Enhancing Security in Smart Contract Wallets : An OTP Based 2-Factor Authentication Approach

Kalash
National Institute of Technology Karnataka Surathkal (India)
Surathkal, India
kalash.222is008@nitk.edu.in

Bishakh Chandra Ghosh
Indian Institute of Technology Kharagpur (India)
Kharagpur, India
ghoshbishakh@iitkgp.ac.in

Sourav Kanti Addya
National Institute of Technology Karnataka Surathkal (India)
Surathkal, India
souravkaddya@nitk.edu.in

Abstract

As cryptocurrencies have gained widespread popularity, the security and handling of crypto-assets have become increasingly crucial. Numerous attacks targeting both users and blockchain platforms have led to substantial financial losses. This paper proposes a system for 2-factor authentication (2FA) for smart contract wallets, providing users with a flexible, secure, and customizable way of managing their crypto assets. The proposed methodology utilizes cryptographic hash functions and hash chains to generate One-Time Passwords (OTPs) for authentication, ensuring protection against unauthorized access. The 2FA setup involves a client interacting with a smart contract along with an authenticator and software wallet while using the public-private key pair of wallet as the first factor, and OTPs as the second factor. This is done through a two-stage protocol for bootstrapping and operation execution, and offers a level of security similar to traditional authentication schemes like HOTP. Using a novel pre-commitment scheme we also defend the users from front-running attacks. The implementation of the system is done in the context of public blockchain evaluating the practicality and effectiveness of the 2FA model. We open source our implementation for the Ethereum platform and make it available for the community. Furthermore, we analyse the cost incured based on gas consumption, space requirements and payload. In addition we suggest future enhancements for shorter OTP lengths and time based OTPs.

CCS Concepts

• **Security and privacy → Distributed systems security**.

Keywords

Blockchain, Smart Contract, Security, Two factor authentication, Hash chains

ACM Reference Format:
Kalash, Bishakh Chandra Ghosh, and Sourav Kanti Addya. 2025. Enhancing Security in Smart Contract Wallets : An OTP Based 2-Factor Authentication Approach. In *26th International Conference on Distributed Computing and Networking (ICDCN 2025), January 04–07, 2025, Hyderabad, India.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3700838.3700868

ICDCN 2025, January 04–07, 2025, Hyderabad, India

ACM ISBN 979-8-4007-1062-9/25/01
https://doi.org/10.1145/3700838.3700868

1 Introduction

The extraordinary proliferation of cryptocurrencies has catalyzed the emergence of numerous open and decentralized platforms, facilitating financial transactions, self-executing contracts, and the exchange of digital assets [28, 46, 47]. The financial volumes in cryptocurrency transactions are considerably immense, approximating the market cap of 2.42 trillion USD [40]. The volume of transactions in the cryptocurrency platforms is ever increasing. However, just like in any other sector, numerous cases of fraud and security breaches also persist in the cryptocurrency landscape [32, 45]. These breaches are often due to stolen keys [10, 12, 15, 45] or server-side wallets that require users to trust the provider [34, 41]. Cryptocurrency platforms offer tokens, known as crypto-tokens such as Bitcoin, Ethereum, Monero, etc. [40], that can be transferred through blockchain transactions. These transactions are authenticated by the owners' digital signature using the owners' private keys. These private keys are managed by a wallet application that gives the user an interface to interact with the blockchain network. Typically, the private key is encrypted using the user's chosen password or a seed phrase (a recovery phrase) [11]. Unfortunately, people often choose weak passwords that can be brute-forced if their devices are compromised by some malware [31]. In the event of a leak of the private key, there is no way for the legitimate owner of the crypto assets to prevent fraudulent transactions. Despite the robust security provided by digital signatures, the increasing value of crypto assets and the sophistication of cyber threats necessitate enhanced security measures to protect users and their investments. Employing an additional layer of authentication, that is multi-factor authentication (MFA) can be an optimal and elegant solution towards elevating wallet security. This system allows the spending of cryptocurrency tokens only when a combination of multiple secrets is utilized. With MFA, even in the event of leak of the wallet private key, attackers cannot launch fraudulent transactions. This buys time for the crypto asset owners to move their assets to an uncompromised account.

Multi-factor authentication is already a standard practice in traditional web based applications [14, 18, 20, 37, 39] where typically a one-time password (OTP) is used as the second factor in addition to the user's password. The first popular protocol for these OTP based MFA was HMAC-based one-time password (HOTP) [35]. In a standard HOTP scheme, apart from the application server, a separate authenticator device is used that generate the one time passwords. This authenticator device is usually air-gapped with the client (browser in case of a web application). Both the server and the authenticator use a secret key and a counter. They use a pseudo-random function *hmac* to create one-time passwords. During setup,

both sides agree on a secret key k and start with a counter value c set to 0. The passwords are then generated using $hmac(k, c)$. The counter is incremented each time a password is generated, while the server increments it following each successful authentication. Similarly, in the following years, standard time-based one-time password schemes (TOTP) [36] were developed to make one-time passwords valid only for a specific period. In TOTP, instead of a shared counter like in HOTP, a timestamp-based counter is used. During setup, the server and device agree on the starting time t_0, (usually the UNIX epoch), a time interval I (typically 30 seconds), and a secret key k. Then, when the client wants to authenticate, it calculates $hmac(k, (t - t_0)/I)$, where t is the current time.

While these traditional OTP-based techniques deliver robust second factor authentication, their implementation within blockchain system presents several technical challenges. The challenges arise primarily because the web applications are replaced by smart contracts in case of blockchains. Instead of a secure server, the data is recorded in a public ledger. Moreover, in place of encrypted HTTPS requests from the browser to the server, blockchain transactions are collected in transaction pools where every transaction is public. Concretely, the challenges are as follows. **First,** in both TOTP and HOTP schemes, the server must retain the secret which is shared with the authenticator device. However, storing the seed on the blockchain would expose it to everyone, compromising the security of the system. **Second,** attempting to generate a random secret within our smart contract is also unfeasible owing to the deterministic characteristics of the blockchain platform. This is because, given a specific input and code, the blockchain platform is designed to consistently produce the same output. **Third,** the computational overhead and the large size of signatures generated by conventional asymmetric cryptography techniques [19] result in higher execution costs. For the blockchain platforms, there is a gas limit tied to the computational effort required, which these techniques can easily surpass. Consequently, these methods are not optimal for our specific blockchain environment. **Fourth,** submitting OTPs through smart contract transactions is susceptible to front running attacks. Attackers can scan the transactions from the transaction pool, recover the OTP, and use the same OTP to create their own fraudulent transactions. Through front running [21], the fraudulent transactions can be made to be mined before the legitimate transactions by offering higher transaction fees.

Contributions. Overcoming the above limitations, we propose a smart contract wallet framework which provides 2FA for blockchain transactions. Our proposed system allows the spending of cryptocurrency tokens only when a combination of multiple secrets is utilized. In this context, our focus lies on developing state-aware smart contract wallets which enforce spending regulations and security attributes directly within the smart contracts.

