We provide concrete expressions for $\alpha(i)$ and $\beta(i)$ in Equation 4, which involves all proposed improvements and optimizations. A derivation of $\mathcal{R}$ from the OTP at $\mathbb{S}$ needs to be updated as well (see Algorithm 6 in Appendix). In detail, $\mathbb{S}$ executes $P - \alpha(i) - 1 = \left\lfloor \frac{iP}{N} \right\rfloor$ hash computations, which is a complementary number to the number of hash computations at $\mathbb{A}$ with regard to P. Also, $\mathbb{C}$ has to be modified, requiring computation of a proof to use the leaf index relative to the current iteration layer of OTPs (i.e., $i \ \% \ \frac{N}{P}$).

With this improvement, given the number of leaves equal to 2^{20} and $P = 2^{12}$, $\mathbb{C}$ stores only $33.6MB$ of data and it has 2^{32} OTPs available. On the other hand, this modification implies, on average, the execution of additional $P/2$ hash computations at $\mathbb{S}$, imposing additional costs. However, our experiments show the benefits of this approach (see Section 6.1).

4.5 Depletion of OTPs

Even with the previous modification, the number of OTPs remains bounded, therefore they may be depleted. We propose handling of depleted OTPs by a special operation that replaces the current tree with a new one. To introduce a new tree securely, we propose updating $\mathcal{R}$ value while using the last OTP of the current tree for

[5]E.g., several smart contracts in Ethereum have over 2^{20} transactions made.

[6]Note that this figure contains further, not yet described, improvements.

[7]For simplicity, we assume that $GCD(N, P) = P$.

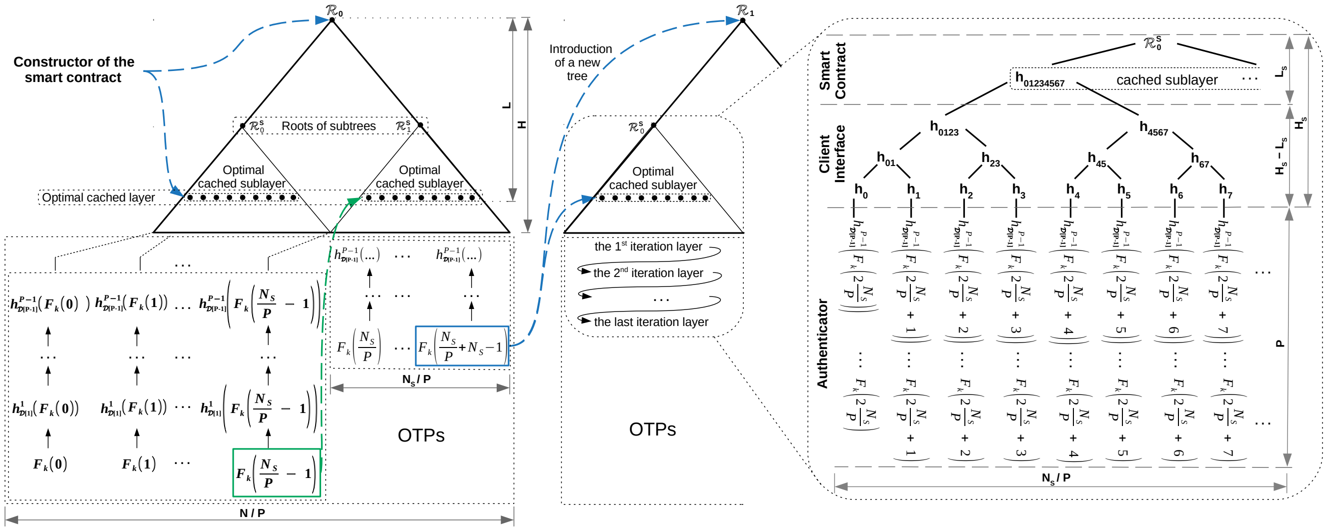


Figure 3: An overview of our approach and its improvements.

confirmation. Nevertheless, for this purpose we cannot use Π_O consisting of two stages, as $\mathcal{A}$ possessing $SK_{\mathbb{U}}$ could be "faster" than the user and might initialize the last operation and thus block all the user's funds. 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 protocol Π_{NR} that replaces $\mathcal{R}$ during three stages of interaction with the blockchain, which requires two append-only lists L_1 and L_2 (see Algorithm 2):

(1) $\mathbb{U}$ enters OTP_{N-1} to $\mathbb{C}$. $\mathbb{C}$ sends $h(OTP_{N-1} \parallel \mathcal{R}^{new})$ to $\mathbb{S}$, which appends it to L_1.
(2) $\mathbb{C}$ sends $\mathcal{R}^{new}$ to $\mathbb{S}$, which appends it to L_2.
(3) $\mathbb{C}$ passes OTP_{N-1} with π_{N-1} to $\mathbb{S}$, where the first matching entries of L_1 and L_2 are located to perform the introduction of $\mathcal{R}^{new}$. Finally, the lists are cleared for future updates.

Locating the first entries in the lists relies on the append-only feature of lists, hence no $\mathcal{A}$ can make the first valid pair of entries in the lists. Similarly as in Π_B, we propose two variants of Π_{NR} intended for secure (i.e., $\Pi_{NR}^{\mathcal{S}}$) and insecure environment (i.e.,

Algorithm 2: Introduction of a new $\mathcal{R}$ in $\mathbb{S}$

```
$L_1 \leftarrow []$;                                   ▷ Items have form $< h(\mathcal{R}^{new} \parallel OTP) >$
$L_2 \leftarrow []$;                                   ▷ Items have form $< \mathcal{R}^{new} >$
function 1_newRootHash(hRootAndOTP) public
    assert $\Sigma.verify(tx.\sigma, PK_{\mathbb{U}})$;
    assert $nextOpID \ \% \ N = N-1$;                  ▷ The last oper. of tree
    $L_1$.append(hRootAndOTP);
function 2_newRootHash($\mathcal{R}^{new}$) public
    assert $\Sigma.verify(tx.\sigma, PK_{\mathbb{U}})$;
    assert nextOpID $\% \ N = N-1$;                    ▷ The last oper. of tree
    $L_2$.append($\mathcal{R}^{new}$);
function 3_newRootHash(otp, $\pi$) public
    assert nextOpID $\% \ N = N-1$;                    ▷ The last oper. of tree
    verifyOTP(otp, $\pi$, nextOpID);
    if $L_1.len > LEN_{MAX} \mid L_2.len > LEN_{MAX}$ then
        $L_1, L_2 \leftarrow [], []$;
        return;                                        ▷ To avoid $\mathcal{A}$ DoS-ing $\mathbb{S}$ by gas depletion.
    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
                $\mathcal{R} \leftarrow L_2[i]$;
                $L_1, L_2 \leftarrow [], []$;
                nextOpID++;
```

$\Pi_{NR}^{\mathcal{I}}$). In $\Pi_{NR}^{\mathcal{I}}$ (see Appendix A.5), $\mathbb{A}$ must compute and display $h(OTP_{N-1} \parallel \mathcal{R}^{new})$ and $\mathcal{R}^{new}$ to enable protection against $\mathcal{A}$ that tampers with $\mathbb{C}$. Hence, $\mathbb{U}$ can verify the equality of items displayed at $\mathbb{W}$ with the ones displayed at $\mathbb{A}$ during the first and the second stage of $\Pi_{NR}^{\mathcal{I}}$, preventing $\mathcal{A}$ from forging the tree. To adapt this improvement at $\mathbb{C}$, $\mathbb{C}$ needs to store all nodes of the new tree. Therefore, $\mathbb{U}$ provides $\mathbb{C}$ with all nodes of the new tree, transferred from $\mathbb{A}$ on a microSD card. In the case of $\Pi_{NR}^{\mathcal{S}}$, the nodes of the new tree are transferred by a transcription of k from $\mathbb{A}$ to $\mathbb{C}$ and no values are displayed at $\mathbb{W}$ and $\mathbb{A}$ for $\mathbb{U}$'s verification.

4.6 Cost & Security Optimizations

4.6.1 ***Caching in the Smart Contract.*** With a high Merkle tree, the reconstruction of $\mathcal{R}$ from a leaf node may be costly. Although the number of hash computations stemming 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 $\mathbb{S}$ and do proof verifications against a cached layer. Hence, every call of *deriveRootHash()* will execute L fewer hash computations in contrast to the version that reconstructs $\mathcal{R}$, while $\mathbb{C}$ 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 a high amount of cached data on the blockchain is too expensive. Therefore, this cost optimization must be viewed as a trade-off between the depth L of the cached layer and the price required for the storage of such a cached layer on the blockchain (see Section 6.1).

We depict this modification in the left part of Figure 3, and we show that an optimal caching layer can be further partitioned into caching sublayers of subtrees (introduced later). To enable this optimization, the cached layer of the Merkle tree must be stored in the constructor of $\mathbb{S}$. From that moment, the cached layer replaces the functionality of $\mathcal{R}$, reducing the size of proofs. During the confirmation stage of Π_O, an OTP and its proof are used for the reconstruction of a particular node in the cached layer, instead of $\mathcal{R}$. Then the reconstructed value is compared with an expected node

Algorithm 3: Introduction of the next subtree at $\mathbb{S}$

```
currentSubLayer[];                                  ▷ Adjusted in the constructor
function nextSubtree(nextSubLayer, otp, $\pi_{otp}$, $\pi_{sr}$) public
    assert nextOpID % $N \neq N - 1$;                 ▷ Not the last op. of parent
    assert nextOpID % $N_S = N_S - 1$;                ▷ The last op. of subtree
    assert currentSubLayer.len = nextSubLayer.len;
    assert deriveRootHash(otp, $\pi_{otp}$, nextOpID) = $\mathcal{R}$;
    currentSubLayer ← nextSubLayer;
    $\mathcal{R}^s$ ← reduceMT(currentSubLayer, currentSubLayer.len);
    assert subtreeConsistency($\mathcal{R}^s$, $\pi_{sr}$, $\mathcal{R}$);
    nextOpID++;                     ▷ Accounts for this introduction of a subtree
```

of the cached layer. The index of an expected node is computed as

$$idxInCache(i) \quad = \quad \left\lfloor \left(i \;\%\; \frac{N}{P}\right) \;/\; 2^{H-L} \right\rfloor, \tag{3}$$

where i is the ID of an operation.

4.6.2 ***Partitioning to Subtrees.*** The caching of the optimal layer minimizes the operational costs of SmartOTPs, 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. On top of that, after revealing the first iteration layer of OTPs, the security of our scheme described so far is decreased by $log_2(N/P)$ bits due to the birthday attack (see Section 5) on OTPs. Hence, bigger trees suffer from higher security loss than smaller trees.

