Message Control for Blockchain Rewriting

***Abstract*—Blockchain rewriting is necessary for modifying illegal or invalid messages included in blockchain transactions, while maintaining the consistency of subsequent blocks in the blockchain. However, selecting arbitrary messages for blockchain rewriting is not desirable as it defeats the purpose of blockchain rewriting. In this work, we propose a new security primitive *message-controlled chameleon hash* (MCH) and apply it for message control in blockchain rewriting. The proposed MCH enables permitted parties to select candidate messages from designated message sets for blockchain rewriting at the message level. It can be easily integrated into both permissioned and permissionless blockchains.**

***Index Terms*—Blockchain Rewriting, Chameleon Hash, Message Control**

I. Introduction

Blockchain is a distributed ledger technology that was first introduced in the context of Bitcoin [24]. Blockchain grows by one block at a time, where the hash of each block is linked to its previous block. Each block consists of a set of transactions which is mapped to a single hash value through a Merkle tree [23], and this hash value is used in the computation of the hash of the block. Each transaction, on the other hand, may contain various information objects which could be recorded in the blockchain. A key feature of blockchain is that the transactions cannot be modified once they appear in the blockchain. But, mutability is necessary for blockchain applications. For example, the “Right to be Forgotten” imposed by GDPR in Europe allows people to have the right to ask for deleting their personal data [3].

There are two research questions regarding blockchain rewriting. The first question is which party can rewrite. For example, a user creates a mutable transaction associated with a public key, while another user who possesses the secret key can rewrite the mutable transaction [9], [12]. Another example is the scenario where multiple users (or miners) rewrite a recorded transaction using a consensus-based e-voting system [14], [30]. It requires a certain number of miners to agree on the same modification together. In this work, we denote modifier as any user involved in the blockchain rewriting and denote owner as any user who created the (mutable) transactions.

The second question is what object information is allowed to be rewritten and *how to rewrite it*. For most blockchain systems, a transaction contains various objects, such as in/out-script, transaction amount, and timestamp [2]. We denote each rewritable object as a message for convenience. Many blockchain rewriting solutions have been proposed to rewrite certain messages included in a transaction. However, they lack message control. The modifier can choose arbitrary messages to rewrite the recorded ones in the blockchain. This work focuses on the second research question about message control in both permissioned and permissionless blockchains.

Motivation. Message control is essential to blockchain rewriting. An application scenario is that payment-transactions are required to be recorded in a blockchain, where certain recorded transactions may later be modified under certain conditions. One typical example is selling a non-fungible token (NFT) [5] in an e-auction. There are different types of auctions for which the final transaction price is different from the highest bidding price. Since each transaction price represents the value of a NFT in the auction, it is desirable to enforce message control in blockchain rewriting instead of using multiple transactions to revise the transaction price (e.g., give the change to the bidder in another transaction).

Consider a Vickrey auction [8] in which the highest bidder wins, but the striking price is equal to the second highest-bid. For instance, the seller (i.e., owner) of a NFT may choose a bid set. For example, he chooses the reserve price to be ₿3, and every bid is ₿1 up. In this case, the possible *message set* is $(₿3, ₿4, ₿5, ₿6, \cdots)$. If Alice bids for ₿4, she will sign a mutable transaction with a message set (₿4, ₿3). If Bob bids for ₿5, he will sign a mutable transaction with a message set (₿5, ₿4, ₿3). If the e-auction is executed in a smart contract, the smart contract itself can be considered as the modifier. During the bidding period, if the smart contract receives 10 bids over the reserve price, it can compare all prices and transfer the money from the highest bidder to the seller of the NFT. The transaction price is modified to the second-highest price in the Vickrey auction, and all other bids are not executed[1]. If Bob wins the auction and the second-highest price is ₿4, the final transaction price paid by Bob will be modified to ₿4. There are also other types of auction with changing bidding price, such as a Dutch auction (a descending-price auction where the price drops continuously with time, until a bid is received), open-outcry double auction (buyers and sellers can modify bids based on knowledge gained from other bids). We stress that the striking or changing price (e.g., the second-highest price in a Vickrey auction or the continuously dropped price in a Dutch auction) is assumed to be acceptable numbers, not arbitrary ones.

Another example is a *buy limit order* transaction in the market, where a payer (i.e., owner) can buy a cryptocurrency at a specified price or better. A payer is willing to buy one BTC with ETH with the exchange rate from 1:30 up to 1:32. The payer creates a transaction containing a statement “I spend ETH 30 to buy BTC 1 from the payee” and appends it to the blockchain. The payer additionally selects a *message set* including monetary values ranging from ETH 30 to ETH 32, and links the message set to that transaction. Later, the payee (i.e., modifier) chooses a value from the message set to rewrite the transaction amount. The payee may change the amount

[1]Alternatively, Alice can add ₿0 to her message set and the message is changed to ₿0 if she lose her bid. However, it will incur some transaction fee in the blockchain.

from ETH 30 to ETH 31 if the market rate is 1:31 when the exchange takes place.

Arbitrary or malicious rewriting by the modifier in the examples above is dangerous. For example, a modifier may abuse their rewriting privilege and replace the recorded messages with incorrect ones, such as "I spend ETH 100 to buy BTC 1 from the payee." To ensure message control for blockchain rewritings, we re-design the underlying cryptographic primitive: chameleon hash [21]. Chameleon hash enables a modifier to rewrite the hashed message without changing the hash output and without breaking the links between consecutive blocks in a chain. In this work, the core objective is to design a message-controlled chameleon hash (MCH), which ensures that: 1) a message m can be redacted to m' if m' is selected from a designated message set $\mathsf{M} = \{m, m', ...\}$ (which includes multiple messages m, m') specified by an owner such that

$$\mathsf{Hash}_{\mathtt{pk}}(m; r) \quad = \quad \mathsf{Hash}_{\mathtt{pk}}(m'; r'),$$

where $\mathtt{pk}$ is the chameleon public key ($\mathtt{sk}$ is the corresponding trapdoor), and r, r' are randomness; and 2) the proposed MCH can be integrated into permissioned and permissionless blockchains.

Our Contributions. The challenge of designing for MCH is to prevent the modifier from finding collisions with respect to arbitrarily selected messages. Since the modifier has the rewriting privilege (i.e., holding the trapdoor $\mathtt{sk}$), she can always find a collision for any arbitrary message which may appear in the blockchain. This contradicts our design goal. The ideal solution is to enforce control on the message selection at the time of collision finding.

To address this challenge, we apply a message-controlled technique to the collision finding algorithm. Specifically, MCH requires two trapdoors to find a collision. One is the chameleon trapdoor $\mathtt{sk}$; the other is a *relationship value*, which is derived from an original message and every new message. The original message is the hashed one present in the chain that is to be replaced by one of the new messages. We call them in general candidate messages, which belong to the message set. The relationship value between the original and new messages can be regarded as the second trapdoor, which serves as an input to the collision finding algorithm.

We rely on a chameleon hash-based accountable assertion to generate relationship values between candidate messages. Accountable assertion binds messages (i.e., statements in the original paper [26]) to contexts in an accountable way. If two contradicting messages are detected in the same context, any public user can extract the trapdoor of the accountable assertion. It was initially used to prevent double-spending attacks: the trapdoor is extracted if two pairs of colliding (message, assertion) values are given. We use the extracted trapdoor as a relationship value between a modified message and the original message of a mutable transaction.

The major contributions are summarized as follows.

- *New Primitive.* We introduce a new primitive called message-controlled chameleon hash (MCH), which is used to secure message-level blockchain rewriting. The control on the rewritten message is essential to blockchain rewriting because it *prevents* a chameleon trapdoor holder from modifying the hashed message maliciously.
- *Practical Instantiation.* We present a practical instantiation of MCH, and we show the detailed evaluation to validate its practicality. The evaluation shows that MCH incurs a moderate computational and storage cost as compared to the standard chameleon hash.
- *Applications for Blockchain Rewriting.* We show that MCH can be integrated into both permissioned and permissionless blockchains for message-level rewriting. The proof-of-concept implementation shows that blockchain rewriting based on our approach incurs almost no overhead to chain validation when compared to the immutable blockchain.

A. Related Work

Blockchain Rewriting. Blockchain rewriting was firstly introduced by Ateniese et al. [9]. The proposed solution enables a block in a blockchain to be modified by a modifier, by replacing the use of regular SHA256 hash function with a chameleon hash (CH) in block generation [21]. The hashing of CH is parametrized by a public key $\mathtt{pk}$, and CH behaves like a collision-resistant hash function if the secret key $\mathtt{sk}$ (or trapdoor) is unknown. The modifier who has the trapdoor can find collisions and output new (message, randomness) values without changing the hash value.

Camenisch et al. [10] introduced a new cryptographic primitive: chameleon hash with ephemeral trapdoor (CHET), which can be regarded as a variant of CH. CHET requires that a modifier must have two trapdoors for rewriting: one trapdoor $\mathtt{sk}$ is associated with the public key $\mathtt{pk}$; the other one is an ephemeral trapdoor chosen by the party who initially computed the hash value. CHET provides more control in rewriting in the sense that the party, who computed the hash value, can decide whether the modifier (i.e., holder of $\mathtt{sk}$) shall be able to rewrite the hash by providing/withholding the ephemeral trapdoor.

Derler et al. [12] proposed a policy-based chameleon hash (PCH) that supports fine-grained blockchain rewriting. PCH replaces the public-key encryption scheme in CHET by ciphertext-policy attribute-based encryption (ABE) scheme. A modifier possessing a chameleon trapdoor can modify the transaction if her attribute set corresponding to the chameleon trapdoor satisfies the embedded access policy. PCH enables rewriting at a transaction-level – which is more practical compared to block-level rewriting as described in [9]. However, PCH is suitable for permissioned blockchain, since a key generation center (KGC) is needed for ABE. Later, Tian et al. proposed an accountable PCH for blockchain rewriting (PCHBA) [31]. The proposed PCHBA enables the modifiers of transactions to be held accountable for the modified transactions. In particular, PCHBA allows KGC to resolve any dispute over modified transactions.

Puddu et al. [25] proposed μchain: a mutable blockchain. A transaction owner introduces a set of transactions, including an active transaction and multiple inactive transactions, where the inactive transactions are possible versions of the transaction data (namely, mutations) encrypted by the transaction owner. The decryption keys are distributed among miners using Shamir's secret sharing scheme [29]. The transaction owner enforces certain policies to define who is allowed to trigger mutations in which context. Upon receiving a mutation-trigger request, a set of miners runs a Multi-Party Computation (MPC) protocol to reconstruct the decryption key, decrypt the appropriate version of the transaction and publish it as an active transaction. μchain incurs considerable overhead due to the use of MPC protocols across multiple miners.

Deuber et al. [14] introduced an efficient redactable blockchain in the permissionless setting. The proposed protocol relies on a consensus-based e-voting system [20], such that the modification is executed in the chain if a modification request from a user gathers enough votes from miners (we call it V-CH for convenience). When integrated into the existing Bitcoin system, V-CH incurs only a small additional overhead. In a follow-up work, Thyagarajan et al. [30] introduced a protocol to repair blockchains (e.g., fix buggy contracts, or remove illicit contents), which acts as a publicly verifiable layer on top of any permissionless blockchain. The proposed protocol allows a user to propose a repair first. Then, a group of users output a decision on the repair proposal after a publicly verifiable deliberation process. Eventually, miners check whether the decision adheres to the repair policy and update the contents in the repaired block.

Jia et al. [19] introduced a new fine-grained redactable blockchain. Specifically, a transaction includes two parts: personal data (that could be recorded), and security-related data (e.g., in/out-script and transaction amount). Their solution allows the user to rewrite the recorded personal data, and the security-related data can be rewritten by a semi-trusted regulator. In particular, they propose a new CH scheme to achieve the fine-grained blockchain rewriting. We show the differences as follows: 1) our solution focuses on message control during rewriting. Their focus is access control on personal/security-related data. 2) our solution can *prevent* any trapdoor holders from maliciously modifying any data inside a transaction. In contrast, their solution allows the regulator (resp. miner) to check/monitor the malicious rewriting by the user (resp. regulator). 3) our proposed MCH scheme ensures indistinguishability (IND), which is a fundamental security property in most CH-based schemes. However, their CH scheme lacks IND property.

Derler and Slamanig [13] introduced an extended sanitizable signature with controlled modifications. Specifically, the signer specifies admissible modifications ADM and a set limitations V. Then, the sanitizer can modify the message-signature pair if and only if the modifications are compatible with ADM and V. Our proposed solution focuses on chameleon hash functions for redactable blockchains. Thus, the research on sanitizable signatures with controlled modifications is beyond the scope of this work.

There are mainly two types of blockchain rewritings in the literature: CH-based [9], [12], [31], [19], and non CH-based [25], [14], [30]. Specifically, CH-based blockchain rewritings allow one or more trusted modifiers to rewrite mutable transactions. The non-CH-based solution requires a threshold number of parties (or miners) to rewrite the redactable blockchains. We stress that one can apply both the CH-based schemes (including our proposed MCH) and the non-CH-based solutions to redactable blockchains.

TABLE I: The comparison between the blockchain rewriting solutions. CH-based means CH-based blockchain rewriting. Setting indicates either permissioned (perm) or permissionless (permless) setting, or both of them. Granularity means the blockchain rewriting is performed in a block, transaction (trans), or message (msg) level.

	[9]	[25]	[12]	[14]	[31]	[19]	Ours
CH-based	✓	×	✓	×	✓	✓	✓
Setting	both	both	perm	permless	perm	both	both
Granularity	block	trans	trans	trans	trans	msg	msg
Msg-Control	×	×	×	×	×	×	✓

In this work, we use chameleon hash cryptographic primitive to secure blockchain rewriting. Table I shows that our proposed MCH supports a controlled rewriting at the message-level in both permissioned and permissionless settings. Message-level rewriting means that a permitted modifier can modify various messages (e.g., transaction amount) inside a mutable transaction, which is more fine-grained than block-level and transaction-level modifications. Message-controlled (i.e., Msg-Control) rewriting means that a message in a transaction can be modified in a controlled manner (to be precise, the modification value must be chosen from a permissible message set only).