(1) In addition to the user's private key for primary authentication factor, we introduce a secondary factor that uses a hash-based cryptographic construct to generate OTPs, utilizing hash chains for security.
(2) We introduce a novel pre-commitment scheme to mitigate the front running man in the middle attacks while submitting OTPs.
(3) Compared to other cryptographic constructs like Merkle tree based approach [25], our hash chain-based model effectively reduces the proof payload and eliminates the necessity for client storage space during OTP generation.
(4) We have implemented our system using *Ethereum* as the public blockchain platform, and made it open-source for the community [3].
(5) For a better user-experience, we have implemented a QR code based OTP entry mechanism in the application.

We conducted thorough experiments and analysis to evaluate our system, and demonstrate its light payload and zero client storage requirements. Furthermore, we validate the security of our model against front-running transactions by malicious attackers. Additionally, we have performed a gas requirement analysis of our model to establish its practicality.

2 Related Work

The second factor authentication was initially introduced as one-time passwords [29], which were subsequently realized as S/Key [24]. Later, HOTP and TOTP were suggested in [35] and [36], respectively. In [43], a thorough analysis was conducted on two-factor authentication schemes, presenting a range of efficient protocols with different considerations regarding usability and security.

While Merkle signatures [33] utilize merkle trees to combine multiple one-time verification keys, the size of these keys and signatures is significantly larger compared to the size of OTPs. SmartOTPs [25] is a notable work which utilizes merkle trees for one-time password based 2-factor authentication in blockchains. The merkle tree is generated by using the hashed OTPs as leaf nodes, whereas the root of the merkle tree is stored on the blockchain to verify the OTPs at a later stage. To generate the proof for authentication, the client device constructs the proof from hashes after fetching the merkle tree nodes from its storage. This proof along with OTP is sent to smart contract where the smart contract does the comparison with the stored root after proof and OTP validation. However, the problem lies in storing the merkle tree on the client side, especially when dealing with a larger number of OTPs, the tree size increases, leading to greater space consumption. Additionally, when submitting the OTP, the proof must also be sent to the smart contract wherein proof's length is also dependent on size of tree. Furthermore, the SmartOTP does not address the potential risk of front-running attacks that could occur when submitting both the merkle root hash and the OTPs to the smart contract.

Use of other cryptographic constructs like hash chain for OTP generation is also extensive in traditional centralized systems. In [23], a reinitializable hash chain is introduced, which is a type of hash chain designed to be securely reinitialized once its root is reached. A detailed review of hashchains and their applications have been done in [16, 26] . A 2-factor authentication system over standard TOTP and HOTP using hashchains is provided by T/key [27]. It blends concepts from both S/key and TOTP to harness the optimal features of each. T/key also doesn't retain any confidential information on the server and ensures that passwords have a limited validity period. To the best of our knowledge, our approach of utilizing hash chain constructs as the primary method for generating OTPs to address security challenges in state-aware smart contracts is unique and distinctive.

3 System Model and Design Challenges

We design our 2-factor authentication approach for public (permissionless) blockchains. These blockchains operate with widely-recognized consensus protocols, such as Proof of Work (PoW) [38] and Proof of Stake (PoS) [13]. These protocols ensure the immutability and integrity of the ledger by requiring participants to perform computational work or hold a stake in the network, respectively, for mining each block. The applications for which we design our 2-factor system are smart contracts deployed on these public blockchains. Given the public nature of the blockchain, all data in the ledger including any smart contract data is accessible to any participant within the network. Moreover, any transaction initiated by any participant of the blockchain is visible to everyone else since a common transaction pool is created from which the miners select some of the transactions to mine in a block.

We assume that the cryptographic constructs, including cryptographic hash algorithms and digital signatures, are secure and resistant to compromise, meaning that adversaries will be unable to break these cryptographic components or the guarantees of the underlying blockchain platform. We also assume that the probability of a hash collision, where two distinct inputs produce the same hash output, is negligible, and that reversing the hash function used in our hash chains is computationally infeasible.

Based on traditional 2-factor authentication mechanisms, such as those employed by Microsoft Authenticator [5], we consider that the client and the authenticator are distinct devices or apps which are not connected. For instance, the client device might be a laptop, desktop, that houses the software wallet with private keys, while the authenticator could be an application on a separate device like mobile phone. This setup creates an air gap between the client and authenticator, meaning that they do not need to connect via USB or any other data transfer method, enhancing security by isolating the authentication process from direct data connections.

Simply sharing a secret between the smart contract and the authenticator, as is done in traditional server-authenticator setups, is impractical in this context. If we treat the smart contract as our server, any data stored on it becomes accessible to the entire public blockchain network, making the concept of a shared secret untenable. Utilizing public key cryptography [17] to tackle these challenges will incur significant computational costs for the verification process on the smart contract side. All the blockchain operations are governed by a specific gas limits, and exceeding those gas limits due to high computational demands results in transaction failure, high processing fees and ultimately state reversions. Thus, incorporating traditional public key cryptography is also not viable due to its substantial computational overhead.

3.1 Threat Model

We assume an adversary aiming to execute unauthorized transactions on the user's behalf and control the user's wallet. The adversary can potentially take part in the consensus protocols of the blockchain platform and intercept or eavesdrop the network traffic. In our model, we consider two potential adversaries: 1) someone with access to the user's private key, or 2) someone with access to the authenticator. The adversary is not able to compromise both the wallet and the authenticator together. These scenarios are taken into account during operation authentication and not during the bootstrap phase. We also hypothesize that attacker possesses the capability to intercept and front run the user's transactions. This could involve executing a man-in-the-middle (MITM) attack or crafting a malicious transaction that conflicts with the user's intended transaction, potentially offering a higher fee to miners. In such cases, miners may prioritize attacker's transaction, leading to the user's transaction being disregarded, a phenomenon often referred to as transaction front-running.

4 Methodology

In this section we first describe the building blocks used to construct our proposed 2-factor authentication approach. Then we describe in detail the architecture of the 2-factor authentication system and its protocols.

4.1 Building blocks

We consider a generic cryptocurrency where records are organized into blocks within a continuously expanding public distributed ledger known as a blockchain. This blockchain is inherently immutable and resistant to alterations by design. In this architecture, blocks are linked together by cryptographic hashes, and every new block must be approved by members, usually through mining. These blocks could include token transfers for cryptocurrencies, executable software written in a language specific to the platform, and the instructions needed to execute those programs. These executable scripts, often known as smart contracts, can store several types of processing logic, including contracts. One example of such blockchain platform is Ethereum [47].

Hash chain: A hash chain is a cryptographic data structure that follows a specific pattern for verification and authentication. In a hash chain, each link in the chain contains the cryptographic hash of the previous link, forming a sequence of interconnected hashes. By comparing the final hash of a chain with a computed hash from the initial data and subsequent hashes, one can confirm the validity of the entire chain. The process of reconstructing the final hash has a linear time complexity, which renders hash chains a practical and efficient method for ensuring data integrity and authenticity. However, given a final hash, or intermediate hash values, it is computationally infeasible to construct the previous hashes in the chain. We use this property for the construction of OTPs without having a shared secret between the smart contract and the authenticator.