To overcome the prepayment issue and to mitigate the birthday attack, we propose partitioning an optimal cached layer to smaller groups having the same size, forming sublayers that belong to subtrees (see the left part of Figure 3). The obtained security loss is $log_2(N_S/P)$, $N_S \ll N$.

Starting with the deployment of $\mathbb{S}$, the cached sublayer of the first subtree and the "parent" root hash (i.e., $\mathcal{R}$) are passed to the constructor; the cached sublayer is stored on the blockchain and its consistency against $\mathcal{R}$ is verified. Then during the operational stage of Π_O, when confirmation of operation is performed, the passed OTP is verified against an expected node in the cached sublayer of the current subtree, saving costs for not doing verification against $\mathcal{R}$ (see Algorithm 5 in Appendix).

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). We propose a protocol Π_{ST} for the introduction of the next subtree (see Appendix A.5 for the detailed description). Namely, $\mathbb{C}$ introduces the next subtree in a single step by calling a function *nextSubtree()* of $\mathbb{S}$ with the arguments containing: (1) the last OTP of the current subtree $OTP_{(N_S-1)+\delta N_S}$, $\delta \in \{1, \ldots, N/N_S - 1\}$, (2) its proof π_{otp}, (3) the cached sublayer of the next subtree, and (4) the proof π_{sr} of the next subtree's root; all items but OTP are computed by $\mathbb{C}$. The pseudo-code of the next subtree introduction at $\mathbb{S}$ is shown in Algorithm 3. The current subtree's cached sublayer is replaced by the new one, which is verified by the function *subtreeConsistency*() against $\mathcal{R}$ with the use of the passed proof π_{sr} of the new subtree's root hash $\mathcal{R}^s$. Note that introducing a new subtree invalidates all initialized yet to be confirmed operations of the previous subtree.

At $\mathbb{A}$, this improvement requires accommodating the iteration over layers of hash chains in shorter periods. Hence, $\mathbb{A}$ provides OTPs by Equation 2 with the following expressions:

$$\begin{aligned} \alpha(i) &= \quad P - \left\lfloor \frac{(i \;\%\; N_S)P}{N_S} \right\rfloor - 1, \\ \beta(i) &= \quad \left\lfloor \frac{i}{N_S} \right\rfloor \frac{N_S}{P} + \left(i \;\%\; \frac{N_S}{P}\right), \end{aligned} \tag{4}$$

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 $\mathcal{R}$ as well as the constructor of $\mathbb{S}$ requires, additionally to Algorithm 2 and Algorithm 1, the introduction of a cached sublayer of the first subtree (omitted here for simplicity).

5 SECURITY ANALYSIS

We analyze the security of SmartOTPs and its resilience to attacker models under the assumption of random oracle model $\mathcal{RO}$.

5.1 Security of OTPs

OTPs in our scheme are related to two cryptographic constructs: a list of hash chains and the Merkle tree aggregating their last values. In this subsection, we assume an adversary $\mathcal{A}$ who is trying to invert OTPs, and we give a concrete expressions for security of our scheme. Since we employ the hash domain separation technique [49] for hash chains, each hash execution can be seen as an execution of an independent hash function. For such a construction, Kogan et al. give the following upper bound (see Theorem 4.6 in [43]) on the advantage of $\mathcal{A}$ breaking a chain:

$$Pr[\mathcal{A} \textit{ breaks a chain}] \leq \frac{2Q + 2P + 1}{2^S}, \tag{5}$$

where Q is the number of queries that $\mathcal{A}$ can make to $h(.)$, P is the chain length, and S is the bit-length of OTPs (and the output of $h(.)$). Kogan et al. [43] proved that inverting a hash chain hardened by the domain separation imposes a loss of security equal to the factor of 2. Therefore, to make a hardened hash chain as secure as λ-*bit* $\mathcal{RO}$, it is enough to set $S = \lambda + 2$. E.g., to achieve 128-bit security, S should be equal to 130.

SmartOTPs without Subtrees. This scheme (see Section 4.6) uses a Merkle tree that aggregates $\mathcal{L} = \frac{N}{P}$ hash chains, where the chains are created independently of each other; they have the same length and the same number of OTPs. $\mathcal{A}$ can win by inverting any of the chains; hence, the probability that this scheme is secure is

$$Pr[\textit{Scheme is secure}] = \left(1 - \frac{2Q + 2P + 1}{2^S}\right)^{\mathcal{L}}. \tag{6}$$

We can apply the alternative form of Bernoulli's inequality $(1 - x)^{\mathcal{L}} \geq 1 - x\mathcal{L}$, where $\mathcal{L} \geq 1$ and $0 \leq x \leq 1$ must hold. In our case, the input conditions hold since the number of hash chains is always greater than one and the probability that $\mathcal{A}$ breaks a single chain from Equation 5 fits the range of x (i.e., $0 \leq \frac{2Q+2P+1}{2^S} \leq 1$). Hence, we lower-bound the probability from Equation 6 as follows:

$$Pr[\textit{Scheme is secure}] \geq 1 - \frac{\mathcal{L}(2Q + 2P + 1)}{2^S}. \tag{7}$$

Corollary 5.1. *To make SmartOTPs without partitioning into subtrees as secure as λ-bit $\mathcal{RO}$, it is enough to set $S = \lambda + 2 + log_2(\mathcal{L})$.*

For example, to achieve 128-bit security with $\mathcal{L} = 64$ and $P \geq 1$, S should be equal to 136, and thus an OTP can be transferred by one QR code v1 or 13 mnemonic words.

Full SmartOTPs. The full SmartOTPs scheme contains partitioning into subtrees, in which all leaves of the next subtree "are visible" only after depleting OTPs of the current subtree (and using OTPs from the 1st iteration layer of the next subtree). This improves the security of our scheme under the assumption that $\mathbb{C}$'s storage is not compromised by $\mathcal{A}$, which is true for $\mathcal{A}$ that possesses $PK_{\mathbb{U}}$ or $\mathbb{A}$. Therefore, we replace $\mathcal{L}$ in Equation 7 for $\mathcal{L}_{\mathcal{S}} = \frac{N_S}{P}$, $N_S \ll N$.

Corollary 5.2. *To make the full scheme of SmartOTPs as secure as λ-bit $\mathcal{RO}$, it is enough to set $S = \lambda + 2 + log_2(\mathcal{L}_{\mathcal{S}})$.*

Therefore, to achieve 128-bit security with $\mathcal{L} = \frac{N}{N_S}\mathcal{L}_{\mathcal{S}}$, $\mathcal{L}_{\mathcal{S}} = 64$, and $P \geq 1$, S should be equal to 136, and thus an OTP can be transferred by a QR code v1 or 13 mnemonic words. To achieve the same security with $\mathcal{L}_{\mathcal{S}} = 1024$, we need to set $S = 140$, and thus an OTP can be transferred in a QR code v2 or 13 mnemonic words.

5.2 The Attacker Possessing $SK_{\mathbb{U}}$

Theorem 5.3. *$\mathcal{A}$ with access to $SK_{\mathbb{U}}$ is able to initiate operations by Π_O but is unable to confirm them.*

Justification. The security of Π_O is achieved by meeting all requirements on general OTPs (see Section 3.2). In detail, the requirement on the *independence* of two different OTPs is satisfied by the definition of $F_k(.) \equiv h(k \parallel .)$, where $h(.)$ 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 OTPs belonging to all previous iteration layers by a sliding window at $\mathbb{S}$ (see Section 4.4). The requirement on the *linkage* of each OTP_i with operation O_i is satisfied due to (1) $\mathcal{RO}$ used for instantiation of $h(.)$ and (2) by the definition of the Merkle tree, preserving the order of its aggregated leaves. By meeting these requirements, $\mathcal{A}$ is able to initiate an operation O_j in the first stage of Π_O but is unable to use an OTP_i intercepted in the second stage of Π_O 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 $\mathcal{R}$ associated with k at the constructor of $\mathbb{S}$. □

Theorem 5.4. *Assuming $\delta \in \{0, \ldots, \frac{N}{N_S} - 2\}$, $\mathcal{A}$ with access to $SK_{\mathbb{U}}$ is unable to deplete all OTPs or misuse a stolen OTP that introduces the $(\delta + 1)$th subtree by Π_{ST}.*

Justification. When all but one OTPs of the δth subtree are depleted, the last remaining operation $O_{(N_S-1)+\delta N_S}$, $\delta \in \{0, \ldots, \frac{N}{N_S} - 2\}$ is enforced by $\mathbb{S}$ to be the introduction of the next subtree. This operation is executed in a single transaction calling the function *nextSubtree*() of $\mathbb{S}$ (see Algorithm 3) requiring the corresponding $OTP_{(N_S-1)+\delta N_S}$ that is under control of $\mathbb{U}$; hence $\mathcal{A}$ cannot execute the function to proceed with a further depletion of OTPs in the $(\delta + 1)th$ subtree. If $\mathcal{A}$ were to intercept $OTP_{(N_S-1)+\delta N_S}$ during the execution of Π_{ST} by $\mathbb{U}$, he could use the intercepted OTP only for the introduction of the next valid subtree since the function *nextSubtree*() also checks a valid cached sublayer of the $(\delta + 1)$th subtree against the parent root hash $\mathcal{R}$. □

Theorem 5.5. *Assuming $\delta = \frac{N}{N_S} - 1$, $\mathcal{A}$ with access to $SK_{\mathbb{U}}$ is neither able to deplete all OTPs nor introduce a new parent tree nor render SmartOTPs 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 $\mathbb{S}$ against any paramount field. If we were to use Π_O while constraining to the last initialized operation $O_{(N-1)+\eta N}$, $\eta \in \{0, 1, \ldots\}$ of the parent tree, then $\mathcal{A}$ could render SmartOTPs unusable by submitting an arbitrary $\mathcal{R}$ in *initOp*(), thus blocking 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 protocol Π_O, using two unlimited append-only lists L_1 and L_2 that are manipulated in three stages of interaction with the blockchain (see Section 4.5). In the first stage, $h(\mathcal{R}^{new} \parallel OTP_{(N-1)+\eta N})$ is appended to L_1, hence $\mathcal{A}$ cannot extract the value of OTP. In the second stage, $\mathcal{R}^{new}$ 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 $\mathcal{A}$ 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 $\mathbb{S}$, the user's entries will match as the first ones. □

5.3 The Attacker that Tampers with the Client

Theorem 5.6. *If $\mathbb{C}$ is tampered with after Π_B, $\mathbb{U}$ can detect such a situation and prevent any malicious operation from being initialized.*

