Transaction-level Blockchain Rewrites with Revocation and Traceability using Attribute-based Cryptosystems

GAURAV PANWAR, New Mexico State University, USA
ROOPA VISHWANATHAN, New Mexico State University, USA
SATYAJAYANT MISRA, New Mexico State University, USA

In this paper, we study efficient and authorized rewriting of transactions already written to a blockchain. Mutable transactions will make a fraction of all blockchain transactions, but will be a necessity to meet the needs of privacy regulations, such as the General Data Protection Regulation (GDPR). The state-of-the-art rewriting approaches have several shortcomings, such as lack of user anonymity, inefficiency, and absence of revocation mechanisms for entities authorized to mutate transactions. To address this challenge we present ReTRACe, an efficient framework for blockchain rewrites. ReTRACe is designed by composing a revocable chameleon hash scheme with an ephemeral trapdoor, a revocable fast attribute based encryption scheme, and a dynamic group signature scheme. In this paper, (i) we discuss ReTRACe and its constituent primitives in detail, (ii) present security analyses of the primitives, and (iii) present experimental results to demonstrate the scalability of ReTRACe.

CCS Concepts: • **Security and privacy → Public key encryption**; **Access control**; **Pseudonymity, anonymity and untraceability**; *Privacy-preserving protocols*; *Distributed systems security*.

Additional Key Words and Phrases: Blockchain rewrite; chameleon hash; anonymity; attribute-based encryption.

1 INTRODUCTION AND RELATED WORK

In blockchains, transactions are typically collected into blocks that are linked together to form an immutable ledger. As blockchains continued to get used in practical applications in the real-world, there could be many situations where transactions need to be overwritten or redacted. For instance, there are governmental regulations that mandate stringent privacy regulations for users whose data is collected by entities such as companies. The European Union General Data Protection Regulation (GDPR) empowers users to demand erasure of their personal data by entities collecting such data. In U.S., California has enacted the California Consumer Privacy Act [12], which has similar regulations. Going forward, with users becoming increasingly sensitive about personal data and wanting to strictly control their data privacy, many regulatory bodies will enact similar legislations.

Blockchain technology has found use in many real-world applications, such as healthcare, banking transactions, and records management. Governments have also recommended the use of blockchains for approval-chain processes and supply chain monitoring [22]. According to privacy regulations, such as GDPR users should be able to request deletion/modification of their personal data from the blockchain. Consequently, many applications that store user related data on the blockchain will have to support user such deletion/modification operations. One of the desirable properties of these applications would be for users to be able to independently verify that their

Authors' addresses: Gaurav Panwar, New Mexico State University, Las Cruces, USA, gpanwar@nmsu.edu; Roopa Vishwanathan, New Mexico State University, Las Cruces, USA, roopav@nmsu.edu; Satyajayant Misra, New Mexico State University, Las Cruces, USA, misra@cs.nmsu.edu.

ACM 2769-6480/2025/5-ART
https://doi.org/10.1145/3736655

requested data deletion/modification has indeed taken place. The most efficient way to enable such independent verification is to have a mechanism for blockchain rewrites.

In addition to personally identifiable data deletion, there could be situations where wrong data gets written on the blockchain. Take for example a global consortium of banks, such as R3 consortium [40], which uses a shared blockchain called Corda. Incorrect data may end up as transactions on the blockchain, either as a honest mistake or through malicious intent, e.g., a rogue employee posting personally identifiable information (PII) of colleagues or customers. In such use cases, it is useful to have the inaccurate or objectionable transaction removed from the blockchain. These situations formed the motivation for the development of our scheme, Revocable and Traceable Blockchain Rewrites using Attribute-based Cryptosystems, ReTRACe. While in this paper, we concentrate on a permissioned blockchain for our use-case, ReTRACe is consensus-agnostic, and can be deployed on permissionless blockchains as well.

1.1 Related Work

In recent years, a number of blockchain rewriting solutions have been proposed. A simple, but inefficient approach is to do a *hard fork* of the blockchain at the point where a transaction in a block needs to be rewritten. However this approach has obvious inefficiencies. It results in creating a separate blockchain at the point of the fork, thus invalidating all later blocks in the original chain, which now need to be mined again. Besides, this approach does not remove the original transactions from the blockchain–the old copy will still be visible. Another technique is to modify a message m to m', post m' on the blockchain and have m' point to m, thus indicating m' is a newer copy. This again is not helpful in removing PII that may be in m. Further, neither of the above approaches allow the exercise of any control over who rewrites the messages. Ateniese et al. [2, 48] in one of the first works in this area, designed a method to do block-level rewrites. Similarly, Reparo [44] offered redaction at the block-level, with users proposing a redact operation on a block, which is accepted if a majority of other users vote in its favor. But, this redaction is not only for the transaction but the whole block. Both systems are not only block-level rewrites, but they also do not allow for revocation of rewrite privileges.

Intuitively, ReTRACe applies in scenarios where sensitive (or offensive) content needs to be overwritten or redacted on the blockchain, without having to rewrite the entire block, e.g., a single transaction that exposes someone's social security number or a single transaction containing objectionable content. It also has applications where the key for encrypted content, say a message m posted on the BC is leaked, and m needs to be re-encrypted and posted on the BC. Note that in all these cases, it is not sufficient to simply create a new transaction m' and have m' point to m, or do a hard fork, etc. The original content needs to be expunged from the BC.