4.2 Architecture and Protocols

Design of hash chain for OTPs. We initiate the generation of our hash chain with the starting seed S_k, which undergoes hashing at each step using a different hash function represented by h_i described in Equation (1). Upon computing all the hashes, we obtain the final hash at the end of the hash chain, referred as $rootHash$, which serves for verification in subsequent stages. Such a hash chain is depicted in Figure 1. To generate OTPs, we traverse the hash chain in reverse, starting from the tail which is $rootHash$ towards the head which is seed. The node immediately preceding the tail is regarded as the first OTP, and so forth until reaching the head. The head represents the final OTP or the k_{th} OTP, where k is the length

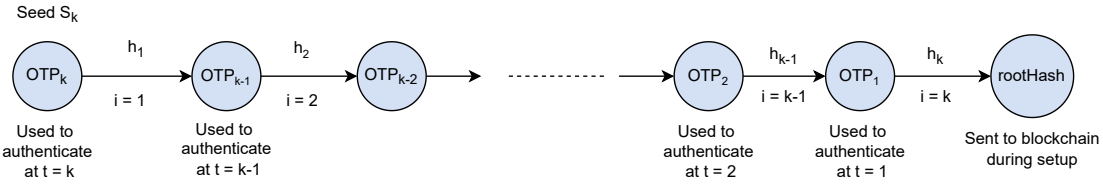

Figure 1: Design of hash chain for OTPs

of the hash chain. In this context, even if OTP_m is revealed, it is very easy to compute the $rootHash$. But the derivation of OTP_n from OTP_m where $m < n$ is computationally infeasible due to the one way nature of cryptographic hash functions [22]. To verify an OTP_m, we simply require to hash it m times with corresponding required values of i, get the hashed output and then further compare it with the $rootHash$. If the $rootHash$ is equal to the generated hash output, then the OTP_m is deemed valid indicating that OTP_m is indeed a constituent of the current hash chain.

We propose a 2-factor authentication for blockchain which consists of : (1) a client C, (2) a private key software wallet W, (3) a smart contract S, and (4) an authenticator A. The client is a device running an application using which the transactions to the blockchain are initiated. The client device also holds the wallet containing the private key and public key of the user (SK_U, PK_U). The authenticator can be an application running on smart phone or similar device that is air-gapped from the client. The first factor is traditional public key authentication where the client prepares a transaction and digitally signs it using the private key SK_U from the wallet. The transaction is then submitted to the blockchain for executing a smart contract. The blockchain network derives the user's public key from the provided signature and the address, and verifies the authenticity of the transaction. The second factor are the OTPs which are produced by the authenticator and inputted to the client by the user to initiate a transaction. The provided OTP is then validated by a specialized smart contract that enforces the second factor of the authentication. This specialized smart contract acts as a gatekeeper, validating transactions based on encoded spending rules and security features to determine if crypto token transfers will proceed or not. We assume a two stage protocol for the entire setup denoted as Π_O, where first phase is bootstraping (Π_B) and second phase is operation exceution (Π_E). Table 1 shows all the parameters utilized in the equations, along with their respective lengths.

Parameter	Length	Description
id	32 bits	Salt length
k	256 bits	Chain length
c	256 bits	Commitment for OTP
OTP	128 bits	One time password length
t	256 bits	Number of bits used for time
rootHash	128 bits	Hash root stored in smart contract

Table 1: Parameters and their lengths

Bootstrap / Intial setup (Π_B) : We illustrate the initial setup process in Figure 2. As common in other schemes and protocols, by default, we assume a secure environment for initial setup, which means that C is trusted and cannot be compromised during execution of Π_B. Initially, C creates a confidential seed S_k, generates a random salt denoted as id, records the time of setup initiation as a counter initiation value denoted as t_{init} and determines the maximum number of OTPs (hash chain length) denoted as k. Note that the time t and t_{init} here does not represent physical clock time but functions as a logical clock. For instance, during bootstrap, t_{init} can be initialized to zero, while for each operation verification, the current time t increments by one with each new OTP request. Subsequently, the user securely transfers this seed S_k, id, t_{init} and k from C to A through an air-gapped process, such as transcribing a selection of mnemonic words or scanning a QR code. Next, the client uses the hash functions defined by $h_i(x)$ to generate a hash chain. It's worth noting that for each node of the hash chain, a distinct and independent hash function is utilized as shown in the folllowing equation.

$$h_i(x) = H(< t_{init} + k - i > \| \ id \ \| \ x) \tag{1}$$

The approach of deriving independent hash functions from a singular hash function across an expanded domain, commonly referred to as domain separation, and it is often attributed to Leighton and Micali [30]. Drawing upon the described hash functions, the client iteratively computes the $rootHash$ as shown in Algorithm 1 by applying the hash functions k times, where i ranges from 1 to k in $h_i(x)$.

$$rootHash = h_k(h_{k-1}(h_{k-2}(.....(h_1(S_k)....)) \tag{2}$$

Algorithm 1: Generation of rootHash

function $genRootHash(t_{init}, k, id, S_k)$ returns $(32\ byte\ hash)$
  $rootHash \leftarrow S_k$
  for $(i = 1; i <= k; i++)$ **do**
    $rootHash \leftarrow genHashFunc(t_{init}, k, id, i, rootHash)$
  return $rootHash$

function $genHashFunc(t_{init}, k, id, i, x)$ returns $(32\ byte\ hash)$
  $y \leftarrow t_{init} + k - i$
  $hash \leftarrow sha256(y \ \| \ id \ \| \ x)$
  return $hash$

After this stage, the secret seed S_k as well as the hash chain is erased from the client's storage. The client then initiates the transaction to store values in the smart contract S using the wallet W, requesting the user to sign the transaction using the private

key stored in the wallet. The signed transaction is then broadcasted to the blockchain network. The transaction is then mined by the miners who verify the validity of the transaction signature. S stores all public parameters like t_{init}, k, id and $rootHash$ illustrated in Figure 2. The smart contract uses its state variables and initialises t_{prev} as t_{init} and $pprev$ as $rootHash$. Hence, the keys (SK_U, PK_U) in wallet W enables whether the arbitrary transaction was signed by the user or not while $rootHash$ enables the verification whether the given OTP was produced by user's authenticator A or not.

