An Enhanced Long-term Blockchain Scheme Against Compromise of Underlying Cryptography

Long Meng, Liqun Chen

ABSTRACT

Blockchain is a decentralized ledger applying the peer-to-peer (P2P) network, cryptography and consensus mechanism over distributed network. Especially, the underlying cryptographic algorithms protect the blockchain integrity and data authenticity. However, it is well-known that every single algorithm is associated with a limited lifespan due to the increasing computational power of attackers. The compromise of algorithms directly leads to the compromise of blockchain validity. There are two existing long-term blockchain schemes dealing with this problem, but we observe that in these schemes: 1) the calculation of block hash values is not compatible with existing blockchains; 2) the hash transition procedure is only specified from the first algorithm to the second one, there are multiple possibilities to implement the scheme for a longer time, some of them may lead to the failure of the scheme; 3) the security of their schemes are not formally analyzed and proved. In this paper, we propose an enhanced long-term blockchain scheme as a solution to issue 1 and 2, and we formally prove that our scheme is secure without the limitation of cryptographic algorithms. Besides, we implement our scheme, the results show that our hash transition procedure can be completed between 20 minutes (best case) and several hours (worst case) for a current Bitcoin and Ethereum blockchain, which is very efficient.

KEYWORDS

Blockchain, Cryptographic protocols, Hash functions, Digital signatures, Long-term security

1 INTRODUCTION

In the recent years, the blockchain technology has been widely adopted in various application fields, especially the emerging cryptocurrencies such as "Bitcoin" [15], "Ethereum" [6, 21], "Litecoin" [4], "Ripple" [3] etc. Blockchain is a decentralized ledger that stores information as a chain of blocks, the main technology under blockchain include the P2P network, cryptography and consensus mechanism over distributed network, the security of blockchain relates to the security of each component. Especially, cryptography plays an important role in blockchain security. In specific, blockchain makes use of signature schemes to protect the authenticity and integrity of block data, and of hash functions to prevent the tampering of blocks and guarantee the order of blocks. Therefore, the security of blockchain is associated with the security of underlying cryptographic algorithms.

However, it is well-known that any single algorithm has a limited lifespan due to the limited operational life cycle or increasing computational power of attackers. For instance, the upcoming quantum computers are considered to break most of the broadly-used signature algorithms [19] and to increase the speed of attacking hash functions [11]. For a blockchain needs to be maintained for decades or even permanently, the blockchain security will be threatened after the cryptographic algorithms are no longer secure.

In this paper, we discuss how to make a blockchain holding long-term security. For the purpose of this work, if a blockchain is secure in a long period of time that is not bounded with the lifetimes of its underlying cryptographic algorithms, we say that the blockchain is *long-term secure.* If a blockchain is long-term secure, we refer to it as a *long-term blockchain* (LTB). The idea to build such a solution is to transfer the weak algorithms to secure ones before they are actually compromised, but it is not trivial due to the complexity of a blockchain system and the sensitivity of the renewal time.

The impact of broken cryptographic primitives in Bitcoin has been analyzed by Giechaskiel et al. [9]. They claimed that the compromise of hash functions and signature schemes leads to the problems of stealing coins, double spending, repudiated payments, changing existing payments etc. These results can be extended to other blockchains that huge security problems will occur if the underlying cryptographic algorithms are broken.

In 2017, Sato et al. proposed the first LTB scheme to renew the underlying algorithms in public blockchains [18]. The main idea is to compute new hash values of previous blocks using stronger hash functions and archive the new hash values in future blocks, and each user's asset should be transferred to a new signature pair before the old signature scheme is broken. After that, an improved LTB scheme [7] proposed by Chen et al. to avoid the hard fork caused by the hash transition procedure in [18].

We observe three issues in the existing LTB schemes [7, 18]: 1) the computation of block hash values is not compatible with existing blockchains, 2) the schemes only describe the transition procedure from the first hash function to the second one, but how to extend this procedure to the further transition process is not claimed clearly and several possible cases could happen, some of them may cause the failure of the LTB scheme, and 3) the security analysis is not given for both schemes, whether the schemes are secure or not becomes a question. That is the reason why issue 2 exists.

In this paper, we proposed an enhanced LTB scheme that addresses the technical issues 1 and 2 in the existing schemes [7, 18], and we provide a formal security model to analyze our proposed scheme. We proved that our scheme is secure in long-term periods that are not bounded with the lifetimes of underlying cryptographic algorithms. Finally, we implement our proposed scheme and evaluate the performance. We surprisingly find that for current Bitcoin blockchain, a hash transition can be finished only within 20 - 70 minutes; for current Ethereum blockchain, a hash transition can be completed in 38 - 158 minutes. These results show that our scheme is very efficient and practical.

2 PRELIMINARIES

2.1 Hash functions

A secure hash function [1] maps a string of bits of variable (but usually upper bounded) length to a fixed-length string of bits, satisfying the following three properties:

- *Preimage Resistance*: it is computationally infeasible to find, for a given output, an input which maps to this output.
- *Second Preimage Resistance*: it is computationally infeasible to find a second input which maps to the same output.
- *Collision Resistance*: it is computationally infeasible to find any two distinct inputs which map to the same output.

2.2 Digital signature schemes

A signature scheme [10] is a tuple of probabilistic polynomial-time algorithms (Gen, Sign, Vrfy) satisfying the following:

(1) $(sk, pk) \leftarrow \mathrm{Gen}()$: The key generation algorithm generates a secret private key sk and a public verification key pk.
(2) $s \leftarrow \mathrm{Sign}(sk, m)$: The signing algorithm takes as input a private key sk and a message m. It outputs a signature s.
(3) $b \leftarrow \mathrm{Vrfy}(pk, m, s)$: The verification algorithm takes as input a public key pk, a message m, and a signature s. It outputs a bit $b = 1$ if s is valid, it outputs $b = 0$ if s is invalid.

Let S = (Gen, Sign, Vrfy) be a signature scheme, and consider the following signature experiment $\mathrm{Sig{-}forge}^{cma}_{S,A}(n)$ for an adversary A and parameter n:

(1) $\mathrm{Gen}(1^n)$ is run to obtain keys (pk, sk).
(2) Adversary A is given pk and oracle access to $Sign(\cdot)$. This oracle returns a signature $\sigma \leftarrow \mathrm{Sign}(sk, m)$ for any message m of the A's choice. The adversary then outputs (m, σ).
(3) Let Q denote the set of messages whose signatures were requested by A during its execution. The output of the experiment is 1 if $m \notin Q$ and $\mathrm{Vrfy}(pk, m, \sigma) = 1$.

Definition 2.1. (Unforgeability.) A signature scheme S is existentially unforgeable under an adaptive chosen-message attack if for all probabilistic polynomial-time adversaries $\mathcal{A}$, there exists a negligible function $negl$ such that $\Pr[\mathrm{Sig{-}forge}^{cma}_{S,A}(n) = 1] \leq negl(n)$.

2.3 Merkle trees

Merkle tree is an efficient and secure structure for verification of large amount of data [14], Fig. 1 shows an example Merkle tree for 4 data items D_1, D_2, D_3, D_4. First, the lowest leaf nodes h_1, h_2, h_3, h_4 are the hash values of D_1, D_2, D_3, D_4 computed through hash function H_0 respectively. Second, the parent node h_5 is the hash result of the concatenation h_1 and h_2 through H_0, so as to h_6 from h_3 and h_4. Finally, the concatenation of h_5 and h_6 is hashed through H_0 to obtain the root hash value h_r.

In order to verify a data item is a part of the root hash value, the used hash function, the neighbor nodes, and the concatenation vectors are required as a hash path to recompute the root hash value. For instance, to verify that the data item D_2 is a part of the root hash value h_r, we need to reconstruct the Merkle tree with hash function H_0 and a hash path $c_2 = ((\text{left}, h_1), (\text{right}, h_6))$.

2.4 Blockchains

Blockchains are distributed digital ledgers of cryptographically signed transactions that are grouped into blocks. Each block is linked to the previous one by cryptographic hash functions after validation and undergoing a consensus decision [22]. In specific, blockchains are comprised of blocks, each block is comprised of a block header and block data. As Fig. 2 shows, a block header contains the block number, a nonce, the hash value of the previous block header, a time-stamp, and a Merkle tree root hash value of all block data. The block data contains a list of transactions along with digital signatures within the block.

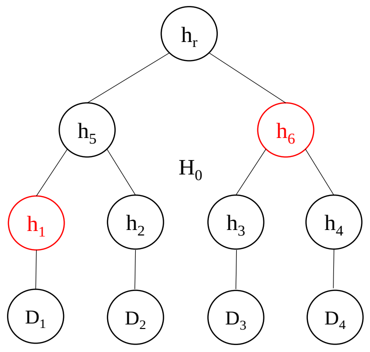

Figure 1: Computation of a Merkle tree root hash value

Blockchains can be categorized based on their permission models [12]: a permissionless blockchain is open to anyone publishing blocks without needing permission from any authority (e.g., Bitcoin, Ethereum), a permissioned blockchain is private and limit to a number of trusted entities that got permission to join the network in order to validate transactions (e.g., Hyperledger [2]).

In a permissionless blockchain, the proof-of-work (PoW) consensus model is usually used to reach agreements of accepting new blocks. In PoW model, miners publishes the next block by being the first to solve a computationally intensive puzzle. Commonly, the puzzle is to compute the hash value of a block header, and find one be less than a target value by changing the nonce field in the header. Changes to a blockchain protocol and data structures are called forks. If the change to a blockchain is not backward compatible, it is called a hardfork; otherwise, it is called a soft fork [22].

Blockchain technology utilizes cryptographic hash functions and signature schemes. As the block B_i in Fig. 1, each transaction is signed by the user who initiates the transaction, then all the transaction and signature pairs $(\mathrm{Tx}_{i1}, \mathrm{Sig}_{i1})$, ..., $(\mathrm{Tx}_{ij}, \mathrm{Sig}_{ij})$ in the block are aggregated together by using a Merkle tree, with only the root hash value $mkroot_i$ stored in the block header for simplified verification [15]. The block header is then hashed into a hash value h_i that is stored in the block header of the next block B_{i+1}. The signatures enables the network nodes to verify the integrity and authenticity of transactions, the chaining of hash values between blocks protects the integrity of all block data.

3 REVIEW THE EXISTING LTB SCHEMES

In this section, we review the details of the two existing LTB schemes [7, 18] and discuss their issues.