II. Building Blocks

In this section, we introduce chameleon hash functions and accountable assertions, which are used in our proposed construction.

A. Chameleon Hash

We will use chameleon hash function for blockchain rewriting. A chameleon hash (CH) function is a randomized hash function that is collision-resistant but provides a trapdoor [21]. Given the trapdoor $\mathtt{sk}$, a message m with a randomness r, and a new message m', it computes a new randomness r', such that $h = \mathsf{Hash}_{\mathtt{pk}}(m; r) = \mathsf{Hash}_{\mathtt{pk}}(m'; r')$. We call them collided (message, randomness) values (or collision).

Definition 1: A public-coin chameleon hash (CH) function consists of the following algorithms.

- $\mathsf{KeyGen}(1^\lambda)$: It takes a security parameter λ as input, outputs a chameleon key pair $(\mathtt{sk}, \mathtt{pk})$.
- $\mathsf{Hash}(\mathtt{pk}, m)$: It takes the chameleon public key $\mathtt{pk}$, and a message $m \in \mathcal{M}$ as input, outputs a chameleon hash h, a randomness r. Note that $\mathcal{M} = \{0,1\}^*$ denotes a general message space. We sometimes write $h = \mathsf{Hash}_{\mathtt{pk}}(m; r)$ when we talk about collision.
- $\mathsf{Verify}(\mathtt{pk}, m, h, r)$: It takes the chameleon public key $\mathtt{pk}$, message m, chameleon hash h and randomness r as input, output a bit $b \in \{0,1\}$.
- $\mathsf{Adapt}(\mathtt{sk}, m, m', h, r)$: It takes the chameleon secret key $\mathtt{sk}$, messages m, m', chameleon hash h and randomness r as input, outputs a new randomness r'.

We assume that the Adapt algorithm always verifies if the chameleon hash h given is valid, or it outputs $\perp$. Correctness is straightforward and is given in [21].

A chameleon hash function needs to achieve collision-resistance and indistinguishability. Specifically, collision-resistance means that any malicious party cannot find two pairs (m, r) and (m', r') that map to the same chameleon hash h. Indistinguishability means that a randomness r does not reveal if it was generated through Hash or Adapt. A formal definition on collision-resistance and indistinguishability can be found at [11].

A secret-coin CH is different from a public-coin CH in that a check string ξ is output from the function Hash. The check string ξ replaces the randomness r in Verify and Adapt [9].

1) Semi-public-coin Chameleon Hash: In this paper, we use a new variant called *semi-public-coin* CH with $(h, \xi) \leftarrow \mathsf{Hash}(\mathtt{pk}, m)$, and use the check string ξ to replace the randomness r in Verify only. The output of Adapt is modified to $(r', \xi') \leftarrow \mathsf{Adapt}(\mathtt{sk}, m, m', h, r)$.

Consider the discrete logarithm (DL)-based CH with public key $\mathtt{pk} = (g, \chi = g^{\mathtt{sk}})$. Then $\mathsf{Hash}(\mathtt{pk}, m)$ outputs $h = \chi^m g^r$ and $\xi = (R = g^r, \pi)$, where π is a NIZK for the value $\log_g R$. In Verify, the hash value h can be verified by checking if $h = \chi^m R$ and if π is valid. In Adapt, we compute $r' = (m - m') \cdot \mathtt{sk} + r$ and $\xi' = (R' = g^{r'}, \pi')$, where π' is a NIZK for $\log_g R'$. Hence, it is a semi-public-coin CH. We use Schnorr NIZK [18] in this DL-based CH. Roughly, it is a non-interactive variant of Schnorr identification scheme via Fiat-Shamir transformation [16].

B. Accountable Assertions

The accountable assertion (AA) was proposed in [26] to bind messages in an accountable way. If a user asserts two contradicting messages, the user's trapdoor (i.e., chameleon secret key $\mathtt{sk}$) can be extracted. Accountable assertion includes two security guarantees: extractability and secrecy. Extractability guarantees that the trapdoor is extracted given two pairs of (message, assertion) values. Secrecy prevents the extraction of the trapdoor if no collision is given.

The concrete construction of AA in [26] is based on the idea of chameleon authentication trees (CAT) [28]. Loosely speaking, it is a combination of collision-resistant hash functions $\mathtt{H}$ and chameleon hash functions (CH) to built a Merkle-style tree. Let chameleon hash be $(h, r) = \mathsf{Hash}(\mathtt{pk}, m)$ for some chameleon public key $\mathtt{pk}$. We show a simple tree with 3-arity in Figure 1. Specifically, each tree node has three slots (A, B, C) and each slot may have a child (represented by a solid line). Initially, each leaf slot is associated with a dummy (message, randomness) values $(u_{i,j}, v_{i,j})$, where $i \in \{1, 2, 3\}$ is the depth of the node, and $j \in \{1, \cdots, 6\}$ is the position of the node. E.g., $(B_{3,4}, v_{3,4}) = \mathsf{Hash}(\mathtt{pk}, u_{3,4})$.

We show the assertion operation in Figure 1. Assume a user asserts a message m to a slot $B_{3,4}$. To do so, the user computes the first collision in $(B_{3,4}, v_{3,4}) = \mathsf{Hash}(\mathtt{pk}, u_{3,4})$. That is, the user uses his trapdoor $\mathtt{sk}$ to find a randomness $r_{3,4}$ such that $B_{3,4} = \mathsf{Hash}_{\mathtt{pk}}(u_{3,4}; v_{3,4}) = \mathsf{Hash}_{\mathtt{pk}}(m; r_{3,4})$. Next, the user needs to compute the elements $A_{3,4}, C_{3,4}, A_{2,2}, B_{2,2}, C_{2,2}, A_{1,1}, B_{1,1}, C_{1,1}$ which are

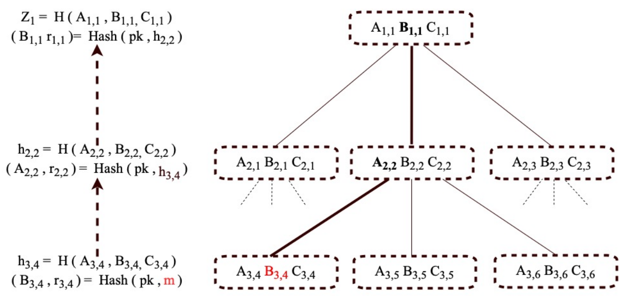
$Z_1 = \mathrm{H}(A_{1,1}, B_{1,1}, C_{1,1})$
$(B_{1,1}\ r_{1,1}) = \mathrm{Hash}(pk, h_{2,2})$
$h_{2,2} = \mathrm{H}(A_{2,2}, B_{2,2}, C_{2,2})$
$(A_{2,2}, r_{2,2}) = \mathrm{Hash}(pk, h_{3,4})$
$h_{3,4} = \mathrm{H}(A_{3,4}, B_{3,4}, C_{3,4})$
$(B_{3,4}, r_{3,4}) = \mathrm{Hash}(pk, m)$
$A_{1,1}\ \mathbf{B_{1,1}}\ C_{1,1}$
$A_{2,1}\ B_{2,1}\ C_{2,1}$
$\mathbf{A_{2,2}}\ B_{2,2}\ C_{2,2}$
$A_{2,3}\ B_{2,3}\ C_{2,3}$
$A_{3,4}\ B_{3,4}\ C_{3,4}$
$A_{3,5}\ B_{3,5}\ C_{3,5}$
$A_{3,6}\ B_{3,6}\ C_{3,6}$

Fig. 1: An assertion in a Merkle-style tree. $\mathtt{H}$ indicates collision-resistant hash function, Hash means chameleon hash function CH.

the slots along the assertion path from $B_{3,4}$ to the root. These elements can be computed from the corresponding values $(u_{i,j}, v_{i,j})$. To assert $(A_{3,4}, B_{3,4}, C_{3,4})$ to the parent $A_{2,2}$, the user computes $h_{3,4} = \mathtt{H}(A_{3,4}, B_{3,4}, C_{3,4})$. Again, the user uses the trapdoor $\mathtt{sk}$ to find a randomness $r_{2,2}$ such that $A_{2,2} = \mathsf{Hash}_{\mathtt{pk}}(u_{2,2}; v_{2,2}) = \mathsf{Hash}_{\mathtt{pk}}(h_{3,4}; r_{2,2})$. The user repeats this procedure up to the root. In this example, the assertion τ on message m includes three randomness $(r_{3,4}, r_{2,2}, r_{1,1})$ so that any public user can verify the (message, randomness) values along the assertion path from $B_{3,4}$ to the root. This completes the informal description of the underlying assertion data structure. Here is the resulting assertion public key $\mathtt{apk} = (\mathtt{pk}, Z_1)$, assertion secret key $\mathtt{ask} = \mathtt{sk}$ and some auxiliary data (e.g., a pseudo-random key κ for generating dummy (message, randomness) values).

If the user asserts two messages in slot $B_{3,4}$, anyone can extract trapdoor $\mathtt{sk}$ because there exist two pairs of collided (message, randomness) values such that $B_{3,4} = \mathsf{Hash}_{\mathtt{pk}}(m; r_{3,4}) = \mathsf{Hash}_{\mathtt{pk}}(m'; r'_{3,4})$, where $r'_{3,4}$ is a randomness associated with message m'. Similarly, if two pairs of collided (message, randomness) values occur in slot $A_{2,2}$, the trapdoor $\mathtt{sk}$ can also be extracted.

C. Verifiable Secret Sharing

Verifiable Secret Sharing (VSS) scheme allows a dealer to share a secret s among n parties, and s can be recovered from $k+1$ parties. We give the Feldman's VSS scheme [15] below:

- $\mathsf{Setup}(1^\lambda, s)$: For a secret $s \in \mathbb{Z}_q$, the dealer picks some random coefficients $\alpha_1, \ldots \alpha_k \in \mathbb{Z}_q$. Define a polynomial $p(x) = s + \alpha_1 x + \ldots + \alpha_k x^k$. The dealer publishes a set of commitments $\mathbb{C} = (C_0 = g^s, C_1 = g^{\alpha_1}, \ldots, C_k = g^{\alpha_k})$ and stores coefficients $(\alpha_1, \ldots \alpha_k)$.
- $\mathsf{Share}(s, (\alpha_1, \ldots \alpha_k), e_i)$: For a secret s, coefficients $(\alpha_1, \ldots \alpha_k)$ and an index e_i, the dealer sends a share $s_i = p(e_i)$ to party i for $i \in \{1, n\}$.
- $\mathsf{Check}(s_i, e_i, \mathbb{C})$: Each party i can validate his share by checking if $g^{s_i} = \prod_{j=0}^{k} C_j^{e_i^j}$. Output 1 if it is correct.

D. Extended Accountable Assertions

We introduce a new primitive called extended accountable assertion (eAA) to provide message control in blockchain. AA was designed to *prohibit* changes: if there are two asserted messages, the assertion secret key is extracted. On the other hand, we design eAA to *control* changes with the use of VSS: if there are two asserted messages, a relationship value is extracted; and the relationship value is a key share that can be verified in VSS.

1) Our eAA Construction: The proposed eAA works as follows: 1) the Feldman's VSS scheme [15] is used to split an assertion secret key into multiple shares; 2) a share s_i is used as a relationship value and it is extracted from two pairs of collided (message, assertion) values; 3) the extracted share (i.e., relationship value) can be checked by the VSS scheme. Now, we describe our eAA, which is built on top of AA in [26] and VSS in [15].

- $\mathsf{KeyGen}(1^\lambda, \mathsf{M})$: It takes a security parameter λ as input, runs $(\mathtt{apk}', \mathtt{ask}', \mathtt{auxsk}') \leftarrow \mathsf{AA.KeyGen}(1^\lambda)$, and $(\mathbb{C}, (\alpha_1, \ldots, \alpha_k)) \leftarrow \mathsf{VSS.Setup}(1^\lambda, \mathtt{ask}')$, where $C_0 = \mathtt{apk}' \in \mathbb{C}$. For each message $m \in \mathsf{M}$, it generates a CH public key Z and secret key z. It outputs $\mathtt{ask} = \mathtt{ask}', \mathtt{auxsk} = (\mathtt{auxsk}', (\alpha_1, \ldots, \alpha_k), \{z\}), \mathtt{apk} = (\mathtt{apk}', \{m, Z\})$.
- $\mathsf{Assert}(\mathtt{ask}, \mathtt{auxsk}, m)$: It takes the assertion secret keys $\mathtt{ask}, \mathtt{auxsk}$ and a message $m \in \mathsf{M}$ as input. The context C includes the commitments $\mathbb{C}$ in VSS. It computes $e_i = G(m, C_i)$ for some pseudorandom function G and $C_i \in \mathbb{C}$, and generates a share by $s_i \leftarrow \mathsf{VSS.Share}(\mathtt{ask}', (\alpha_1, \ldots, \alpha_k), e_i)$ for all i indicating the depth of the tree. It returns $\tau/\perp \leftarrow \mathsf{AA.Assert}(\{s_i\}, \mathtt{auxsk}, C, m)$, where the share s_i is used when computing the nodes in depth i. In particular, z is used as the secret key of CH in the leaf node when running $\mathsf{AA.Assert}$ in [26].
- $\mathsf{Verify}(\mathtt{apk}, m, \tau)$: It takes the assertion public key $\mathtt{apk}$, a message m, and an assertion τ as input, outputs $b \leftarrow \mathsf{AA.Verify}(\mathtt{apk}, m, \tau)$. In particular, Z is used as the public key of CH in the leaf node when running $\mathsf{AA.Verify}$ in [26].
- $\mathsf{Extract}(\mathtt{apk}, m, m', \tau, \tau')$: It takes the assertion public key $\mathtt{apk}$, a collided (message, assertion) values $[(m; \tau), (m'; \tau')]$ as input, outputs $s_i/\perp \leftarrow \mathsf{AA.Extract}(\mathtt{apk}, m, m', \tau, \tau')$. The extracted share s_i shows that m and m' collide at a node of depth i, and s_i is a valid share since $\mathsf{VSS.Check}(s_i, G(m, C_i), \mathbb{C}) = 1$ for some $C_i \in \mathbb{C}$. Note that this check process involves $\mathtt{apk}'$ as $C_0 = \mathtt{apk}' \in \mathbb{C}$.