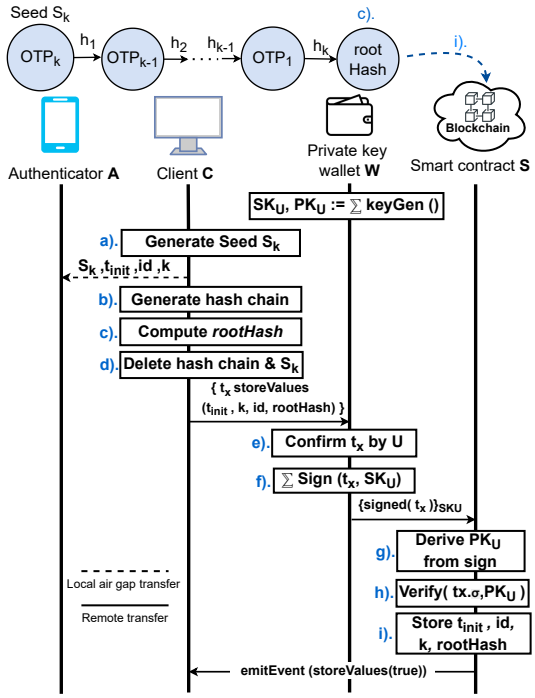

Figure 2: Initial setup (π_B)

Execute operation (Π_E) : Once the wallet framework is initialized, it becomes ready to execute operations and do the 2-factor authentication. The initial factor involves verification through the transaction signature, while the second factor involves OTP verification. We illustrate this execution operation flow in Figure 3.

The user while initiating a transaction operation O_t first enters the details of operation into C creating a transaction $initOp()$. This needs operation parameters like the type of operation (e.g., transfer), a numerical parameter (e.g., amount or daily limit), an address parameter (e.g., recipient), the time t when operation is initiated and the commitment for the OTP. To generate the required OTP, OTP_t for the transaction O_t, the client first displays the parameter t to the user. Based on t, the authenticator A generates the OTP_t (Equation 3). In addition, a commitment c for the OTP is also generated by the authenticator.

$$OTP_t = h_{t_{max}-t}(h_{t_{max}-t-1}(.....(h_1(S_k)....))) \quad (3)$$

The commitment c plays a vital role to prevent front-running attacks. Sending the commitment first without revealing the OTP to the client prevents the attackers having possession of SK_U from forging a valid OTP to perform their own fraudulent transactions.

Algorithm 2: Generation of OTP and commitment

function $genOTP(t_{init}, k, id, S_k, t)$ returns (otp)
  $OTP_t \leftarrow S_k$
  for $(i = 1; i <= t_{max} - t; i++)$ **do**
    $OTP_t \leftarrow genHashFunc(t_{init}, k, id, i, OTP_t)$
  return OTP_t

function $genCommitment(OTP_t)$ returns $(commitment)$
  $c \leftarrow sha256(OTP_t)$
  return c

function $genHashFunc(t_{init}, k, id, i, x)$ returns $(32\ byte\ hash)$
  $y \leftarrow t_{init} + k - i$
  $hash \leftarrow sha256(y \parallel id \parallel x)$
  return $hash$

The use of commitment protocol forces the attacker and user both to send the commitment first and the original OTP is revealed in later stage which results in attacker being unaware of OTPs in earlier phase of verification. Algorithm 2 shows the OTP generation and commitment generation schemes. We have discussed about the commitment scheme and its role in defending against front running attacks in detail in Section 6.4. Client sends the commitment c for OTP_t to the smart contract via wallet by signing the transaction and then S stores the commitment for verification at a later phase to prevent MITM front running attacks. After providing the commitment, the client now requests the authenticator for the corresponding OTP_t. After getting the OTP from authenticator through air gapped means, the client sends OTP_t to the smart contract through W as shown in Figure 3.

When the confirmation of verifying the signature on blockchain network comes, the first factor authentication is over which establishes the fact that the transaction for request of executing operation is coming from user having private key. After the factor one authentication, the smart contract validates the commitment by hashing the OTP and comparing the resultant hashed ouptut with the stored commitment. After successful verification of commitment, the time range is verified as in $t \in (t_{init}, t_{max}]$. Here, t_{max} represents the maximum time limit with respect to hash chain length k after which the authentication will fail and you will require reinitialisation.

Hence, the $OTP_{t_{max}}$ is the last valid OTP for this hash chain. To check the validity of password, the smart contract uses the stored values, t_{prev} and the OTP_t and generates the resultant expected hash root to do the verification. Equation 4 shows the generation of the validation factor $pverify$ using the OTP_t.

$$pverify = h_{t_{max}-t_{prev}}(h_{t_{max}-t_{prev}-1}(..(h_{t_{max}-t+1}(OTP_t)..) \quad (4)$$

If $pprev = pverify$, then authentication is successful, and the smart contract updates $pprev$ to OTP_t and t_{prev} to t and allows the operation O_t to execute and transfer of crypto tokens as illustrated in Algorithm 4. Updating the values is essential to maintain consistency in the hash chain, as the $rootHash$ value must be shifted left after successful verification. If no successful match is found in $pverify$ and $pprev$, the smart contract rejects the password as well as operation O_t. Now user needs to reinitiate the new operation

and thus the entire process of operation execution will get repeated for the new value of t.

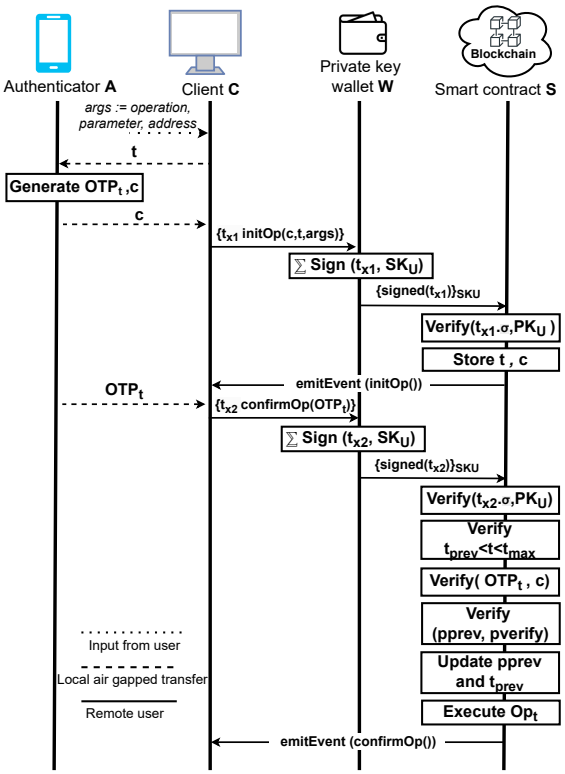

Figure 3: Operation execution (π_E)

Reinitialization: Users who conduct numerous transactions would frequently need to generate new OTPs. So, A limited number of OTPs can pose challenges for both usability and security. We discussed about $OTP_{t_{max}}$ being the last OTP for our hash chain in section 3, when user asks for authentication at $t > t_{max}$, the reinitialisation gets invoked. In our model, reinitialisation requires the client to generate a new seed and salt and then it is shared with authenticator via air gapped means similar to the setup phase, there is no requirement of sharing t_{init} and k as t_{init} is replaced with the t_{max} and k remains same as described in Algorithm 3. Again a new hash chain is generated at the client and the calculated $rootHash_{new}$ is shared with smart contract. The main difference between initial setup and reinitialisation is that, the client only provides the salt and the new $rootHash$ to the smart contract. After required updation in both authenticator and smart contract, the system is again ready to execute operations. The finite length of the hash chain necessitates periodic reinitialization, with the reinitialization frequency being dependent on the system's operation execution rate. Since each operation requires one OTP, a higher number of operation executions will result in greater OTP consumption, thus accelerating the need for hash chain reinitialization. Considering the chain length as $k = 5 * 10^5$ and time step $G = 60$ seconds, it results in a valid hash chain without reintialization for nearly 1 year. Considering that for each minute the system requires an authentication, the given value of k would suffice for a year. Since, key rotation is crucial for updating security and is recommended (For instance, NIST suggests "cryptoperiod" of 1–2 years for private authentication keys. [9]) , we don't consider periodic reinitalisation as a massive limitation of our scheme.