3.1 The Sato et al. scheme [18]

In 2017, Sato et al. proposed the first LTB scheme that describes the transition procedure from a compromised hash function or signature scheme to a secure one [18]. The transition procedure can be divided into a basic version and a supplement version:

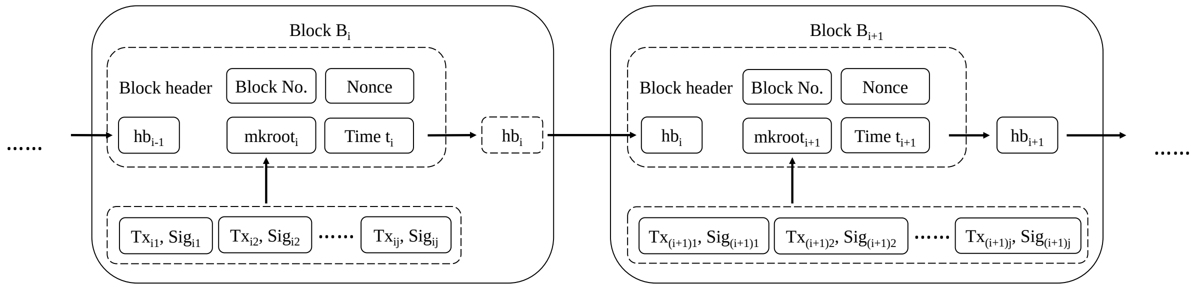

Figure 2: The general structure of a blockchain

Basic procedure: let M denote the number of existing blocks in the blockchain using hash function H_1, $b_i (i \in \{1, M\})$ denote the i-th block. Let $[tx_{ij}, sig_{ij}]$ denote the j-th transaction and signature pair in block b_i ($j \in \{1, N\}$, tx_{iN} denote the last transaction in b_i), hb_i represent the hash value of block b_i, $mkroot_i$ represent the Merkle tree root hash value for all transaction and signature pairs in block b_i. Then the generation of block b_i can be described as:

(1) Compute Merkle tree root value $mkroot_i$ for $[tx_{i1}, sig_{i1}]$, ..., $[tx_{iN}, sig_{iN}]$ in b_i using H_1.
(2) Compute the hash value of the previous block b_{i-1} using H_1: $hb_{i-1} = H_1(b_{i-1})$.
(3) Construct $b_i = (hb_{i-1}, mkroot_i, [tx_{i1}, sig_{i1}], ..., [tx_{iN}, sig_{iN}])$.

Now assume H_1 is threatened but still secure, there is a stronger hash function H_2. The transition scheme starts from $i = M+1$, the previous M blocks are divided into r sets of blocks, each set contains s blocks, i.e., $M = r \times s$. Let $b'_k (k \geq 1)$ be the index of new block generated by H_2, let $tx'_{kj} (j \in \{1, N\})$ be the transaction in block b'_k and sig'_{kj} be the signature for tx'_{kj}. Then b'_k is generated as:

(1) Calculate a new hash value of a group of previous blocks using H_2: $archiveHash_k = H_2(b_{(k-1)s+1}, b_{(k-1)s+2}, ..., b_{(k-1)s+s}, hb_{(k-1)s+s})$.
(2) Compute Merkle tree root value $mkroot'_k$ for $[tx'_{k1}, sig'_{k1}]$, ..., $[tx'_{kN}, sig'_{kN}]$ using H_2.
(3) Compute the hash value of the last block b'_{k-1} using H_2: $hb'_{k-1} = H_2(b'_{k-1})$. When $k=1$, $hb'_{k-1} = H_2(b_M)$.
(4) Construct block $b'_k = (archiveHash_k, hb'_{k-1}, mkroot'_k, [tx'_{k1}, sig'_{k1}], ..., [tx'_{kN}, sig'_{kN}])$.

Compare a transition block b'_k with a original block b_i, in block b'_k, the new Merkle tree root value and block hash value are determined using H_2, and there is a new field $archiveHash_k$ in its block header, which is a new hash value of a group of original blocks using H_2. After the generation of block b_r, the transition from block b_1 to b_M is completed.

The verification procedures of b'_k: 1) calculate the Merkle tree root value from $[tx'_{k1}, sig'_{k1}]$,..., $[tx'_{kN}, sig'_{kN}]$, check if it equals to $mkroot'_k$; 2) calculate $H_2(b'_{k-1})$ from b'_{k-1}, check if it equals to hb'_{k-1}; 3) calculate $H_2(b'_k)$, check if it equals to hb'_k in b'_{k+1}; 4) retrieve $b_{(k-1)s+1}$, $b_{(k-1)s+2}$, ..., $b_{(k-1)s+s}$, $hb_{(k-1)s+s}$ to calculate $archiveHash_k$, check if $archiveHash_k$ is equal to the one in b'_k; 5) check if hash value of $b_{(k-1)s+p} (p \in \{1, s\})$ with H_1 is same as $hb_{(k-1)s+p}$ in block $b_{(k-1)s+p+1}$.

The signature transition happens when the signature scheme is threatened but not practically broken. In specific, each user applies a stronger signature scheme, generates a new key pair, then transfer assets or status to the new key pair.

Supplement procedure: Sato et al. claimed that in their hash transition scheme, the overhead of the additional field *archiveHash* decreases the number of transactions in new blocks. To deal with this issue, they proposed a supplement procedure: the new field of *archiveHash* is adding to a second chain called a supply chain, which is maintained by the same or a part of miners or the original chain, while block structure of original chain remains the same as before. The proof-of-work competition is applied to one of these two chains, and both chains store the same transactions after completion of all *archiveHash*. Transactions verification is only conducted in original chain for most of time, the support chain is used for the verification process when a dispute occurs.

3.2 The Chen et al. scheme [7]

In 2018, Chen et al. argued that the hash transition in scheme [18] is a hardfork in proof-of-work blockchains [7], which addresses both the basic procedure and the supplement procedure. They stressed that the hardfork may cause disagreement and split in the blockchain community, and they proposed an improved scheme to solve this issue as follows:

Transition procedure: assume there are M original blocks generated using H_1 with the same process as Section 3.1. The scheme starts from b_{M+1}, and the previous M blocks are divided into r sets with s blocks in each set. Let $target_k$,$nonce_k$, and tsp_k separately represent the mining target for b_k using hash function H_1, the field that can be filled with random value to meet $target_k$, and the time-stamp stored in b_k. Let $target'_k$,$nonce'_k$, and tsp'_k denote the corresponding parameters for proof of work with H_2. The generation process of a new block $b'_{M+k} (k \geq 1)$ is listed as follows:

(1) Construct an inner block b_{M+k} using H_1: $b_k = (hb_{M+k-1}, mkroot_{M+k}, [tx_{(M+k)1}, ..., tx_{(M+k)N}], target_{M+k}, nonce_{M+k}, tsp_{M+k})$.
(2) Solve the proof-of-work puzzle using H_1 so that satisfies $target_{M+k}$: $H_1(b_{M+k}) \leq target_{M+k}$.
(3) Compute $archiveHash_k = H_2(b_{(k-1)s+1}, b_{(k-1)s+2}, ..., b_{(k-1)s+s})$.
(4) Compute the hash value of the last block using H_2: $hb'_{M+k-1} = H_2(b_{M+k-1})$, where $hb'_{M+k-1} = H_2(b_M)$ for $k=1$.

(5) Compute Merkle tree root value $mkroot'_{M+k}$ for $[tx_{(M+k)1}, ..., tx_{(M+k)N}]$ using H_2.
(6) Construct outer block $ob_k = (hb'_{M+k-1}, mkroot'_{M+k}, target'_{M+k}, nonce'_{M+k}, tsp'_{M+k})$.
(7) Construct block $b'_{M+k} = (archiveHash_k, b_{M+k}, ob_{M+k})$.
(8) Solve the proof-of-work puzzle using H_2 so that satisfies $target'_k$: $H_2(b'_{M+k}) \leq target'_{M+k}$.

The verification procedures of block b'_{M+k} include to: 1) calculate $archiveHash_k$ from original M blocks, and check whether it is equal to the one stored in b'_{M+k}; 2) calculate the Merkle tree root value from $[tx_{(M+k)1}, ..., tx_{(M+k)N}]$, and check whether it is equal to $mkroot'_{M+k}$; 3) calculate $H_1(b_{M+k})$ and check if it meets the $target_{M+k}$; 4) calculate $H_2(b'_{M+k})$ and check if it meets the target $target'_{M+k}$; 5) check if $H_2(b'_{M+k-1})$ is equal to hb'_{M+k-1} in b'_{M+k}; and 6) check if $H_2(b'_{M+k})$ is equal to hb'_{M+k} in b'_{M+k+1}.

The improved scheme constructs two layers of proof-of-work separately using H_1 and H_2, so that the scheme overcomes the hardfork problem as it is backward compatible with old miners who still use H_1. Besides, the scheme provides stronger security than the one-layer proof-of-work against malicious split attack.

3.3 Discussions

After the overview of the two existing LTB schemes, we have following observations:

Issue 1: the computation of block hash value is not compatible with the existing blockchains in the world. In both schemes, every block hash value is calculated as the hash value of the whole block, e.g., $hb_i = H_1(b_i)$, $b_i = (hb_{i-1}, mkroot_i, [tx_{i1}, sig_{i1}], ..., [tx_{iN}, sig_{iN}])$. They defined hb_i as the "proof of existence" of block b_i, and it is this hash value stored in the next block b_{i+1}. This is not the case of a general blockchain structure. As Section 2.4 shows, a block hash value is the hash value of only the block header rather than include the block data. In this case, their schemes may not be easily adopted for many blockchains at the implementation stage.

Issue 2: both schemes only specify the hash transition procedure from H_1 to H_2. It is not clearly displayed how to extend this procedure for further transition process, there are multiple possible methods to implement the LTB scheme, some of them may cause the failure of a long-term blockchain. For instance, after the transition from H_1 to H_2, the validity of b_1, ..., b_M has been maintained by all *archiveHash* fields stored in b_{M+1}, ..., b_r. Then for a specific transaction tx stored in block $b_i (i \in \{1, M\})$, the computation of *archiveHash* can be regarded as $H_2(tx, H_1(tx))$.

Now we assume H_2 is threatened, the blockchain needs to transfer all blocks to a stronger hash function H_3, the validity of blocks under H_2 can be renewed as the same steps from H_1 to H_2, but the procedure to extend the validity of blocks b_1, ..., b_M under H_1 could be two possible cases: 1) recompute the *archiveHash* of b_1, ..., b_M using H_3 and store them in future blocks, but at the time of calculation, H_1 might has been compromised, then block data in b_1, ..., b_M could have been tampered and cannot be trusted anymore; 2) compute the *archiveHash* of blocks b_{M+1}, ..., b_r since the *archiveHash* fields in these blocks maintain the validity of b_1, ..., b_M, i.e., $archiveHash_2 = H_3(archiveHash_1)$. However, this transition method for transaction tx can be regarded as $H_3(H_2(tx, H_1(tx))$. After H_2 is compromised, the $(tx, H_1(tx))$ pair is vulnerable to collision attacks that it can be modified to another pair $(tx', H_1(tx'))$ with the same hash values to $H_2(tx, H_1(tx))$, which means b_1, ..., b_M are no longer valid after the broken of H_2.

Issue 3: the formal security analysis of both schemes are not presented. Since blockchain is a complex system that utilizes hash functions and signature schemes in many places, the transition scheme for each algorithm should be secure. Any single place of broken will lead to the compromise of blockchain validity. The existence of Issue 2 is an example, if such a transition scheme is not analyzed under a formal security model, the LTB scheme may be vulnerable to security failures or attacks.