The eAA is *correct* if for all security parameters λ, all keys $(\mathtt{ask}, \mathtt{apk}, \mathtt{auxsk}) \leftarrow \mathsf{KeyGen}(1^\lambda, \mathsf{M})$, all $m \in \mathsf{M}$, and all $\tau \leftarrow \mathsf{Assert}(\mathtt{ask}, \mathtt{auxsk}, m)$, we have $1 = \mathsf{Verify}(\mathtt{apk}, m, \tau)$. Note that only key shares $\{s_i\}$ can be extracted in case of extractability, not $\mathtt{ask}$. The auxiliary secret key $\mathtt{auxsk}$ includes some secret values used to derive key shares from $\mathtt{ask}$. For example, one can split assertion secret key $\mathtt{ask}$ into multiple key shares using VSS. The $\mathtt{auxsk}$ also includes secret values associated with messages in M. We associate each message with a secret value. The intention is to enforce the assertion key holder chooses a message $m \in \mathsf{M}$ to generate an assertion τ. If an asserted message $m \notin \mathsf{M}$, then $\tau = \perp$. The key differences between AA and eAA include: 1) an asserted message is chosen from a message set. 2) an extracted key is a key share of $\mathtt{ask}$. The eAA needs to achieve indistinguishability and collision-resistance. Indistinguishability means that the randomness involved in an assertion does not reveal whether it was obtained from directly hashing or computing them using the assertion key shares. Note that an assertion includes a set of (message, randomness) values. Collision-resistance means that an adversary cannot generate assertions on messages when the assertion secret key is unknown. The indistinguishability and collision-resistance of our eAA is referred to Appendix A.

2) Message Control Technique: Let M be a set of individual messages. Message-control requires two publicly verifiable conditions: 1) A candidate message $m \in \mathsf{M}$ is verified by a corresponding CH public key Z in $\mathtt{apk}$. 2) Two candidate messages are linked via a relationship value. This condition is achieved using two pairs of collided (message, assertion) values. One is $\tau \leftarrow \mathsf{Assert}(\mathtt{ask}, \mathtt{auxsk}, m)$, the other is $\tau' \leftarrow \mathsf{Assert}(\mathtt{ask}, \mathtt{auxsk}, m')$. Each assertion corresponds to a tree path in the Merkle-style tree (see Figure 2), and a relationship is established if two tree paths have overlap. We show the detailed generation of the relationship value below.

We consider a general case where any candidate message pair can establish a unique relationship. The proposed eAA scheme is designed to handle this case, as the assertion secret key is split into multiple key shares. Each key share may act as a relationship value between two candidate messages. The idea of message-control is to let each generation of an assertion link to a tree path, and use key shares to find collisions from a leaf node to the tree root. If two tree paths have an overlap point, a particular key share can be extracted from two pairs of collided (message, randomness) values. The extracted key share confirms the relationship between two candidate messages, and we denote the extracted key share as a relationship value c. We show how to find various relationships between different message pairs in Figure 2. For example, assertion τ on message m includes three slots in a tree path $(A_{3,4}, A_{2,2}, B_{1,1})$, where $(A_{3,4}, r_{3,4}) = \mathsf{Hash}(Z, m)$. Assertion τ' on message m' in another tree path includes slots $(C_{3,4}, A_{2,2}, B_{1,1})$, where $(C_{3,4}, r'_{3,4}) = \mathsf{Hash}(Z', m')$. One can see that the first overlap point is $A_{2,2} \leftarrow (m, m')$, which means a relationship between two messages is established. In particular, a key share can be extracted by any public user because the key share was used to find the collided (message, randomness) values at slot $A_{2,2}$. Similarly, we can find another relationship for another message pair: $B_{1,1} \leftarrow (m, m'')$.

If a relationship value is used in a chameleon hash CH, we can achieve the claimed message control. For example, a relationship value c is used to generate a chameleon hash (h, r) on a message $m \in \mathsf{M}$. Later, a trapdoor holder selects a new message $m' \in \mathsf{M}$ and generates the chameleon hash (h, r') using the same c. In this example, message m links to m' via the relationship value c.

III. The Proposed Construction

In this section, we present the definition and the construction of MCH, respectively.

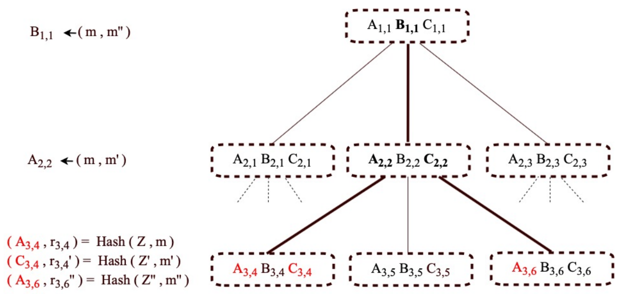
$B_{1,1} \leftarrow (m, m'')$
$A_{2,2} \leftarrow (m, m')$
$(A_{3,4}, r_{3,4}) = \text{Hash}(Z, m)$
$(C_{3,4}, r_{3,4}') = \text{Hash}(Z', m')$
$(A_{3,6}, r_{3,6}'') = \text{Hash}(Z'', m'')$
$A_{1,1}$ $B_{1,1}$ $C_{1,1}$
$A_{2,1}$ $B_{2,1}$ $C_{2,1}$
$A_{2,2}$ $B_{2,2}$ $C_{2,2}$
$A_{2,3}$ $B_{2,3}$ $C_{2,3}$
$A_{3,4}$ $B_{3,4}$ $C_{3,4}$
$A_{3,5}$ $B_{3,5}$ $C_{3,5}$
$A_{3,6}$ $B_{3,6}$ $C_{3,6}$

Fig. 2: Assertions in a Merkle-style tree. The tree depth is three, and each tree node has three slots. Slot $A_{2,2}$ indicates a relationship between messages (m, m'), and slot $B_{1,1}$ indicates a relationship between messages (m', m''). Hash means chameleon hash function CH.

A. Definition

A message-controlled chameleon hash (MCH) consists of the following algorithms.

- $\mathsf{Setup}(1^\lambda)$: It takes a security parameter λ as input, outputs a chameleon key pair $(\mathtt{sk}, \mathtt{pk})$.
- $\mathsf{KeyGen}(\mathtt{pk}, \mathsf{M})$: It takes the chameleon public key $\mathtt{pk}$, and a set of candidate messages M as input, outputs an assertion key pair $(\mathtt{ask}, \mathtt{apk})$. Note that the relationship values between candidate messages in M are implicitly established after this key generation.
- $\mathsf{Hash}(\mathtt{pk}, \mathtt{ask}, m)$: It takes the chameleon public key $\mathtt{pk}$, the assertion secret key $\mathtt{ask}$, and a message $m \in \mathsf{M}$ as input, outputs a chameleon hash h, a randomness r, and a relationship value c.
- $\mathsf{Verify}(\mathtt{pk}, \mathtt{apk}, m, h, r)$: It takes the chameleon public key $\mathtt{pk}$, assertion public key $\mathtt{apk}$, message m, chameleon hash h and randomness r as input, output a bit $b \in \{0, 1\}$.
- $\mathsf{Adapt}(\mathtt{sk}, \mathtt{ask}, c, m, m', h, r)$: It takes the chameleon secret key $\mathtt{sk}$, the assertion secret key $\mathtt{ask}$, messages m, m', relationship value c, chameleon hash h and randomness r as input, outputs a new randomness r'.

The MCH is *correct* if for all security parameters λ, all keys $(\mathtt{sk}, \mathtt{pk}) \leftarrow \mathsf{Setup}(1^\lambda)$, $(\mathtt{ask}, \mathtt{apk}) \leftarrow \mathsf{KeyGen}(\mathtt{pk}, \mathsf{M})$, for all $m \in \mathsf{M}$, for all $(h, r, c) \leftarrow \mathsf{Hash}(\mathtt{pk}, \mathtt{ask}, m)$, for all $m' \in \mathsf{M}$, for all $r' \leftarrow \mathsf{Adapt}(\mathtt{sk}, \mathtt{ask}, c, m, m', h, r)$, we have $1 = \mathsf{Verify}(\mathtt{pk}, \mathtt{apk}, m, h, r) = \mathsf{Verify}(\mathtt{pk}, \mathtt{apk}, m', h, r')$.

B. Security Models

We consider two security guarantees, including indistinguishability and collision-resistance. Since the definitions are various in the literature, we follow the standard indistinguishability and collision-resistance models defined in [11].

Indistinguishability. Informally, for a chameleon hash, an adversary cannot decide whether its randomness was freshly generated using Hash algorithm or was created using Adapt algorithm. We define a formal experiment between an adversary $\mathcal{A}$ and a challenger $\mathcal{S}$ in Figure 3. The security experiment allows $\mathcal{A}$ to access a $\mathsf{HashOrAdapt}$ oracle which ensures that the randomness does not reveal whether it was obtained from Hash or Adapt algorithm. Note that $\mathcal{A}$ generates a message set M, and $\mathcal{A}$ chooses candidate messages from M to query $\mathsf{HashOrAdapt}$ oracle.
We define the advantage of $\mathcal{A}$ as

$$\mathtt{Adv}_{\mathcal{A}}^{\mathrm{IND}}(\lambda) = |\Pr[\mathsf{Exp}_{\mathcal{A}}^{\mathrm{IND}}(\lambda) \to 1] - 1/2|.$$

Definition 2: A MCH scheme is indistinguishable if for any probabilistic polynomial-time (PPT) $\mathcal{A}$, $\mathtt{Adv}_{\mathcal{A}}^{\mathrm{IND}}(\lambda)$ is negligible in λ.

Collision Resistance. Informally, an adversary attempts to find collisions without using relationship values. The adversary gets access an Adapt' oracle: it takes messages (m, m') and a relationship value c as input, outputs a collision for the adversarially chosen hash and records (m, m') under a list $\mathcal{Q}$. Note that $m, m' \in \mathsf{M}$ and c is associated with messages (m, m'). In this experiment, $\mathcal{S}$ chooses a message set M and the corresponding relationship values. $\mathcal{A}$ wins if it outputs a collision for an adversarially generated chameleon hash h^*, while the message m^* was not previously queried to the Adapt' oracle. We define a formal experiment in Figure 4.
We define the advantage of $\mathcal{A}$ as

$$\mathtt{Adv}_{\mathcal{A}}^{\mathrm{CR}}(\lambda) = \Pr[\mathsf{Exp}_{\mathcal{A}}^{\mathrm{CR}}(\lambda) \to 1].$$

Definition 3: A MCH scheme is collision-resistant if for any PPT $\mathcal{A}$, $\mathtt{Adv}_{\mathcal{A}}^{\mathrm{CR}}(\lambda)$ is negligible in λ.

```
Experiment $\mathsf{Exp}_{\mathcal{A}}^{\mathrm{IND}}(\lambda)$
$(\mathsf{sk}, \mathsf{pk}) \leftarrow \mathsf{Setup}(1^\lambda), b \leftarrow \{0, 1\}$
$(\mathsf{ask}, \mathsf{apk}) \leftarrow \mathsf{KeyGen}(\mathsf{pk}, \mathsf{M})$
$b' \leftarrow \mathcal{A}^{\mathsf{HashOrAdapt}(\mathsf{sk}, \mathsf{ask}, \cdots, b)}(\mathsf{pk}, \mathsf{apk})$
  where $\mathsf{HashOrAdapt}(\mathsf{sk}, \mathsf{ask}, \cdots, b)$ on input $m, m'$ :
    return $\perp$, if $m, m' \notin \mathsf{M} \vee c = \perp$
    $(h_0, r_0, c) \leftarrow \mathsf{Hash}(\mathsf{pk}, \mathsf{ask}, m')$
    $(h_1, r_1', c) \leftarrow \mathsf{Hash}(\mathsf{pk}, \mathsf{ask}, m)$
    $r_1 \leftarrow \mathsf{Adapt}(\mathsf{sk}, \mathsf{ask}, c, m, m', h_1, r_1')$
    return $(h_b, r_b, c)$
    return $\perp$ if $r_0 = \perp \vee r_1 = \perp$
return 1, if $b' = b$; else, return 0.
```

Fig. 3: Indistinguishability.

```
Experiment $\mathsf{Exp}_{\mathcal{A}}^{\mathrm{CR}}(\lambda)$
$(\mathsf{sk}, \mathsf{pk}) \leftarrow \mathsf{Setup}(1^\lambda), (\mathsf{ask}, \mathsf{apk}) \leftarrow \mathsf{KeyGen}(\mathsf{pk}, \mathsf{M}),$
$\mathcal{Q}_1, \mathcal{Q}_2 \leftarrow \emptyset, (m^*, r^*, m'^*, r'^*, h^*) \leftarrow \mathcal{A}^{\mathsf{Hash}', \mathsf{Adapt}'}(\mathsf{pk}, \mathsf{apk})$
  where $\mathsf{Hash}'(\mathsf{pk}, \mathsf{ask}, \cdot)$ on input $m$ :
    $(h, r, c) \leftarrow \mathsf{Hash}(\mathsf{pk}, \mathsf{ask}, m)$
    let $\mathcal{Q}_1 \leftarrow \mathcal{Q}_1 \cup \{(\mathsf{pk}, h, m, c)\}$
    return $(h, r, c)$
  where $\mathsf{Adapt}'(\mathsf{sk}, \mathsf{ask}, \cdots)$ on input $m, m', h, r, c$ :
    return $\perp$, if $(\mathsf{pk}, h, m, c) \notin \mathcal{Q}_1$
    return $\perp$, if $m, m' \notin \mathsf{M} \vee c = \perp$
    $r' \leftarrow \mathsf{Adapt}(\mathsf{sk}, \mathsf{ask}, c, m, m', h, r)$
    let $\mathcal{Q}_2 \leftarrow \mathcal{Q}_2 \cup \{(\mathsf{pk}, h, m, m', c)\}$
    return $r'$
return 1, if $1 = \mathsf{Verify}(\mathsf{pk}, \mathsf{apk}, m^*, h^*, r^*)$
    $= \mathsf{Verify}(\mathsf{pk}, \mathsf{apk}, m'^*, h^*, r'^*) \wedge (\mathsf{pk}, h^*, m^*, \cdot) \in \mathcal{Q}_1$
    $\wedge (\mathsf{pk}, h^*, m^*, c^*, \cdot) \notin \mathcal{Q}_2 \wedge m^* \neq m'^*$
else, return 0.
```

Fig. 4: Collision-Resistance.