A more scalable and relevant blockchain rewrite solution was proposed by Derler et al. [21], which involved transaction-level rewriting using a combination of chameleon hash functions and attribute-based encryption. Chameleon hash functions [30] are a versatile primitive and have found applications in online/offline signatures [17], identity-based encryption [52], policy-based sanitizable signatures [41], sanitizable signatures [11, 13], distributed hashing [23], and direct anonymous attestation [6]. Chameleon hash functions allow users to find collisions in the domain of an otherwise collision-resistant hash function. This is done with the help of a "trapdoor" that is tied to the digest. Knowledge of the trapdoor enables one to create as many collisions as one wants on the corresponding preimage. While in traditional chameleon hashes, there is only one trapdoor linked to a digest, Camenisch et al. [13] proposed a variant of chameleon hash functions that supports two trapdoors, and knowledge of both is needed for adapting a given preimage or message: a *long term trapdoor*, and an *ephemeral trapdoor*. The ephemeral trapdoor is picked per message, while the long term trapdoor is a system parameter. Krenn et al. [31] presented a dual-trapdoor scheme where the ephemeral trapdoor is not chosen per message, but did not explore its applications to blockchain rewrites. Lai et al. [32] proposed a weaker version, called "double trapdoor" chameleon hash functions where knowledge of either of the trapdoors enables the creation of

a collision, and showed how to apply the primitive to design sanitizable signature schemes. Derler et al. [21] use Camenisch et al. [13] method. The problem with the above schemes is that access to the ephemeral trapdoor, once issued, cannot be revoked, i.e., once given out, the ephemeral trapdoor is accessible to users in perpetuity. With ReTRACe, we also solve provide this missing revocation functionality.

There has been some recent work in trying to provide revocation for authority to blockchain rewrites. Tian et al. [45] presented a way to identify a user who misuses their rewriting privileges to write wrong or malicious information, but this approach does not offer a way to cryptographically revoke rewriting privileges of a user. Jia et al. [26] proposed a rewrite mechanism using chameleon hashes, where a user's ephemeral trapdoor can be revoked by an authority, but requires the revoked user to broadcast an additional transaction, called *witness transaction*, which miners need to validate. Xu et al. [50] proposed a scheme for rewriting transactions using a revocable attribute-based encryption scheme. In their construction, rewrites can be done in specific time intervals, and key update materials are distributed periodically by a key generation authority. But this approach does not allow for revocation on demand, which is detrimental since it allows the malicious users to continue to operate until the next revocation update.

In our work, we provide a revocable chameleon hash scheme which can revoke rewriting privileges of users on demand. Miners do not need to check if a given modification is valid or not, per the transaction rewriting rules, and the miner does not need to do more work than in a regular blockchain system. Ma et al. [35] built a system that employs multi-authority attribute-based encryption (ABE); this work takes an important step towards decentralized blockchain rewrites, but unfortunately, is proven secure in a weak security model, that is, static security, where the adversary submits a set of key queries, list of corrupted authorities, and messages once, and cannot adaptively corrupt authorities or issue key queries.

There have been other recent works such as [53], that do not provide the revocation feature, while others [46, 50] require revocation lists and/or offer only static revocation where users' keys are valid for certain time intervals. Xu *et al.* [49] do not offer revocation capability. Shao *et al.* [42] presented a blockchain rewriting scheme using RSA; consequently, it requires a revocation list for revoked users. The scheme requires a trusted consortium of parties called auditors who manage the revocation list to ensure revoked users cannot rewrite messages on the blockchain, this undermines the scalability of this approach. Some works in the literature, such as, Zhang *et al.* [54], Xu *et al.* [51], and Wang *et al.* [47] provide dynamic revocation. However, these schemes are either proposed for the cloud or leverage the cloud setting, use trusted proxy re-encryption servers in the cloud, use revocation lists to provide the revocation functionality. For instance, Zhang *et al.* [54] proposed a scheme, which closely followed that proposed by Lewko and Waters [33], extending it to a cloud server that maintains a revocation list with user identities and the attributes they possess. In the event of revocation, the cloud server acts as a proxy re-encrypting the data for use by the non-revoked users. These centralized schemes cannot be used in the distributed blockchain environments as they need an always-online trusted server and rely on revocation lists circulated to relevant entities.

Our framework, RFAME does not use revocation lists as they are not scalable, nor does it need an external third-party cloud server nor a trusted entity to act as a proxy. Further, given the significant construction efficiencies our scheme is novel and more scalable in the blockchain environment than the state-of-the-art. Table 1 presents a qualitative comparison of ReTRACe with other works in the area on qualitative metrics such as hardness assumptions, coarse-grained or fine-grained blockchain rewriting, revocation capability, and type of revocation. The novelty of ReTRACe is that it provides dynamic revocation, while not needing revocation lists, nor trusted parties, which none of the state-the-art schemes accomplish.

Table 1. Comparison of state-of-the-art blockchain-rewrite frameworks. *Legend for Computational hardness assumptions:* DL: discrete logarithm, DLIN: decisional linear, DDH: decisional Diffie Hellman, CDH: computational Diffie Hellman, SXDH: symmetric external Diffie Hellman, k-LIN: k linear, q-BDHE: q bilinear decisional Diffie Hellman. Rev. denotes revocation.