In following sections, we will propose an enhanced LTB scheme with following improvements compared to the existing schemes: 1) our scheme is compatible with the existing blockchain structure; 2) our scheme describes how to implement the hash transition and signature procedure from a long-term view; 3) we propose a formal security model for a LTB scheme and analyze the security of our proposed scheme under the model.

4 THE PROPOSED LTB SCHEME

In our proposed LTB scheme, there are three entities: a set of users, a blockchain system, and a verifier. The scheme is comprised of a hash transition procedure and a signature transition procedure. The hash transition procedure is performed by the blockchain system, which target is to extend the validity of blocks when the security of underlying block hash functions or Merkle tree hash functions are threatened. The signature transition procedure is performed by each user, which purpose is to extend the validity of users' signatures on their transactions or assets when the signature schemes are about to be compromised. Thus, the proposed scheme has a tuple of algorithms for each procedure introduced in Section 4.1 and 4.2 separately. The notation is shown in Table 1.

Our scheme is not limited to a specific type of blockchain, hence it is not constructed as [7] to solve the hardfork issue. However, if our scheme is implemented on a proof-of-work blockchain, we recommend the approach of [7] to create a two-layer proof-of-work so that the scheme is backward compatible with the old algorithm.

4.1 Hash transition procedure

The hash transition procedure is consist of three algorithms (BGen, BRen, HVer), which separately stands for block generation, block renewal and hash verification. The algorithm BGen and BRen are implemented for generating blocks as the timeline shown in Fig. 3. Let M be the total number of hash transitions, the timeline is divided into a total of 2M+1 phases (M could be continuously on the timeline):

- Phase 0 ($t \in [t_0, t'_0]$): let H_0 be a secure hash function, blocks b_1, ..., b_{F_0} are generated by algorithm BGen using H_0.
- Phase 1' ($t \in [t'_0, t_1]$): H_0 becomes weak but still secure, let H_1 be a stronger hash function. The previous F_0 blocks are divided into r_1 sets, with s_1 blocks in each set. i.e., $P_1 = F_0 = r_1 \times s_1$. Blocks b_{P_1+1}, ..., $b_{P_1+r_1}$ are generated by algorithm BRen using H_1.
- Phase 1 ($t \in [t_1, t'_1]$): H_0 could already be compromised, H_1 is still secure. Blocks $b_{P_1+r_1+1}$, ..., $b_{P_1+r_1+F_1}$ are generated by algorithm BGen using H_1.

Hash transition notation

$M \in \mathbb{N}$	total number of hash transition	$m \in [0, M]$	index number of hash transition
Phase m	the phase when BGen takes place	Phase m'	the phase when BRen takes place
t_m	time point when BGen starts	t'_m	time point when BRen starts
blc	the blockchain used in the LTB scheme	H_m	the m-th hash function used in the blockchain
b_k	the k-th block in blc	bh_k, bd_k	the block header and block data of b_k
$(H_m, t) \to \text{VD}$	H_m is secure at time t	hb_k	hash value of bh_k
$j \in \{1, J\}$	index number of transaction in a block	tx_{kj}	the j-th transaction in b_k
sig_{kj}	signature of tx_{kj}	$mkroot_k$	Merkle tree root value of block data in b_k
r_m	the set number of m-th hash transition	s_m	number of blocks in set r_m
P_m	number of blocks before m-th hash transition	VD	verification data used in HVer and SVer algorithms
F_m	number of blocks generated by BGen algorithm using H_m	t_v	the verification time
$archiveHash_{mk}$	the archive hash value stored in b_k calculated using H_m	ts_k	time-stamp in b_k

Signature transition notation

$N \in \mathbb{N}$	total number of signature transition	$n \in [0, N]$	index number of signature transition
Phase n	the phase when SGen takes place	Phase n'	the phase when SRen takes place
t_n	time point when SGen starts	t'_n	time point when SRen starts
S_n	the n-th signature scheme used by the user	pk_n	the public key associated with S_n
$tx_{n \to n}$	transactions between two addresses generated by S_n	s_n	the signature of $tx_{n \to n}$
$tx_{(n-1) \to n}$	transactions from address generated by S_{n-1} to the one by S_n	s'_n	the signature of $tx_{(n-1) \to n}$
STX_n	signed transaction of $tx_{n \to n}$	STX'_n	signed transaction of $tx_{(n-1) \to n}$
$(S_n, t) \to \text{VD}$	S_n is secure at time t		

Table 1: Notation

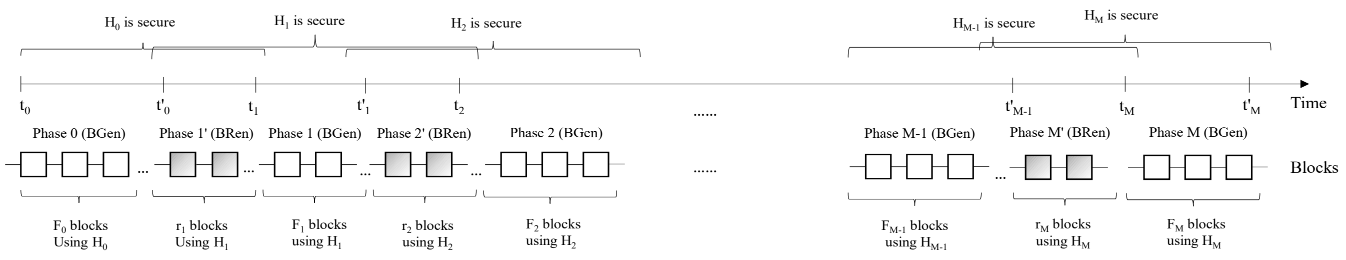

Figure 3: The timeline of hash transition scheme

- ...
- Phase M' ($t \in [t'_{M-1}, t_M]$): H_0, ..., H_{M-2} could already be compromised, H_{M-1} is threatened but still secure, let H_M be a stronger hash function. There are total P_M previous blocks, i.e., $P_M = P_{M-1} + r_{M-1} + F_{M-1}$. The P_M blocks are divided into r_M sets, with s_M blocks in each set. i.e., $P_M = r_M \times s_M$. Blocks $b_{P_M+1}, ..., b_{P_M+r_M}$ are generated by algorithm BRen using H_M.
- Phase M ($t \in [t_M, t'_M]$): H_0, ..., H_{M-1} could be compromised, H_M is still secure. Blocks $b_{P_M+r_M+1}$, ..., $b_{P_M+r_M+F_M}$ are generated by algorithm BGen using H_M.

BGen: $b_k \leftarrow \text{BGen}(H_m;\ bd_k,\ bh_{k-1})$. The block generation algorithm BGen takes place at Phase m, i.e., $t \in [t_m, t'_m] (m \in [0, M])$. For $k \in [P_m + r_m + 1,\ P_m + r_m + F_m]$ (P_0 and r_0 do not exist), BGen takes input the block data $bd_k = ([tx_{k1}, sig_{k1}], ..., [tx_{kJ}, sig_{kJ}])$, the block header of block b_{k-1}, outputs a block b_k using H_m. The generation process of b_k is listed as below.

(1) $sig_{k1}, ..., sig_{kJ}$ are the signatures of transactions $tx_{k1}, ..., tx_{kJ}$ generated by one or a set of secure signature schemes, here we collectively denote them as S. i.e., $sig_{k1} \leftarrow S(tx_{k1}), ..., sig_{kJ} \leftarrow S(tx_{kJ})$.
(2) Compute Merkle tree root value of block data bd_k using H_m: $mkroot_k \leftarrow MT(H_m; bd_k)$.
(3) Compute the hash value of the block header of b_{k-1} using H_m: $hb_{k-1} = H_m(bh_{k-1})$.
(4) Construct the block header of b_k: $bh_k = (hb_{k-1}, mkroot_k, ts_k)$.
(5) Construct block $b_k = (bh_k, bd_k)$.

BRen: $b_k \leftarrow \text{BRen}(H_m;\ bd_k,\ bh_{k-1},\ b_{(k-P_m-1)s_m+1}, ..., b_{(k-P_m)s_m})$. The block renewal algorithm BRen takes place at Phase m', i.e., $t \in [t'_{m-1}, t_m] (m \in [1, M])$. For $k \in [P_m+1, P_m+r_m]$, BRen takes input the block data $bd_k = ([tx_{k1}, sig_{k1}], ..., [tx_{kJ}, sig_{kJ}])$, the block header of b_{k-1}, and a set of previous blocks $b_{(k-P_m-1)s_m+1}, ..., b_{(k-P_m)s_m}$, outputs an block b_k using H_m. b_k is generated as follows.

(1) For $m \in [2,\ M]$, divide b_1, ..., $b_{P_{m-1}}$ into r_{m-1} sets, with each set of s_{m-1} blocks. Then compute the hash values of each set of blocks using H_{m-1} and compare each hash value with $archiveHash_{(m-1)0}$, ..., $archiveHash_{(m-1)r_{m-1}}$ stored

in $b_{P_{m-1}+1}$, ..., $b_{P_{m-1}+r_{m-1}}$ respectively (the calculation of *archiveHash* field is introduced in step 5). If all matches or for $m=1$, perform following steps:

(2) sig_{k1}, ..., sig_{kJ} are the signatures of transactions tx_{k1}, ..., tx_{kJ} generated by one or a set of secure signature schemes S. Note that S could be the same or not same with the ones used in BGen algorithm. i.e., $sig_{k1} \leftarrow S(tx_{k1})$, ..., $sig_{kJ} \leftarrow S(tx_{kJ})$.

(3) Compute Merkle tree root value of block data using H_m: $mkroot_k \leftarrow MT(H_m; bd_k)$.

(4) Compute the hash value of the block header of b_{k-1} using H_m: $hb_{k-1} = H_m(bh_{k-1})$.

(5) Divide the previous blocks $b_1, ..., b_{P_m}$ into r_m sets, each set has s_m blocks, then compute a new hash value of a group of previous blocks using H_m: $archiveHash_{mk} = H_m(b_{(k-P_m-1)s_m+1}, ..., b_{(k-P_m)s_m})$.

(6) Construct the block header of b_k: $bh_k = (archiveHash_{mk}, hb_{k-1}, mkroot_k, ts_k)$.

(7) Construct block $b_k = (bh_k, bd_k)$.

HVer: $0/1 \leftarrow$ HVer(b_k, blc, t_v, VD). At verification time $t_v > t'_M$, the hash verification algorithm HVer takes input a block b_k, the copy of the whole blockchain blc, the verification time t_v and the verification data VD, outputs a bit 1 if b_k is a valid block on blc, otherwise outputs a bit 0. Assume the underlying hash function of b_k is $H_{M-x}(x \in \{0, M\})$, the verification of b_k can be divided into two cases: $x=0$ and $x \in [1, M]$.

Case 1 ($x=0$): the algorithm HVer checks whether $archiveHash_{Mk}$ is included in b_k. If not, it checks whether the following conditions are satisfied:

(1) H_M is secure at t_v, and the signature scheme S used for generating $sig_{k1}, ..., sig_{kJ}$ is valid when b_k is generated: $(H_M, t_v) \rightarrow$ VD, $(S, ts_k) \rightarrow$ VD.