C. Construction

An Envisioned Solution. We consider a two-party setting: a content owner and a permitted modifier. The content owner publishes its original transaction that includes a set of acceptable values and the authorized modifier's public key. Later, the modifier selects an appropriate value from the set of acceptable values, signs the value with her secret key, and updates the transaction with the selected value. Once the updated transaction is included in the blockchain, it represents the new state of the application while the original transaction remains in the blockchain.

Suppose the modifier selects an *arbitrary* value which is not from the set of acceptable values and signs the value with her secret key. As a result, the blockchain will not accept the updated transaction. This is because if such updated transactions are submitted to the blockchain, all participants, including miners (the number of honest miners is $> 51\%$), can verify that they contain *unacceptable* values and ignore such transactions. Using our proposed MCH, we can guarantee that only updated transactions that contain acceptable values can be accepted into the blockchain.

High-level Description. We assume that every modifier has two key pairs $(\mathtt{sk}_c, \mathtt{pk}_c), (\mathtt{sk}_m, \mathtt{pk}_m)$ and that modifiers' public keys are known to all users. The key pair $(\mathtt{sk}_c, \mathtt{pk}_c)$ is associated with a semi-public-coin chameleon hash scheme CH, and key pair $(\mathtt{sk}_m, \mathtt{pk}_m)$ is associated with a public key encryption scheme PKE. For example, a transaction owner creates a mutable transaction T, including a chameleon hash h, a message m, a randomness r, and a message set $\mathsf{M} = \{m, m'\}$ (assume two messages only). The randomness r contains a ciphertext C encrypting a randomness ρ associated with CH under $\mathtt{pk}_m$, and an assertion τ. During rewriting, the modifier verifies message-randomness pair (m, r) under h and $\mathtt{pk}_c$, computes a new randomness r' containing a new ciphertext C' encrypted a randomness ρ' under $\mathtt{pk}_m$, and a new assertion τ'. The randomness ρ' is derived from $\mathtt{sk}_c$ and a relationship value c. We stress that c is associated with two messages (m, m'), and is decided by the transaction owner. The proposed MCH consists of the following building blocks.

- An extended accountable assertion scheme $\mathsf{eAA} = (\mathsf{KeyGen}, \mathsf{Assert}, \mathsf{Verify}, \mathsf{Extract})$.
- A semi-public-coin chameleon hash scheme $\mathsf{CH} = (\mathsf{KeyGen}, \mathsf{Hash}, \mathsf{Verify}, \mathsf{Adapt})$.
- A public key encryption scheme $\mathsf{PKE} = (\mathsf{KeyGen}, \mathsf{Enc}, \mathsf{Dec})$.

We assume that a transaction owner hashes a message m, while a transaction modifier replaces it by a new message m'.

- $\mathsf{Setup}(1^\lambda)$: It generates a CH key pair $(\mathtt{sk}_c, \mathtt{pk}_c) \leftarrow \mathsf{CH.KeyGen}(1^\lambda)$ and an encryption key pair $(\mathtt{sk}_e, \mathtt{pk}_e) \leftarrow \mathsf{PKE.KeyGen}(1^\lambda)$ for the modifier. It outputs $\mathtt{pk}_m = (\mathtt{pk}_c, \mathtt{pk}_e)$ and $\mathtt{sk}_m = (\mathtt{sk}_c, \mathtt{sk}_e)$.
- $\mathsf{KeyGen}(\mathtt{pk}_m, \mathsf{M})$: The owner outputs an assertion key pair $(\mathtt{ask}', \mathtt{auxsk}', \mathtt{apk}) \leftarrow \mathsf{eAA.KeyGen}(1^\lambda, \mathsf{M})$. It outputs $\mathtt{apk}$, $\mathtt{ask} = (\mathtt{ask}', \mathtt{auxsk}', \mathtt{apk})$. In addition, the ciphertext $C_{\mathtt{ask}} \leftarrow \mathsf{PKE.Enc}(\mathtt{pk}_e, \mathtt{ask})$ is sent to the modifier. The transaction contains $(\mathtt{pk}_m, \mathtt{apk})$.
- $\mathsf{Hash}(\mathtt{pk}_m, \mathtt{ask}, m)$: The owner takes a message $m \in \mathsf{M}$ as input, he computes $\tau \leftarrow \mathsf{eAA.Assert}(\mathtt{ask}', \mathtt{auxsk}', m)$.
  He picks a tree depth i^* and computes a share e_{i^*} using $\mathsf{VSS.Share}$ in eAA. He computes a CH public key $\mathtt{pk}'$ using $\mathtt{pk}_c$ and e_{i^*}. Then he computes $(h, \xi) \leftarrow \mathsf{CH.Hash}(\mathtt{pk}', m; \rho)$ using a randomness ρ, and computes the ciphertext $C_\rho \leftarrow \mathsf{PKE.Enc}(\mathtt{pk}_e, \rho)$. He outputs a hash h, a randomness $r = (\xi, C_\rho, \tau, \mathtt{pk}')$ and a relationship value $c = e_{i^*}$.
- $\mathsf{Verify}(\mathtt{pk}_m, \mathtt{apk}, m, h, r = (\xi, C_\rho, \tau, \mathtt{pk}'))$: Any public user can verify whether a given hash h is valid. Output 1 if: $1 \leftarrow \mathsf{CH.Verify}(\mathtt{pk}_c, m, h, \xi)$ and $1 \leftarrow \mathsf{eAA.Verify}(\mathtt{apk}, m, \tau)$. Otherwise output 0.
- $\mathsf{Adapt}(\mathtt{sk}_m, \mathtt{ask}, c, m, m', h, r = (\xi, C_\rho, \tau, \mathtt{pk}')$: The modifier with a chameleon secret key $\mathtt{sk}_m$, a message $m' \in \mathsf{M}$, and the relationship value $\mathtt{sk}_c$ performs the following operations
  1) Verify $1 \leftarrow \mathsf{Verify}(\mathtt{pk}_m, \mathtt{apk}, m, h, r)$.
  2) Compute $\tau' \leftarrow \mathsf{eAA.Assert}(\mathtt{ask}', \mathtt{auxsk}', m')$.
  3) Compute a CH secret key $\mathtt{sk}'$ by using the share e_{i^*} and $\mathtt{sk}_c$. Compute $(\rho', \xi') \leftarrow \mathsf{CH.Adapt}(\mathtt{sk}', m, m', \rho, h)$, where $\rho \leftarrow \mathsf{PKE.Dec}(\mathtt{sk}_e, C_\rho)$. Compute the ciphertext $C_{\rho'} \leftarrow \mathsf{PKE.Enc}(\mathtt{pk}_e, \rho')$.
  4) Output $(h, r = (\xi', C'_\rho, \tau', \mathtt{pk}'), c)$, where $C_{\rho'} \leftarrow \mathsf{PKE.Enc}(\mathtt{pk}_e, \rho')$.

Correctness and Security. Given two pairs of collided (message, assertion) values (m, τ) and (m', τ') that corresponds to the same h, any public user can extract an assertion key share $c \leftarrow \mathsf{eAA.Extract}(\mathtt{apk}, m, m', \tau, \tau')$, verify the chameleon hash $\mathsf{CH.Verify}(\mathtt{pk}_c, m, h, \xi) = \mathsf{CH.Verify}(\mathtt{pk}_c, m', h, \xi') = 1$. The public verification requires that both the owner and the modifier generate assertions (τ, τ'). If the modifier avoids using the assertion key share c to generate assertion τ', the relationship between two messages cannot be publicly verified. The security analysis is referred to Appendix B.

IV. Instantiation and Evaluation

Let $\mathbb{G}$ denote a cyclic group with prime order q and generator g. Let $\mathtt{H}_0 : \{0,1\}^* \rightarrow \mathbb{Z}_q$, and $\mathtt{H}_1 : \{0,1\}^* \rightarrow \mathbb{G}$ be collision-resistant hash functions. Let F_κ be a pseudo-random function, where κ is a secret key. All candidate messages can be converted from bit-strings to values in $\mathbb{Z}_q$ by a pseudo random function $\mathcal{F}_z : \mathcal{M} \times \mathcal{K} \rightarrow \mathbb{Z}_q$ for some key z in a key space $\mathcal{K}$. We denote that the message set M contains the converted messages in $\mathbb{Z}_q$.

A. Instantiation

- $\mathsf{Setup}(1^\lambda)$: It takes a security parameter λ as input, a modifier outputs a chameleon key pair $(\mathtt{sk}_c, \mathtt{pk}_c)$, where $\mathtt{sk}_c \in \mathbb{Z}_q^*$, $\mathtt{pk}_c = g^{\mathtt{sk}_c}$. It generates an ElGamal encryption key pair $(\mathtt{sk}_e, \mathtt{pk}_e = g^{\mathtt{sk}_e})$. It outputs $\mathtt{pk}_m = (\mathtt{pk}_c, \mathtt{pk}_e)$ and $\mathtt{sk}_m = (\mathtt{sk}_c, \mathtt{sk}_e, \mathtt{pk}_m)$.
- $\mathsf{KeyGen}(\mathtt{pk}_m, \mathsf{M})$: A transaction owner chooses a pseudo-random key $\kappa \leftarrow \{0,1\}^\lambda$, and publishes a Merkle-style tree Λ with depth k. Each tree node includes a number of dummy (message, randomness) values: $\{(u_j^i, v_j^i)\}^{n_0}$, where $u_j^i = \mathsf{F}_\kappa(p, a_j, 0)$, $v_j^i = \mathsf{F}_\kappa(p, a_j, 1)$, $i \in \{1, k\}$ is the depth of the node, $j \in \{1, n_0\}$ is the index inside the node, p denotes the unique identifier for the position of a tree node in Λ, a_j denotes the position of u_j^i inside a tree node, n_0 denotes

the number of dummy messages in a tree node, and F_κ denotes a pseudorandom function. Meanwhile, it maintains an initially empty set L of *used* leaf positions.

The owner runs $\mathsf{eAA.KeyGen}(1^\lambda, \mathsf{M})$ to obtain $\mathtt{apk}'$, $\mathtt{ask}'$ and $\mathtt{auxsk}'$. It outputs $\mathtt{apk} = (\mathtt{apk}', \Lambda)$, $\mathtt{ask} = (\mathtt{ask}', \mathtt{auxsk}', \kappa, \mathtt{apk})$. The ciphertext $C_{\mathtt{ask}} \leftarrow \mathsf{ElGamal.Enc}(\mathtt{pk}_e, \mathtt{ask})$ is sent to the modifier.

- $\mathsf{Hash}(\mathtt{pk}_m, \mathtt{ask}, m)$: To hash a candidate message m at a leaf node Y_k of depth k, recall that $\mathtt{ask} = (\mathtt{ask}', \mathtt{auxsk}', \kappa, \mathtt{apk})$ and:
  - the dummy (message, randomness) values are in the form of $(u^i_j, v^i_j) = [\mathsf{F}_\kappa(p, a_j, 0), \mathsf{F}_\kappa(p, a_j, 1)]$, where a_j denotes the position of u^i_j within Y_i.
  - for the VSS scheme, $\mathtt{ask}'$ is shared by the polynomial $f(x) = \mathtt{ask}' + \sum_{\ell=1}^{k} \alpha_\ell \cdot x^\ell$ for some $\alpha_\ell \in \mathtt{auxsk}'$. The commitments $C_i = g^{\alpha_i}$ are used to check the validity of the shares. Denote $F(x) = g^{f(x)}$.
  - since $m \in \mathsf{M}$, there is a corresponding value $z \in \mathtt{auxsk}'$.

  The owner performs the following operations. Firstly, we show the generation of an assertion τ. The owner computes an assertion path $(Y_k, a_k, Y_{k-1}, a_{k-1}, \cdots, Y_1, a_1)$ from a leaf Y_k to the root Y_1.

  1) Computing $y^k_{a_k}$: Compute dummy values $(u^k_{a_k}, v^k_{a_k})$ and $e_k = \mathtt{H}_0(m, C_k, a_k)$. Assert the message m to leaf node Y_k by computing the collision $v'^k_{a_k} = v^k_{a_k} + (u^k_{a_k} - m) \cdot z/f(e_k)$. The message m links to a slot a_k in Y_k, so the owner adds a_k to L. Compute the entry $y^k_{a_k} = \mathtt{H}_1(g^{z\cdot m} \cdot g^{f(e_k)\cdot v'^k_{a_k}}, v'^k_{a_k})$.
  2) Compute the remaining entries $y^k_j = g^{u^k_j} \cdot g^{f(e_k)\cdot v^k_j}$ for $j \in \{1, n_0\} \setminus \{a_k\}$. The leaf Y_k stores the entries $(y^k_1, \cdots, y^k_{n_0})$.
  3) Set an intermediate value $Z_k = \mathtt{H}_0(y^k_1, \cdots, y^k_{n_0})$, and $\zeta_k = (y^k_1, \cdots, y^k_{a_{k-1}}, F(e_k), y^k_{a_{k+1}} \cdots, y^k_{n_0})$.

  Next, it computes the remaining nodes up to the root for $w = k-1, \cdots, 1$ as follows.

  1) Assert the intermediate value Z_{w+1} to node Y_w by computing $v'^w_{a_\mathrm{w}} = v^w_{a_\mathrm{w}} + (u^w_{a_\mathrm{w}} - Z_{w+1})/f(e_w)$, where $e_w = \mathtt{H}_0(u^w_{a_\mathrm{w}}, C_w, a_\mathrm{w})$. Compute the entry $y^w_{a_\mathrm{w}} = g^{Z_{w+1}} \cdot g^{f(e_w)\cdot v'^w_{a_\mathrm{w}}}$.
  2) Compute the remaining entries $y^w_j = g^{u^w_j} \cdot g^{f(e_w)\cdot v^w_j}$ for $j \in \{1, n_0\} \setminus \{a_\mathrm{w}\}$. The leaf Y_w stores the entries $(y^w_1, \cdots, y^w_{n_0})$.
  3) Set $Z_w = \mathtt{H}_0(y^w_1, \cdots, y^w_{n_0})$, and $\zeta_w = (y^w_1, \cdots, y^w_{a_{\mathrm{w}-1}}, F(e_w), y^w_{a_{\mathrm{w}+1}} \cdots, y^w_{n_0})$.