Algorithm 3: Reinitialisation

function $reinit(id_{new}, S_{new})$
  $t_{init} \leftarrow t_{max}, \quad id \leftarrow id_{new}, \quad t_{max} \leftarrow t_{init} + k$
  $rootHash_{new} \leftarrow genRootHash(t_{init}, k, id_{new}, S_{new})$
  $storeVal(t_{init}, k, id_{new}, rootHash_{new})$ // Reintialise smart contract

Algorithm 4: Smart Contract

State variables : $t_{init}, k, id, pprev, t_{prev}, t_{max}, commitment$
transfer(val,addr) : transfer val crypto-tokens from smart contract to addr
balance : current balance of smart contract

function $storeVal(initialTime, numOfOtp, salt, rootHash)$ **public**
  $t_{init} \leftarrow initialTime, \quad k \leftarrow numOfOtp, \quad id \leftarrow salt$
  $pprev \leftarrow rootHash, \quad t_{prev} \leftarrow t_{init}, \quad t_{max} \leftarrow t_{init} + k$
  emitEvent $valuesStored(true)$

function $initOp(c, currentTime, operationArguments)$ **public**
  $commitment \leftarrow c$
  $t \leftarrow currentTime$
  $args \leftarrow operationArguments$
  emitEvent $initOp()$

function $genHashFunc(i, x)$ **view** returns $(32\ byte\ hash)$
  $y \leftarrow t_{init} + k - i$
  $hash \leftarrow sha256(y \parallel id \parallel x)$
  return $hash$

function $confirmOp(OTP_t)$ **public** returns $(bool)$
  $authSuccess = false$
  if $(commitment\ != sha256(OTP))$ **then**
    emitEvent $confirmOp(authSuccess)$
    return $authSuccess$
  if $(t < t_{prev}$ **or** $t > t_{max})$ **then**
    emitEvent $confirmOp(authSuccess)$
    return $authSuccess$
  $pverify = OTP_t$
  for $(i = t_{max} - t + 1; i <= t_{max} - t_{prev}; i++)$ **do**
    $pverify \leftarrow genHashFunc(i, pverify)$
  if $(pprev == pverify)$ **then**
    $pprev \leftarrow OTP_t$
    $t_{prev} \leftarrow t$
    $authSuccess \leftarrow true$
    execute $Op_t(args)$
  emitEvent $confirmOp(authSuccess)$
  return $authSuccess$

function $executeOp_t(args)$ **private**
  if $(operation_t.type == transfer)$ **then**
    check $operation_t.par <= balance$
    $transfer(operation_t.par, operation_t.addr)$

5 Implementation

To evaluate the potential of our system, we made a proof-of-concept implementation of smart contract in solidity compatible with Solidity [7] *version 0.5.0* and higher. Scalability, availability, and compatibility are critical requirements for authentication platforms and hence we used *Ethereum* for public blockchain environment because of its high scalability and large ecosystem as compared to other platforms.

Metamask [4] functioned as the software wallet for private keys, with the injected provider of *Remix IDE* facilitating the seamless integration of Metamask with the smart contract for managing account access and transaction signing. The development of the client and authenticator components was executed in JavaScript. Figure 4 presents a pictorial demonstration of the user interface that we have developed for our implementation. For the communication between the frontend and the blockchain infrastructure, we integrated web3.js [8] version *1.10.0* for streamlined smart contract interactions. Cryptographic hash functions responsible for generating rootHash and OTPs were implemented using the Crypto-JS [1] library version *4.2.0*. For testing purposes, the Sepolia testnet [6] served as the designated testing environment, affording the capability to simulate real-world blockchain scenarios effectively.

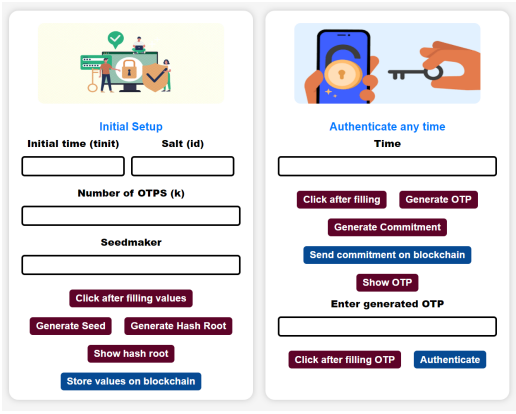

Figure 4: Demo user interface for 2-FA

Due to the increased length of OTPs in our scheme compared to traditional OTPs (128 bits versus 20 bits), encoding them as short numeric codes is impractical. Instead, we explore alternative encoding methods that are more appropriate for these longer password lengths while preserving the air-gapped environment.

QR codes. QR codes have been widely used for second factor authentication to transmit information from the authenticating device to the client device [42–44]. For instance, In Google Authenticator, a website shows a QR code containing the shared secret for the traditional TOTP scheme. The user scans this QR code using their mobile phone, which allows the authenticator app to access the secret. Similarly, in our setup, both the client and the authenticator must have camera setups for two-way scanning. The client encodes the random secret for the authenticator to scan, while the authenticator encodes the 128-bit one-time password and 256-bit commitment as QR codes which is displayed to the client as shown in the Figure 5.

Manual Entry. The QR code method poses challenges with systems without camera. Alternatively, one-time passwords can be generated as arbitrary strings for manual entry by users. Assuming each character in these strings holds 6 bits of entropy (a common measure for case-sensitive alphanumeric strings) and 4 bits (for hexadecimal), the resulting passwords would be composed of 22 characters and 32 characters respectively. While entering these passwords manually may pose some challenges, they remain operationally viable. We have made the source code of our implementation available for the community [3].

Figure 5: QR encodings of OTP and its commitment

6 Results

In this section, we analyze the gas consumption of our model and compare the client storage costs and payload costs with other existing models that use a merkle tree approach. We then validate the security of our model by addressing front-running attacks.

6.1 Payload cost

SmartOTPs [25] utilizes a merkle tree to store OTPs and sends both the OTP and its proof for verification. This approach increases the payload size sent to the blockchain, as it includes not only the OTP but also its proof. As the number of OTPs (leaves in the Merkle tree) grows, the size of the proof also increases logarithmically, resulting in larger payloads. In contrast, our hash chain-based model does not require sending OTP proofs to the smart contract. Therefore, our payload consists solely of the OTP and its commitment, both of which maintain a constant size regardless of the number of OTPs in the hash chain. Figure 6 illustrates the variation in payload size relative to the number of OTPs for both of the models. This efficiency ensures that even with longer hash chains, the payload size remains constant, demonstrating superior performance compared to the SmartOTP model.