(2) Signatures sig_{k1}, ...,sig_{kJ} are correct: Vrfy(pk, sig_{k1}, tx_{k1}) = 1, ..., Vrfy(pk, sig_{kJ}, tx_{kJ}) = 1.

(3) The Merkle tree root hash value is correctly calculated: $mkroot_k \leftarrow MT\{H_M; bd_k\}$.

(4) The hash value of the previous block is correctly calculated: $hb_{k-1} = H_M(bh_{k-1})$.

(5) The hash value of the current block is correct: $hb_k = H_M(bh_k)$.

If $archiveHash_{Mk}$ is included in b_k, two additional conditions are needed to be satisfied:

(1) b_k is generated when H_{M-1} and H_M are secure: $([H_{M-1}, H_M], ts_k) \rightarrow$ VD.

(2) $archiveHash_{Mk}$ field is correctly calculated: $archiveHash_{Mk} = H_M(b_{(k-P_M-1)s_M+1}, ..., b_{(k-P_M)s_M})$.

Case 2 ($x \in [1, M]$): the algorithm HVer checks $archiveHash_{(M-x)k}$ is included in b_k or not. If not, then it checks whether the following conditions are satisfied:

(1) Every *archiveHash* in $archiveHash_{M-x+1}$, ..., $archiveHash_M$ that takes input of b_k is correctly calculated.

(2) At least one hash function is secure at phase $M-x$, $M-x+1$, ..., M: for $m \in [M-x, M]$, $(H_m, [t_m, t'_m]) \rightarrow$ VD.

(3) At least two hash functions are secure at phase $(M-x+1)'$, $(M-x+2)'$, ..., $(M-1)'$: for $m \in [M-x+1, M-1]$, $(H_m, H_{m+1}, [t'_m, t_{m+1}]) \rightarrow$ VD.

(4) Step 1 - 5 as Case 1 above.

If yes, there are two additional conditions are needed to be satisfied:

(1) b_k is generated when H_{M-x-1} and H_{M-x} are secure: $([H_{M-x-1}, H_{M-x}], ts_k) \rightarrow$ VD.

(2) $archiveHash_{(M-x)k}$ field is correct: $archiveHash_{(M-x)k} = H_{M-x}(b_{(k-P_{M-x}-1)s_{M-x}+1}, ..., b_{(k-P_{M-x})s_{M-x}})$.

For both cases, the algorithm HVer outputs 1 if all above verification details are satisfied, which means the block b_k is a valid block on blockchain blc at time t_v. Otherwise, the algorithm outputs 0.

4.2 Signature transition procedure

The signature transition scheme is comprised of three algorithms (SGen, SRen, SVer), which separately represents signature generation, signature renewal and signature verification. The algorithm SGen and SRen are performed as the timeline shown in Fig. 4. Let N be the total number of hash transitions, the timeline is divided into a total of 2N+1 phases (N could be continuously on the timeline):

- Phase 0 ($t \in [t_0, t'_0]$): let S_0 be a secure signature scheme. Signed transaction STX_0 is generated by the algorithm SGen using S_0.
- Phase 1' ($t \in [t'_0, t_1]$): S_0 is threatened but still secure, let S_1 be a stronger signature scheme. Signed transaction STX'_1 is generated by the algorithm SRen using S_0.
- Phase 1 ($t \in [t_1, t'_1]$): S_0 could be already compromised, S_1 is still secure. Signed transaction STX_1 is generated by the algorithm SGen using S_1.
- ...
- Phase N' ($t \in [t'_{N-1}, t_N]$): S_0, ..., S_{N-2} could be compromised, S_{N-1} is threatened but still secure, let S_N be a stronger hash function. Signed transaction STX'_N is generated by the SRen algorithm using S_{N-1}.
- Phase N ($t \in [t_N, t'_N]$): S_0, ..., S_{N-1} could be compromised, S_N is secure. Signed transaction STX_N is generated by SGen algorithm using S_N.

SGen: $STX_n \leftarrow$ SGen(S_n; $tx_{n\rightarrow n}$). The signature generation algorithm SGen is implemented at Phase n, i.e., $t \in [t_n, t'_n] (n \in [0, N])$, which takes input a transaction $tx_{n\rightarrow n}$, outputs a signed transaction STX_n using signature scheme S_n. The generation process of STX_n is listed as follows:

(1) A user generates the signature of $tx_{n\rightarrow n}$ using S_n: $s_n \leftarrow S_n(tx_{n\rightarrow n})$.

(2) The user forms the signed transaction $STX_n = (s_n, tx_{n\rightarrow n})$.

(3) STX_n is submitted to a blockchain that uses a secure hash function H: $STX_n \rightarrow blc$.

SRen: $STX'_n \leftarrow$ SRen(S_{n-1}; $tx_{n-1\rightarrow n}$). The signature renewal algorithm SRen takes place at Phase n', i.e., $t \in [t'_{n-1}, t_n] (n \in [1, N])$, which takes input a transaction $tx_{n-1\rightarrow n}$, outputs a signed transaction STX'_n using S_n. This algorithm means that the user updates the signature scheme from S_{n-1} to S_n, such as transfer his coins from the address derived from S_{n-1} to a new address derived from S_n. The generation of STX'_n is listed as follows:

(1) A user generates the signature of $tx_{n-1\rightarrow n}$: $s'_n \leftarrow S_n(tx_{n-1\rightarrow n})$.

(2) The user forms the signed transaction $STX'_n = (s'_n, tx_{n-1\rightarrow n})$.

(3) STX'_n is submitted to the blockchain that uses a secure hash function H: $STX'_n \rightarrow blc$.

SVer: $0/1 \leftarrow$ SVer(STX_N, blc, t_v, VD). At time $t_v > t'_N$, the signature verification algorithm SVer takes input a signed transaction STX_N,

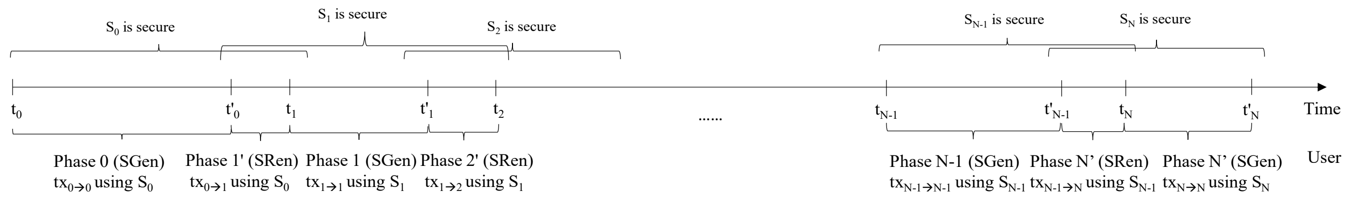
S_0 is secure
S_1 is secure
S_2 is secure
S_{N-1} is secure
S_N is secure
t_0
t'_0
t_1
t'_1
t_2
......
t_{N-1}
t'_{N-1}
t_N
t'_N
Time
Phase 0 (SGen) $tx_{0\to0}$ using S_0
Phase 1' (SRen) $tx_{0\to1}$ using S_0
Phase 1 (SGen) $tx_{1\to1}$ using S_1
Phase 2' (SRen) $tx_{1\to2}$ using S_1
Phase N-1 (SGen) $tx_{N-1\to N-1}$ using S_{N-1}
Phase N' (SRen) $tx_{N-1\to N}$ using S_{N-1}
Phase N' (SGen) $tx_{N\to N}$ using S_N
User

Figure 4: The timeline of signature transition scheme

the copy of the whole blockchain blc, the verification time t_v and the verification data VD, outputs a bit 1 if STX_n is valid, otherwise outputs a bit 0. The verification procedures are to check whether the following conditions are satisfied:

(1) The blocks contain STX_0, STX'_1, STX_1, ..., STX'_N, STX_N are valid by using the HVer algorithm.
(2) Signatures generated by SGen algorithm are correct: for $n \in [0, N]$, $\text{Vrfy}(pk_n, s_n, tx_{n\to n)=1}$.
(3) Signatures generated by SRen algorithm are correct: for $n \in [1, N]$, $\text{Vrfy}(pk_{n-1}, s'_n, tx_{n-1\to n}) = 1$.
(4) S_0 is valid at t_0, S_N is valid at t_v: $(S_0, t_0) \to \text{VD}$, $(S_N, t_v) \to \text{VD}$.
(5) At least one signature scheme is valid at phase n: for $n \in [0, N]$, $(S_n, [t_n, t'_n]) \to \text{VD}$.
(6) At least two signature schemes are valid at phase n': for $n \in [1, N]$, $(S_{n-1}, S_n, [t'_{n-1}, t_n]) \to \text{VD}$.

4.3 Discussions

Now we discuss following aspects of the scheme:

Relations between two procedures. The hash transition procedure is taken by the blockchain system, and the signature transition procedure is completed by users. They are not contradicted with each other and can be implemented simultaneously. Notice that the hash transition procedure only extends the validity of block hash function and Merkle tree hash function. If the signature scheme applies a hash function, the validity of this hash function is treated together with the signature algorithm. In other words, the signature transition procedure should be conducted either the signature algorithm or the signature hash function is threatened.

Permission model. The blockchain in our scheme could either be a permissionless or a permissioned blockchain. In a permissioned blockchain, the transition scheme can be implemented by the authority who control the blockchain access. In a permissionless blockchain, the transition scheme can be performed by miners who participate in generating new blocks, and we assume that there is a agreement on when to start the transition scheme from a weak algorithm to a stronger one.

Verification data. The verification data VD should contain necessary data used for the HVer and SVer algorithms. Apart from the ones can be directly collected from the blockchain blc, such as the identifiers of hash functions and signature schemes, the public key certificates for verifying signatures, relevant block information etc, VD must also contain the information indicating the start time and breakage time of hash functions and signature schemes. This information can be collected from reliable sources.

For instance, after discovering theoretical attacks or flaws of a cryptographic algorithm, the ISO/IEC or NIST standards will recommend that the usage of this algorithm should be ended by a specific date [16, 17]. Then at the time of verifying the validity of algorithms, the block time-stamps and the VD time should be synchronized with a same criteria, e.g., the global time.

Time-stamp. The time-stamps contained in blocks is assumed accurate and reliable to verify the start time and breakage time of hash functions and signature schemes. This assumption could be achieved by existing researches [20] and [13]. The authors proposed similar ideas to create accurate time-stamps in Bitcoin blockchain by leveraging an external time-stamping authority. Their methods could be adopted if our scheme is implemented on Bitcoin blockchain.

Group number. For the hash transition procedure, both existing LTB schemes [7, 18] claim to divide previous blocks into r groups with s blocks in each group, but neither of them discuss how to set the values of r and s. Our scheme follows this group division idea, and we believe it depends on the blockchain size and the average block interval. In Section 7, we implement the hash transition procedure and evaluate the performance for Bitcoin and Ethereum as examples, then discuss how to choose the r and s values.

Overhead. Regard to the overhead issue and the supply chain solution proposed in [18], we calculate the overhead of SHA3-256, SHA3-384 and SHA3-512 hash functions as renewal candidates, and discuss the impacts to Bitcoin and Ethereum blockchains as examples in Section 7.