  The assertion on message m is $\tau = \{(v'^k_{a_k}, \zeta_k, a_k), \cdots, \underline{(v'^i_i, \zeta_i, a_i)}, \cdots, (v'^1_1, \zeta_1, a_1), \underline{Z_1}\}$. Note that the component ζ_i includes the value $F(e_i) = g^{f(e_i)}$. The assertion public key in the root node Y_1 is computed as $Z_1 = \mathtt{H}_0(y^1_1, \cdots, y^1_{n_0})$.

  Denote the index $i^* \in [1, k]$ which allows modification and compute key share $c = f(e_{i^*})$ and $\mathtt{pk}' = g^c$. Pick a random $\rho \in \mathbb{Z}^*_q$ and compute chameleon hash $h = g^{c\cdot m} \cdot \mathtt{pk}^\rho_c$, $\xi = (R = \mathtt{pk}^\rho_c, \pi)$, where π is the ZK proof of ρ. Encrypt ρ by $C_\rho \leftarrow \mathsf{ElGamal.Enc}(\mathtt{pk}_e, \rho)$. It outputs a hash h, a randomness $r = (\xi, C_\rho, \tau, \mathtt{pk}')$ and a relationship value c.
- $\mathsf{Verify}(\mathtt{pk}_m, \mathtt{apk}, m, h, r = (\xi, C_\rho, \tau, pk'))$: Any public user can verify whether a given chameleon hash h is valid or not. The algorithm outputs 1, if the following conditions hold:
  1) Compute $e_\mathrm{w} = \mathtt{H}_0(m, C_\mathrm{w}, a_\mathrm{w})$ and verify the key shares $F(e_\mathrm{w})$ for w$= 1, \ldots, k$:
  $$F(e_\mathrm{w}) \overset{?}{=} g^{\mathtt{ask}'} \cdot {C_1}^{e_\mathrm{w}} \cdot {C_2}^{e^2_\mathrm{w}} \cdots {C_\mathrm{w}}^{e^\mathrm{w}_\mathrm{w}}.$$
  2) Parse $\xi = (R, \pi)$. Verify the chameleon hash $h \overset{?}{=} F(e_{i^*})^m \cdot R$ and π is a valid ZK proof for $\mathtt{pk}_c$.
  3) For level k, compute $y^k_{a_k} = \mathtt{H}_1((g^z)^m \cdot F(e_k)^{v'^k_{a_k}}, v'^k_{a_k})$ and $Z_k = \mathtt{H}_0(y^k_1, \cdots, y^k_{n_0})$ (the remaining inputs y^k_i are from ζ_k). Compute recursively, for level $w = k-1, \ldots 1$, that $y^w_{a_\mathrm{w}} = g^{Z_{w+1}} \cdot F(e_w)^{v'^w_{a_\mathrm{w}}}$ and $Z_w = \mathtt{H}_0(y^w_1, \cdots, y^w_{n_0})$. Finally check if the assertion public key is equal to Z_1.
- $\mathsf{Adapt}(\mathtt{sk}_m, \mathtt{ask}, c = f(e_{i^*}), m, m', h,\ r = (\xi, C_\rho, \tau, \mathtt{pk}')$: The modifier performs the following operations
  1) Verify the chameleon hash tuple as described above.
  2) Generate a new assertion τ' on the message m' using the assertion secret key $\mathtt{ask}$. Specifically, the modifier performs the following steps to assert the message m' to a node Y_k.
     a) Compute the collision $v'^k_{a_k} = v^k_{a_k} + (u^k_{a_k} - m') \cdot z'/f(e'_k)$, where $f(e'_k) = \mathtt{ask}' + \sum_{\ell=1}^{k} \alpha_\ell \cdot e'^\ell_k$, $e'_k = \mathtt{H}_0(m', C_k, a_k)$. In addition, the modifier adds a_k to L.
     b) Compute the entry $y^k_{a_k} = \mathtt{H}_1(g^{z'\cdot m'} \cdot g^{f(e'_k)\cdot v'^k_{a_k}}, v'^k_{a_k})$.
     c) Compute the remaining entries $y^k_j = g^{u^k_j} \cdot g^{f(e'_k)\cdot v^k_j}$ for $j \in \{1, n_0\} \setminus \{a_k\}$. The leaf Y_k stores the entries $(y^k_1, \cdots, y^k_{n_0})$.
     d) Set $Z_k = \mathtt{H}_0(y^k_1, \cdots, y^k_{n_0})$, and $f_k = (y^k_1, \cdots, F(e'_k), \cdots, y^k_{n_0})$.
     e) Compute the remaining nodes up to the root for $w = k-1, \cdots, 1$ using the same method as described in the Hash algorithm. For example, the modifier uses a key share $f(e_w)$ for $e_w = \mathtt{H}_0(u^w_j, C_w, a_j)$ to assert an intermediate value Z_{w+1} to node Y_w. Eventually, the assertion on message m' is $\tau' = \{(v'^k_{a_k}, f_k, a_k), \cdots, \underline{(v'^i_i, f_i, a_i)}, \cdots, (v'^1_1, f_1, a_1), Z_1\}$.
  3) Compute a new randomness as $\rho' = \rho + (m - m') \cdot c/\mathtt{sk}_c$, where $\rho \leftarrow \mathsf{ElGamal.Dec}(\mathtt{sk}_e, C_\rho)$, $c = f(e_{i^*})$.
  4) Compute the value $\xi' = (R' = \mathtt{pk}^{\rho'}_c, \pi')$ where π' is a ZK proof for ρ' and compute an ElGamal ciphertext $C_{\rho'} \leftarrow \mathsf{ElGamal.Enc}(\mathtt{pk}_e, \rho'))$.
  5) Output $(h, r' = (\xi', C_{\rho'}, \tau', \mathtt{pk}'), c)$.

B. Evaluation

We evaluate the performance of MCH for blockchain rewriting based on a proof-of-concept implementation in Java. On a high-level, two schemes are constructed. Scheme 1 creates an immutable blockchain system with basic functionalities and a PoW consensus mechanism. The simulated scheme 1 is "healthy": satisfying both persistence and liveness according to [17]. The scheme 1 is specifically designed to include ten blocks, and each block consists of 100 transactions (one can extend it to real-world transactions such as a block containing

3500 transactions). We generate ten threads to simulate ten nodes in a peer-to-peer network, and each thread can be regarded as a single node. All these threads are managed by a modular for broadcasting transactions and new mined blocks to all nodes. A chain of blocks is established with PoW by consolidating transactions broadcast by the ten nodes. The code is available on GitHub [6].

Scheme 2 is similar to Scheme 1, except that the proposed MCH is applied to certain mutable transactions (to be precise, some of the mutable contents present in these transactions) in Scheme 2. In this scheme, a randomly chosen node serves as a modifier to rewrite the blockchain, while the remaining nodes can contribute to the consensus. Any node can generate mutable transactions associated with a modifier's chameleon public key. Later, the modifier holding the chameleon secret key can perform rewriting.

C. Benchmarking

Our benchmarking experiments are performed on a desktop computer with Inter Core i7 at 2.9GHz, and 16GB RAM. We choose a hash function SHA-256. In Scheme 2, if there is no mutable transaction in a block, the block is formed and validated in the same way as in Scheme 1. We thus focus on the evaluation of mutable transactions in Scheme 2.

We first evaluate the performance of MCH which includes the evaluation of: Hash, Verify, Adapt algorithms and correctness check (i.e., checking the relationship between two (message, randomness) values with the same chameleon hash output). Then, we measure the running time for generating a Merkle tree on a set of transactions (including mutable ones) and for validating blocks in PoW. Each mutable transaction is set to contain two candidate messages in our evaluation.

Figure $5(a)$ shows that the running time of Hash algorithm ranges between 0.5 ms and 4.7 ms as the depth of the tree changes from five to fifteen, and the number of dummy messages in each tree node varies from five to fifteen. Figures $5(b)$–(d) show similar trends for the running time of Verify, Adapt, and correctness check.

Table II shows the performance comparison between Scheme 1 and Scheme 2 in terms of transaction generation time, transaction overhead, Merkle tree generation time, and PoW validation time. We set ten mutable transactions in each block, which means 10% of the transactions are mutable according to [14], and each transaction is assumed to contain one mutable content. We also set each mutable transaction to contain two candidate messages and tree parameters $k = 10$ and $n_0 = 10$. Let $\mathcal{L}_{\text{CH}}$ denote the length of chameleon hash CH, which includes a chameleon hash $\mathcal{L}_{\text{CH}_h}$ and a randomness $\mathcal{L}_{\text{CH}_\rho}$. Let the size of (plain) candidate messages associated with a mutable transaction be $\mathcal{L}_{\mathsf{M}}$. We denote the length of eAA and PKE as $\mathcal{L}_{\text{eAA}}$ and $\mathcal{L}_{\text{PKE}}$, respectively.

From Table II, we observe that the generation of a mutable transaction in Scheme 2 takes approximately 2 ms, which is longer than 0.002 ms in Scheme 1. The storage overhead of a mutable transaction in Scheme 2 is higher than a normal transaction in Scheme 1. Specifically, the additional storage overhead includes $\mathcal{L}_{\text{CH}_h} + \mathcal{L}_{\text{eAA}} + 2\mathcal{L}_{\text{PKE}} + \mathcal{L}_{\mathsf{M}}$, where $2\mathcal{L}_{\text{PKE}}$ consists of one encryption on an assertion secret key `ask`, and one encryption on a randomness ρ. Here, we summarize two points. First, the additional computational and storage costs depend on the structure of the assertion tree and the number of candidate messages involved in each mutable transaction. The additional overhead in Scheme 2 seems not to be negligible. But, we argue that some scalability techniques (e.g., Sharding [22]) can be used to reduce the additional overhead, thus improving scheme 2's performance. Second, the performance of Merkle tree generation and PoW validation in Scheme 2 is nearly identical to that in Scheme 1, which confirms that the proposed MCH can be easily integrated into the existing blockchain systems.

TABLE II: The performance comparison between Scheme 1 and Scheme 2. Tx (time) denotes the time to generate a transaction, and Tx (overhead) denotes the transaction's storage cost.

	Tx (time)	Merkle	PoW	Tx (overhead)
Scheme 1:	0.002 ms	75710 ns	24 ns	256-bit
Scheme 2:	2.04 ms	73935 ns	22 ns	256-bit + $*$

V. Integrating MCH into Blockchains

In this section, we show how the proposed message-controlled chameleon hash (MCH) can be integrated into a blockchain. Specifically, we consider an immutable blockchain in the permissioned setting (where certain permission is required to access the blockchain) as well as in the permissionless setting (where anyone can join the network as a node and access the blockchain). In addition, we address the issue regarding transaction inconsistency in the blockchain [25], [14], when the monetary value involved in a transaction already present in the blockchain needs to be modified.

A. Transaction Inconsistency

MCH is designed to modify data present in the blockchain in a controlled manner. The types of modification could be various depending on the application scenarios. For example, one might want to modify the monetary amount of a transaction already registered in the blockchain. However, this can potentially result in transaction inconsistency in the blockchain [25], [14]. This is due to the fact that modifying the amount of a transaction may affect the subsequent transactions present in the blockchain. To the best of our knowledge, transaction inconsistency due to a change in transaction amount has never been rigorously addressed in the context of blockchain rewriting. Note that we address the transaction inconsistency in a single blockchain system. The transaction inconsistency between two or more blockchains is beyond the scope of this work.

Our solution is to use the time-lock mechanism [1] to address transaction inconsistency when the action of rewriting affects transaction amounts. The key idea is to allow rewriting of a transaction's amount prior to a specified timestamp by which the transaction is locked. Once the timestamp is reached, the transaction becomes *spendable* and it cannot

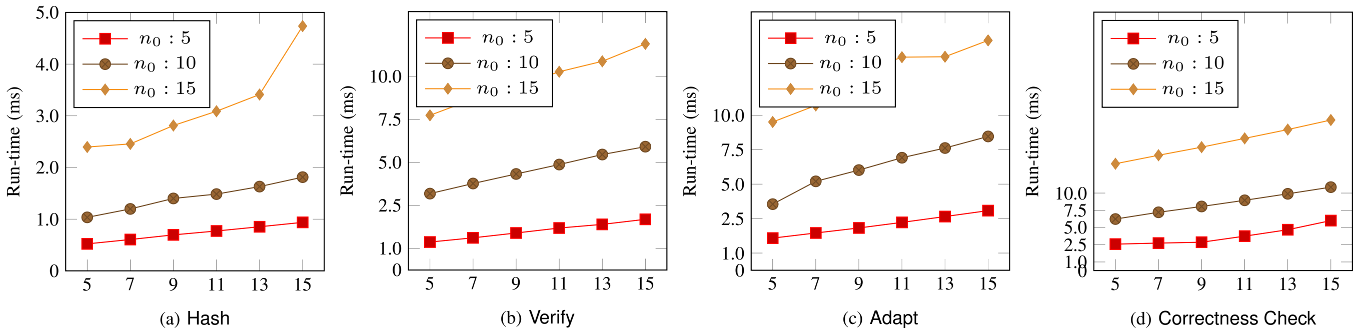

Fig. 5: Run-time of Hash, Verify, Adapt algorithms and correctness check, where n_0 denotes the number of dummy messages in a tree node, and the running time varying with respect to the depth of tree $k \in \{5, 15\}$.

be rewritten further. We consider three types of transactions in our mutable blockchain: normal (immutable) transactions, mutable transactions (that can be modified later) and revision-transactions (or dummy transactions that specify the changes pertaining to mutable transactions previously logged in the blockchain). We note that modifying a transaction must not affect the transaction fee associated with it. So, the set of messages must be chosen in such a way that the transaction fee remains unchanged.