Scheme	**Hardness assumptions**	**Block-level/TX-Level**	**Rev**	**Rev. List**	**Revocation: Static (time-bound) or Dynamic (instant)**	**Trusted parties**
Ateniese et al. [2, 48]	DL, DDH, SXDH, k-LIN,	Block	No	–	–	No
Reparo [44]	–	Block	No	–	–	No
Derler et al. [21]	DLIN, Factoring	TX	No	–	–	No
Tian et al. [46]	DLIN, DL	TX	Yes	Yes	Static	Yes
Xu et al. [50]	DLIN	TX	Yes	No	Static	Yes
Xu et al. [49]	DLIN, qBDHE	TX	No	–	–	Yes
Shao *et al.* [42]	Strong RSA, DL, CDH	TX	Yes	Yes	Dynamic	Yes
ReTRACe (this)	DLIN, DL	TX	Yes	No	Dynamic	No

1.2 Our Contributions

We extend ReTRACe proposed by us in [37] comprehensively in this paper with theoretical analyses and use-case discussions to motivate the myriad uses of rewritable blockchains. The contributions of this paper include:
1) Detailed presentation of the design of a new *revocable chameleon hash with ephemeral trapdoor scheme*, RCHET, which guarantees that a revoked user cannot use the ephemeral trapdoor he possesses to adapt a blockchain message or trapdoor, and the ephemeral trapdoor can be quickly modified by authorities upon user revocation.
2) Detailed discussion of the design of our *revocable and traceable blockchain rewriting framework*, ReTRACe, using RCHET, a revocable attribute-based encryption scheme, and a dynamic group signature scheme as building primitives. In ReTRACe, authorized users can adapt messages, using the ephemeral trapdoor of a message's chameleon hash digest. Access to the ephemeral trapdoor can be revoked instantly as needed. Authorized users can *anonymously* post to and adapt messages on a blockchain, and their identities can be unmasked by legitimate oversight authorities if needed.
3) We rigorously prove the security of our framework by showing that it has four security properties: indistinguishability, private collision resistance, public collision resistance, and revocation collision resistance.
4) Implementation of RCHET and ReTRACe to demonstrate scalability accompanied by their detailed security analyses.

We note that in a blockchain adapted with the ReTRACe functionality, the only part that is changed is the way the rewritable transaction is verified; there is no other change. Both the ReTRACe messages and non-ReTRACe messages can co-exist in the underlying permissioned blockchain, that is, after integrating ReTRACe into the blockchain, the chain can accept mutable (messages with trapdoors controlled with ABE policies) as well as regular immutable messages, such as financial transactions and records of payments between different entities.
Organization: In Section 2, we discuss applications and use-cases for ReTRACe. In Section 3, we discuss the constituents of our system and threat models, and cover preliminaries and assumptions. In Section 4, we give a short technical overview of ReTRACe. In Sections 5 we introduce RCHET, its definition, security analysis, and

constructions. In Section 6 we briefly discuss the revocable ABE scheme and the dynamic GSS schemes used, and in Section 7, we give the definition, security analysis, and construction of ReTRACe. In Section 8 we present our implementation of ReTRACe and analyze the implementation, and Section 9 concludes the paper.

2 PRACTICAL USE CASES FOR ReTRACe

Before we get into the details of ReTRACe, we first present two practical use cases, which underscore the need for a framework such as ReTRACe in real-world applications.

2.1 Smart Contracts

Smart contracts are high-level computer programs submitted to a BC by users in the system. Once written to the BC, users can interact with the smart contract by submitting input transactions calling functions defined within the smart contract. Miners of the system execute the code of smart contracts on receiving input transactions. Smart contracts enforce agreements between two or more parties, and can be used in place of a required trusted third party or arbiter, e.g., to enforce fair exchange of goods and services. There has been extensive research that address interesting facets of smart contract deployment, such as off-chain execution of smart contracts [19, 20], and addressing and mitigating software bugs [27]. One of the problems of smart contracts is that the money locked up in them, e.g., due to errors in the smart contract code or due to user error while submitting a transaction, cannot be reclaimed. Roughly $174 million dollars are locked up at the $0x0$ address in Ethereum due to user errors during transactions submission; about $1.2 million are locked up in the *ENG Smart Contract*, just to name a few [36]. With the use of ReTRACe this problem can be prevented. The smart contract can be posted as a ReTRACe message, allowing it to be updated. A set of miners in the system can be authorized to decrypt the trapdoor and update the smart contract through a specific process which validates the correctness and authorization of the update–ReTRACe provides this functionality. This will allow the miners, backed by the consensus algorithm, to help the senders reclaim their lost tokens.

2.2 Financial Services Consortium

R3 [40] is a company that leads a global consortium of over two hundred members consisting of banks, trade associations and fintech companies, including U.S.-based members such as Bank of America and Goldman Sachs, international members such as Credit Suisse, Nomura, and Deutsche Bank. R3 developed *Corda*, an open-source permissioned distributed ledger platform, designed to operate and execute financial transactions, while restricting access to transaction data. Corda has already been deployed to codify financial agreements, securities trading, inter-bank transactions, etc., between the members of the consortium on the ledger.