5 SECURITY MODEL AND DEFINITIONS

In this section, we propose a security model for a LTB scheme. Firstly, we make following assumptions in a LTB scheme:

(1) When the current hash function or signature algorithm used in the blockchain becomes weak but not actually compromised, there is a stronger hash function or signature algorithm secure in the next time period.
(2) The verification data VD is trusted.

A secure LTB scheme should satisfy three properties: correctness, long-term integrity, and long-term unforgeability. The definitions of these properties are provided as follows.

5.1 Correctness

Correctness means that if all entities legitimately perform their functions, a LTB scheme is able to maintain the blockchain validity in long-term periods that are not bounded with the lifetimes of underlying cryptographic algorithms. Assume all blocks in the blockchain are generated in terms of the phases described in Section 4.1:

- Phase 0, $k \in [1, F_0]$: $b_k \leftarrow \text{BGen}(H_0; bd_k, bh_{k-1})$

- Phase 1', $k \in [P_1+1, P_1+r_1]$:
  $b_k \leftarrow \text{BRen}(H_1; bd_k, bh_{k-1}, b_{(k-P_1-1)s_1+1}, ..., b_{(k-P_1)s_1})$
- Phase 1, $k \in [P_1+r_1+1, P_1+r_1+F_1]$:
  $b_k \leftarrow \text{BGen}(H_1; bd_k, bh_{k-1})$
- ......
- Phase M', $k \in [P_M+1, P_M+r_M]$:
  $b_k \leftarrow \text{BRen}(H_M; bd_k, bh_{k-1}, b_{(k-P_M-1)s_M+1}, ..., b_{(k-P_M)s_M})$
- Phase M, $k \in [P_M+r_M+1, P_M+r_M+F_M]$:
  $b_k \leftarrow \text{BGen}(H_M; bd_k, bh_{k-1})$

At a point in time $t_v > t'_M$, assume the latest hash function H_M and the latest signature scheme S are secure. The algorithm HVer takes input a block b_k, the blockchain blc, the verification data VD and verification time t_v.

Then for a user, assume all blocks in the blockchain are legitimately generated by BGen and BRen as the above process, the signed transactions STX_0, STX'_1, STX_1, ..., STX'_N, STX_N are generated in terms of the phases described in Section 4.2:

- Phase 0: $STX_0 \leftarrow \text{SGen}(S_0; tx_{0\rightarrow 0})$, $STX_0 \rightarrow blc$
- Phase 1': $STX'_1 \leftarrow \text{SRen}(S_0; tx_{0\rightarrow 1})$, $STX'_1 \rightarrow blc$
- Phase 1: $STX_1 \leftarrow \text{SGen}(S_1; tx_{1\rightarrow 1})$, $STX_1 \rightarrow blc$
- ......
- Phase N': $STX'_N \leftarrow \text{SRen}(S_{N-1}; tx_{N-1\rightarrow N})$, $STX'_N \rightarrow blc$
- Phase N: $STX_N \leftarrow \text{SGen}(S_N; tx_{N\rightarrow N})$, $STX_N \rightarrow blc$

At a point in time $t'_v > t'_N$, assume the latest signature scheme S_N is secure. The algorithm SVer takes as input a block STX_N, the blockchain blc, the verification data VD and verification time t'_v.

Definition 5.1. (Correctness.) Let LTB = ([BGen, BRen, HVer], [SGen, SRen, SVer]) be a LTB scheme. For the scheme to be correct, it must satisfy that if every block and signed transaction in the blockchain are generated following the above process, the verification results $\text{HVer}(b_k, blc, \text{VD}, t_v) = 1$ and $\text{SVer}(STX_N, blc, \text{VD}, t'_v) = 1$ hold.

5.2 Long-term integrity

The concept of long-term integrity is associated with the concept of compromising a LTB scheme. By intuition, we say that an attacker is able to compromise a LTB scheme, if it is able to claim non-existed data or to tamper data in any of blocks on the blockchain without being detected. Thereby, we say that a LTB scheme has long-term integrity if any polynomial time adversary is unable to compromise the LTB scheme in long-term periods that are not bounded with the lifetimes of underlying cryptographic algorithms.

To formalize this, the long-term integrity model is defined as a game running between a long-lived adversary $\mathcal{A}$ and a simulator $\mathcal{B}$. The computational power of $\mathcal{A}$ is as specified in Section 4.1: in each phase, $\mathcal{A}$ is able to break some cryptographic primitives and restricted against some others within the phase. $\mathcal{B}$ has computational resources comparable to $\mathcal{A}$. The hash transition procedure is performed by the blockchain system. $\mathcal{A}$ is able to access a clock oracle $Clk(\cdot)$, and a blockchain oracle $Blc(\cdot)$, which are defined as follows:

(1) $Clk(\cdot)$: $P_{cur} \leftarrow Clk(t_{cur})$. $\mathcal{A}$ inputs the current time t_{cur} to the clock oracle, the oracle returns the corresponding computational power P_{cur} in terms of which phase t_{cur} lies in.
(2) $Blc(\cdot)$: $b_x \leftarrow Blc(tx, sig)$, $R \leftarrow R \parallel b_x$. $\mathcal{A}$ could input transaction and signature pairs (tx, sig). The blockchain oracle verifies the validity of the pairs. If any of them is not valid, the oracle returns a $\perp$. If all of them are valid, the oracle forms blocks of the blockchain blc with these pairs, and records every block $b_x (x \geq 1)$ in a list R.

The long-term integrity experiment is shown as Algorithm 1:

Algorithm 1: Long-term integrity experiment $\text{Exp}^{\text{LTI}}_{\text{LTB}}(\mathcal{A})$

```
1  Input: M, $blc$, VD
2  Output: a bit 1 or 0
3  set PHASE = [phase 0, phase 1', phase 1, ..., phase M', phase M]
4  set $R = []$
5  for $m=0; m \leq 2M; m{+}{+}$ do
6      In PHASE[m]:
7      $b'_k \leftarrow \mathcal{A}^{Clk(\cdot),\, Blc(\cdot)}$
8      if HVer$(b'_k, blc, \text{VD}, t_v) = 1$ and $b'_k \notin R$ then
9          Return 1, breaks;
10     else
11         Return 0;
12 Return 0;
```

We use $\Pr[\text{Exp}^{\text{LTI}}_{\text{LTB}}(\mathcal{A}) = 1]$ to denote the probability of $\mathcal{A}$ winning the game. By the time t_v, we denote the probability that $\mathcal{B}$ breaks at least one hash function and at least one signature scheme within its validity period separately as $\mathcal{B}^{Com}_{\mathcal{H}}$ and $\mathcal{B}^{Com}_{\mathcal{S}}$.

Definition 5.2. (Long-term Integrity.) Let LTB = ([BGen, BRen, HVer], [SGen, SRen, SVer]) be a LTB scheme, let $\mathcal{A}$ and $\mathcal{B}$ be an adversary and a simulator respectively as specified above. Then a LTB scheme holds the long-term integrity property if there exists a constant c for $\mathcal{B}$ at any t_v, $\Pr[\text{Exp}^{\text{LTI}}_{\text{LTB}}(\mathcal{A}) = 1] \leq c \cdot (\mathcal{B}^{Com}_{\mathcal{H}} + \mathcal{B}^{Com}_{\mathcal{S}})$.

5.3 Long-term unforgeability

The concept of Long-term unforgeability relates to the concept of forging a signature. By intuition, if an attacker is given a public key pk generated by a signer S, we say an adversary outputs a forgery if it outputs a message m along with a valid signature s on m, and m was not previously signed by S. We say that a LTB scheme is long-term unforgeable, if any polynomial time adversary is unable to forge signatures on the blockchain in long-term periods that are not bounded with the lifetimes of underlying cryptographic algorithms.

Similar to the long-term integrity model, the long-term unforgeability model is defined as a game between a long-lived adversary $\mathcal{A}$ and a simulator $\mathcal{B}$. $\mathcal{A}$ could communicate with the clock oracle as specified in Section 5.2, and a signing oracle $Sign(\cdot)$ defined as below. The computational power of $\mathcal{A}$ and $\mathcal{B}$ follow Section 4.2, and the long-term unforgeability model is listed as Algorithm 2.

- $Sign(\cdot)$: $s \leftarrow Sign(sk, tx)$, $Q \leftarrow Q \parallel tx$. $\mathcal{A}$ inputs a transaction tx to the signing oracle, the signing oracle returns a signature s of tx. Then every input tx is stored in a list Q.

We use $\Pr[\text{Exp}^{\text{LTU}}_{\text{LTB}}(\mathcal{A}) = 1]$ to denote the probability that $\mathcal{A}$ wins the game, $\mathcal{B}^{Com}_{\mathcal{H}}$ and $\mathcal{B}^{Com}_{\mathcal{S}}$ have the same meaning as in Section 5.2.

Definition 5.3. (Long-term Unforgeability.) Let LTB = ([BGen, BRen, HVer], [SGen, SRen, SVer]) be a LTB scheme, let $\mathcal{A}$ and $\mathcal{B}$ be

Algorithm 2: Long-term unforgeability experiment $\text{Exp}_{\text{LTB}}^{\text{LTU}}(\mathcal{A})$

```
1  Input: N, $blc$, VD
2  Output: a bit 1 or 0
3  set PHASE = [phase 0, phase 1', phase 1, ..., phase N', phase N]
4  set $Q = []$
5  for $n = 0; n \leq 2N; n{+}{+}$ do
6      In PHASE[n]:
7      if $n \in 2k\ (k \in \mathbb{N})$ then
8          $(pk, sk) \leftarrow Gen(), \mathcal{A} \leftarrow pk$
       /* signature generation phases */
9      else
10         $(pk_1, sk_1) \leftarrow Gen(), (pk_2, sk_2) \leftarrow Gen(),$
11         $\mathcal{A} \leftarrow pk_1, pk_2$
       /* signature renewal phases */
12     $STX = (tx, s) \leftarrow \mathcal{A}^{Clk(\cdot),\, Sign(\cdot)}$
13     if SVer$(STX, blc, \text{VD}, t_v) = 1$ and $tx \notin Q$ then
14         Return 1, breaks;
15     else
16         Return 0;
17 Return 0;
```

an adversary and a simulator respectively as specified above. Then a LTB scheme holds long-term unforgeability property if there exists a constant c for $\mathcal{B}$ at any t_v, $\Pr[\text{Exp}_{\text{LTB}}^{\text{LTU}}(\mathcal{A}) = 1] \leq c \cdot (\mathcal{B}_{\mathcal{H}}^{Com} + \mathcal{B}_{\mathcal{S}}^{Com})$.

6 SECURITY ANALYSIS

In this section, we prove that the proposed LTB scheme holds each security property in terms of the security model in Section 5.