Justification. If we were to assume that $\mathbb{W}$ is implemented as a software wallet (or hardware wallet without a display), then $\mathcal{A}$ tampering with $\mathbb{C}$ might also tamper with the $\mathbb{W}$'s software running on the same machine. This would in turn enable a malicious operation to be initialized and further confirmed by $\mathbb{U}$, since $\mathbb{U}$ would be presented with a legitimate data in $\mathbb{C}$ and $\mathbb{W}$, while the transactions would contain malicious data. Therefore, we require that $\mathbb{W}$ is implemented as a hardware wallet with a display, which exposes only signing capabilities, while $SK_{\mathbb{U}}$ never leaves the device (e.g., [11, 31, 42, 73]). Due to it, $\mathbb{U}$ can verify the details of a transaction being signed in $\mathbb{W}$ and confirm signing only if the details match the information shown in $\mathbb{C}$ (for Π_O) or $\mathbb{A}$ (for $\Pi_{NR}^{\mathcal{I}}$). We refer the reader to the work of Arapinis et al. [5] for the security analysis of hardware wallets with displays. □

Theorem 5.7. *If $\mathbb{C}$ is tampered with during an execution of $\Pi_B^{\mathcal{I}}$, $\mathcal{A}$ can neither intercept k nor forge $\mathcal{R}$ nor forge $PK_{\mathbb{U}}$.*

Justification. When the protocol $\Pi_B^{\mathcal{I}}$ is used, instead of an air-gapped transfer of k from $\mathbb{A}$ to $\mathbb{C}$, $\mathbb{U}$ transfers leaves of the Merkle tree by microSD card. The leaves represent hashes of OTPs in the base version or the hashes of the last items of hash chains in the full version of SmartOTPs. In both versions, the transferred data do not contain any secrets, hence $\mathcal{A}$ cannot take advantage of intercepting them. The next option that $\mathcal{A}$ may seek for is to forge $\mathcal{R}$ for $\mathcal{R}'$ and $PK_{\mathbb{U}}$ for $PK_{\mathcal{A}}$, which results in different $\mathbb{S}^{ID}$ than in the case of $\mathcal{R}$ and $PK_{\mathbb{U}}$, since $\mathbb{S}^{ID}$ is computed as $h(PK_{\mathbb{U}} \parallel \mathcal{R})$. While $PK_{\mathbb{U}}$ is stored at $\mathbb{W}$, the authenticity of $\mathcal{R}$ needs to be verified by $\mathbb{U}$ who compares displays of $\mathbb{A}$ and $\mathbb{W}$. Only in the case of equality, $\mathbb{U}$ knows that $\mathbb{S}^{ID}$ displayed in $\mathbb{W}$ maps to legitimate $PK_{\mathbb{U}}$ and $\mathcal{R}$. □

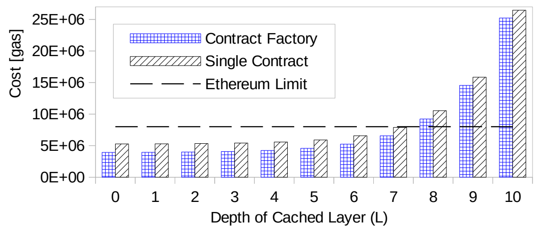


Figure 4: Deployment costs ($H = H_S$).

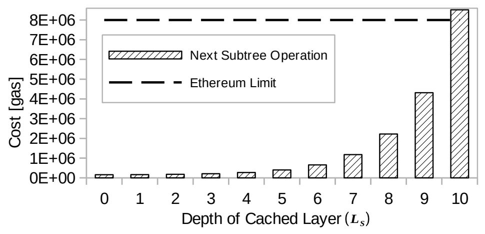


Figure 5: Cost of introducing the next subtree ($H = 20$, $H_S = 10$).

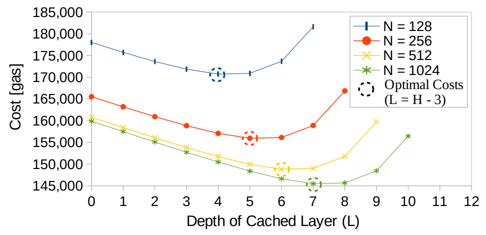


Figure 6: Average total cost per transfer ($H = H_S$).

5.4 The Attacker Possessing the Authenticator

It is trivial to see that $\mathcal{A}$ with access to $\mathbb{A}$ is unable to initialize any operation with SmartOTPs since he does not hold $PK_{\mathbb{U}}$.

5.5 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 [33], HoneyBadgerBFT [52], or StrongChain [71]. In 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 $SK_{\mathbb{U}}$ 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., ~3 minutes). Note that such waiting can be done as a background task of $\mathbb{C}$, hence $\mathbb{U}$ does not have to wait: (1) considering that $\mathcal{A}$ possesses $SK_{\mathbb{U}}$, $\mathbb{C}$ can detect such a fork during the wait and resubmit the *initOp*() transaction, (2) in the case of $\mathcal{A}$ tampering with $\mathbb{C}$, no operation can be initialized since $\mathbb{U}$ never signs $\mathcal{A}$'s transaction (due to the hardware wallet), and (3) $\mathcal{A}$ possessing $\mathbb{A}$ cannot initialize any operation as well.

Attacks with a Post Quantum Computer. Although a resilience to quantum computing (QC) is not the focus of this paper, it is of worthy 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 $h(.)$. A generic QC attack against $h(.)$ is Grover's algorithm [35], providing a quadratic speedup in searching for the input of the black box function. As indicated by Amy et al. [4], using this algorithm under realistic assumptions, the security of SHA-3 is reduced from 256 to 166 bits. Applying these results to OTPs with 128-bit security from examples in Section 5.1, we obtain 98-bits post-QC security. Further, when assuming the example with $\mathcal{L} = 64$ from Section 5.1 and [4], to achieve 128-bits of post-QC security, we estimate the length of OTPs to 205-bits.

6 REALIZATION IN PRACTICE

We have selected the Ethereum platform and the Solidity language for the implementation of $\mathbb{S}$, HTML/JS for DAPP of $\mathbb{C}$, Java for smartphone App of $\mathbb{A}$, and Trezor T&One [73] for $\mathbb{W}$. We selected $S = 128$ bits, which has practical advantages for an air-gapped $\mathbb{A}$, producing OTPs that are 12 mnemonic words long or a QR code v1 (with a capacity of 17B). Next, we used SHA-3 with truncated output to 128 bits as $h(.)$. We selected the size of k equal to 128 bits, fitting 12 mnemonic words $\simeq$ 1 QR code v1.

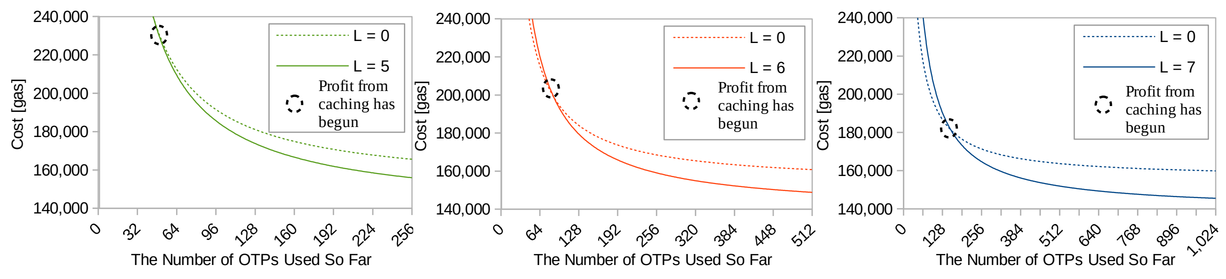
So far, we have considered only the crypto-token transfer operation. However, our proposed protocol enables us to extend the set of operations. For demonstration purposes, we extended the operation set by supporting daily limits and last resort information (see Appendix A.3). We also tested our contracts by static/dynamic analysis tools Mythril [23], Slither [70], and ContractGuard [36]; none of them detected any vulnerabilities. In addition, we made a hardware implementation of $\mathbb{A}$ using NodeMCU [58] equipped with ESP8266 (see Appendix A.6). The source code of our implementation and videos are available at https://github.com/ivan-homoliak-sutd/SmartOTPs.

6.1 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 bit-length S for $h(.)$ as well as for OTPs. S significantly influences the gas consumption for storing the cached layer on the blockchain. We remark that measured costs can also be influenced by EVM internals (e.g., 32B-long words/alignment).

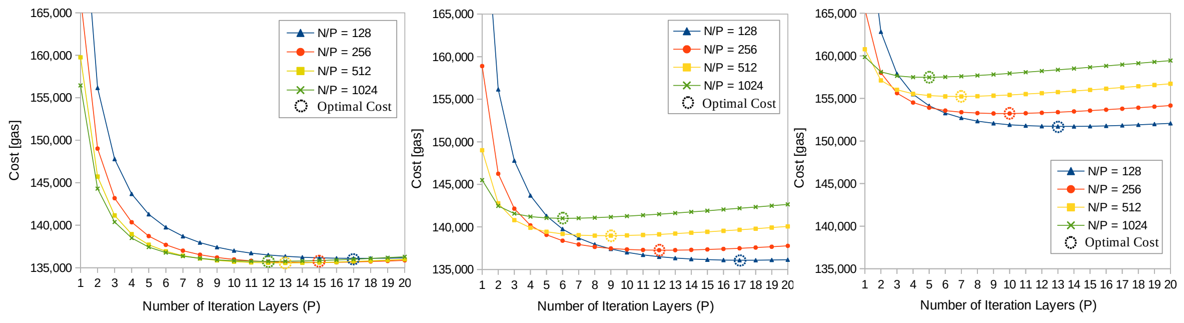
6.1.1 *Costs Related to the Merkle Tree.*

Deployment Cost. The cost of a smart contract deployment is driven mainly by the L_S (related to the first subtree) and S. A less significant factor is the consistency check of a Merkle tree, which is driven by L_S: the higher L_S is, more layers have to be reduced. Similarly, the greater $H - H_S$ is, more steps have to be done in the 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 the depth L ($\equiv L_S$) of the cached layer are presented in Figure 4. The figure depicts two cases: one uses a single $\mathbb{S}$ and the second assumes a contract factory producing instances of $\mathbb{S}$. Thanks to the contract



(a) $H = 8$ **(b)** $H = 9$ **(c)** $H = 10$

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



(a) $L = H$ **(b)** $L = 7$ **(c)** $L = 0$

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

factory, we managed to save a constant amount of gas equal to $\sim 1.3M$, regardless of L_S. Since we assume $8M$ as the maximum gas limit at the Ethereum main network, we can build a caching layer with $L_S = 7$ at maximum. Later, we will see that 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 Π_O is similar, here we selected the transfer of crypto-tokens O^t, and we measured the total cost of O^t as follows:

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

where *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 on 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 cost, we measured the average cost of a transaction (i.e., $\overline{cost}(O^t(L, N, P))$ for both stages of Π_O; note that the cost of $initOp() \simeq 70k$ of gas in all operations. For completeness, we present the transaction costs of all proposed operations in Appendix A.4. In Figure 6, 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 $cost(O^d(L))$ and $\overline{cost}(O^t(L, N, P))$. We see that $L = H - 3$ for such an optimal point. 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, respectively. Next, we explored the number of transfer operations to be executed until a profit of the caching has begun (see Figure 7). We computed a rolling average cost per O^t, while distinguishing between the optimal caching layer and disabled caching – the 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 5). We found out that when subtrees (and their cached sublayers) are introduced within a dedicated operation, it is significantly cheaper compared to the introduction of a subtree during the deployment.