Contracts between multiple banks involving trade instruments and securities, such as debts, bonds, and equity warrants, are posted on Corda. In any contract, parties retain the right to change the terms of the contract(s), provided all involved agree to revised terms. Change could be on account of contract renegotiation, re-financing, financial emergencies, human error, or a honest mistake in the contract. Other situations could arise necessitating a blockchain rewrite, such as rogue employees posting sensitive information and/or offensive content and/or leaked private keys, or errors in contracts that need to be expunged, etc. We envision ReTRACe to be very useful for such purposes, resulting in minimal update of the Corda ledger.

In this scenario, R3, in its capacity as leader of the consortium can manage edit access for the members of the consortium. Although the members of the consortium include varied entities, let us consider banks as an example. A bank may have various internal departments, e.g., retail banking, credit operations, loan operations, and private (high net-worth individuals) banking, which are further divided into sub-departments, e.g., the loan operations departments could have auto-loan, mortgage and student loan sub-departments. Each department or

sub-department could be visualized as a group being headed by a group manager (GM) who issues signing keys to the group's members.

In this situation, a bank can post a contract to the BC, and enable specific banks' departments (after mutual consultations) to update the contract by giving them access to a trapdoor.[1] The trapdoor could be encrypted using R3's public key, under an ABE policy that specifies the aforementioned departments relevant information, thus allowing for access control of the trapdoor. Similar to the ABE policy, the group signature policy of a given message details the group signature membership requirements for the given contract. Only parties belonging to the groups detailed in the signature policy can pass signature validation when they try to update the contract.

3 SYSTEM MODEL AND THREAT MODEL

ReTRACe uses three underlying primitives in its construction: 1) RCHET, a revocable chameleon hash scheme which provides an ephemeral trapdoor required for updating a message posted on the Blockchain (BC), 2) RFAME, a revocable attribute-based encryption scheme used to control access to the ephemeral trapdoor; and 3) a dynamic group signature scheme (DGSS), which helps legitimate users knowing the current ephemeral trapdoor, to sign message updates anonymously before posting them to the blockchain. Both, RFAME and DGSS are associated with policies which help with access control to the RCHET trapdoor. In this section, we discuss the parties involved in ReTRACe and the policy structures associated with access control mechanisms in the framework.

Naming Conventions: In our discussions, we refer to the blockchain as BC and we use *message* and transaction interchangeably to depict a BC transaction that contains the pre-image of the chameleon hash function. In our discussion, by "trapdoor", we mean the ephemeral trapdoor of the chameleon hash, unless otherwise specified. A chameleon hash function's output has mutable and immutable parts, the immutable part contains the digest of the hash function, the mutable part includes the trapdoor needed for creating a message collision, among other things.

3.1 Policies

ReTRACe consists of four policies, all represented as Boolean predicates: 1) The trapdoor associated with the digest of the given message is encrypted under an ABE policy, $\Upsilon_{\mathbf{ABE}}$. 2) The policy $\Upsilon_{\mathbf{GS}}$, spells out certain DGSS groups to which authorized users need to belong to be able to anonymously sign a valid message update, and post it on the BC. 3) For access control to the trapdoor, we define a policy, $\Upsilon_{\boldsymbol{ABE}\mathbf{admin}}$, that describes which ABE attributes are required to create and update ephemeral trapdoor, and encrypt it under a new policy, Υ'_{ABE}. 4) Finally, $\Upsilon_{\boldsymbol{GS}\mathbf{admin}}$ governs who can generate valid signatures when they update the Υ_{ABE} and/or update the ephemeral trapdoor.

For simplicity, we assume that the $\Upsilon_{ABE\text{admin}}$ and $\Upsilon_{GS\text{admin}}$ policies are set when the message is first created and cannot be changed (which can be relaxed on an as-needed basis by setting up a higher level of control). We stress that it is the $\Upsilon_{ABE\text{admin}}$ and $\Upsilon_{GS\text{admin}}$ *policy clauses* that are set as immutable; our design allows for the *set of people* satisfying them to be dynamic and ever-changing.

Unlike ABE schemes, DGSS does not have the concept of policies built into it; we introduce this notion for ReTRACe. We define a DGSS policy as a Boolean predicate associated with a message m that describes which DGSS group membership can produce valid signatures on m, e.g., $\{m, \Upsilon_{\mathsf{GS}} =$ "Admin" AND "Payroll"$\}$, indicates users belonging to both groups "Admin" and "Payroll" can produce valid signatures on m. In case of Boolean predicates with conjunctive clauses, the signatures (σ) and corresponding group public keys (gpk) are collected into a set, ξ_m and included in the BC message, where $\xi_m = (\sigma_{\text{Admin}}, gpk_{\text{Admin}}), (\sigma_{\text{Payroll}}, gpk_{\text{Payroll}})$. Any public verifier can check the validity of a set of signatures for a message w.r.t. a given policy.

[1] If a contract is unilaterally changed, the party that made the change can be traced and be penalized.