6.1 Proof of correctness

THEOREM 6.1. *The proposed LTB scheme holds correctness property.*

PROOF. In our proposed scheme LTB = ([BGen, BRen, HVer], [SGen, SRen, SVer]), we assume that every block in the blockchain is generated through algorithm BGen and BRen, and all signed transactions of users are generated by the algorithm SGen and SRen as the process described in Section 5.1. At time $t_v > t'_M$, the input values of algorithm HVer are b_k, VD, blc and t_v. At time $t'_v > t'_N$, the input values of algorithm SVer are STX, VD, blc and t'_v. We now analyze the output of HVer and SVer in terms of the verification procedures specified in Section 4.1 and 4.2 respectively:

Hash transition. For case 1 ($x = 0$), the hash function used in b_k is $H_{M-x} = H_M$. If b_k does not contain $archiveHash_{Mk}$ field, it is generated through BGen algorithm legitimately. Therefore, the signatures $sig_{k1}, ..., sig_{kJ}$ are correct, and signature scheme S is secure at the block generation time t_k. Then the Merkle tree root value is determined from $bd_k = [tx_{k1}, sig_{k1}], ..., [tx_{kJ}, sig_{kJ}]$, the block hash values hb_{k-1} and hb_k are calculated from b_{k-1} and b_k correctly. If b_k includes $archiveHash_M$ field, it is generated by BRen algorithm by following the hash transition timeline. Thus, $archiveHash_{Mk}$ is correctly calculated when H_{M-1} and H_M are secure.

For case 2 ($x \in [1, M]$), the hash function used in b_k is H_{M-x}. Since all blocks are generated by BGen and BRen algorithms legitimately by following the timeline, $H_{M-x}, ..., H_M$ are secure when used in BGen algorithm, and each pair of hash function (H_{M-x}, H_{M-x+1}), ..., (H_{M-1}, H_M) is secure when used in BRen algorithm. If b_k does not have $archiveHash_{(M-x)k}$ field, it is generated by BGen algorithm. Thus, the validity of signatures, signature schemes, Merkle tree root hash value, and block hash values are all guaranteed as case 1. If b_k has $archiveHash_{(M-x)k}$ field, then it is generated by BRen algorithm. The calculation result and time of $archiveHash_{(M-x)k}$ are correct as case 1. With the assumption that H_M is secure at time t_v, we have HVer$(b_k, blc, t_v, \text{VD}) = 1$.

Signature transition. First, with the assumption that all blocks in the blockchain are correctly produced by BGen and BRen algorithms, the blocks containing $STX_0, STX'_1, STX_1, ..., STX'_N, STX_N$ are valid. Second, since every signed transaction is created by SGen and SRen algorithms legitimately, the signatures $s_0, ..., s_N$ and $s'_1, ..., s'_N$ are correct. Third, since each time algorithms SGen and SRen take place by following the signature transition timeline, signature schemes $S_0, ..., S_N$ are secure when used in SGen algorithm, and each pair of signature schemes $(S_0, S_1), ..., (S_{N-1}, S_N)$ are secure when used in SRen algorithm. With the assumption that S_N is secure at t'_v, we get SVer$(STX, blc, t'_v, \text{VD}) = 1$.

Based on the above analysis on the verification results of HVer and SVer algorithms, the proposed LTB scheme holds the correctness property, the theorem 6.1 follows. □

6.2 Proof of long-term integrity

THEOREM 6.2. *In the proposed LTB scheme, if each time the current hash function or signature scheme used in the blockchain becomes weak but not actually compromised, a stronger one is used in the next time period, and the verification data VD is trusted, the proposed LTB scheme holds the long-term integrity property.*

PROOF. If the adversary $\mathcal{A}$ wins the game, it must output a block b'_k, which is not an original block on blockchain blc, but somehow to manage letting HVer$(b'_k, blc, \text{VD}, t_v) = 1$. Now we analyse the probability of $\mathcal{A}$ winning the game separately from the Phase 0, Phase 1', Phase 1, ..., Phase M', and Phase M that are defined in Section 5.2.

At Phase 0, $\mathcal{A}$ could query the oracle $Clk(\cdot)$ to set computational power P_0 at time $t \in [t_0, t'_0]$, which is bounded to break a secure signature scheme S and a hash function H_0 with negligible probability. H_0 is the only hash function used in the blockchain for Merkle tree and block hash value computation. If $\mathcal{A}$ produces signatures or hash values by using broken signature schemes or hash functions, and inputs them into the blockchain by oracle $Blc(\cdot)$, the oracle will check the algorithm identifiers and verify them as invalid transactions, then returns a $\perp$. Hence, if $\mathcal{A}$ wins the game, one of the following three cases must happen:

(1) $\mathcal{A}$ finds $sig \leftarrow S(tx_1) = S(tx_2)$.
(2) $\mathcal{A}$ finds $root \leftarrow MT(H_0; [tx_1, sig_1]) = MT(H_0; [tx_2, sig_2])$.
(3) $\mathcal{A}$ finds $H_0(b_k) = H_0(b'_k)$.

Then $\mathcal{B}$ can obtain the pair (tx_1, tx_2), $([tx_1, sig_1], [tx_2, sig_2])$, (b_k, b'_k) to break the security of S or collision resistance of H_0 within their validity periods. This result is contradict to the assumption that S and H_0 are secure at Phase 0. If $\mathcal{A}$ does not win the game, it must legitimately input a valid (tx, sig) pair into the blockchain, then let us carry on with our reasoning.

At Phase 1', $\mathcal{A}$ could set computational power P_1 at time $t \in [t_0', t_1]$ by oracle $Clk(\cdot)$. Signature scheme S and hash functions H_0, H_1 are secure against P_1. The blockchain generates new blocks by using H_1, existing blocks could be generated using H_0 or H_1. If $\mathcal{A}$ wins the game, one of the following three cases must happen:

(1) $\mathcal{A}$ finds $sig \leftarrow S(tx_1) = S(tx_2)$.
(2) $\mathcal{A}$ finds $root \leftarrow MT(H_0; [tx_1, sig_1]) = MT(H_0; [tx_2, sig_2])$, or $root \leftarrow MT(H_1; [tx_1, sig_1]) = MT(H_1; [tx_2, sig_2])$.
(3) $\mathcal{A}$ finds $H_0(b_k) = H_0(b_k')$, or $H_1(b_k) = H_1(b_k')$.

Then $\mathcal{B}$ can obtain the pair (tx_1, tx_2), $([tx_1, sig_1], [tx_2, sig_2])$, (b_k, b_k') to break the security of S, or collision resistance of H_0, or collision resistance of H_1 within their validity periods. This is contradict to the assumption that S, H_0, H_1 are secure at Phase 1'. If $\mathcal{A}$ does not win the game, it must input a valid (tx, sig) pair into the blockchain, we can carry on with our reasoning.

At Phase 1, $\mathcal{A}$ could set computational power P_1 at time $t \in [t_1, t_1']$ by oracle $Clk(\cdot)$. Signature scheme S and hash functions H_1 are secure against P_1, H_0 can be compromised by P_1. The blockchain generates new blocks by using H_1, and existing blocks could be generated using H_0 or H_1. Similar to phase 0, if $\mathcal{A}$ wins the game, $\mathcal{B}$ can obtain the pair (tx_1, tx_2), $([tx_1, sig_1], [tx_2, sig_2])$, (b_k, b_k') to break the security of S or collision resistance of H_1 within their validity periods. This is contradict to the assumption that S and H_1 are secure at Phase 1. If $\mathcal{A}$ inputs a valid (tx, sig) pair into the blockchain, the reasoning continues.

Carrying on our argument as before until Phase M', $\mathcal{A}$ could set computational power P_M at time $t \in [t_{M-1}', t_M]$ by oracle $Clk(\cdot)$. Signature scheme S and hash functions H_{M-1}, H_M are secure against P_M. H_0, ..., H_{M-2} could be compromised by P_M. Similar to phase 1', if $\mathcal{A}$ wins the game, $\mathcal{B}$ can break the security of S, or collision resistance of H_{M-1}, or collision resistance of H_M within their validity periods. This is contradict to the assumption that S, H_{M-1}, H_M are secure at Phase M'. If $\mathcal{A}$ inputs a valid (tx, sig) pair into the blockchain, the reasoning continues.

At Phase M, $\mathcal{A}$ set computational power P_M at time $t \in [t_M, t_M']$ by oracle $Clk(\cdot)$. Signature scheme S and hash functions H_M are secure against P_M, $H_0, ..., H_{M-1}$ could be compromised by P_M. Same as previous phases, if $\mathcal{A}$ wins the game, $\mathcal{B}$ can break the security of S or collision resistance of H_M within their validity periods. This is contradict to the assumption that S and H_M are secure at Phase M.

In summary, based on the above reasoning, the winning probability of $\mathcal{A}$ is reduced to the same level of the probability that $\mathcal{B}$ breaks at least one signature scheme or hash function within its validity period. Assume until time t_v, the probability of the former term is denoted as $\mathcal{B}_{\mathcal{S}}^{Com}$, the probability of the latter term is denoted as $\mathcal{B}_{\mathcal{H}}^{Com}$. There exists a constant c such that:

$$\Pr[\mathrm{Exp}_{\mathrm{LTB}}^{\mathrm{LTI}}(\mathcal{A}) = 1] \leq c \cdot (\mathcal{B}_{\mathcal{H}}^{Com} + \mathcal{B}_{\mathcal{S}}^{Com}).$$

Thus, we have proved Theorem 6.2. □

6.3 Proof of long-term unforgeability

Theorem 6.3. *In the proposed LTB scheme, if each time the current hash function or signature scheme used in the blockchain becomes weak but not actually compromised, a stronger one is used in the next time period, and the verification data VD is trusted, the proposed LTB scheme holds the long-term unforgeability property.*

Proof. If the adversary $\mathcal{A}$ wins the game, it must output a signed transaction $STX = (tx, s)$, in which tx is not queried from the signing oracle, but somehow to manage letting $\mathrm{SVer}(STX, blc, \mathrm{VD}, t_v) = 1$. In terms of the model defined in Section 5.3, we analyze the probability of $\mathcal{A}$ winning the game from Phase 0, Phase 1', Phase 1, ..., Phase N', and Phase N.

At Phase 0, $\mathcal{A}$ could query the oracle $Clk(\cdot)$ to set computational power P_0 at time $t \in [t_0, t_0']$, in which a signature scheme S_0 is secure against P_0. If $\mathcal{A}$ produces signatures by using compromised signature schemes and submits these signatures to blc, the verification procedure $(S_0, [t_0, t_0']) \rightarrow \mathrm{VD}$ is not hold. If $\mathcal{A}$ modifies the signed transaction STX_0 without breaking the Merkle tree or block hash functions within their validity periods, the output of algorithm HVer will be 0. For both of above cases, the output of SVer algorithm will be 0. Thus, if $\mathcal{A}$ wins the game, one of following three cases must happen:

(1) $\mathcal{A}$ finds $H(STX) = H(STX')$.
(2) $\mathcal{A}$ finds $root \leftarrow MT(H; STX) = MT(H; STX')$.
(3) $\mathcal{A}$ finds a pair (tx, s) that satisfies $\mathrm{Vrfy}(pk_0, s, tx) = 1 \wedge tx \notin Q$.