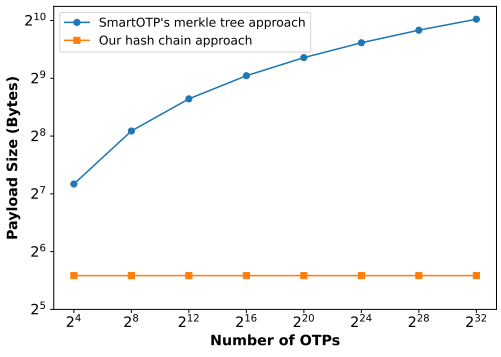

Figure 6: Payload cost comparison

6.2 Client storage

In SmartOTP's merkle tree implementation, generating a proof requires storing the entire merkle tree in client storage, where each leaf corresponds to a hash of an OTP. Consequently, as the number of OTPs increases, so does the size of the entire tree. In the basic version of SmartOTP, storing 2^{20} OTPs requires 33.6 MB of client storage data, a figure that increases significantly even in the improved version with 2^{32} OTPs.

In contrast, our model eliminates the need for client-side storage of hash chains since OTPs are dynamically generated by the authenticator each time an operation is invoked using the variable t. OTPs are securely transferred to the client in an air-gapped manner. By eliminating the need for proof generation, our approach eliminates the entire storage space requirement on the client side as shown in Figure 7. Although storage space is initially required to create the hash chain, but it is subsequently deleted. Therefore, after the hash chain is created, there is zero space requirement for verification and further processes.

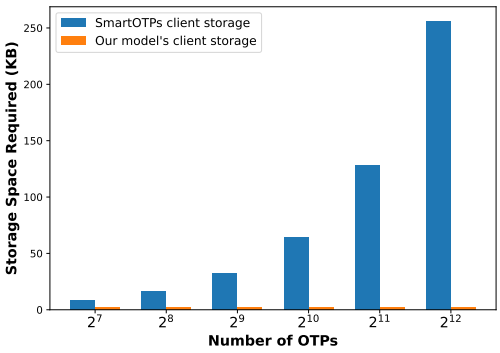

Figure 7: Client storage cost

6.3 Gas consumption

Executing smart contracts on the blockchain involves costs associated with computations and data storage. In the Ethereum Virtual Machine (EVM), these costs are quantified in terms of gas, which reflects the complexity of executing specific instructions. The figure 8 examines the expenses incurred by our approach across various transactions. The deployment of the contract and the *storeValues()* function incur one-time costs when a new hash chain is created and all values are initially stored in the blockchain. On the other hand, the *initOp()* and *confirmOp()* functions involve costs for each operation, which are repeated whenever the new operations are invoked. The costs for these two transactions vary slightly for each operation , although the difference is marginal and not significant depending upon the blockchain network congestion at the time of transaction exceution.

6.4 Front running attacks

As outlined in section 3.1, front-running attacks arise from the intruder's capability to intercept transactions and introduce conflicting actions against the user's intended transactions. Assume

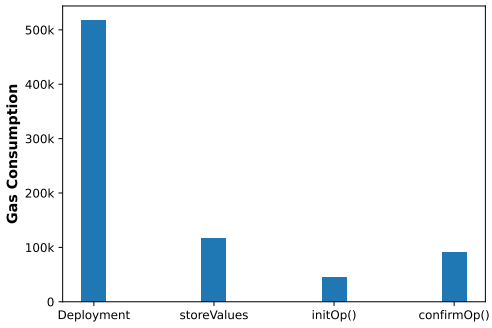

Figure 8: Gas consumption

the case when our setup lacks a commitment scheme. A MITM attacker with access to the private key SK_U can initiate an operation O_i. Being part of the blockchain network, the attacker can monitor all network updates, including transactions in the mempool before they are mined. However, the attacker lacks access to the authenticator and thus cannot directly obtain the OTP.

In this scenario, the attacker initiates operation O_i and waits for the intended user to initiate a follow-up operation O_j where $j > i$. When the intended user initiates the follow-up operation O_j and submits the OTP in the second phase, the OTP, without a commitment, is sent in plaintext and becomes visible to the attacker before mining. Given the block time of approximately 13-15 seconds, the attacker can intercept the OTP and front-run the user's transaction by offering higher transaction fees. This incentivizes miners to prioritize the attacker's transaction over the user's. Consequently, the malicious transaction is executed first, causing the intended transaction to fail or execute later resulting in a successful MITM front-running attack.

To address front-running transactions, we introduced a commit-reveal scheme in which the user first generates a commitment of the OTP before submitting the OTP on the blockchain network. The commitment is generated by hashing the 128-bit OTP to produce a 256-bit commitment value. This commitment is published on the blockchain without revealing the actual OTP, ensuring that even if an intruder intercepts the commitment, they cannot deduce the OTP. In the next step, the user sends the OTP to the smart contract, where the smart contract first verifies the commitment by hashing the OTP and then checks the correctness of the OTP itself. The OTP remains concealed until it is revealed, and by that time, the user's transaction is already being verified. This approach prevents the attacker from front-running the transaction.

One could argue that an intruder might simply duplicate the user's commitment and attempt to front-run the commitment transaction with the anticipation of obtaining the OTP later and front-running it as well. However, given the public nature of the blockchain, the user will also be aware if the intruder has copied their commitment. If the user detects the copied commitment mined in the next block, they will ask the authenticator to generate a new OTP, rendering the previous OTP invalid. Although this cycle could theoretically continue, the intruder incurs substantial fees for front-running each transaction. Repeating this process would rapidly

deplete the intruder's funds, making the attack on this model via front-running financially unsustainable.

7 Discussion and Limitations

We utilize a single OTP for each operation, generating a new OTP for every subsequent operation, and performing verification accordingly. The generated OTP remains valid until the operation is completed, regardless of the time it takes. Consequently, the OTP in the hash chain shifts leftward with each new operation, moving towards the head of the hash chain. This mechanism closely mirrors the HOTP version of traditional authentication architecture, where a counter increments, and the OTP is derived from both the counter and a seed. In our scenario, the entire hash chain nodes function as OTPs, which are derived from a seed. The leftward shift within the hash chain represents the counter increment.

Limitations. Incorporating time as a factor in using OTP, our setup necessitates two transactions to be verified from blockchain for OTP verification along with the network congestion delay. We establish the time step G between t_i and t_{i+1} as 1 minute, signifying the duration for which an OTP remains valid. Therefore, we introduce a time dependency in our setup, making time a crucial validating factor for OTP verification. Within this 1-minute timeframe, the expected increase in block numbers is approximately 4 for the Ethereum blockchain. The average time it takes for a new block to be added to the public blockchain is referred as blocktime which is 13 seconds for Ethereum. It's important to note that the block time and time interval may fluctuate across different blockchain platforms. Given our implementation on the Ethereum test network, we assume a time step of roughly 1 minute, or 4 Ethereum block times. Therefore, if an operation commences at block number x, the OTP_t can only be authenticated within the subsequent block numbers up to $x + 4$. Only OTP_{t+1} can be authenticated between the blocknumber $\in [x + 4, x + 8]$ and hence OTP_t becomes invalid for that time frame which gives our model a similar behaviour as of $TOTP$.