3.2 Parties

The parties involved in ReTRACe are categorized into:
1) Originator: The originator creates a message, its digest and trapdoor, and sets the four policies that regulate future message updates, Υ_{ABE}, Υ_{GS}, $\Upsilon_{ABE\text{admin}}$, and $\Upsilon_{GS\text{admin}}$.
2) Set of Authorized Users, AuthU: The users who can create a valid collision on a message posted on the BC by the originator (**AuthU** could include the originator as well) as long as they possess enough attributes as required in Υ_{ABE}, and are a member of a DGSS group satisfying Υ_{GS}.
3) Set of Authorized User Administrators, AuthUAdmins: This is the set of users who are authorized to modify the trapdoor, as well as update the message (**AuthUAdmins** could include the originator), as long as they possess enough attributes to satisfy $\Upsilon_{ABE\text{admin}}$ and are a member of a DGSS groups required by $\Upsilon_{GS\text{admin}}$.
4) Attribute-issuing authority and Group Manager: The attribute-issuing authority (AIA) for the ABE scheme, and the Group Manager (GM) for the DGSS issue keys to their respective users in the system.

For clarity of presentation, we use a single AIA and GM, but in practice, several of these entities can be used in ReTRACe with minor modifications.

3.3 Blockchain Operations

A ReTRACe message consists of a mutable portion (plaintext message, message update policies, trapdoor, ciphertexts, signatures, etc.) and an immutable portion (digest, admin policies). When a ReTRACe transaction is included in a block by the miners, only the immutable part of the ReTRACe transaction is used in the Merkle tree calculation of the given block instead of the hash of the whole transaction. This makes it possible to update a transaction in the future as long as the mutable part of the updated transaction verifies with the immutable digest on the BC which in turn is tied to the Merkle tree of the corresponding block.

ReTRACe can be incorporated with any kind of BC system with modifications and does not impact or depend on other features like the consensus mechanism, e.g., proof of work/stake, as long as the security requirements for ReTRACe defined in Section 3.4 are met. A user in ReTRACe needs to be previously onboarded with an AIA and GM in the given ReTRACe system for them to be able to post any ReTRACe messages in the BC.

To use ReTRACe with a given BC system, the hash verification function of the respective BC first needs to be modified. The verification function needs to include some extra checks apart from the regular block hash verifications, including the chameleon hash verifications of the ReTRACe messages. But ReTRACe does not leak sensitive information to the miners and the miners are not required to be onboarded by the AIA and GM to be able to carry out their transaction verification operations. Hence, adding or removing miners in a system that incorporates ReTRACe is exactly the same as a system that doesn't use ReTRACe.

Each block on a BC using ReTRACe could have mutable as well as immutable transactions, while the block-level hash of the block will always be immutable like in the original BCs. The modified verification algorithm can verify mutable as well as immutable transactions, thus allowing a ReTRACe implemented blockchain to support both, mutable and immutable transactions. Note that once an immutable transaction is written to the BC, it cannot be modified, only transactions which were originally posted as ReTRACe transactions and hence have a corresponding trapdoor and access policies are updatable. The AIA and GM in the ReTRACe system are publicly available to all users in the system. They can support any user that needs to post mutable transactions on the BC. However, users not interested in posting mutable transactions do not need to register with the AIA and GM.

3.4 Trust Assumptions and Threat Model

We make a few trust assumptions in ReTRACe as described below. As in most BC-enabled systems, we assume honest and trustworthy operation by miners of the system enforced by the consensus protocol being used. We assume the AIA will generate trustworthy attributes and keys. One could relax this assumption by using

multi-AIA techniques detailed in [16, 33], where certain parts of users' secret keys are contributed to by multiple AIAs. Similarly for the DGSS scheme, we assume that the GM for a group will issue signing keys properly, and the user tracing operation will be carried out honestly. Again, we could relax this assumption by using techniques of Bootle et al. [10], by introducing a *tracing manager*, and separating the responsibilities of group managers and tracing managers. We do not discuss these relaxations for narrative simplicity.

Threat Model and Security Goals: Our main objective is to protect against adversaries that do not have access to the long-term trapdoor and/or the current version of the ephemeral trapdoor for a given message, m, posted on the BC. The adversaries also do not satisfy the policies associated with m, yet they try to update m or its trapdoors. Another goal is to protect the privacy of, and provide anonymity to, the individuals who post messages or update messages and/or corresponding trapdoors on the BC. The only way for an adversary to violate our goals is to break the security of our cryptographic constructs. In this work, we prove the strong security properties of our constructs. We note that we do not consider network attacks (e.g., eclipse attacks, traffic analysis, etc.) in this paper, however, prior works [9, 39] that tackle these problems can be easily used in conjunction with ReTRACe in a BC system.

3.5 Computational Assumptions

The security of ReTRACe is derived from well-known assumptions based on the Discrete Log problem, the Decision Linear problem (DLIN), and the Decisional Diffie-Hellman (DDH) problem. As is common in the literature, we model the ABE access control policies as Boolean formulas with AND and OR gates, where each input is associated with an attribute, and the Boolean formulas are represented as monotone span programs [28].

4 ReTRACe SYSTEM OVERVIEW

In this section, we give a high-level, brief technical overview of ReTRACe. Without loss of generality, let us consider a single AIA and GM in the system. Let $[1..n]$ represent a set of users, the AIA issues sets of secret keys, $\{\mathbb{SK}_1, \ldots, \mathbb{SK}_n\}$, and the GM issues sets of signing keys, $\mathsf{sk}_1, \ldots, \mathsf{sk}_n$ to the n users. Let $\mathrm{mpk}_{\mathrm{ABE}}$ and gpk denote the public keys for the ABE and DGSS protocols respectively.