*6.1.2 **Costs Related to Hash Chains.***

Since each iteration layer of hash chains contributes to an average cost of $confirmOp()$ 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 $H = H_S$, 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 6), we achieved savings of 27.80%, 19.61%, 14.95%, and 12.51% for trees with H equal to 7, 8, 9, and 10, respectively. For 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 the proof verification in contrast to bigger trees; these operations consume substantially more gas than operations related to hash chains. Although 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. Hence, one may set this parameter even at higher values, depending on the use case.

6.1.3 ***Costs in Fiat Money.*** We assume the average exchange rate of ETH/USD equal to 211 and the "standard" gas price 5 GWEI as of May 2, 2020. For example, in the case of $N = 2^{25}$ (i.e., $H = 20$, $H_S = 10$, $P = 2^5$, $L_S = 7$), expenses per transfer operation are \$0.2, while expenses for deployment and introduction of a new subtree are \$6.90 and \$1.23, respectively.

7 RELATED WORK

In this section, we compare SmartOTPs with other hash-based approaches and other smart-contract wallets.

Hash-Based Approaches. Although Merkle signatures [51] utilize Merkle trees for aggregation of several one-time verification keys (e.g., [46]), the size of these keys and signatures is substantially larger than the size of OTPs in SmartOTPs. Even further optimization of the signature size (i.e., Winternitz OTS [28]) does not make signatures as short as in SmartOTPs. Next, we highlight that we utilize hash chains for multiplication of OTPs, which is different than their application in Winternitz OTS [28] that utilize them for the purpose of reducing the size of a single Lamport-Diffie OTS [46] by encoding multiple bits of a message digest into the number of recurrent hash computations. The next related schemes are Lamport's hash chain [47] and its modification T/Key [43] that applies the domain separation. However, since they contain only a single chain, they are not secure in the setting of the public blockchain (see Section 3.2) in contrast to SmartOTPs that never consecutively iterate OTPs within a single hash chain. Moreover, T/Key [43] is using OTPs expiring in 30s to mitigate phishing attacks, which are unrelated in our case. TESLA [63, 64] is another related scheme that utilizes a single hash-chain in a centralized setting of time-based multi-cast authentication of streamed messages.

Smart Contract Wallets. An example of the 2-of-3 multi-signature approach that only supports Trezor wallets is *TrezorMultisig2of3* [75]. A disadvantage of this solution is that $\mathbb{U}$ has to own three Trezor devices, which might be an expensive solution. The n-of-m multi-signature scheme is provided by *Gnosis Wallet* [22], which currently holds a significant amount of Ether across various smart contracts. Similar to the previous example, a disadvantage of this wallet is that $\mathbb{U}$ has to own two hardware wallets for 2FA.

The main reason why existing smart contract wallets using asymmetric cryptography are not suitable for an air-gapped authentication is due to the signature size of 64B. Hence, to input OTP, $\mathbb{U}$ has to transcribe 48 mnemonic words in the case of lacking a camera on $\mathbb{C}$, which would take ~4x longer than in the case of SmartOTPs. When $\mathbb{C}$ is equipped with a camera, $\mathbb{A}$ implemented as an embedded device might not be capable of displaying a single OTP as a small QR code since the minimal required QR code having enough data capacity is v4. Therefore, several QR codes of a lower version would be needed, which introduces additional complexity for $\mathbb{U}$.

Another drawback of asymmetric cryptography (used in these wallets) stems from its resource demands that increase the operational costs, both on $\mathbb{S}$ and $\mathbb{W}$: (1) smart contract platforms place a high execution cost for asymmetric cryptography, and (2) $\mathbb{W}$ requires more advanced MCU for cryptographic computations, while $\mathbb{A}$ from SmartOTPs requires only a secure hash function. Based on the latter, we believe that hardware realization of $\mathbb{A}$ (see Appendix A.6) in SmartOTPs is less expensive than the second hardware wallet used in multi-signature smart contracts. Moreover, we note that if SmartOTPs were to use only mnemonic words and omit QR codes, then hardware requirements of $\mathbb{A}$ (and thus the overall cost) would be even lower – mnemonic words can be displayed even on a smart-card-embedded display, such as in CoolBitX [24].

8 DISCUSSION

Vulnerability in HW Wallets. We found out that two used hardware wallets do not display all data of transactions being signed: Trezor One displays first 24B of data and Trezor T displays 35B. With regard to Ethereum transactions, this means that used wallets display only the first 8B and 19B of data representing the parameters of a contract call. Hence, $\mathcal{A}$ that tampers with $\mathbb{C}$ might purposely preserve user expected values in the displayed data while forging data that are not displayed. We reported this vulnerability to the vendor, and as a mitigation, we put the most critical parameter (i.e., address) of all concerning functions at the first displayed position.

Usability. Our approach inherits the common usability characteristics of 2FA schemes, such as an extra device to carry,[8] effort for securely storing the recovery phrase k, effort for recalling/entering passwords, and effort for a transfer of OTPs, which can be made by scanning a QR code or transcription of mnemonic words. Note that these usability implications are almost the same as in the case of existing smart contract wallets with 2FA [22, 75]. In addition to the previous, SmartOTPs requires $\mathbb{U}$ 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 6.1.3, $\mathbb{U}$ has to introduce the next subtree after using $\sim 32K$ OTPs, while $\sim 33.5M$ OTPs are available to use before re-initialization of the parent tree. Next, we note that entering *opID* into $\mathbb{A}$ might be seen as a usability limitation, especially when N is large. However, *opID* can be reset after each iteration layer of the current subtree, thus fitting a small range (i.e., $\langle 1, \frac{N_S}{P} \rangle$).

To compare SmartOTPs with Gnosis Wallet [22], we counted the number of elementary actions (i.e., clicks, button presses, inputs of form fields, QR code scanning) required to make a transfer of funds. In the result, SmartOTPs required 33 actions while Gnosis wallet required 39 actions.

Costs. With consumption of up to $\sim 150k$ gas units per operation, our approach is comparable to equivalent 2FA solutions using smart contracts: Gnosis Wallet [22] requires $\sim 275k$ gas units[9] and TrezorMultisig2of3 [75] requires $\sim 95k$ gas units[10] per operation.

Lost Secrets. When $\mathbb{U}$ loses access to $\mathbb{A}$, he can initialize a new instance of $\mathbb{A}$ from the backup of seed k. Moreover, if $\mathbb{U}$ losses access to $\mathbb{A}$ and $\mathbb{W}$ at the same time, he can still recover the funds with the last resort functionality that we implemented (see Section 6).

[8] Assuming that the user already has a hardware wallet (the first factor).
[9] https://etherscan.io/tx/0xdb6e938... and https://etherscan.io/tx/0x328a7cc...
[10] https://etherscan.io/tx/0xfc7bbdd... (2 signatures in a single transaction).

State at the Client. The only state $\mathbb{C}$ has to store is the cache of hashes of OTPs from the first iteration layer. This might be seen as a limitation when $\mathbb{U}$ changes a client device. However, the state can be recovered anytime from the seed k or transferred by microSD card from $\mathbb{A}$ to $\mathbb{C}$ (see Section 4.2.1 and Section 4.3).

Transaction Size. Although the base version of SmartOTPs might slightly bloat the transaction due to H items of the proof, this is improved with the caching at $\mathbb{S}$, which reduces the number of items in the proof to H - L. For example, in the case of $H = 10$ and optimal caching (see Section 6.1.1) where $L = 7$, only three items of the proof are required. In this case, SmartOTPs consume 68B of transaction data (assuming 4B for operation ID), which is similar to asymmetric cryptography used in most of the blockchains.

9 CONCLUSION

In this paper, we have proposed SmartOTPs, a smart-contract wallet framework that provides a secure and usable method of managing crypto-tokens. The framework provides 2FA that is executed in two stages of interaction with the blockchain and protects against the attacker possessing a user's private key or a user's authenticator or the attacker that tampers with the client. Our framework uses OTPs constructed using a pseudo-random function, Merkle trees, and hash chains. We combine these primitives in a novel way, which enables an air-gapped setting using transcription of mnemonic words or scanning of small QR codes. Our protocol is general and can be utilized, besides the wallets, in any smart contract application for the purpose of 2FA. The provided smart contract is self-contained but its operation set can be extended by the community.

ACKNOWLEDGMENT

This work was supported by the IT4Innovations Excellence in Science project LQ1602, H2020 ECSEL project VALU3S (876852), and the internal project of Brno University of Technology (FIT-S-20-6427). This research was partially supported by ST Electronics and National Research Foundation (NRF), Prime Minister's Office Singapore, under Corporate Laboratory @ University Scheme (Programme Title: STEE Infosec – SUTD Corporate Laboratory).