We address transaction inconsistency due to changes in the transaction amount as follows. The transaction modifier is not allowed to modify a transaction that is already spent; otherwise it would require tracing all the subsequent transactions (in a way similar to taint analysis in software security) and changing some of them (which may cause a dispute among multiple parties — we leave this as a future work). To achieve our goal, a time-lock [1] is embedded in each mutable transaction using a timestamp t that is agreed upon and signed by both the payer and the payee (to simplify our description, we focus on the case in which each transaction is associated with a single payer and payee; it is not difficult to extend the case for multiple payers and payees). The modification on the transaction amount must be performed prior to the specified timestamp t. In other words, the payee cannot spend this mutable transaction before time t, and the transaction becomes unchangeable and spendable once t is reached; therefore, monetary transactions can only be rewritten when they are in the *unspent* state. Suppose the blockchain includes a valid transaction Tx with a time-lock t. The transaction modifier later may wish to modify the monetary value of the transaction within time $t - \delta$, where δ is the time elapsed between a transaction getting broadcasted and the transaction getting included in the blockchain, and broadcasts a revision-transaction Tx$'$. Before including the revision-transaction in a block, a miner verifies if the current timestamp is less than the time-lock t mentioned in the previous transaction Tx (i.e., whether Tx is still unspendable).

We note that the time-locked transaction-rewriting mechanism is meaningful in practice. For example, in the NFT scenario, a transaction can be locked for a time period denoting the end of the corresponding auction. On the other hand, for a limit buy order, a transaction can be locked until a future time depending on when the actual exchange is expected to take place.

B. Mutable Blockchains Using MCH

We show how MCH can be used to rewrite blockchains both in the permissioned and permissionless settings. Here we provide a generic description that works with both the settings. In a blockchain, each block is a compact representation of a set of valid transactions stored at the leaf-nodes of a Merkle tree [23]. The hash of the root-node of the tree (say, TX_ROOT) corresponding to a block acts as a commitment to all the transactions included in that particular block. MCH enables one to rewrite some parts of the transactions without changing TX_ROOT. Consequently, the hash values of all the subsequent blocks present in the blockchain also remain intact. We distinguish three types of transactions using two bits (b, b'): $(0, 0)$ for normal transactions, $(0, 1)$ for mutable transactions, and $(1, 0)$ for revision-transactions.

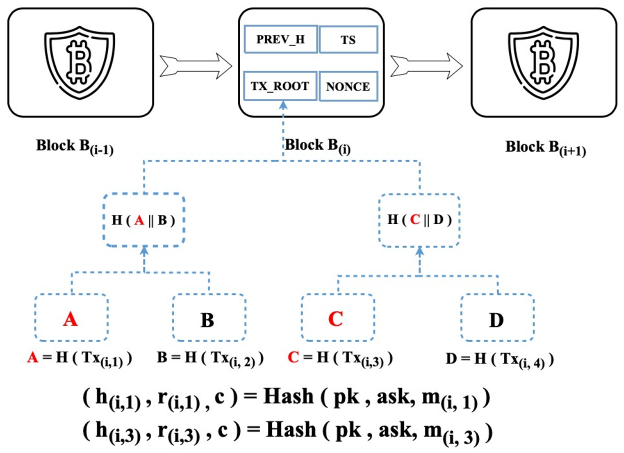

Fig. 6: An example of a block containing both normal and mutable transactions.

We assume that each party in the network has a chameleon secret key sk and the corresponding public key is pk. If a payer (transaction owner) wants to include a mutable content (or message) m along with other immutable content (say, $\bar{m}$) in a transaction, then the transaction includes $\mathsf{Hash}(\mathtt{pk}, \mathtt{ask}, m)$ along with the immutable content $\bar{m}$, where $\mathtt{pk}$ is the public key of the transaction modifier, and $\mathtt{ask}$ is the assertion secret key shared between the transaction owner and the transaction modifier (i.e., the owner chooses $\mathtt{ask}$ and encrypts it using the modifier's public key under the PKE scheme.) Subsequently, all mutable transactions and normal transactions to be included in a block are hashed using the conventional collision-resistant hash function $\mathtt{H}$. Figure 6 illustrates a block B_i containing four transactions $\mathsf{Tx}_{(i,1)}, \mathsf{Tx}_{(i,2)}, \mathsf{Tx}_{(i,3)}$, and $\mathsf{Tx}_{(i,4)}$, where $\mathsf{Tx}_{(i,1)}$ and $\mathsf{Tx}_{(i,3)}$ are mutable transactions chosen by the transaction owner, $\mathsf{Tx}_{(i,2)}$ and $\mathsf{Tx}_{(i,4)}$ are normal transactions. In particular, the mutable transactions $\mathsf{Tx}_{(i,1)}$ and $\mathsf{Tx}_{(i,3)}$ include the ("simplified" version of) chameleon hash tuples $(h_{(i,1)}, r_{(i,1)}, c) = \mathsf{Hash}(\mathtt{pk}, \mathtt{ask}, m_{(i,1)})$ and $(h_{(i,3)}, r_{(i,3)}, c) = \mathsf{Hash}(\mathtt{pk}, \mathtt{ask}, m_{(i,3)})$ along with the other immutable contents $\bar{m}_{(i,1)}$ and $\bar{m}_{(i,3)}$, respectively. The hash values A, B, C, and D are derived by applying the collision-resistant hash function $\mathtt{H}$ on the respective transactions, except that the randomness values corresponding to the mutable transactions $\mathsf{Tx}_{(i,1)}$ and $\mathsf{Tx}_{(i,3)}$ are not included in the computation of A and C using $\mathtt{H}$. TX_ROOT is obtained by constructing Merkle tree on A, B, C, and D. The payer also includes a set of candidate messages M in a transaction (as auxiliary data). Depending on a specific application, the set M is signed by the payer (or the payee), if the payee (or the payer) acts as the transaction modifier. Similarly, every node in the network can validate such a mutable transaction just like a normal transaction with an additional membership check for the mutable content in the set M. Then, it first evaluates the MCH over the mutable content (using the corresponding randomness) and then computes the conventional collision-resistant hash function $\mathtt{H}$ over the output of the MCH (concatenated with the immutable content of the transaction). It verifies all transactions in a block and finally validates TX_ROOT. It then verifies the block using the hash of the previous block and the nonce provided by the respective miner. We note that the chaining among the successive blocks in the blockchain remain the same, i.e., it still uses the conventional collision-resistant hash function $\mathtt{H}$ for this purpose.

When a mutable transaction (more precisely, its mutable content or message) needs to be modified, the transaction modifier with the corresponding secret key $\mathtt{sk}$ can compute a collision in the chameleon hash $h_{(i,1)}$ (or $h_{(i,3)}$) for a new randomness value $r'_{(i,1)}$ (or $r'_{(i,3)}$). We note that the randomness values $r'_{(i,1)}$ and $r'_{(i,3)}$ are not included in the hash computations, and they are included as non-hashed parts of the transaction. As modifying a mutable content of a transaction does not change the corresponding hash due to the collision, TX_ROOT computed over all the transactions present in the corresponding block also remains the same as before. Moreover, $PREV\_H$ value present in the next block in the blockchain does not change, no further modifications are thus required in any of the subsequent blocks in the blockchain.

If certain contents in mutable transactions are hashed using MCH, then the following properties hold: 1) *indistinguishability* — given a transaction containing a mutable content, any outsider cannot distinguish whether the content is the original one or it is modified using MCH; 2) *collision-resistance* — a transaction modifier with rewriting permission can perform modifications only if the chosen new content belongs to set M selected and signed by the other party.

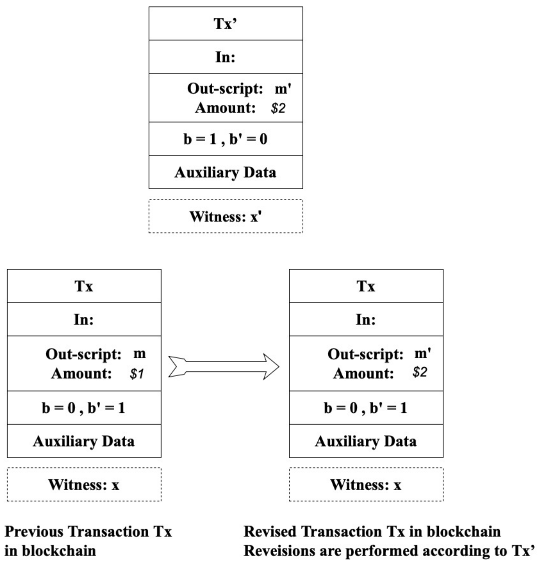

Fig. 7: An example of a revision of a mutable transaction Tx according to a revision-transaction Tx'.

For a transaction Tx with $(b = 0, b' = 1)$, miners treat it as a mutable transaction to be included in a block in the blockchain. On the other hand, a revision-transaction Tx' indicates that a certain previous transaction Tx needs to be modified (e.g., due to changes in the data m' present in the out-script or the transaction amount ₿2 as shown in Figure 7). We note that a revision-transaction Tx' with $(b = 1, b' = 0)$ is always unspendable, and a mutable transaction Tx with $(b = 0, b' = 1)$ is unspendable unless its time-lock has expired. A witness x denotes a digital signature computed on the transaction using the corresponding secret key [2]. While rewriting a valid transaction Tx already included in the blockchain with a time-lock t, the transaction modifier chooses one message from the message set M. Then, the transaction modifier generates a revision-transaction Tx' having the same structure as Tx, except that the new changes in transaction amounts are incorporated in Tx'. We note that Tx' points to the previous transaction Tx (e.g., by using the transaction id of Tx), and the main purpose of a revision-transaction is to help rewriting a previous (mutable) transaction in the blockchain. In other words, one cannot spend a revision-transaction (with $b = 1$), i.e., any other transaction having a revision-transaction in its input script is not considered as valid.

Finally, the transaction modifier broadcasts Tx' to all the peer nodes in the blockchain. Tx' is included in the blockchain (by some miners) in the same way as other transactions are included (i.e., through the consensus). Before including the revision-transaction Tx' (with $b = 1$) in a potential block, a miner checks:

1) whether Tx' is a valid transaction pointing to another valid transaction Tx already present in the blockchain.

2) whether the current timestamp is less than the time-lock t mentioned in the previous transaction Tx (i.e., whether Tx is still unspendable);
3) whether $₿1, ₿2 \in \mathsf{M}$ (i.e, whether the old and new monetary values in Tx and Tx' are selected from the same message set M.
4) whether $1 = \mathsf{Verify}(\mathtt{pk}, \mathtt{apk}, ₿2, h, r')$, where $\mathtt{pk}$ denotes the transaction modifier's chameleon public key, $\mathtt{apk}$ denotes the assertion public key.

Once Tx' is included in the blockchain through consensus, every node in the network modifies the previous transaction Tx (more precisely, the mutable monetary value in it) according to the changes specified in Tx'. After time t, the updated transaction Tx becomes spendable — which eliminates the possibility of transaction inconsistency. We emphasize that we rely mostly on the existing consensus mechanism for registering transactions (both normal transactions and revision-transactions) into the blockchain — which requires small changes in order to incorporate rewriting.

Comparison with [14]. In [14], Deuber et al. proposed a consensus-based e-voting system to integrate redactable transactions into the blockchain. Informally, a user first broadcasts a modification request to the network, which includes a candidate block. Then, there is a voting period when all miners may vote for the candidate block. Eventually, if the request is approved (i.e., a certain number of votes are received in its favor), the original block will be replaced by the candidate block. We summarize the advantages of our techniques over those of [14] as follows: 1) Rewriting the blockchain in our case requires small changes in the existing consensus mechanism (e.g., transactions include fields b and auxiliary data, and the validation procedure for mutable transactions includes MCH verification as well). However, we do not need any separate mechanism to incorporate changes in the blockchain. On the contrary, [14] relies on a separate consensus-based e-voting mechanism to handle modifications in the blockchain (along with the existing consensus for selecting new blocks to be appended to the blockchain). 2) The permissible changes both in our scheme and [14] are fine-grained as the content of a transaction (except its transaction amount) can be modified. However, changes in the monetary amounts of a transaction are allowed in our case but *not* in [14]. 3) The arbitrarily chosen messages (or unacceptable values) never appear in the blockchain using MCH. However, a modification request containing illicit content may gain enough votes under certain collusion attacks using a consensus-based e-voting mechanism [14].

Limitations. Our techniques for rewriting a blockchain have the following limitations. 1) The blockchain records the revision histories of all mutable transactions. 2) In order to maintain a synchronized view of the blockchain with respect to its peers, each node storing a copy of the blockchain should modify original mutable transactions according to revision-transactions. While it is difficult to enforce that each node in the blockchain updates its transactions, the whole system works if a majority of nodes follow its protocol as in the original blockchain system. Nonetheless, a malicious node can always keep separate copies of the original transactions. However, we note that all existing schemes, that rewrite blockchains, suffer from this limitation. 3) Our solution focuses on systems based on unspent transaction outputs or UTXO [7] (e.g., Bitcoin and some of its derivatives such as Zcash [27], Litecoin [4]), where unspent transactions are stored in a list and they are spent in an all-or-nothing fashion. We use the time-lock provision available in Bitcoin transactions in order to handle transaction inconsistency (for changes in monetary values). The time-lock for a mutable transaction in our scheme denotes a time-bound that the message of the transaction (e.g., monetary values) must be finalized within. We note that this restriction is not suitable for certain applications where changes need to be made irrespective of such time-bounds. 4) The transaction owner is assumed to create a valid transaction associated with a message set that contains certain acceptable values. For rewriting, the modifier needs to choose a message from this message set only. We do not consider the scenario where the owner creates an invalid transaction with an empty message set or the modifier rejects to redact the mutable transactions. If such a malicious behavior occurs, one may use other blockchain rewriting approaches with no message control (e.g., [25], [14], [30], which are complementary to this work) to redact those (invalid) transactions.

VI. Conclusion

In this paper, we proposed the first blockchain rewriting based on message-controlled chameleon hash MCH and applied it to address the issue of arbitrary or malicious rewriting in blockchains. Our solution enforces that a permitted modifier should be able to replace a message already in a blockchain by a message chosen from a specified set of messages only. Unlike the previous blockchain rewritings, which support blockchain rewriting at either block-level or transaction-level, in either permissioned setting or permissionless setting, our solution allows blockchain rewriting at message-level in both settings.