In this model, the authentication cost is contingent upon time. Following the initialization time t_0, for every G interval, t increases by one. Suppose the last successful authentication occurred at time t using OTP_t, in which case $pprev$ must get updated with OTP_t for subsequent authentications. Now, assuming an authentication request arises at arbitrary time x after t, the subsequent OTP utilized for authentication will be OTP_x now. So, now the authentication gap will be $(x - t)$ and thus $(x - t)$ hashes are required to be computed at the smart contract to do the verification of current $rootHash$. Therefore, the authentication cost is dependent upon the gap between successive authentication attempts.

Here, the authentication cost is not same for every *confirmOp()* as it differs for each time x based on the time gap between the last successful authentication t and the current request time x. Hence, If $(x - t)$ is large, the authentication cost will be large and gas consumption will be more. We have done an analysis upon the behaviour of gas consumption with respect to $(x - t)$ in figure 10 which shows the linear relationship in rise of gas consumption with the difference $(x - t)$.

As of July 2024, the Ethereum gas limit per transaction stands at 30 million [2]. Assuming a limit of 15 million and a maximum gap of $(x - t)$ being 4500, with G set at 1 minute, this translates to a maximum authentication gap of 4500 minutes, or roughly 3 days. Therefore, our model can be functional within a maximum authentication gap of 3 days if you are willing to give 1 sepoliaETH as a transaction fee for authentication. We can also observe the transaction fee relative to the authentication time gap in Figure 9.

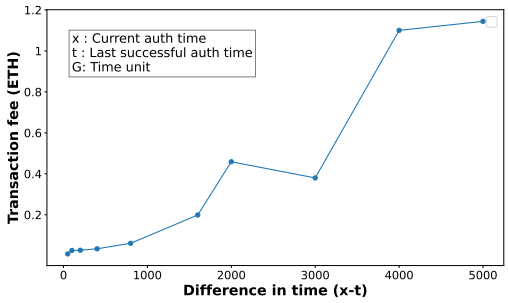

Figure 9: Gas fee analysis for authentication gap

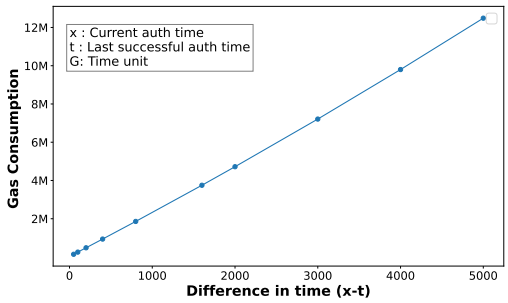

Figure 10: Gas consumption for TOTP setup

8 Conclusion and future work

We have presented a system for smart-contract wallets that offers a safe and practical way to handle cryptocurrency tokens. The framework safeguards against an attacker who can compromise the client or who has access to the user's private key. This is done through a two-factor authentication scheme where an air-gapped authenticator device generates the OTPs. The use of cryptographic hash chains make it possible to implement an HOTP like protocol without a shared secret between the smart contract and the authenticator. Our novel commitment protocol mitigates front-running attacks. The proof of concept implementation of the system demonstrates its feasibility and practicality. We evaluated the entire setup and assessed the gas usage involved in deploying the smart contract as well as its different transactions. While some might argue that employing this model leads to additional fees for transferring funds, but it also provides an entirely secure method for fund transfers.

For future work, there is potential to decrease the length of OTPs, which currently stands at 128 bits, which is larger than traditional OTPs. Reducing it to 20 bits would result in OTP length closer to traditional six digit OTPs, offering practical benefits for air gapped authentications without QR. Additionally, integrating time as a

factor in generating OTPs to accurately reflect the behavior of TOTP could be another improvement in existing framework.