REFERENCES

[1] 2015. Cryptocurrency-Stealing Malware Landscape. (2015). http://www.opensource.im/cryptocurrency/cryptocurrency-stealing-malware-landscape-dell-secureworks.php
[2] Rachel Abrams and Nathaniel Popper. 2014. Trading Site Failure Stirs Ire and Hope for Bitcoin. (2014). https://dealbook.nytimes.com/2014/02/25/trading-site-failure-stirs-ire-and-hope-for-bitcoin/
[3] Fadi Aloul, Syed Zahidi, and Wassim El-Hajj. 2009. Two factor authentication using mobile phones. In *Computer Systems and Applications, 2009. IEEE/ACS International Conference on.* IEEE, 641–644.
[4] Matthew Amy, Olivia Di Matteo, Vlad Gheorghiu, Michele Mosca, Alex Parent, and John Schanck. 2016. Estimating the cost of generic quantum pre-image attacks on SHA-2 and SHA-3. In *International Conference on Selected Areas in Cryptography.* Springer, 317–337.
[5] Myrto Arapinis, Andriana Gkaniatsou, Dimitris Karakostas, and Aggelos Kiayias. 2019. A Formal Treatment of Hardware Wallets. In *Financial Cryptography.* Springer.
[6] Armory Technologies, Inc. 2016. Bitcoin Armory. (2016). https://www.bitcoinarmory.com
[7] Daniel J Bernstein. 2009. Introduction to post-quantum cryptography. In *Post-quantum cryptography.* Springer, 1–14.
[8] Daniel J Bernstein, Niels Duif, Tanja Lange, Peter Schwabe, and Bo-Yin Yang. 2012. High-speed high-security signatures. *Journal of Cryptographic Engineering* 2, 2 (2012), 77–89.
[9] Binance. 2019. Binance Security Breach Update. (2019). https://binance.zendesk.com/hc/en-us/articles/360028031711-Binance-Security-Breach-Update
[10] Binance.com. 2020. Binance. (2020). https://www.binance.com/
[11] BitLox. 2019. BitLox wallet. (2019). https://www.bitlox.com
[12] Dan Boneh, Ben Lynn, and Hovav Shacham. 2001. Short signatures from the Weil pairing. In *International Conference on the Theory and Application of Cryptology and Information Security.* Springer, 514–532.
[13] Dan Boneh, Riad S. Wahby, Sergey Gorbunov, Hoeteck Wee, and Zhenfei Zhang. 2019. RFC Internet-Draft: BLS signature. (2019). https://tools.ietf.org/html/draft-irtf-cfrg-bls-signature-00
[14] Joseph Bonneau, Andrew Miller, Jeremy Clark, Arvind Narayanan, Joshua A Kroll, and Edward W Felten. 2015. Sok: Research perspectives and challenges for bitcoin and cryptocurrencies. In *S&P.* IEEE, 104–121.
[15] Thanh Bui, Siddharth Prakash Rao, Markku Antikainen, Viswanathan Manihatty Bojan, and Tuomas Aura. 2018. Man-in-the-machine: exploiting ill-secured communication inside the computer. In *27th USENIX Security Symposium (USENIX Security 18).* USENIX Association, 1511–1525.
[16] Jean-Pierre Buntinx. 2016. Brain Wallets Are Not Secure and 'No One Should Use Them,' Says Study. (2016). https://news.bitcoin.com/brain-wallets-not-secure-no-one-use-says-study/
[17] CarbonWallet.com. 2019. Multi Signature Online Cryptocurrency Wallet. (2019). https://carbonwallet.com/
[18] Vincent Chia, Pieter Hartel, Qingze Hum, Sebastian Ma, Georgios Piliouras, Daniel Reijsbergen, Mark van Staalduinen, and Pawel Szalachowski. 2018. Rethinking Blockchain Security: Position Paper. In *Blockchain.*
[19] Citowise Developments. 2019. Citowise wallet. (2019). https://citowise.com/wallet
[20] coinbase. 2020. Coinbase. (2020). https://www.coinbase.com/
[21] Coinomi Ltd. 2019. Coinomi Wallet. (2019). https://coinomi.com/
[22] ConsenSys. 2019. Gnosis Wallet. (2019). https://github.com/Gnosis/MultiSigWallet
[23] ConsenSys. 2019. Mythril. (2019). https://github.com/ConsenSys/mythril
[24] CoolBitX. 2019. The CoolWallet S. (2019). https://coolwallet.io/
[25] Copay. 2019. The Secure, Shared Bitcoin Wallet. (2019). https://copay.io/
[26] Nicolas Courtois, Guangyan Song, and Ryan Castellucci. 2016. Speed optimizations in Bitcoin key recovery attacks. *Tatra Mountains Mathematical Publications* 67(1) (2016), 55–68.
[27] Vivek Dhakal, Anna Maria Feit, Per Ola Kristensson, and Antti Oulasvirta. 2018. Observations on Typing from 136 Million Keystrokes. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems.* ACM, 646–658.
[28] Chris Dods, Nigel P Smart, and Martijn Stam. 2005. Hash based digital signature schemes. In *IMA International Conference on Cryptography and Coding.* Springer, 96–115.
[29] Donjon Team. 2019. Extracting seed from Ellipal wallet. (2019). https://donjon.ledger.com/Ellipal-Security/
[30] Electrum Technologies GmbH. 2019. Electrum Bitcoin wallet. (2019). https://electrum.org/
[31] ELLIPAL. 2019. ELLIPAL Hardware Wallet 2.0. (2019). https://www.ellipal.com/
[32] Shayan Eskandari, Jeremy Clark, David Barrera, and Elizabeth Stobert. 2018. A first look at the usability of bitcoin key management. *preprint arXiv:1802.04351* (2018).
[33] Yossi Gilad, Rotem Hemo, Silvio Micali, Georgios Vlachos, and Nickolai Zeldovich. 2017. Algorand: Scaling byzantine agreements for cryptocurrencies. In *SOSP.*
[34] Steven Goldfeder, Rosario Gennaro, Harry Kalodner, Joseph Bonneau, Joshua A Kroll, Edward W Felten, and Arvind Narayanan. 2015. Securing Bitcoin wallets via a new DSA/ECDSA threshold signature scheme. (2015).
[35] Lov K Grover. 1996. A fast quantum mechanical algorithm for database search. In *STOC.* ACM, 212–219.
[36] GuardStrike. 2019. ContractGuard. (2019). https://contract.guardstrike.com/
[37] Johan Håstad and Mats Näslund. 2001. Practical construction and analysis of pseudo-randomness primitives. In *ASIACRYPT.* Springer, 442–459.
[38] Martin Hellman. 1980. A cryptanalytic time-memory trade-off. *IEEE transactions on Information Theory* 26, 4 (1980), 401–406.
[39] Yih-Chun Hu, Markus Jakobsson, and Adrian Perrig. 2005. Efficient constructions for one-way hash chains. In *International Conference on Applied Cryptography and Network Security.* Springer, 423–441.
[40] Infinity Blockchain Labs Europe. 2019. Infinito wallet. (2019). https://www.infinitowallet.io/
[41] Don Johnson, Alfred Menezes, and Scott Vanstone. 2001. The elliptic curve digital signature algorithm (ECDSA). *International journal of information security* 1, 1 (2001), 36–63.
[42] KeepKey. 2019. The Simple Cryptocurrency Hardware Wallet. (2019). https://www.keepkey.com/
[43] Dmitry Kogan, Nathan Manohar, and Dan Boneh. 2017. T/key: second-factor authentication from secure hash chains. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security.* ACM, 983–999.
[44] Kraken. 2019. Inside Kraken Security Labs: Flaw Found in Keepkey Crypto Hardware Wallet. (2019). https://blog.kraken.com/post/3245/

flaw-found-in-keepkey-crypto-hardware-wallet/
[45] Kraken. 2020. Kraken Identifies Critical Flaw in Trezor Hardware Wallets. (2020). https://blog.kraken.com/post/3662/kraken-identifies-critical-flaw-in-trezor-hardware-wallets/
[46] Leslie Lamport. 1979. *Constructing digital signatures from a one-way function.* Technical Report. Technical Report CSL-98, SRI International Palo Alto.
[47] Leslie Lamport. 1981. Password authentication with insecure communication. *Commun. ACM* 24, 11 (1981), 770–772.
[48] Ledger. 2018. Ledger Nano. (2018). https://www.ledgerwallet.com/products/1-ledger-nano
[49] Frank T Leighton and Silvio Micali. 1995. Large provably fast and secure digital signature schemes based on secure hash functions. (1995). US Patent 5,432,852.
[50] Luno. 2019. Luno wallet. (2019). https://www.luno.com/wallet/
[51] Ralph C Merkle. 1989. A certified digital signature. In *Conference on the Theory and Application of Cryptology.* Springer, 218–238.
[52] Andrew Miller, Yu Xia, Kyle Croman, Elaine Shi, and Dawn Song. 2016. The honey badger of BFT protocols. In *ACM CCS.*
[53] Tyler Moore and Nicolas Christin. 2013. Beware the middleman: Empirical analysis of Bitcoin-exchange risk. In *International Conference on Financial Cryptography and Data Security.* Springer, 25–33.
[54] D M'raihi, M Bellare, F Hoornaert, D Naccache, and O Ranen. 2005. *HOTP: An HMAC-based one-time password algorithm.* Technical Report.
[55] David M'Raihi, Salah Machani, Mingliang Pei, and Johan Rydell. 2011. *Totp: Time-based one-time password algorithm.* Technical Report.
[56] Mycelium Holding LTD. 2019. Mycelium Entropy. (2019). https://mycelium.com/mycelium-entropy.html
[57] Mycelium LTD. 2019. Mycelium wallet. (2019). https://wallet.mycelium.com/
[58] NodeMcu Team. 2018. NodeMCU. (2018). https://nodemcu.readthedocs.io/en/master/
[59] Marek Palatinus, Pavol Rusnak, Aaron Voisine, and Sean Bowe. 2013. BIP-39. (2013). https://github.com/bitcoin/bips/blob/master/bip-0039.mediawiki
[60] Parity Technologies. 2017. The Multi-sig Hack: A Postmortem. (2017). https://paritytech.io/the-multi-sig-hack-a-postmortem/
[61] Paxful, Inc. 2020. Paxful. (2020). https://paxful.com/wallet
[62] Payward, Inc. 2020. Kraken. (2020). https://www.kraken.com/
[63] Adrian Perrig, Ran Canetti, J Doug Tygar, and Dawn Song. 2000. Efficient authentication and signing of multicast streams over lossy channels. In *Proceeding 2000 IEEE Symposium on Security and Privacy. S&P 2000.* IEEE, 56–73.
[64] A Perrig, D Song, R Canetti, JD Tygar, and B Briscoe. 2005. RFC4082: Timed Efficient Stream Loss-Tolerant Authentication (TESLA): Multicast Source Authentication Transform Introduction. *Request for Comments. IETF* (2005).
[65] Antony Peyton. 2017. Cyren sounds siren over Bitcoin siphon scam. (2017). https://www.bankingtech.com/2017/01/cyren-sounds-siren-over-bitcoin-siphon-scam/
[66] Polo Digital Assets, Ltd. 2020. Poloniex. (2020). https://poloniex.com/
[67] QRStuff.com. 2011. What Size Should A Printed QR Code Be? (2011). https://blog.qrstuff.com/2011/01/18/what-size-should-a-qr-code-be
[68] Reuters. 2016. Bitcoin Worth $72M Was Stolen in Bitfinex Exchange Hack in Hong Kong. (2016). http://fortune.com/2016/08/03/bitcoin-stolen-bitfinex-hack-hong-kong/
[69] Bruce Schneier. 2005. Two-factor authentication: too little, too late. *Commun. ACM* 48, 4 (2005), 136.
[70] Slither Team. 2019. Slither. (2019). https://github.com/crytic/slither
[71] Pawel Szalachowski, Daniël Reijsbergen, Ivan Homoliak, and Siwei Sun. 2019. StrongChain: Transparent and Collaborative Proof-of-Work Consensus. In *USENIX.* 819–836.
[72] Parity Technologies. 2019. Parity Wallet. (2019). https://www.parity.io/
[73] Trezor. 2019. Trezor. (2019). https://trezor.io/
[74] TrustedCoin, LLC. 2019. TrustedCoin cosigning service. (2019). https://trustedcoin.com
[75] Unchained Capital. 2019. TrezorMultisig2of3. (2019). https://github.com/unchained-capital/ethereum-multisig
[76] Marie Vasek and Tyler Moore. 2015. There's no free lunch, even using Bitcoin: Tracking the popularity and profits of virtual currency scams. In *Financial Cryptography.* Springer, 44–61.
[77] Wolfie Zhao. 2018. Bithumb $31 Million Crypto Exchange Hack: What We Know (And Don't). (2018). https://www.coindesk.com/bithumb-exchanges-31-million-hack-know-dont-know/