Let us consider a user, u, who creates a message, m, to be posted on the BC. User u creates an *adaptable* (i.e., updatable) trapdoor τ, that enables future modifications of m (using our novel RCHET scheme), sets a policy, Υ_{ABE}, which defines the set of authorized users, **AuthU**, who can update the message. User u encrypts τ with $\mathrm{mpk}_{\mathrm{ABE}}$ (using our novel revocable ABE scheme, RFAME), $E_{\mathrm{mpk}_{\mathrm{ABE}}}(\tau, \Upsilon_{\mathsf{ABE}}) \rightarrow X$. For controlling who can update τ in the future, u picks an $r \leftarrow\$ \mathbb{Z}_q$ (where $\mathbb{G}$ and $q = |\mathbb{G}|$ are part of the public parameters, and will be used in the cryptographic operations of ReTRACe). User u then sets the $\Upsilon_{ABE\mathrm{admin}}$ that defines the set of authorized user administrator(s), **AuthUAdmins**, and computes $E_{\mathrm{mpk}_{\mathrm{ABE}}}(r, \Upsilon_{ABE\mathrm{admin}}) \rightarrow X_r$. A user in **AuthU** updates m during a message adaptation, a user in **AuthUAdmins** updates m, X, and X_r during a trapdoor update.

A user in **AuthUAdmins** that satisfies $\Upsilon_{ABE\mathrm{admin}}$ can obtain r, prove knowledge of r to the miner(s), and update the trapdoor τ. User u also sets the DGSS policies, Υ_{GS}, and $\Upsilon_{GS\mathrm{admin}}$ that stipulate only members of **AuthU** and **AuthUAdmins** are authorized to produce valid anonymous signatures on an updated message and updated trapdoor respectively, before posting to the BC. Finally u posts tuple $t = (m, X, X_r, \Upsilon_{\mathrm{info}} = (\Upsilon_{\mathsf{ABE}}, \Upsilon_{\mathsf{GS}}), \Upsilon_{\mathrm{admin}} = (\Upsilon_{ABE\mathrm{admin}}, \Upsilon_{GS\mathrm{admin}}))$, along with a signature on t to the BC. We assume standard techniques such as nonces/timestamps to prevent replay attacks are used. Any user $i \in [1..n]$, s.t. $i \in$ **AuthU** whose secret key set $\mathbb{SK}_i \in \{\mathbb{SK}_1, \ldots, \mathbb{SK}_n\}$ satisfies Υ_{ABE}, can decrypt X, obtain τ, and update m to m' (using RCHET). Note that being able to satisfy Υ_{ABE} only allows i to decrypt the trapdoor, τ, and update m, but not update τ. User i will produce a signature on m' using sk_i that satisfy Υ_{GS}, and post m' and the signature on the BC.

The hash of a given transaction only corresponds to the m contained in it. The miner's verification function in ReTRACe ensures that only members of **AuthU** can update m and C_1, and only members of **AuthUAdmins** have

permission to update m, C_1, and C_2. The miner's in ReTRACe do not need any extra or privileged information other than what is already available to the rest of the system as part of the public parameters of ReTRACe, hence they *do not* need to be onboarded with the AIA or GM in the system.

Revocation of users from **AuthU** is handled by either a member of **AuthUAdmins** updating Υ_{ABE} to Υ'_{ABE} (RFAME), or by the AIA/GM revoking individual users. Any user $j \in [1..n]$, s.t. $j \in$ **AuthUAdmins** whose secret key set $\mathbb{SK}_j \in \{\mathbb{SK}_1, \ldots, \mathbb{SK}_n\}$ satisfies $\Upsilon_{ABE\text{admin}}$, can decrypt X_r, obtain r, update τ to τ' (using RCHET), such that τ' will only be decryptable by non-revoked users (using RFAME for access control). User j will compute $E_{\text{mpk}_{\text{ABE}}}(\tau', \cdot) \rightarrow C'_1$, j will prove knowledge of r to the miner, thus proving it can satisfy $\Upsilon_{ABE\text{admin}}$, and is a member of **AuthUAdmins**. Then j will sign and post m' and X' on the BC. The DGSS signature will be produced using j's set of signing keys, sk_j, that satisfy $\Upsilon_{GS\text{admin}}$.

5 REVOCABLE CHAMELEON HASH WITH EPHEMERAL TRAPDOORS

We create a revocable CHET scheme, where the long-term trapdoor remains permanent, but access to the ephemeral trapdoor can be revoked at will. Intuitively, for performing revocation, we update the ephemeral trapdoor, and prevent the revoked user from accessing the updated trapdoor.