That means, simulator $\mathcal{B}$ can obtain the (STX, STX') or (tx, s) pair to compromise the collision resistance of H or unforgeability of S_0 within their validity periods. This is contradict to the assumption that H and S_0 are secure at Phase 0. If $\mathcal{A}$ does not win the game, it must legitimately input a valid $(s_0, tx_{0\rightarrow 0})$ pair into the blockchain, in which $s_0 \leftarrow Sign(sk_0, tx_{0\rightarrow 0})$. Then we can carry on with our reasoning.

At Phase 1', $\mathcal{A}$ could set computational power P_1 at time $t \in [t_0', t_1]$ by oracle $Clk(\cdot)$, signature schemes S_0, S_1 are secure against P_1. Same as Phase 0, $\mathcal{A}$ is not able to win the game by submitting invalid signatures, or modify signed transactions without breaking the Merkle or block hash functions. If $\mathcal{A}$ wins the game, one of the following three cases must happen:

(1) $\mathcal{A}$ finds $H(STX) = H(STX')$.
(2) $\mathcal{A}$ finds $root \leftarrow MT(H; STX) = MT(H; STX')$.
(3) $\mathcal{A}$ finds a pair (tx, s) that satisfies $\mathrm{Vrfy}(pk_0, s, tx) = 1 \wedge tx \notin Q$, or $\mathrm{Vrfy}(pk_1, s, tx) = 1 \wedge tx \notin Q$.

Then the simulator $\mathcal{B}$ can obtain the (STX, STX') or (tx, s) pair to compromise the collision resistance of H, the unforgeability of S_0 or unforgeability of S_1 within their validity periods, which is contradict to the assumption that H, S_0 and S_1 are secure at Phase 1'. If $\mathcal{A}$ does not win the game, it must input a valid $(s_0, tx_{0\rightarrow 0})$ pair or $(s_1, tx_{1\rightarrow 1})$ pair into the blockchain, in which $s_0 \leftarrow Sign(sk_0, tx_{0\rightarrow 0})$ and $s_1 \leftarrow Sign(sk_1, tx_{1\rightarrow 1})$. Then we can carry on with our reasoning.

At Phase 1, $\mathcal{A}$ could set computational power P_1 at time $t \in [t_1, t_1']$ by oracle $Clk(\cdot)$, signature scheme S_1 is secure against P_1, S_0 could be compromised by P_1. Similar to Phase 0 and 1, $\mathcal{A}$ is not able to win the game by submitting invalid signatures, or modify signed transactions without breaking the Merkle or block hash functions. If $\mathcal{A}$ wins the game, the simulator $\mathcal{B}$ can obtain the (STX, STX') or (tx, s) pair to compromise the collision resistance of H or unforgeability of S_1 within their validity periods, which is contradict to the assumption that H and S_1 are secure at Phase 1. If $\mathcal{A}$ does not win the game, then the reasoning continues.

Carrying on our argument as before until Phase N', $\mathcal{A}$ sets computational power P_N at time $t \in [t_N, t_N']$ by oracle $Clk(\cdot)$, signature schemes S_{N-1}, S_N are secure against P_N, $S_0, ..., S_{N-2}$ could be compromised by P_N. Same as previous phases, if $\mathcal{A}$ wins the game, the simulator $\mathcal{B}$ can compromise the collision resistance of H, the unforgeability of S_{N-1}, or unforgeability of S_N within their validity periods, which is contradict to the assumption that H, S_{N-1} and S_N are secure at Phase N'. If $\mathcal{A}$ does not win the game, we keep on with our reasoning.

At Phase N, $\mathcal{A}$ could set computational power P_N at time $t \in [t'_N, t_{N+1}]$, signature scheme S_N is secure against P_N, $S_0, ..., S_{N-1}$ could be compromised by P_N. Same as before, if $\mathcal{A}$ wins the game, the simulator $\mathcal{B}$ can compromise the collision resistance of H or the unforgeability of S_N within its validity period, which is contradict to the assumption that H and S_N are secure at Phase N.

In conclusion, based on the above reasoning, the winning probability of $\mathcal{A}$ is reduced to the same level of the probability that $\mathcal{B}$ breaks at least one hash function or signature scheme within its validity period. Assume until time t_v, the probability of the former term is denoted as $\mathcal{B}^{Com}_{\mathcal{H}}$, the probability of the latter one is denoted as $\mathcal{B}^{Com}_{\mathcal{S}}$. There exists a constant c such that:

$$\Pr[\mathrm{Exp}^{\mathrm{LTU}}_{\mathrm{LTB}}(\mathcal{A})=1] \leq c \cdot (\mathcal{B}^{Com}_{\mathcal{S}} + \mathcal{B}^{Com}_{\mathcal{H}}).$$

Thus, we have proved Theorem 6.3. □

7 IMPLEMENTATIONS

As an implementation, we have simulated the hash transition procedure, evaluated the time consumption and overhead for generating an *archiveHash* field. Based on the resulting performances, we discuss the group number and overhead issues mentioned in Section 4.3.

We do not implement the signature transition procedure because it is taken by the blockchain users with simple actions: they only need generate a new account from the stronger signature scheme, then transfer their assets from the current account to the new one. The implementation of the whole process is same as generating and submitting a new transaction on the blockchain.

7.1 Experiments

Our experiments use a Desktop with a AMD Ryzen 5 3600 6-Core Processor, a 16 - GB RAM and a 64-bit operating system. Firstly, we create a blockchain on python that applies ECDSA signature scheme for users' transactions and SHA-256 hash function for Merkle tree and hashing blocks. Then we use the 'hashlib' library in python to measure the hashing time for SHA3-256, SHA3-384 and SHA3-512 hash functions as renewal candidates of SHA-256. The hash input could be any part or whole of blockchain, we set different hash input size by adjusting transaction numbers in each block, and copy a number of same blocks to reach our target values.

The blocks in our blockchain is formed in "dictionary" type in python, which is a big list comprised of two tuples "block header" and "block data". However, the hash functions in hashlib only accepts "bytes" type input values. That means a converting from "dictionary" to "string" type, and an encoding from "string" to "bytes" type are required before hashing. Thereby, we also measure the string converting time and encoding time along with the hashing time for corresponding data sizes.

We summarize our results into three diagrams as Fig. 5, Fig. 6 and Fig. 7, which indicate the time performance of calculating an *archiveHash* field for various input data sizes using SHA3-256, SHA3-384, and SHA3-512 respectively. The x-axis represents the scale of input data size (GB). The y-axis represents the time (seconds). The five colored lines in each diagram represent the relations between input data size and time of string converting, encoding, hashing, encoding + hashing, and string converting + encoding + hashing. Notice that the string converting and encoding can be implemented before hashing, here we measure them together to simulate multiple situations, so they remain the same in the three diagrams.

For example, the five time points corresponding to block size = 200 GB with SHA3-256 (the five red crossed points between the vertical white line and coloured lines on Fig. 5) are measured as follows: we

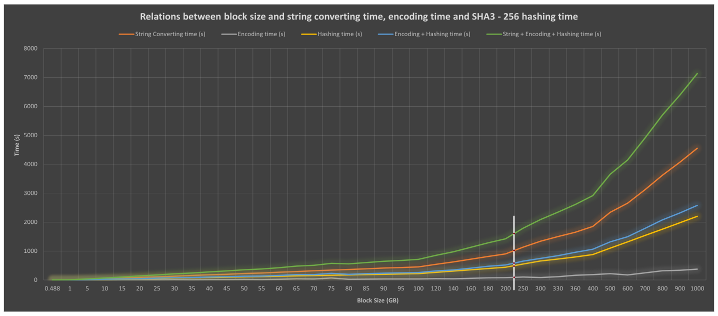

Figure 5: Hashing time, Encoding time and string converting time performance for SHA3 - 256

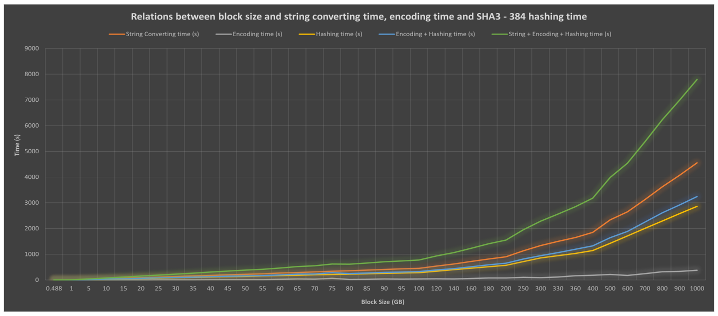

Figure 6: Hashing time, Encoding time and string converting time performance for SHA3 - 384

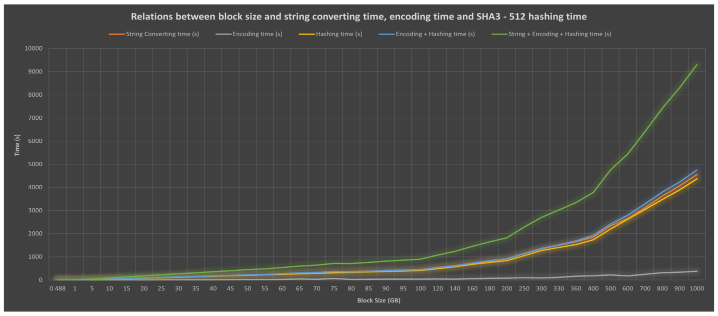

Figure 7: Hashing time, Encoding time and string converting time performance for SHA3 - 512

Measured target	Size - Time function
SHA3 - 256	$y = 2.2x$
SHA3 - 384	$y = 2.86x$
SHA3 - 512	$y = 4.3x$
String Converting	$y = 4.55$
Encoding	$y = 0.4x$

Table 2: Functions between input size (GB) and hashing time (Sec) of SHA3-256, SHA3-384, SHA3-512, string converting time (Sec), and encoding time (Sec)

create a block containing 1380 transactions with block size of 539 KB, then we copy the same block 389082 times to form a 200 GB input data size, and measure the string converting time, encoding time, and SHA3-256 hashing time for this 200 GB input data as 902.26s, 78.89s, and 839.32s respectively. Finally, we determine the encoding + hashing time as $78.89s + 839.32s = 918.21s$, and string + encoding + hashing time as $902.26s + 78.89s + 839.32s = 1820.47s$.

Based on our results, we observe that the hashing speeds comparison is: SHA3-256 > SHA3-384 > SHA3-512, the consumed time comparison with taking input same data sizes are: string converting > hashing > encoding. Besides, the relations between input data size and string converting, encoding and hashing time are all close to linear, so we determine the size-time functions as Table 2. In these functions, y represents the time measured in seconds and x represents the input data size measured in gigabytes. Our diagrams only scale some block sizes from 0.488 GB to 1000 GB, but we can use these functions to calculate the time of other input data sizes.

7.2 Group number discussion

Now we can evaluate how to set the group number and group size so that achieves best performance to complete a hash transition procedure. As two examples, we collect the relevant block statistics of Bitcoin and Ethereum from [5] and [8] respectively, which are presented in Table 3. The statistics include the whole blockchain size, average block size, average transactions per block, total number of transactions, total number of blocks, average block interval, and max/min block interval collected by 00:00 15/10/2021.