A APPENDIX

A.1 Notation

We use the following notation in addition to the notation used so far: $LSB(.)$ extracts a value of the least significant bit; $a \ll b$ represents the bitwise left shift of a by b bits; a & b represents bitwise AND; $a \oplus b$ represents bitwise exclusive OR; and $h_{\mathcal{D}[a:b]}(.)$ represent $(b - a)$-times chained function $h(.)$ with embedded domain separation respecting interval $\langle a,b \rangle$, e.g., $h_{\mathcal{D}[2:3]}(.) = h(3 \parallel h(2 \parallel .))$,

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

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

Algorithm 5: A reconstruction of a node in a cached sub-layer of a subtree from OTP and its proof π

```
function $deriveNodeInCache(otp, \pi, opID)$
    assert $\pi$.len = $H_S - L_S$;                              ▷ $H_S = log_2\left(\frac{N_S}{P}\right)$
    eci ← $getExpectedIdxInCache\left(opID \% \left(\frac{N_S}{P}\right)\right)$;
    assert eci = deriveIdxInCache($\pi$);
    $a \leftarrow \lfloor (opID \% N_S) * P / N_S \rfloor$;
    res ← $h^{a}_{\mathcal{D}[a:P]}(otp)$;                           ▷ Resolve hash chain
                                                                 ▷ Then resolve $\pi$
    for $\{i \leftarrow 0;\ i < \pi.len;\ i{+}{+}\}$ do
        if $1 = LSB(\pi[i])$ then
            res ← h(res ‖ $\pi$[i]);                    ▷ A node of $\pi[i]$ is on the right
        else
            res ← h($\pi$[i] ‖ res);                    ▷ A node of $\pi[i]$ is on the left
    return res;

function $deriveIdxInCache(\pi)$
    idx ← 0;
    for $\{i \leftarrow 0;\ i < H_S - L_S;\ i{+}{+}\}$ do
        if $1 = LSB(\pi[i])$ then
            idx ← idx | (1 ≪ i);
    return idx;

function $getExpectedIdxInCache(childLeafID)$
    mask ← 0xFFFFFFFF ≡ $2^{32} - 1$;                   ▷ Assuming max. $H_S = 32$
    retID ← $childLeafID$;
    for $\{i \leftarrow H_S - L_S;\ i < H_S;\ i{+}{+}\}$ do
        bitToClear ← 0x01 ≪ i;
        retID ← retID & (mask ⊕ bitToClear);
    return retID;
```

Algorithm 6: A reconstruction of $\mathcal{R}$ from OTP and π

```
function $deriveRootHash(otp, \pi, opID)$
    assert $\pi$.len = H;                                        ▷ $H = log_2\left(\frac{N}{P}\right)$
    assert $opID \% \left(\frac{N}{P}\right)$ = deriveIdx($\pi$);
    $a \leftarrow \lfloor (opID \% N_S) * P / N_S \rfloor$;
    res ← $h^{a}_{\mathcal{D}[a:P]}(otp)$;                           ▷ Resolve hash chain
                                                                 ▷ Then resolve $\pi$
    for $\{i \leftarrow 0;\ i < \pi.len;\ i{+}{+}\}$ do
        if $1 = LSB(\pi[i])$ then
            res ← h(res ‖ $\pi$[i]);                    ▷ A node of $\pi[i]$ is on the right
        else
            res ← h($\pi$[i] ‖ res);                    ▷ A node of $\pi[i]$ is on the left
    return res;

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

Algorithm 7: Aggregation of OTPs

```
function aggregateOTPs(OTPs)
    hOTPs ← [];
    for {$i \in [0, \ldots, OTPs.len - 1]$} do
        hOTPs[i] ← $h^P_{\mathcal{D}}(i + 1 \,\|\, OTPs[i])$;        ▷ Leaves of par. tree
    return reduceMT(hOTPs, hOTPs.len);

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

A.2 Details of Algorithms and Implementation

When bootstrapping $\mathbb{C}$, OTPs of the last iteration layer are generated by Algorithm 4. Generated OTPs are then processed by hash chains, obtaining the first iteration layer of OTPs; this layer is further aggregated into $\mathcal{R}$ by Algorithm 7, which contains recursive in-situ implementation. When the OTP_{opID} is used for the authentication of the operation O_{opID}, $\mathcal{R}$ is reconstructed from the OTP and its proof π_{opID}; first, by resolving hash chains and then π_{opID} (see Algorithm 6).

A.3 Functionality Extension of the Wallet

Daily Limit. Adjusting a daily limit is a functionality that contributes primarily to $\mathbb{U}$'s self-monitoring of expenses but at the same time it avoids typos in transfers that exceeds a daily limit. This operation has the only argument representing an amount that can be spent in a single calendar day. Security implications for this operation are the same as in the case of the transfer crypto-tokens operation (see Section 5).

Last Resort Address and Timeout. As users may lose all 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 Π_O: one for the adjustment of the last resort address (enforced to be different than the address of $\mathbb{U}$) 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 funds 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 $\mathcal{A}$) when $\mathbb{U}$ loses all secrets. Note that update of the activity is made only in the second stage of Π_O, requiring an OTP.

Operation	Stage	Mean [gas]	Standard Deviation [gas]	Sum [gas]
Transfer	Init. Confirm.	70,558 68,540	0 129	139,098
Set Daily Limit	Init. Confirm.	69,342 64,596	0 129	133,938
Set Last Resort Timeout	Init. Confirm.	69,342 64,982	0 474	134,324
Set Last Resort Address	Init. Confirm.	70,366 65,238	0 129	135,604
Introduction of the Next Parent Tree	Stage 1 Stage 2 Stage 3	34,223 49,459 1,082,009	- - -	1,165,691
Introduction of the Next Subtree	Depends mainly on L_S (see Figure 5)			
Send Crypto-Tokens to the Last Resort Address	-	13,887	-	13,887

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

A.4 Cost of All Operations

Operational costs of all implemented operations are shown in Table 1. In the table, we do not account for deployment costs, hence we measure only instant gas consumption of the function calls. The cost measurements were obtained using configuration with the optimal cost (i.e., $L_S = H_S - 3$), $H = H_S$ and $P = 1$, which are independent of H.

A.5 Detailed Description of Protocols

Bootstrapping – protocol Π^S_B
(for a secure environment)

- **Authenticator** $\mathbb{A}$**:** Generate $k \leftarrow random()$ and display k to $\mathbb{U}$.
- **Client** $\mathbb{C}$**:** Upon k, N, N_S, and P are entered by $\mathbb{U}$ into $\mathbb{C}$, compute $OTPs_{LL} \leftarrow F_k(\eta * \frac{N}{P} + i),\ \ i \in \{0,\ldots,\frac{N}{P} - 1\},\ \ \eta \in \{0,1,\ldots\}$. Then compute and store $hOTPs \leftarrow h^P_{\mathcal{D}}(OTPs_{LL}[i]),\ \ i \in \{0,\ldots,\frac{N}{P} - 1\}$ (leaves of the parent tree). Then delete $OTPs_{LL}$ and k. Then compute $\mathcal{R} \leftarrow reduceMT(hOTPs)$ by Algorithm 7, the cached sublayer *cache* of the first subtree and the proof π_{sr} of that subtree's root hash $\mathcal{R}^s$ against $\mathcal{R}$. Then create $tx_{constructor}(\mathcal{R}, cache, \pi_{sr})$ and send it to $\mathbb{W}$. Upon receiving $\{tx_{constructor}(\mathcal{R}, cache, \pi_{sr}, PK_{\mathbb{U}})\}$ from $\mathbb{W}$, forward it to $\mathbb{S}$. Upon receiving the event *ContrDeployed*($\mathbb{S}^{ID}$) from $\mathbb{S}$, update UI and inform $\mathbb{U}$ about the deployment and display $\mathbb{S}^{ID}$.
- **User** $\mathbb{U}$**:** Once k is generated by $\mathbb{A}$, transfer k from $\mathbb{A}$ to $\mathbb{C}$ in an air-gapped manner. Once $\mathbb{C}$ displays $\mathbb{S}^{ID}$, record $\mathbb{S}^{ID}$ as a public reference to $\mathbb{S}$.
- **Private Key Wallet** $\mathbb{W}$**:** Generate private/public key-pair $SK_{\mathbb{U}}$, $PK_{\mathbb{U}} \leftarrow \Sigma.KeyGen()$. Upon receiving $tx_{constructor}(\mathcal{R}, cache)$ from $\mathbb{C}$, add $PK_{\mathbb{U}}$ to this transaction and send it to $\mathbb{C}$.
- **Smart Contract** $\mathbb{S}$**:** Upon receiving $\{tx_{constructor}(\mathcal{R}, cache, \pi_{sr}, PK_{\mathbb{U}})\}$ from $\mathbb{C}$, deploy the code of $\mathbb{S}$ (i.e., Algorithm 1 enriched by storing of *cache*) on the blockchain, assigning $\mathbb{S}^{ID}$ to $\mathbb{S}$. During the deployment, store $\mathcal{R}$, $PK_{\mathbb{U}}$, *cache*, and adjust $nextOpID \leftarrow 0$. Next, compute root hash from *currentSubLayer* and verify its consistency against $\mathcal{R}$ using π_{sr}. Finally, send event *ContrDeployed*($\mathbb{S}^{ID}$) to $\mathbb{C}$.