References

[1] Bitcoin Developer Guide. https://bitcoin.org/en/developer-guide.
[2] Bitcoin Script. https://en.bitcoin.it/wiki/Script.
[3] General Data Protection Regulation. https://gdpr-info.eu.
[4] Litecoin. https://litecoin.org.
[5] Non-fungible token. https://en.wikipedia.org/wiki/Non-fungible_token.
[6] Rewritable Chain. https://github.com/SMC-SMU/rewritableBlockchain.
[7] UTXO. https://en.wikipedia.org/wiki/Unspent_transaction_output.
[8] Vickrey auction. https://en.wikipedia.org/wiki/Vickrey_auction.
[9] G. Ateniese, B. Magri, D. Venturi, and E. Andrade. Redactable blockchain–or–rewriting history in bitcoin and friends. In *EuroS&P*, pages 111–126, 2017.
[10] J. Camenisch, D. Derler, S. Krenn, H. C. Pöhls, K. Samelin, and D. Slamanig. Chameleon-hashes with ephemeral trapdoors. In *PKC*, pages 152–182, 2017.
[11] D. Derler, K. Samelin, and D. Slamanig. Bringing order to chaos: The case of collision-resistant chameleon-hashes. In *PKC*, pages 462–492, 2020.
[12] D. Derler, K. Samelin, D. Slamanig, and C. Striecks. Fine-grained and controlled rewriting in blockchains: Chameleon-hashing gone attribute-based. In *NDSS*, 2019.
[13] D. Derler and D. Slamanig. Rethinking privacy for extended sanitizable signatures and a black-box construction of strongly private schemes. In *ProvSec*, pages 455–474, 2015.

[14] D. Deuber, B. Magri, and S. A. K. Thyagarajan. Redactable blockchain in the permissionless setting. In *IEEE S&P*, pages 124–138, 2019.
[15] P. Feldman. A practical scheme for non-interactive verifiable secret sharing. In *FOCS*, pages 427–438, 1987.
[16] A. Fiat and A. Shamir. How to prove yourself: Practical solutions to identification and signature problems. In *CRYPTO*, pages 186–194, 1986.
[17] J. Garay, A. Kiayias, and N. Leonardos. The bitcoin backbone protocol: Analysis and applications. In *EUROCRYPT*, pages 281–310, 2015.
[18] F. Hao. Schnorr non-interactive zero-knowledge proof. Technical report, 2017.
[19] Y. Jia, S.-F. Sun, Y. Zhang, Z. Liu, and D. Gu. Redactable blockchain supporting supervision and self-management. In *AsiaCCS*, pages 844–858, 2021.
[20] T. Kohno, A. Stubblefield, A. D. Rubin, and D. S. Wallach. Analysis of an electronic voting system. In *IEEE S&P*, pages 27–40, 2004.
[21] H. Krawczyk and T. Rabin. Chameleon signatures. In *NDSS*, pages 143–154, 2000.
[22] L. Luu, V. Narayanan, C. Zheng, K. Baweja, S. Gilbert, and P. Saxena. A secure sharding protocol for open blockchains. In *CCS*, pages 17–30, 2016.
[23] R. C. Merkle. A certified digital signature. In *CRYPTO*, pages 218–238, 1989.
[24] S. Nakamoto. Bitcoin: A peer-to-peer electronic cash system. 2008.
[25] I. Puddu, A. Dmitrienko, and S. Capkun. μchain: How to forget without hard forks. *IACR Cryptology ePrint Archive*, 2017:106, 2017.
[26] T. Ruffing, A. Kate, and D. Schröder. Liar, liar, coins on fire!: Penalizing equivocation by loss of bitcoins. In *CCS*, pages 219–230, 2015.
[27] E. B. Sasson, A. Chiesa, C. Garman, M. Green, I. Miers, E. Tromer, and M. Virza. Zerocash: Decentralized anonymous payments from bitcoin. In *IEEE S&P*, pages 459–474, 2014.
[28] D. Schröder and H. Schröder. Verifiable data streaming. In *CCS*, pages 953–964, 2012.
[29] A. Shamir. How to share a secret. *Communications of the ACM*, 22(11):612–613, 1979.
[30] S. A. K. Thyagarajan, A. Bhat, B. Magri, D. Tschudi, and A. Kate. Reparo: Publicly verifiable layer to repair blockchains. *arXiv preprint arXiv:2001.00486*, 2020.
[31] Y. Tian, N. Li, Y. Li, P. Szalachowski, and J. Zhou. Policy-based chameleon hash for blockchain rewriting with black-box accountability. In *ACSAC*, pages 813–828, 2020.

Appendix

A. Security of eAA

The security of the extended accountable assertion (eAA) includes indistinguishability and collision-resistance.

```
Experiment $\mathsf{Exp}_{\mathcal{A}}^{\mathrm{IND}}(\lambda)$
$b' \leftarrow \mathcal{A}^{\mathsf{Assert}_{\mathsf{IND}}(b,\cdots)}(\mathsf{M}), b \leftarrow \{0,1\}$
  where $\mathsf{Assert}_{\mathsf{IND}}(b,\cdots)$ on input $m, m'$ :
    return $\perp$, if $m, m' \notin \mathsf{M}$
    $(\mathtt{ask}_0, \mathtt{apk}_0, \mathtt{auxsk}_0) \leftarrow \mathsf{KeyGen}(1^\lambda, m')$
    $(\mathtt{ask}_1, \mathtt{apk}_1, \mathtt{auxsk}_1) \leftarrow \mathsf{KeyGen}(1^\lambda, m)$
    $\tau_0 \leftarrow \mathsf{Assert}(\mathtt{ask}_0, \mathtt{auxsk}_0, m')$
    $\tau_1 \leftarrow \mathsf{Assert}(\mathtt{ask}_1, \mathtt{auxsk}_1, m')$
    return $(\mathtt{apk}_b, \tau_b)$
return 1, if $b' = b$; else, return 0.
```

Fig. 8: eAA Indistinguishability.

```
Experiment $\mathsf{Exp}_{\mathcal{A}}^{\mathrm{IND}}(\lambda)$
$(\mathtt{sk}, \mathtt{pk}) \leftarrow \mathsf{KeyGen}(1^\lambda), b \leftarrow \{0,1\}$
$b' \leftarrow \mathcal{A}^{\mathsf{HashOrAdapt}(\mathtt{sk},b,\cdots)}(\mathtt{pk})$
  where $\mathsf{HashOrAdapt}(\mathtt{sk}, b, \cdots)$ on input $m, m'$ :
    $(h_0, r_0) \leftarrow \mathsf{Hash}(\mathtt{pk}, m')$
    $(h_1, r_1') \leftarrow \mathsf{Hash}(\mathtt{pk}, m)$
    $r_1 \leftarrow \mathsf{Adapt}(\mathtt{sk}, m, m', h_1, r_1')$
    return $(h_b, r_b)$
return 1, if $b' = b$; else, return 0.
```

Fig. 9: CH Indistinguishability.

1) Indistinguishability: We define a formal experiment between an adversary $\mathcal{A}$ and a simulator $\mathcal{S}$ in Figure 8. During key generation process, $\mathcal{S}$ generates $(\mathtt{ask}, \mathtt{auxsk}, \mathtt{apk})$, where $\mathtt{apk} = (\mathtt{pk}, Z_1)$ is associated with dummy message-randomness values $\{(u, v)\}$ such as $(h, v) = \mathsf{Hash}(\mathtt{pk}, u)$. Recall that Z_1 is generated by chameleon hashes (see Figure 1). For asserting a message m, $\mathcal{S}$ generates a message-assertion pair (m, τ) by updating a dummy message-randomness pair (u, v) as (m, r) such that $h = \mathsf{Hash}_{\mathtt{pk}}(m, r) = \mathsf{Hash}_{\mathtt{pk}}(u, v)$ holds. We define an $\mathsf{Assert}_{\mathsf{IND}}$ oracle, showing that the dummy value v involved in the key generation is indistinguishable from the randomness r involved in the generation of τ. In this experiment, we allow $\mathcal{A}$ to decide a message set M, and all oracle queries must be associated with candidate messages in M. We assume adversary submits two candidate messages (m, m') to $\mathcal{S}$. Depending on the bit b, $\mathcal{S}$ either runs KeyGen and Assert on m' $(b = 0)$, or runs KeyGen on m first, then adapts m to m' when running Assert $(b = 1)$. Here are certain insights in Figure 8: 1) $\mathcal{S}$ treats (m, m') as dummy messages when running KeyGen, and $\mathcal{S}$ chooses the remaining dummy message-randomness values to generate $(\mathtt{apk}_0, \mathtt{apk}_1)$, respectively. 2) When $b = 0$, $\mathcal{S}$ simulates τ_0 without using the key shares of $\mathtt{ask}_0$ (i.e., no collision finding in generating τ_0). In other words, $\mathcal{S}$ uses the dummy message-randomness pair (m', r') to generate τ_0, where r' denotes the corresponding randomness used in generating $\mathtt{apk}_0$. Note that $\mathcal{S}$ keeps randomness r' as secret after KeyGen, and publishes it after Assert so that anyone can verify Z_1. 3) When $b = 1$, $\mathcal{S}$ simulates τ_1 using the key shares of $\mathtt{ask}_1$ (i.e., apply collision finding in generating τ_1). 4) When $b = 0$, $\mathtt{apk}_0 = (\mathtt{pk}, Z_{1,0})$. When $b = 1$, $\mathtt{apk}_1 = (\mathtt{pk}, Z_{1,1})$. Essentially, $\mathcal{S}$ simulates two distinct assertion public keys with the same Merkle-style tree and $\mathtt{pk}$.
We define the advantage of the adversary as

$$\mathtt{Adv}_{\mathcal{A}}^{\mathrm{IND}}(\lambda) = |\Pr[\mathcal{S} \rightarrow 1] - 1/2|.$$

Definition 4: An eAA scheme is indistinguishable if for any PPT $\mathcal{A}$, $\mathtt{Adv}_{\mathcal{A}}^{\mathrm{IND}}(\lambda)$ is negligible in λ.

Lemma 5: eAA has indistinguishability if the underlying chameleon hash is indistinguishable.

Proof 1: Let $\mathcal{S}$ be a challenger, who is given a chameleon public key $\mathtt{pk}^*$ and a $\mathsf{HashOrAdapt}$ oracle, aiming to break the CH's indistinguishability (see Figure 9). The reduction works as follows. $\mathcal{S}$ relies on his $\mathsf{HashOrAdapt}$ oracle to generate an assertion key pair $(\mathtt{ask}_b, \mathtt{apk}_b, \mathtt{auxsk}_b)$ and an assertion τ_b. Specifically, $\mathcal{S}$ forwards messages (m, m') to his $\mathsf{HashOrAdapt}$ oracle, and obtains (h_b, r_b). Then, $\mathcal{S}$ generates $\mathtt{apk}_b$ from (h_b, r_b) under $\mathtt{pk}^*$ and constructs τ_b from r_b. If $\mathcal{A}$ can distinguish assertion τ_b, $\mathcal{S}$ will break the CH's indistinguishability. Since there are at most k chameleon hashes involving in the generation of τ_b, the reduction should execute k times using a sequence of games.

2) Collision Resistance: We define an experiment between an adversary $\mathcal{A}$ and a simulator $\mathcal{S}$ in Figure 10. We assume an Assert' oracle, showing that even if an adversary $\mathcal{A}$ has an access to the Assert' oracle, it cannot find any assertions by itself. In this experiment, $\mathcal{S}$ chooses a message set M, and $\mathcal{A}$ chooses $m \in \mathsf{M}$ to query Assert' oracle.

```
Experiment $\mathsf{Exp}_{\mathcal{A}}^{\mathrm{CR}}(1^\lambda)$
$(\mathtt{ask}, \mathtt{apk}, \mathtt{auxsk}) \leftarrow \mathsf{KeyGen}(1^\lambda, \mathsf{M})$, $\mathcal{Q} \leftarrow \emptyset$
$(m^*, \tau^*, m'^*, \tau'^*) \leftarrow \mathcal{A}^{\mathsf{Assert}'(\mathtt{ask}, \mathtt{auxsk}, \cdots)}(\mathtt{apk})$
  where $\mathsf{Assert}'(\mathtt{ask}, \mathtt{auxsk}, \cdots)$ on input $m$ :
    return $\perp$ if $m \notin \mathsf{M}$
    return $\tau \leftarrow \mathsf{Assert}(\mathtt{ask}, \mathtt{auxsk}, m)$
    let $\mathcal{Q} \leftarrow \mathcal{Q} \cup \{(m, \tau)\}$
return 1, if $1 = \mathsf{Verify}(\mathtt{apk}, m^*, \tau^*)$
    $= \mathsf{Verify}(\mathtt{apk}, m'^*, \tau'^*) \wedge (m'^*, \cdot) \notin \mathcal{Q} \wedge m^* \neq m'^*$
else, return 0.
```

Fig. 10: eAA Collision-Resistance.

```
Experiment $\mathsf{Exp}_{\mathcal{A}}^{\mathrm{Secrecy}}(1^\lambda)$
$(\mathtt{ask}, \mathtt{apk}, \mathtt{auxsk}) \leftarrow \mathsf{KeyGen}(1^\lambda)$, $\mathcal{Q} \leftarrow \emptyset$
$\mathtt{ask}^* \leftarrow \mathcal{A}^{\mathsf{Assert}'(\mathtt{ask}, \mathtt{auxsk}, \cdots)}(\mathtt{apk})$
    where $\mathsf{Assert}'(\mathtt{ask}, \mathtt{auxsk}, \cdots)$ on input $h, m$ :
    $\mathcal{Q} \leftarrow \mathcal{Q} \cup \{h, m\}$
    return $\tau \leftarrow \mathsf{Assert}(\mathtt{ask}, \mathtt{auxsk}, h, m)$
return 1, if $\mathtt{ask}^* = \mathtt{ask} \wedge$
$(\nexists h, m_0, m_1 . m_0 \neq m_1 \wedge \{(h, m_0), (h, m_1)\} \in \mathcal{Q})$
else, return 0.
```

Fig. 11: AA Secrecy.