Definition 5.1. *(Revocable chameleon hash with ephemeral trapdoor (*RCHET*) scheme):*
(1) $\mathsf{RCHET.systemSetup}(1^\lambda) \rightarrow (\mathsf{pubpar})$*: This algorithm on input a security parameter outputs the public parameters of the system. We assume that the public parameters* pubpar*, is implicitly passed as input to all other algorithms.*
(2) $\mathsf{RCHET.userKeySetup}(1^\lambda) \rightarrow (sk_{ch}, pk_{ch})$*: This algorithm on input the public parameters returns a long-term trapdoor,* sk_{ch}*, and a public key.*
(3) $\mathsf{RCHET.cHash}(sk_{ch}, pk_{ch}, m) \rightarrow \{(\mathsf{digest}, \mathsf{rand}, \Gamma_{\text{pubinfo}}, \Gamma_{\text{privinfo}}), \perp\}$*: This algorithm takes as input the long-term trapdoor, the public key, and a message m. If successful, it outputs a* digest*, randomness,* rand*, which will be used in verifying the digest, and an ephemeral trapdoor consisting of public/private parts,* $(\Gamma_{\text{pubinfo}}, \Gamma_{\text{privinfo}})$*.*
(4) $\mathsf{RCHET.verifyTransaction}(pk_{ch}, m, \mathsf{digest}, \mathsf{rand}, \Gamma_{\text{pubinfo}}) \rightarrow \{0, 1\}$*: This algorithm takes as input a public key,* pk_{ch}*, message m,* digest*,* rand*, public part of the ephemeral trapdoor, and returns 1 if the* digest *is correct.*
(5) $\mathsf{RCHET.adaptMessage}(sk_{ch}, m, m', \mathsf{digest}, \mathsf{rand}, \Gamma_{\text{pubinfo}}, \Gamma_{\text{privinfo}}) \rightarrow \{\mathsf{rand}', \perp\}$*: This algorithm takes as input the long-term trapdoor, a message m that needs to be adapted to* m'*, the* digest*,* rand*, and the ephemeral trapdoor* $(\Gamma_{\text{pubinfo}}, \Gamma_{\text{privinfo}})$*. If successful, it outputs an updated* rand′*.*
(6) $\mathsf{RCHET.adaptTrapdoor}(sk_{ch}, m, m', \mathsf{digest}, \mathsf{rand}, \Gamma_{\text{pubinfo}}, \Gamma_{\text{privinfo}}) \rightarrow \{(\mathsf{rand}', \Gamma'_{\text{pubinfo}}, \Gamma'_{\text{privinfo}}), \perp\}$*: This algorithm takes in the long-term trapdoor, a message, m, a new message* m'*,* digest*,* rand *and* $\Gamma_{\text{pubinfo}}, \Gamma_{\text{privinfo}}$*. If successful, it outputs an updated* rand′ *and updated public/private parts of the ephemeral trapdoor,* $(\Gamma'_{\text{pubinfo}}, \Gamma'_{\text{privinfo}})$*.*

We present our construction of an RCHET scheme in Figure 1. The security of our RCHET construction is based on the discrete log and DDH assumptions. At a high level, the idea is to hide the long-term and ephemeral trapdoors in the exponent of public parameters and use non-interactive zero knowledge proofs of knowledge (NIZKPoKs) to prove knowledge of them. In the construction, Π is an IND-CCA2 public key encryption scheme, which is used to encrypt randomness associated with the trapdoors. We overload the verification function for both, signatures and zero-knowledge proofs as verify, which will be clear from context.

Our RCHET construction involves a non-interactive zero knowledge proof (NIZK), and hence requires a common reference string (crs), which is a random string produced by a trusted party, or group of parties, that is available to everyone in the system. The crs is generated by the two honest parties in our system: the AIA and the GM. We note that the crs is specific to our construction, and not integral to the idea of an RCHET scheme.

All NIZK proofs require a crs and the use of a crs has precedent: Camenisch et al. [13] used a crs in their CHET constructions and applications to *sanitizable signatures*. Hawk [29], a system for building privacy-preserving

RCHET Algorithms

a) $\mathsf{RCHET.systemSetup}(1^\lambda) \rightarrow (\mathsf{pubpar})$: This algorithm generates the public parameters of the system:

1. $(\mathbb{G}, g, q) \leftarrow \mathsf{GGen}(1^\lambda)$. GGen generates prime-order cyclic group $\mathbb{G}$, $g \in \mathbb{G}$, $q = |\mathbb{G}|$.
2. Pick H's key, $k \in \mathcal{K}$, and $\mathsf{crs} \leftarrow \mathsf{Gen}(1^\lambda)$, where $\mathcal{K}$ is the key-space of H. Set and return $\mathsf{pubpar} = (k, \mathbb{G}, g, q, \mathsf{crs})$. We assume pubpar is implicitly passed as input to all other algorithms.

b) $\mathsf{RCHET.userKeySetup}(1^\lambda) \rightarrow (sk_{ch}, pk_{ch})$: This algorithm generates the long-term trapdoor and a public key:

1. Pick $x \leftarrow \mathbb{Z}_q^*$, $h \leftarrow g^x$, $\pi_{pk} \leftarrow \mathsf{NIZKPoK}\{x : h = g^x\}$, generate keys $(SK, PK) \leftarrow \Pi.\mathsf{KeyGen}(1^\lambda)$.
2. Set $pk_{ch} = (PK, h, \pi_{pk})$ and $sk_{ch} = (SK, x)$. Return (sk_{ch}, pk_{ch}).