Bootstrapping – protocol Π^I_B
(for an insecure environment)

- **Authenticator $\mathbb{A}$:** Generate $k \leftarrow random()$. Once $\mathbb{U}$ enters N, N_S, and P to $\mathbb{A}$, compute $OTPs_{LL} \leftarrow F_k(\eta * \frac{N}{P} + i)$, $i \in \{0, \ldots, \frac{N}{P} - 1\}$. Then compute $hOTPs \leftarrow h^P_{\mathcal{D}}(OTPs_{LL}[i])$, $i \in \{0, \ldots, \frac{N}{P} - 1\}$ and export them to microSD card. Then compute $\mathcal{R} \leftarrow reduceMT(hOTPs)$ by Algorithm 7 and display it to $\mathbb{U}$.
- **Client $\mathbb{C}$:** Upon delivering $hOTPs$ by $\mathbb{U}$ to $\mathbb{C}$, store them in the local storage. Then compute $root \leftarrow reduceMT(hOTPs)$ by Algorithm 7, the cached sublayer $cache$ of the first subtree and the proof π_{sr} of that subtree's root hash $\mathcal{R}^s$. Then create $tx_{constructor}(\mathcal{R}, cache, \pi_{sr})$ and send it to $\mathbb{W}$. Upon receiving $\{tx_{constructor}(\mathcal{R}, cache, \pi_{sr}, PK_{\mathbb{U}})\}$ from $\mathbb{W}$, forward it to $\mathbb{S}$. Upon receiving event $ContrDeployed(\mathbb{S}^{ID})$ from $\mathbb{S}$, inform $\mathbb{U}$ in UI.
- **User $\mathbb{U}$:** Enter N, N_S, and P to $\mathbb{A}$ and $\mathbb{C}$. Upon $hOTPs$ are exported by $\mathbb{A}$ to microSD card, transfer them to $\mathbb{C}$. Upon $\mathcal{R}'$ of $\{tx_{constructor}(\mathcal{R}', cache, \pi_{sr})\}$ is displayed at $\mathbb{W}$, verify whether $\mathcal{R} = \mathcal{R}'$ by reading displays of $\mathbb{W}$ and $\mathbb{A}$. In the positive case, proceed with the deployment by pressing a hardware button of $\mathbb{W}$. Once $\mathbb{W}$ displays $\mathbb{S}^{ID}$, record it as a public reference.
- **Private Key Wallet $\mathbb{W}$:** Generate private/public key-pair $SK_{\mathbb{U}}$, $PK_{\mathbb{U}} \leftarrow \Sigma.KeyGen()$. Upon receiving $tx_{constructor}(\mathcal{R}', cache, \pi_{sr})$ from $\mathbb{C}$, display $\mathcal{R}'$ and $\mathbb{S}^{ID} \leftarrow h(PK_{\mathbb{U}} \parallel \mathcal{R}')$ to $\mathbb{U}$. Upon confirmation by $\mathbb{U}$, add $PK_{\mathbb{U}}$ to this transaction and send it to $\mathbb{C}$.
- **Smart Contract $\mathbb{S}$:** The same as in Π^S_B. The only difference in contrast to Π^S_B is the requirement of a deterministic computation of $\mathbb{S}^{ID}$ by a blockchain platform using both $PK_{\mathbb{U}}$ and $\mathcal{R}$. Hence $\mathbb{S}^{ID}$ can be computed by $\mathbb{W}$ and $\mathbb{S}$ independently.

Operation execution – protocol Π_O

- **Authenticator $\mathbb{A}$:** Upon receiving $opID$ from $\mathbb{U}$, compute $OTP_{opID} \leftarrow h^{\alpha(opID)}_{\mathcal{D}}(F_k(\beta(opID)))$, where $\alpha(opID)$ and $\beta(opID)$ are computed by Equation 4. Then display OTP_{opID} to $\mathbb{U}$.
- **Client $\mathbb{C}$:** Once $args$ are entered by $\mathbb{U}$ into $\mathbb{C}$, construct $tx_{initOp}(args)$ and send it to $\mathbb{W}$. Upon receiving $\{tx_{initOp}(args)\}_{\mathbb{U}}$ from $\mathbb{W}$, forward it to $\mathbb{S}$. Upon receiving event $InitOpEvent(opID)$ from $\mathbb{S}$, update UI and inform $\mathbb{U}$ about initialization of O_{opID}. Upon entering OTP_{opID} by $\mathbb{U}$, create proof π_{opID} from the local storage. Then create $tx_{confirmOp}(OTP_{opID}, \pi_{opID}, opID)$ and send it to $\mathbb{S}$. Upon receiving event $ConfirmOpEvent(opID)$ from $\mathbb{S}$, update UI and inform $\mathbb{U}$.
- **User $\mathbb{U}$:** Enter $args$ of an operation into $\mathbb{C}$. Upon $args'$ of $tx_{initOp}(args')$ are displayed at $\mathbb{W}$, verify whether $args = args'$ by reading display of $\mathbb{W}$ and UI of $\mathbb{C}$. In the positive case, confirm signing of transaction by a hardware button of $\mathbb{W}$. Once $\mathbb{C}$ informs about initialized O_{opID}, enter $opID$ into $\mathbb{A}$. Once $\mathbb{A}$ displays OTP_{opID}, transfer OTP_{opID} to $\mathbb{C}$ in an air-gapped manner.
- **Private Key Wallet $\mathbb{W}$:** Upon receiving $tx_{initOp}(args')$ from $\mathbb{C}$, display $args'$ to $\mathbb{U}$. Upon confirmation of $args'$ by $\mathbb{U}$, sign $tx_{initOp}(args')$ by $\Sigma.Sign(tx, SK_{\mathbb{U}})$ and send it to $\mathbb{C}$.
- **Smart Contract $\mathbb{S}$:** Upon receiving $\{tx_{initOp}(args)\}_{\mathbb{U}}$ from $\mathbb{C}$, verify signature $tx.\sigma$ by $\Sigma.Verify(tx.\sigma, PK_{\mathbb{U}})$. Then create a new operation O_{opID} with $opID \leftarrow nextOpID$ using $args$ and increment $nextOpID$. Then send $InitOpEvent(opID)$ to $\mathbb{C}$. Upon receiving $tx_{confirmOp}(OTP_{opID}, \pi_{opID}, opID)$ from $\mathbb{C}$, verify $O_{opID}.pending = true$. Then verify correctness of OTP_{opID} by checking $currentSubLayer[(opID \ \% \ (N_S \ / \ P)) \ / \ 2^{H_S - L_S}] = deriveNodeInCache(OTP_{opID}, \pi_{opID}, opID)$ from Algorithm 5 (or alternatively $\mathcal{R} = deriveRootHash(OTP_{opID}, \pi_{opID}, opID)$ from Algorithm 6 for the version without subtrees). Then execute O_{opID} and set $O_{opID}.pending \leftarrow false$. Finally, send $ConfirmOpEvent(opID)$ to $\mathbb{C}$.

Introduction of a new parent tree – protocol Π^S_{NR}
(for a secure environment)

- **Authenticator $\mathbb{A}$:** Once $\mathbb{U}$ enters $opID$ into $\mathbb{A}$, check whether $opID \ \% \ N = N - 1$, and if so, notify $\mathbb{U}$ that a new parent tree is being introduced and display k to $\mathbb{U}$. Then compute $OTP_{opID} \leftarrow h^{\alpha(opID)}(F_k(\beta(opID)))$. Next, compute $OTPs_{LL} \leftarrow F_k(\eta \frac{N}{P} + i)$, $i \in \{0, \ldots, \frac{N}{P} - 1\}$, where $\eta \leftarrow \eta + 1$. Then compute $\mathcal{R}^{new} \leftarrow aggregateOTPs(OTPs_{LL})$ by Algorithm 7 and $hRootAndOTP \leftarrow h(\mathcal{R}^{new} \parallel OTP_{opID})$. Finally, show $\mathcal{R}^{new}$ and $hRootAndOTP$ to $\mathbb{U}$.
- **Client $\mathbb{C}$:** •[*Stage I*] $\mathbb{C}$ notifies $\mathbb{U}$ that a new parent tree needs to be introduced and displays $opID = N - 1 + \eta N$, $\eta \in \{0, 1, \ldots\}$. Once $\mathbb{U}$ enters k into $\mathbb{C}$, compute $OTP_{N-1} \leftarrow h^{\alpha(opID)}(F_k(\beta(opID)))$, where $\alpha(opID)$ and $\beta(opID)$ are computed by Equation 4. Then create proof π_{opID} from the local storage. Then compute $OTPs_{LL} \leftarrow F_k(\eta \frac{N}{P} + i)$, $i \in \{0, \ldots, \frac{N}{P} - 1\}$, where $\eta \leftarrow \eta + 1$. Then compute and store $hOTPs \leftarrow h^P_{\mathcal{D}}(OTPs_{LL}[i])$, $i \in \{0, \ldots, \frac{N}{P} - 1\}$. Then delete $OTPs_{LL}$ and k. Then compute $\mathcal{R}^{new} \leftarrow reduceMT(hOTPs)$ by Algorithm 7. Then compute $hRootAndOTP \leftarrow h(\mathcal{R}^{new} \parallel OTP_{opID})$, construct $tx_{1\_newRootHash}(hRootAndOTP)$, and send it to $\mathbb{W}$. Upon receiving $\{tx_{1\_newRootHash}(hRootAndOTP)\}_{\mathbb{U}}$ from $\mathbb{W}$, forward it to $\mathbb{S}$. •[*Stage II*] Upon $newRootHash1(hRootAndOTP)$ event is received from $\mathbb{S}$, construct $tx_{2\_newRootHash}(\mathcal{R}^{new})$ and send it to $\mathbb{W}$. Once $\{tx_{2\_newRootHash}(\mathcal{R}^{new})\}_{\mathbb{U}}$ is received from $\mathbb{W}$, forward it to $\mathbb{S}$. •[*Stage III*] Upon receiving the event $newRootHash2(\mathcal{R}^{new})$ from $\mathbb{S}$, compute the cached sublayer cs of the first subtree in the new parent tree and the proof π_{sr} of the subtree's root hash $\mathcal{R}^s$. Then construct $tx_{3\_newRootHash}(OTP_{opID}, \pi_{opID}, cs, \pi_{sr})$ and send it to $\mathbb{S}$. Upon receiving event $newRootHash3(OTP_{opID})$ from $\mathbb{S}$, update UI and inform $\mathbb{U}$.
- **User $\mathbb{U}$:** Once $\mathbb{C}$ displays $opID = N - 1 + \eta N$, $\eta \in \{0, 1, \ldots\}$ and informs $\mathbb{U}$ about the necessity of introducing a new parent tree, enter $opID$ into $\mathbb{A}$. Once

$\mathbb{A}$ displays k and shows a message that a new parent tree is being introduced, transfer k from $\mathbb{A}$ to $\mathbb{C}$ in an air-gapped manner. Once $hRootAndOTP'$ of $\{tx_{1\_newRootHash}(hRootAndOTP')\}$ is displayed at $\mathbb{W}$, verify $hRootAndOTP' = hRootAndOTP$ by reading displays of $\mathbb{W}$ and $\mathbb{A}$. Once $\mathcal{R}^{new'}$ of $\{tx_{2\_newRootHash}(\mathcal{R}^{new'})\}$ is displayed at $\mathbb{W}$, verify $\mathcal{R}^{new'} = \mathcal{R}^{new}$ by reading displays of $\mathbb{W}$ and $\mathbb{A}$. If so, confirm signing by a hardware button of $\mathbb{W}$.