We define the advantage of the adversary as

$$\mathtt{Adv}_{\mathcal{A}}^{\mathrm{CR}}(\lambda) = |\Pr[\mathcal{A} \rightarrow 1]|.$$

Definition 6: An eAA scheme is collision resistant if for any PPT $\mathcal{A}$, $\mathtt{Adv}_{\mathcal{A}}^{\mathrm{IND}}(\lambda)$ is negligible in λ.

Lemma 7: eAA has collision-resistance if the underlying accountable assertion has secrecy.

Proof 2: Let $\mathcal{S}$ be a challenger, who is given an assertion public key $\mathtt{apk}^*$ and an Assert' oracle, aiming to find an assertion secret key (see Figure 11). The reduction is described as follows.

- $\mathcal{S}$ chooses auxiliary data $((\alpha_1, \cdots, \alpha_k)\{z\})$ and simulates assertion public key shares as $\{g^{s_i} = \mathtt{apk}^* \cdot C_1^{e_i} \cdot C_2^{e_i^2} \cdots C_k^{e_i^k}\}$ using $\mathtt{apk}^*$ and $\mathbb{C} = \{C_1, \cdots, C_k\}$, where $C_0 = \mathtt{apk}^*$, $C_i = g^{\alpha_i}$, and e_i is modelled as a random oracle. $\mathcal{S}$ sets up an assertion tree Λ with depth k, and generates an empty set $\mathcal{Q}$.
- To answer an assertion query on a message m, $\mathcal{S}$ invokes his Assert' oracle to obtain an assertion τ on m. $\mathcal{S}$ returns τ to $\mathcal{A}$ and includes (m, τ) into $\mathcal{Q}$.
- If $\mathcal{A}$ outputs a collided (message, randomness) values $h^* = \mathsf{CH.Hash}(m^*, r^*) = \mathsf{CH.Hash}(m'^*, r'^*)$ with respect to assertions (τ^*, τ'^*), such that: 1) $\mathsf{eAA.Verify}(\mathtt{apk}^*, m^*, \tau^*) = \mathsf{eAA.Verify}(\mathtt{apk}^*, m'^*, \tau'^*)$ = 1; 2) $m^* \neq m'^*$; 3) (m'^*, τ'^*) was not included in $\mathcal{Q}$, then $\mathcal{S}$ can extract a value s_i from the collided (message, randomness) values. Since the extracted share s_i is a key share of $\mathtt{ask}$, the reduction should execute k+1 times using a sequence of games. As a result, $\mathcal{S}$ can recover an assertion secret key $\mathtt{ask}^*$ from the extracted key shares $\{s_i\}^{k+1}$ (i.e., VSS's recover process) and break the accountable assertion's secrecy.

B. Security of MCH

The security analysis of MCH includes indistinguishability, and collision-resistance.

1) Indistinguishability:

Theorem 8: MCH achieves indistinguishability if eAA and semi-public-coin CH are indistinguishable.

Proof 3: We define a sequence of games $\mathbb{G}_i$, $i = 0, \cdots, 2$ and let $\mathtt{Adv}_i$ be the advantage of the adversary in game $\mathbb{G}_i$.

- $\mathbb{G}_0$: This is original game for indistinguishability.
- $\mathbb{G}_1$: This game is identical to game $\mathbb{G}_0$ except that, $\mathcal{S}$ directly hashes a challenge message in the construction of an assertion, instead of calculating the chameleon hash using assertion secret keys. Below we show that the difference between $\mathbb{G}_0$ and $\mathbb{G}_1$ is negligible if eAA is indistinguishable.
  Let $\mathcal{S}$ be a distinguisher against eAA, who is given an assertion public key $\mathtt{apk}^*$ and an $\mathsf{Assert}_{\mathsf{IND}}$ oracle, aims to break eAA's indistinguishability. $\mathcal{S}$ sets the assertion public key as $\mathtt{apk}^*$, and generates the chameleon key pair honestly.
  If $\mathcal{A}$ submits two messages $(m_0, m_1) \in \mathsf{M}$ to $\mathcal{S}$, $\mathcal{S}$ invokes his $\mathsf{Assert}_{\mathsf{IND}}$ oracle to obtain an assertion τ_b on message m_b. Next, $\mathcal{S}$ returns the chameleon hash (h_b, r_b, c) to $\mathcal{A}$, where $r_b = (\xi_b, C_\rho, \tau_b, \mathtt{pk}')$, $(h_b, \xi_b) \leftarrow \mathsf{CH.Hash}(\mathtt{pk}', m_b; \rho)$, $C \leftarrow \mathsf{PKE.Enc}(\mathtt{pk}_e, \rho)$. Note that $\mathtt{pk}'$ is derived from $\mathtt{pk}_c$ and c. Finally, $\mathcal{S}$ outputs whatever $\mathcal{A}$ outputs. If $\mathcal{A}$ guesses the random bit correctly, $\mathcal{S}$ will break eAA' indistinguishability. Therefore, we have
  $$|\mathtt{Adv}_0 - \mathtt{Adv}_1| \leq \mathtt{Adv}_{\mathcal{S}}^{\mathrm{eAA}}(\lambda). \tag{1}$$
- $\mathbb{G}_2$: This game is identical to game $\mathbb{G}_1$ except that in the g-th query, $\mathcal{S}$ directly hashes a challenge message, instead of calculating the chameleon hash using a chameleon secret key. Below we show that the difference between $\mathbb{G}_1$ and $\mathbb{G}_2$ is negligible if the semi-public-coin CH scheme is indistinguishable. Assume that $\mathcal{A}$ activates at most $n(\lambda)$ chameleon hash queries in this game, and let $g \in [1, n(\lambda)]$ be a guess for the index of the $\mathsf{HashOrAdapt}$ query.
  Let $\mathcal{S}$ be a distinguisher against CH, who is given a chameleon public key $\mathtt{pk}^*$ and a $\mathsf{HashOrAdapt}$ oracle, aims to break CH's indistinguishability. $\mathcal{S}$ sets up the CH secret public as $\mathtt{pk}^*$, and simulates the remaining parameters honestly.
  If $\mathcal{A}$ submits two messages $(m_0, m_1) \in \mathsf{M}$ in the g-th query, $\mathcal{S}$ first obtains a chameleon hash (h_b, ξ_b) from his $\mathsf{HashOrAdapt}$ oracle on message m_b. Then, $\mathcal{S}$ simulates the assertion $\tau_b \leftarrow \mathsf{eAA.Assert}(\mathtt{ask}', \mathtt{auxsk}', m_b)$, where $(\mathtt{ask}', \mathtt{auxsk}', \mathtt{apk}) \leftarrow \mathsf{eAA.KeyGen}(1^\lambda, \mathsf{M})$. Eventually, $\mathcal{S}$ returns (h_b, r_b, c) to $\mathcal{A}$, where $r_b = (\xi_b, C_b, \tau_b, \mathtt{pk}')$, $C_b \leftarrow \mathsf{PKE.Enc}(\mathtt{pk}_e, \rho_b)$. Note that $\mathtt{pk}'$ is derived from $\mathtt{pk}^*$ and c. $\mathcal{S}$ outputs whatever $\mathcal{A}$ outputs. If $\mathcal{A}$ guesses the random bit correctly, $\mathcal{S}$ will break CH's indistinguishability. Since $\mathcal{A}$ can issue at most $n(\lambda)$ chameleon hash queries, we have
  $$|\mathtt{Adv}_1 - \mathtt{Adv}_2| \leq n(\lambda) \cdot \mathtt{Adv}_{\mathcal{S}}^{\mathrm{CH}}(\lambda). \tag{2}$$

Combining the above results together, we have

$$\mathtt{Adv}_{\mathcal{A}}(\lambda) \quad \leq \quad \mathtt{Adv}_{\mathcal{S}}^{\mathrm{eAA}}(\lambda) + n(\lambda) \cdot \mathtt{Adv}_{\mathcal{S}}^{\mathrm{CH}}(\lambda).$$

2) Collision Resistance:

Theorem 9: MCH is collision-resistant if eAA and CH are collision-resistant.

Proof 4: We define a sequence of games $\mathbb{G}_i$, $i = 0, \cdots, 3$, and let $\texttt{Adv}_i$ be the advantage of the adversary in game $\mathbb{G}_i$. We assume that $\mathcal{A}$ activates at most $n(\lambda)$ assert and hash queries in each game.

- $\mathbb{G}_0$: This is original game for collision-resistant.
- $\mathbb{G}_1$: This game is identical to game $\mathbb{G}_0$ except the following difference: $\mathcal{S}$ randomly chooses $g \in [1, n(\lambda)]$ as a guess for the index of the forgery such that $\mathcal{A}$ outputs $(h^*, m^*, r^*, m'^*, r'^*)$ under a chameleon public key $\texttt{pk}^*$ and an assertion public key $\texttt{apk}^*$. $\mathcal{S}$ will output a random bit if $\mathcal{A}$'s forgery does not occur in the g-th query. Therefore we have
$$\texttt{Adv}_1 = n(\lambda) \cdot \texttt{Adv}_0 \tag{3}$$
- $\mathbb{G}_2$: This game is identical to game $\mathbb{G}_1$ except that $\mathcal{S}$ will output a random bit if $\mathcal{A}$ outputs a valid forgery in eAA. Below we show that the difference between $\mathbb{G}_1$ and $\mathbb{G}_2$ is negligible if eAA is collision-resistant.
Let $\mathcal{S}$ be an attacker against eAA, who is given an assertion public key $\texttt{apk}^*$ and an Assert' oracle, aims to find a forgery. $\mathcal{S}$ simulates the game for $\mathcal{A}$ as follows.
  - $\mathcal{S}$ sets up the assertion public key in the g-th query as $\texttt{apk}^*$. For the non g-th queries, $\mathcal{S}$ generates a new assertion key pair for an assertion and proceeds honestly, storing the corresponding assertion key pairs.
  - For the g-th query, $\mathcal{S}$ sets $\texttt{apk}^*$ as the assertion public key, invokes his Assert' oracle to obtain τ, and outputs the chameleon hash tuple as (h, m, r, c), where $r = (\xi, C_\rho, \tau, \texttt{pk}')$. Besides, the chameleon hash (h, ξ) can be simulated by $\mathcal{S}$.
  - If $\mathcal{A}$ outputs a chameleon hash tuple $(h^*, m^*, r^*, m'^*, r'^*)$ such that the assertion τ'^* is valid on message m'^* under assertion public key $\texttt{apk}^*$, and it was not previously returned by the Assert' oracle, $\mathcal{S}$ outputs $(m^*, \tau^*, m'^*, \tau'^*)$ as the forgery attempt to eAA. Besides, the chameleon hash $(h^*, m^*, r^*, m'^*, r'^*)$ and the assertion τ^* can be previously simulated by $\mathcal{S}$. Therefore, we have
$$|\texttt{Adv}_1 - \texttt{Adv}_2| \leq \texttt{Adv}_{\mathcal{S}}^{\text{eAA}}(\lambda). \tag{4}$$
- $\mathbb{G}_3$: This game is identical to game $\mathbb{G}_2$ except that in the g-th query, $\mathcal{S}$ will output a random bit if $\mathcal{A}$ outputs a forgery in CH. Below we show that the difference between $\mathbb{G}_2$ and $\mathbb{G}_3$ is negligible if CH is collision-resistant.
Let $\mathcal{S}$ be an attacker against CH, who is given a chameleon public key $\texttt{pk}^*$, and an Adapt' oracle, aims to find a forgery. $\mathcal{S}$ generates an initially empty set $\mathcal{Q}$. $\mathcal{S}$ simulates the game for $\mathcal{A}$ as follows.
  - $\mathcal{S}$ sets up the chameleon public key in the g-th query as $\texttt{pk}^*$. For non g-th queries, $\mathcal{S}$ generates a new chameleon key pair for a chameleon hash and proceeds honestly, storing the corresponding chameleon key pairs.
  - For the g-th query, $\mathcal{S}$ sets $\texttt{pk}^*$ as the chameleon public key, simulates the chameleon hash tuple as (h, m, r, c), where $r = (\xi, C_\rho, \tau, \texttt{pk}')$, the assertion τ is simulated by $\mathcal{S}$ according to the protocol specification. In particular, $\mathcal{S}$ includes $(\texttt{pk}', h, m, r, c)$ into $\mathcal{Q}$. Note that $\texttt{pk}'$ is derived from $\texttt{pk}^*$ and c.
If $\mathcal{A}$ issues an adapt query in the form of $(\texttt{pk}', c, h, m, r, m')$, $\mathcal{S}$ checks whether $(\texttt{pk}', h, m, r, c)$ was previously simulated ($\mathcal{S}$ aborts if not). Then, $\mathcal{S}$ invokes his Adapt' oracle to obtain r' if $m, m' \in \mathsf{M}$ and c is established between (m, m'), and returns the chameleon hash $(\texttt{pk}, h, m', r')$ to $\mathcal{A}$. Besides, the assertion τ' can be simulated by $\mathcal{S}$.
  - If $\mathcal{A}$ outputs a chameleon hash tuple that includes a chameleon hash (h^*, r'^*), which is valid on message m'^* under the chameleon public key $\texttt{pk}'$ and the assertion public key $\texttt{apk}$, and it was not previously returned by the Adapt' oracle. In addition, the relationship between the two messages is established $(m^*, m^{'*}) \in \mathsf{M}$. Then, $\mathcal{S}$ outputs $(\texttt{pk}', m^*, r^*, m'^*, r'^*)$ as the forgery attempt to CH. Therefore, we have
$$|\texttt{Adv}_2 - \texttt{Adv}_3| \leq \texttt{Adv}_{\mathcal{S}}^{\text{CH}}(\lambda). \tag{5}$$

Combining the above results together, we have
$$\texttt{Adv}_{\mathcal{A}}(\lambda) \quad \leq \quad n(\lambda)(\texttt{Adv}_{\mathcal{S}}^{\text{eAA}}(\lambda) + \texttt{Adv}_{\mathcal{S}}^{\text{CH}}(\lambda)).$$