References

[1] 2024. crypto-js. https://github.com/brix/crypto-js Accessed: October 28, 2024.
[2] 2024. Etherscan. https://etherscan.io/blocks Accessed: October 28, 2024.
[3] 2024. Implementation for Enhancing Security in Smart Contract Wallets : An OTP Based 2-Factor Authentication Approach. https://github.com/Kalash1110/Enhancing-Security-in-Smart-Contract-Wallets-An-OTP-Based-2-Factor-Authentication-Approach Accessed: October 28, 2024.
[4] 2024. Metamask. https://github.com/MetaMask/metamask-extension/
[5] 2024. Microsoft Authenticator. https://support.microsoft.com/en-us/account-billing/set-up-an-authenticator-app-as-a-two-step-verification-method-2db39828-15e1-4614-b825-6e2b524e7c95 Accessed: October 28, 2024.
[6] 2024. Sepolia testnet. https://sepolia.etherscan.io/ Accessed: October 28, 2024.
[7] 2024. Solidity. https://github.com/ethereum/solidity
[8] 2024. web3.js. https://github.com/web3/web3.js/releases/tag/v4.0.1-rc.1 Accessed: October 28, 2024.
[9] Elaine Barker and Quynh Dang. 2020. Nist special publication 800–57 part 1, revision 5: Recommendation for key management: Part 1-general, May 2020. *Cited on* (2020), 58.
[10] Binance. 2019. *Binance Security Breach Update.* Technical Report. https://binance.zendesk.com/hc/en-us/articles/360028031711-Binance-Security-Breach-Update
[11] Syeda Tayyaba Bukhari, Muhammad Umar Janjua, and Junaid Qadir. 2024. Secure Storage of Crypto Wallet Seed Phrase Using ECC and Splitting Technique. *IEEE Open Journal of the Computer Society* (2024).
[12] Jean-Pierre Buntinx. 2016. *Brain Wallets Are Not Secure and 'No One Should Use Them.* Technical Report. https://news.bitcoin.com/brain-wallets-not-secure-no-one-use-says-study/
[13] Vitalik Buterin. 2022. *Proof of stake: The making of Ethereum and the philosophy of blockchains.* Seven Stories Press.
[14] Swati Chaudhari, S. S. Tomar, and Anil Rawat. 2011. Design, implementation and analysis of multi layer, Multi Factor Authentication (MFA) setup for webmail access in multi trust networks. In *2011 International Conference on Emerging Trends in Networks and Computer Communications (ETNCC).* 27–32. https://doi.org/10.1109/ETNCC.2011.5958480
[15] Vincent Chia, Pieter Hartel, Qingze Hum, Sebastian Ma, Georgios Piliouras, Daniël Reijsbergen, Mark Van Staalduinen, and Pawel Szalachowski. 2018. Rethinking blockchain security: Position paper. In *2018 IEEE International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData).* IEEE, 1273–1280.
[16] Don Coppersmith and Markus Jakobsson. 2003. Almost optimal hash sequence traversal. In *Financial Cryptography: 6th International Conference, FC 2002 Southampton, Bermuda, March 2002 Revised Papers 6.* Springer, 102–119.
[17] Sergey Gorbunov Dan Boneh, Riad S. Wahby. 2019. *RFC Internet-Draft: BLS signature.* Technical Report. https://tools.ietf.org/html/draft-irtf-cfrg-bls-signature-00
[18] Sanchari Das, Bingxing Wang, Andrew Kim, and L Jean Camp. 2020. MFA is A Necessary Chore!: Exploring User Mental Models of Multi-Factor Authentication Technologies.. In *HICSS.* 1–10.
[19] Chris Dods, Nigel P Smart, and Martijn Stam. 2005. Hash based digital signature schemes. In *Cryptography and Coding: 10th IMA International Conference, Cirencester, UK, December 19-21, 2005. Proceedings 10.* Springer, 96–115.
[20] Mohamed Hamdy Eldefrawy, Khaled Alghathbar, and Muhammad Khurram Khan. 2011. OTP-Based Two-Factor Authentication Using Mobile Phones. In *2011 Eighth International Conference on Information Technology: New Generations.* 327–331. https://doi.org/10.1109/ITNG.2011.64
[21] Shayan Eskandari, Seyedehmahsa Moosavi, and Jeremy Clark. 2020. Sok: Transparent dishonesty: front-running attacks on blockchain. In *Financial Cryptography and Data Security: FC 2019 International Workshops, VOTING and WTSC, St. Kitts, St. Kitts and Nevis, February 18–22, 2019, Revised Selected Papers 23.* Springer, 170–189.
[22] Li Gong. 1989. Using one-way functions for authentication. *ACM SIGCOMM Computer Communication Review* 19, 5 (1989), 8–11.
[23] Vipul Goyal. 2004. How to re-initialize a hash chain. *Cryptology ePrint Archive* (2004).
[24] Neil M. Haller. 1995. *The S/KEY One-Time Password System.* RFC 1760. Network Working Group. https://www.rfc-editor.org/rfc/rfc1760.html
[25] 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 *Proceedings of the 2nd ACM Conference on Advances in Financial Technologies.* 145–162.
[26] Yih-Chun Hu, Markus Jakobsson, and Adrian Perrig. 2005. Efficient constructions for one-way hash chains. In *Applied Cryptography and Network Security: Third International Conference, ACNS 2005, New York, NY, USA, June 7-10, 2005. Proceedings 3.* Springer, 423–441.
[27] 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.* 983–999.
[28] Massimo La Morgia, Alessandro Mei, Francesco Sassi, and Julinda Stefa. 2023. The doge of wall street: Analysis and detection of pump and dump cryptocurrency manipulations. *ACM Transactions on Internet Technology* 23, 1 (2023), 1–28.
[29] Leslie Lamport. 1981. Password authentication with insecure communication. *Commun. ACM* 24, 11 (1981), 770–772.
[30] Frank T Leighton and Silvio Micali. 1995. Large provably fast and secure digital signature schemes based on secure hash functions. US Patent 5,432,852.
[31] Pat Litke and Joe Stewart. 2014. The cryptocurrency-stealing malware landscape. *Secureworks* (Feb 2014). https://www.secureworks.com/research/cryptocurrency-stealing-malware-landscape
[32] Ephrat Livni. 2022. Binance blockchain hit by $570 million hack, exposing crypto vulnerabilities. *The New York Times* (Oct 2022). https://www.nytimes.com/2022/10/07/business/binance-hack.html
[33] Ralph C Merkle. 1989. A certified digital signature. In *Conference on the Theory and Application of Cryptology.* Springer, 218–238.
[34] Tyler Moore and Nicolas Christin. 2013. Beware the middleman: Empirical analysis of Bitcoin-exchange risk. In *Financial Cryptography and Data Security: 17th International Conference, FC 2013, Okinawa, Japan, April 1-5, 2013, Revised Selected Papers 17.* Springer, 25–33.
[35] David M'Raihi, Mihir Bellare, Frank Hoornaert, David Naccache, and Ohad Ranen. 2005. *Hotp: An hmac-based one-time password algorithm.* RFC 4226. Network Working Group, Internet Engineering Task Force (IETF). https://www.rfc-editor.org/rfc/rfc4226.html
[36] David M'Raihi, Johan Rydell, Mingliang Pei, and Salah Machani. 2011. *TOTP: Time-Based One-Time Password Algorithm.* RFC 6238. Network Working Group, Internet Engineering Task Force (IETF). https://www.rfc-editor.org/rfc/rfc6238.html
[37] Sabout Nagaraju and Latha Parthiban. 2015. Trusted framework for online banking in public cloud using multi-factor authentication and privacy protection gateway. *Journal of Cloud Computing* 4 (2015), 1–23.
[38] Satoshi Nakamoto. 2008. Bitcoin: A peer-to-peer electronic cash system. (2008).
[39] Rushikesh Nikam and Manish Potey. 2016. Cloud storage security using multi-factor authentication. In *2016 international conference on recent advances and innovations in engineering (ICRAIE).* IEEE, 1–7.
[40] Arthur AB Pessa, Matjaž Perc, and Haroldo V Ribeiro. 2023. Age and market capitalization drive large price variations of cryptocurrencies. *Scientific reports* 13, 1 (2023), 3351.
[41] Reuters. 2016. Bitcoin Worth $72M Was Stolen in Bitfinex Exchange Hack in Hong Kong. *The Reuters* (Aug 2016). https://fortune.com/2016/08/03/bitcoin-stolen-bitfinex-hack-hong-kong/
[42] SecurEnvoy. 2017. *SecureEnvoy Overview Presentation.* Technical Report. https://www.securenvoy.com/animations/overview/animations.shtm/oneswipe
[43] Maliheh Shirvanian, Stanislaw Jarecki, Nitesh Saxena, and Naveen Nathan. 2014. Two-Factor Authentication Resilient to Server Compromise Using Mix-Bandwidth Devices. In *21st Annual Network and Distributed System Security Symposium, NDSS 2014.*
[44] Guenther Starnberger, Lorenz Froihofer, and Karl M Göschka. 2009. QR-TAN: Secure mobile transaction authentication. In *2009 International Conference on Availability, Reliability and Security.* IEEE, 578–583.
[45] Anton Wahrstätter, Jorão Gomes, Sajjad Khan, and Davor Svetinovic. 2023. Improving cryptocurrency crime detection: Coinjoin community detection approach. *IEEE Transactions on Dependable and Secure Computing* 20, 6 (2023), 4946–4956.
[46] Bryan White, Aniket Mahanti, and Kalpdrum Passi. 2022. Characterizing the OpenSea NFT marketplace. In *Companion Proceedings of the Web Conference 2022.* 488–496.
[47] Gavin Wood et al. 2014. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper* (2014). https://ethereum.github.io/yellowpaper/paper.pdf