- **Private Key Wallet** $\mathbb{W}$**:** The same as in Π_O.
- **Smart Contract** $\mathbb{S}$**:** •[*Stage I*] Upon receiving $\{tx_{1\_newRootHash}(hRootAndOTP)\}_{\mathbb{U}}$ from $\mathbb{C}$, verify signature $tx.\sigma$ by $\Sigma.Verify(tx.\sigma,\ PK_{\mathbb{U}})$. Then verify $nextOpID\ \%\ N = N-1$; if so, append $hRootAndOTP$ into L_1. Then send event $newRootHash1(hRootAndOTP)$ to $\mathbb{C}$. •[*Stage II*] Upon receiving $\{tx_{2\_newRootHash}(\mathcal{R}^{new})\}_{\mathbb{U}}$ from $\mathbb{C}$, verify signature $tx.\sigma$ by $\Sigma.Verify(tx.\sigma,\ PK_{\mathbb{U}})$. Then verify $nextOpID\ \%\ N = N-1$; if so, append $\mathcal{R}^{new}$ into L_2. Then send event $newRootHash2(\mathcal{R}^{new})$ to $\mathbb{C}$. •[*Stage III*] Once $\{tx_{3\_newRootHash}(OTP_{opID},\ \pi_{opID},\ cs, \pi_{sr})\}$ is received from $\mathbb{C}$, verify $nextOpID\ \%\ N = N-1$. Then verify correctness of OTP_{opID} by checking $currentSubLayer[(opID\ \%\ (N_S\ /\ P))\ /\ 2^{H_S-L_S}] = deriveNodeInCache(OTP_{opID},\ \pi_{opID},\ opID)$ from Algorithm 5 (or $\mathcal{R} = deriveRootHash(OTP_{opID},\ \pi_{opID},\ opID)$ from Algorithm 6 in the version without subtrees). Then locate the first entries of L_1 and L_2 that match the condition $h(L_2[i]\ \|\ OTP_{opID}) = L_1[j]$. If matching entries are found, then set $\mathcal{R} \leftarrow L_2[i]$, increment $nextOpID$, adjust $currentSubLayer \leftarrow cs$ and verify its consistency by $subtreeConsistency(\mathcal{R}^s,\ \pi_{sr},\ \mathcal{R})$, where $\mathcal{R}^s \leftarrow reduceMT(currentSubLayer, currentSubLayer.len)$. Finally, clear the lists $L_1, L_2 \leftarrow [], []$.

Introduction of a new parent tree – protocol $\Pi_{NR}^{\mathcal{I}}$ (for an insecure environment)

- **Authenticator** $\mathbb{A}$**:** Once $\mathbb{U}$ enters $opID$ into $\mathbb{A}$, check whether $opID\ \%\ N = N-1$, and if so display $OTP_{opID} \leftarrow h^{\alpha(opID)}(F_k(\beta(opID)))$ and notify $\mathbb{U}$ that a new parent tree is being introduced. Then, compute $OTPs_{LL} \leftarrow F_k(\eta\frac{N}{P} + i),\ i \in \{0, \ldots, \frac{N}{P}-1\}$, where $\eta \leftarrow \eta + 1$. Then compute $hOTPs \leftarrow h^P(OTPs_{LL}[i]),\ i \in \{0, \ldots, \frac{N}{P}-1\}$ and export it to a microSD card. Finally, compute $\mathcal{R}^{new} \leftarrow reduceMT(hOTPs)$ by Algorithm 7 and $hRootAndOTP \leftarrow h(\mathcal{R}^{new}\ \|\ OTP_{opID})$, and display both to $\mathbb{U}$.
- **Client** $\mathbb{C}$**:** [*Stage I*] $\mathbb{C}$ notifies $\mathbb{U}$ that a new parent tree needs to be introduced and displays $opID = N - 1 + \eta N,\ \eta \in \{0, 1, \ldots\}$. Upon entering OTP_{opID} by $\mathbb{U}$, create proof π_{opID} from the local storage. Once leaves of the tree $hOTPs$ are delivered by $\mathbb{U}$ into $\mathbb{C}$, store $hOTPs$ in the local storage. Then compute $\mathcal{R}^{new} \leftarrow reduceMT(hOTPs)$ by Algorithm 7. Then compute $hRootAndOTP \leftarrow h(\mathcal{R}^{new}\ \|\ OTP_{opID})$, construct $tx_{1\_newRootHash}(hRootAndOTP)$, and send it to $\mathbb{W}$. Upon receiving $\{tx_{1\_newRootHash}(hRootAndOTP)\}_{\mathbb{U}}$ from $\mathbb{W}$, forward it to $\mathbb{S}$. [*Stages II*] and [*Stage III*] are the same as in $\Pi_{NR}^{\mathcal{S}}$
- **User** $\mathbb{U}$**:** Once $\mathbb{C}$ displays $opID = N - 1 + \eta N,\ \eta \in \{0, 1, \ldots\}$ and informs $\mathbb{U}$ about necessity of introducing a new parent tree, enter $opID$ into $\mathbb{A}$. Once $\mathbb{A}$ displays OTP_{opID} and notify $\mathbb{U}$ that the new parent tree is being introduced, transfer OTP_{opID} to $\mathbb{C}$ in an air-gapped manner. Once $hOTPs$ are exported by $\mathbb{A}$ to microSD card, transfer them to $\mathbb{C}$. Once $hRootAndOTP'$ of $\{tx_{1\_newRootHash}(hRootAndOTP')\}$ is displayed at $\mathbb{W}$, verify $hRootAndOTP' = hRootAndOTP$ by reading displays of $\mathbb{W}$ and $\mathbb{A}$; in the positive case, confirm signing of transaction within $\mathbb{W}$ by a hardware button. Once $\mathcal{R}^{new'}$ of $\{tx_{2\_newRootHash}(\mathcal{R}^{new'})\}$ is displayed at $\mathbb{W}$, verify $\mathcal{R}^{new'} = \mathcal{R}^{new}$ by reading displays of $\mathbb{W}$ and $\mathbb{A}$; in the positive case, confirm signing of transaction within $\mathbb{W}$ by a hardware button.
- **Private Key Wallet** $\mathbb{W}$**:** The same as in Π_O.
- **Smart Contract** $\mathbb{S}$**:** The same as in Π_{NR_S}.

Introduction of the next subtree – protocol Π_{ST}

- **Authenticator** $\mathbb{A}$**:** The same as in Π_O, while in addition, $\mathbb{A}$ displays a message that the next tree is being introduced.
- **Client** $\mathbb{C}$**:** $\mathbb{C}$ notifies $\mathbb{U}$ that a new subtree needs to be introduced and displays $opID = (N_S - 1) + \delta N_S,\ \delta \in \{0, \ldots, \frac{N}{N_S} - 2\}$. Upon entering OTP_{opID} by $\mathbb{U}$, create the proof π_{opID} from the local storage. Then compute $nextSubLayer$ (i.e., the cached sublayer of the next subtree) and π_{sr} (i.e., the proof of the next subtree's root) from $\mathbb{C}$'s storage. Next construct $tx_{nextSubtree}(nextSubLayer,\ OTP_{opID},\ \pi_{opID},\ \pi_{sr})$ and send it to $\mathbb{S}$. Upon receiving event $newSubtree(opID)$ from $\mathbb{S}$, update UI and inform $\mathbb{U}$.
- **User** $\mathbb{U}$**:** Once $\mathbb{C}$ displays $opID = (N_S - 1) + \delta N_S,\ \delta \in \{0, \ldots, \frac{N}{N_S} - 2\}$ and informs $\mathbb{U}$ about necessity of introducing the next subtree, enter $opID$ into $\mathbb{A}$. Once $\mathbb{A}$ displays OTP_{opID} and confirming that the next tree is being introduced, transfer it to $\mathbb{C}$ in an air-gapped manner.
- **Private Key Wallet** $\mathbb{W}$**:** No interaction required.
- **Smart Contract** $\mathbb{S}$**:** Upon receiving $tx_{nextSubtree}(nextSubLayer,\ OTP_{opID},\ \pi_{opID},\ \pi_{sr})$ from $\mathbb{C}$, verify $nextOpID\ \%\ N \neq N-1 \wedge nextOpID\ \%\ N_S = N_S - 1 \wedge currentSubLayer.len = nextSubLayer.len$.

Then verify correctness of OTP_{opID} by checking $cache[(opID\ \%\ (N_S\ /\ P))\ /\ 2^{H_S-L_S}] = deriveNodeInCache(OTP_{opID},\ \pi_{opID},\ opID)$ from Algorithm 5. Next, update the current cached sublayer $currentSubLayer \leftarrow nextSubLayer$ and check its consistency against $\mathcal{R}$ by a function $subtreeConsistency(\mathcal{R}^s,\ \pi_{sr},\ \mathcal{R})$. Note that this function requires already computed $\mathcal{R}$ of the next subtree using Algorithm 7: $\mathcal{R}^s \leftarrow reduceMT(currentSubLayer,\ currentSubLayer.len)$ and its proof π_{sr}. Finally, increment $nextOpID$ and send event $newSubtree(opID)$ to $\mathbb{C}$.

A.6 Hardware Implementation of $\mathbb{A}$

For demonstration purposes, we constructed a proof-of-concept hardware implementation of the authenticator of SmartOTPs using cheap hardware and C language. In detail, we selected NodeMCU with ESP8266 MCU that costs around $2. Next, we selected a 0.96" OLED display with resolution of 128x64 that was connected to MCU by 4-wire SPI interface (the cost of such a display falls below $2). To control the display, we used Adafruit_SSD1306 a Adafruit_GFX libraries. Finally, we used a simple 4x4 keyboard with 3+4 wires addressing a combination of 3 columns and 4 rows (the cost of the keyboard is around $0.5). To interact with the keyboard, we utilized Keypad library of Adruino that is built for matrix style keyboards. Further, we used software version of hash function, available from https://github.com/ethereum/ethash. The scheme of our hardware implementation is depicted in Figure 9 and the source with a demonstration video is provided at https://github.com/ivan-homoliak-sutd/SmartOTPs.

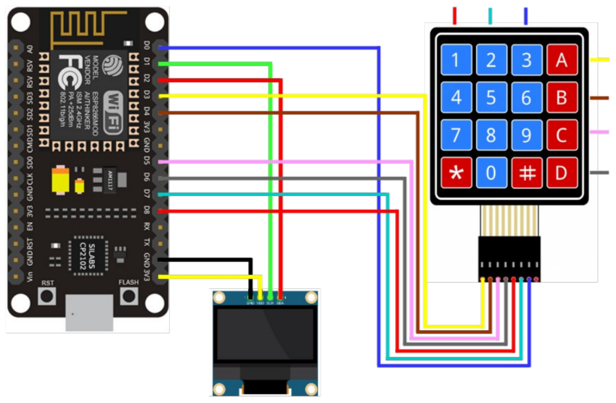


Figure 9: The scheme of a proof-of-concept hardware implementation of the authenticator.