In terms of Table 2, we can determine the time of calculating an *archiveHash* by taking input a whole Bitcoin or Ethereum blockchain without dividing into groups. But the time is normally longer than the average block interval. To avoid possible hard forks, we set the group number satisfying that the generation time of each *archiveHash* field is lower than their average block interval, and compute the total time of a hash transition after dividing the blockchain with the group numbers. We also calculate the group number for the cases of string converting and encoding are taking place at the hash transition. The results are displayed in Table 4 and Table 5.

For instance, the total hash transition time for Bitcoin using SHA3-384 is calculated as follows: 1) compute the hashing time, string converting time and encoding time of whole Bitcoin blockchain following the size-time functions in Table 2. i.e., Hashing time = 17.19 mins, encoding + hashing time = 19.84 mins, string converting + encoding + hashing time = 47.19 mins. 2) Compute the group number that makes each *archiveHash* generation time lower than 10 mins (refer to Table 3) for all three cases, e.g., 17.19 mins/2 = 8.595 mins < 10 mins, so set group number as 2. 4) Calculate the group size, e.g., for group number of 2, the group size is 360.70 GB/2 = 180.35 GB, which means each group should have enough blocks to form 180.35 GB. 5) Calculate the total hash transition time = group number × average block interval. e.g., for 2 groups, the total hash transition time is around 2×10 mins = 20 mins.

From both Table 4 and 5, we can see that the string converting and encoding consumed a lot of extra time that reduce the performances. Including them, the transition for Bitcoin could complete in 5 - 7 blocks with 50 - 70 mins, for Ethereum is in 488 - 630 blocks with 122 - 157.5 mins. Without them, the transition for Bitcoin can finish in 2 - 3 blocks with 20 - 30 mins, for Ethereum is in 150 - 292 blocks within 37.5 - 73 mins. Based on the SHA-1 breakage example, it will take years from theoretical attacks to a practical attack. Thus, even the worst case, i.e., 157.5 mins = 2.625 hours, is very efficient and completely acceptable for a hash transition. Nevertheless, the blockchain size will keep increasing in the future, which will make the hashing time, encoding time, and string converting time much longer. We recommend to complete the string converting and encoding in advance so that improves the efficiency.

7.3 Overhead

The overhead of the hash transition procedure is equal to the size of the *archiveHash* field in each block, which depends on the output size of the new hash function. i.e., 64 KB for SHA3-256, 96 KB for SHA3-384 and 128 KB for SHA3-512. Referred to the average block size in Table 3, the average block size of Bitcoin and Ethereum are 536.456 KB and 79.415 KB respectively.

For Bitcoin, the biggest overhead of 128 KB takes $128/536.456 = 23.9\%$ of a block. Especially, the average block size of Bitcoin in recent three years has raised to 1182.72 KB, that means a 128 KB overhead only takes $128/1182.72 = 10.82\%$ of a block. This overhead could be accepted by reducing 10% - 20% transactions per block or increasing the block size for 10% - 20%, since the block size is not a fixed value. As the evaluation in Section 7.2, the hash transition is efficient and will not take a long time to complete, the changing of block size is temporary and can be recovered soon.

For Ethereum, the overhead of 64 KB takes $64/79.415 = 80.6\%$ of a block, the overhead of 96 KB and 128 KB could be larger than a block, which may be hard to be accepted by the current Ethereum blockchain. In this case, we recommend to adopt the supplement procedure introduced in Section 3.1. That is, to construct a supplement blockchain with bigger block size to contain the *archiveHash* field, the original chain remains the same. Transactions verification is usually conducted in original chain, the support chain is used for the verification process when a dispute occurs. This method can be applied to other blockchains if the overhead cannot be accepted.

8 CONCLUSIONS

In this paper, we have analyzed that the existing long-term blockchain schemes are not compatible with existing blockchain structures, and could possibly be vulnerable to attacks after the first hash transition because the security analysis is missing. Then we have proposed an enhanced version of long-term blockchain scheme with a formal

	Blockchain size (GB)	Average block size (KB)	Average transactions per block	Total number of transactions
Bitcoin	360.6953125	536.456686	986.7616343	678293951
Ethereum	1016.3307030534	79.415	98.2880381	1318964434
	Total number of blocks	Average block interval time	Max/Min Block interval time	
Bitcoin	705027	10 min	24.828 min / 2.081 min	
Ethereum	13419379	15 s	30.31/4.46s	

Table 3: Bitcoin and Ethereum block statistics at 00:00 October 15^{th} 2021

Hash only	SHA3-256	SHA3-384	SHA3-512
No group time	13.23 min	17.19 min	25.85 min
Group number	2	2	3
Group size	180.35 GB	180.35 GB	120.24 GB
Transition time	20 min	20 min	30 min
Encode + Hash	SHA3-256	SHA3-384	SHA3-512
No group time	15.86 min	19.84 min	28.5 min
Group number	2	3	4
Group size	180.35 GB	120.24 GB	90.17 GB
Transition time	20 min	30 min	40 min
Str + Encode + Hash	SHA3-256	SHA3-384	SHA3-512
No group time	43.21 min	47.19 min	55.85 min
Group number	5	6	7
Group size	72.14 GB	60.12 GB	51.53 GB
Transition time	50 min	60 min	70 min

Table 4: Group number evaluation for SHA3-256, SHA3-384, and SHA3-512 in Bitcoin

Hash only	SHA3-256	SHA3-384	SHA3-512
No group time	37.27 min	48.45 min	72.84 min
Group number	150	194	292
Group size	6.78 GB	5.23 GB	3.49 GB
Transition time	37.5 min	48.5 min	73 min
Encode + Hash	SHA3-256	SHA3-384	SHA3-512
No group time	44.72 min	55.9 min	80.29 min
Group number	179	224	322
Group size	5.68 GB	4.53 GB	3.16 GB
Transition time	44.75 min	56 min	80.5 min
Str + Encode + Hash	SHA3-256	SHA3-384	SHA3-512
No group time	121.79 min	132.97 min	157.36 min
Group number	488	532	630
Group size	2.08 GB	1.91 GB	1.6 GB
Transition time	122 min	133 min	157.5 min

Table 5: Group number evaluation for SHA3-256, SHA3-384, and SHA3-512 in Ethereum

security model, and we have analyzed that our scheme achieves correctness, long-term integrity and long-term unforgeability properties under the security model. Finally, we have implemented the hash transition procedure and tested that our scheme is very efficient and practical.

With the advanced technology and computing architecture developing faster, the topic of long-term security becomes more and more significant in the world. Apart from the blockchain technology, other applications that make use of cryptographic algorithms are facing the same problem as well, because most of the algorithms are associated with a limited lifespan. No matter the algorithms are designed for confidentiality or integrity, it is crucial to maintain the security property in long-term periods in order to defend future attacks. In the future, we will carry on our research for other applications that require long-term security.

REFERENCES

[1] ISO/IEC 10118-1. 2016. *Information technology – Security techniques – Hash functions – Part 1: General*. Standard.

[2] Elli Androulaki, Artem Barger, Vita Bortnikov, Christian Cachin, Konstantinos Christidis, Angelo De Caro, David Enyeart, Christopher Ferris, Gennady Laventman, Yacov Manevich, et al. 2018. Hyperledger fabric: a distributed operating system for permissioned blockchains. In *Proceedings of the thirteenth EuroSys conference*. 1–15.

[3] Frederik Armknecht, Ghassan O Karame, Avikarsha Mandal, Franck Youssef, and Erik Zenner. 2015. Ripple: Overview and outlook. In *International Conference on Trust and Trustworthy Computing*. Springer, 163–180.

[4] Jaysing Bhosale and Sushil Mavale. 2018. Volatility of select crypto-currencies: A comparison of Bitcoin, Ethereum and Litecoin. *Annu. Res. J. SCMS, Pune* 6 (2018).

[5] Blockchain.com. 2011. Website. https://www.blockchain.com/. Accessed: 2021-11-11.

[6] Vitalik Buterin et al. 2014. A next-generation smart contract and decentralized application platform. *White Paper* 3, 37 (2014).

[7] Fengjun Chen, Zhiqiang Liu, Yu Long, Zhen Liu, and Ning Ding. 2018. Secure scheme against compromised hash in proof-of-work blockchain. In *International Conference on Network and System Security*. Springer, 1–15.

[8] Etherscan. 2015. Website. https://etherscan.io/. Accessed: 2021-11-11.

[9] Ilias Giechaskiel, Cas Cremers, and Kasper Bonne Rasmussen. 2016. On Bitcoin Security in the Presence of Broken Crypto Primitives. *IACR Cryptol. ePrint Arch.* 2016 (2016), 167.

[10] Shafi Goldwasser, Silvio Micali, and Ronald L Rivest. 1988. A digital signature scheme secure against adaptive chosen-message attacks. *SIAM Journal on computing* 17, 2 (1988), 281–308.

[11] Lov K Grover. 1996. A fast quantum mechanical algorithm for database search. In *Proceedings, 28th Annual ACM Symposium on the Theory of Computing*. 212–219.

[12] Hussein Hellani, Abed Ellatif Samhat, Maroun Chamoun, Hussein El Ghor, and Ahmed Serhrouchni. 2018. On blockchain technology: Overview of bitcoin and future insights. In *2018 IEEE International Multidisciplinary Conference on Engineering Technology (IMCET)*. IEEE, 1–8.

[13] Guangkai Ma, Chunpeng Ge, and Lu Zhou. 2020. Achieving reliable timestamp in the bitcoin platform. *Peer-to-Peer Networking and Applications* 13 (2020), 2251–2259.

[14] Ralph C Merkle. 1989. A certified digital signature. In *Conference on the Theory and Application of Cryptology*. Springer, 218–238.

[15] Satoshi Nakamoto. 2008. *Bitcoin: A peer-to-peer electronic cash system.* Technical Report.

[16] National Institute of Standards and Technology (NIST). 2013. *Digital Signature Standard (DSS)*. Standard.

[17] National Institute of Standards and Technology (NIST). 2017. *NIST Policy on Hash Functions*. Standard. Online available: https://csrc.nist.gov/projects/hash-functions/nist-policy-on-hash-functions.

[18] Masashi Sato and Shin'ichiro Matsuo. 2017. Long-term public blockchain: Resilience against compromise of underlying cryptography. In *2017 26th International Conference on Computer Communication and Networks (ICCCN)*. IEEE, 1–8.

[19] Peter W Shor. 1999. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM review* 41, 2 (1999), 303–332.

[20] Pawel Szalachowski. 2018. (Short Paper) Towards More Reliable Bitcoin Timestamps. In *2018 Crypto Valley Conference on Blockchain Technology (CVCBT)*. IEEE, 101–104.

[21] Gavin Wood et al. 2014. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum Project Yellow Paper* 151, 2014 (2014), 1–32.

[22] Dylan Yaga, Peter Mell, Nik Roby, and Karen Scarfone. 2019. Blockchain technology overview. *arXiv preprint arXiv:1906.11078* (2019).