c) $\mathsf{RCHET.cHash}(sk_{ch}, pk_{ch}, m) \rightarrow \{(\mathsf{digest}, \mathsf{rand}, \Gamma_{\text{pubinfo}}, \Gamma_{\text{privinfo}}), \perp\}$: Creates a chameleon hash for a message m:

1. Check $\mathsf{verify}(\pi_{pk}, h) \stackrel{?}{=} 1$, if not, return $\perp$. Pick $r, \mathsf{etd}, d, \delta \leftarrow \mathbb{Z}_q^*$.
2. Compute $h' \leftarrow g^{\mathsf{etd}}$, $D \leftarrow g^d$, and $\Delta \leftarrow g^\delta$. Do $\pi_t \leftarrow \mathsf{NIZKPoK}\{\mathsf{etd} : h' = g^{\mathsf{etd}}\}$, $\pi_D \leftarrow \mathsf{NIZKPoK}\{d : D = g^d\}$, $\pi_\Delta \leftarrow \mathsf{NIZKPoK}\{\delta : \Delta = g^\delta\}$.
3. Generate hash of message to be posted, $a \leftarrow H_k(m)$, where H is a collision resistant hash function, and create chameleon hash parameters: $\beta \leftarrow (r + \frac{\delta}{x} + \frac{d}{x})$, $p \leftarrow h^r$, $b \leftarrow p \cdot h'^a$. Do $\pi_p \leftarrow \mathsf{NIZKPoK}\{r : p = h^r\}$. Do $C \leftarrow \Pi.\mathsf{Encrypt}(PK, r)$, $C' \leftarrow \Pi.\mathsf{Encrypt}(PK, a)$.
4. Return $\mathsf{digest} = (b, h', \pi_t, C, C')$, $\mathsf{rand} = (\beta, p, \pi_p)$, $\Gamma_{\text{pubinfo}} = (\Delta, \pi_\Delta, D, \pi_D)$, $\Gamma_{\text{privinfo}} = (\delta, d, \mathsf{etd})$.

d) $\mathsf{RCHET.verifyTransaction}(pk_{ch}, m, \mathsf{digest}, \mathsf{rand}, \Gamma_{\text{pubinfo}}) \rightarrow \{0, 1\}$: This algorithm verifies the digest for a message m.

1. Check $\mathsf{verify}(\pi_{pk}, h) \stackrel{?}{=} 1$, $\mathsf{verify}(\pi_p, p) \stackrel{?}{=} 1$, $\mathsf{verify}(\pi_t, h') \stackrel{?}{=} 1$, $\mathsf{verify}(\pi_D, D) \stackrel{?}{=} 1$, and $\mathsf{verify}(\pi_\Delta, \Delta) \stackrel{?}{=} 1$.
2. Check $b \stackrel{?}{=} \frac{h^\beta \cdot h'^a}{D \cdot \Delta}$, where $a \leftarrow H_k(m)$. If check passes, return 1, else return 0.

e) $\mathsf{RCHET.adaptMessage}(sk_{ch}, m, m', \mathsf{digest}, \mathsf{rand}, \Gamma_{\text{pubinfo}}, \Gamma_{\text{privinfo}}) \rightarrow \{\mathsf{rand}', \perp\}$: This algorithm updates a message m:

1. Decrypt $a \leftarrow \Pi.\mathsf{Decrypt}(SK, C')$, check $a \stackrel{?}{\leftarrow} H_k(m)$. Check $b \stackrel{?}{=} \frac{h^\beta \cdot h'^a}{D \cdot \Delta}$. If any check fails, return $\perp$.
2. Check $h \stackrel{?}{=} g^x$, $p \stackrel{?}{=} g^{xr}$, $h' \stackrel{?}{=} g^{etd}$, $D \stackrel{?}{=} g^d$, and $\Delta \stackrel{?}{=} g^\delta$. If checks fail, return $\perp$.
3. Decrypt $r \leftarrow \Pi.\mathsf{Decrypt}(SK, C)$, if $r = \perp$, return $\perp$.
4. Compute $a' \leftarrow H_k(m')$. Compute $r' \leftarrow (\frac{rx + a \cdot \mathsf{etd} - a' \cdot \mathsf{etd} + \delta}{x})$.
5. Set $p' = h^{r'}$ and do $\pi_{p'} \leftarrow \mathsf{NIZKPoK}\{r' : p' = h^{r'}\}$. Compute $\beta' \leftarrow (r' + \frac{d}{x})$. Set and output $\mathsf{rand}' = (\beta', p', \pi_{p'})$.

Fig. 1. Construction of Revocable Chameleon Hash with Ephemeral Trapdoors (RCHET)

smart contracts uses a crs in pursuit of its goals. Succinct non-interactive arguments of knowledge which are used in various systems, the most prominent being ZCash [8], require and use a crs. In our system, since the AIA and GM are trusted, generating a crs is not a challenge; further if one relaxes these trust assumptions as outlined in Section 3 by using multi-AIA techniques, etc., the crs could be chosen using multi-party computations and hence created to be more secure. One could also use the methods of Groth et al. [25], where the crs is updatable

f) RCHET.adaptTrapdoor(sk_{ch}, m, m', digest, rand, $\Gamma_{\text{pubinfo}}, \Gamma_{\text{privinfo}}$) $\rightarrow$ $\{(\text{rand}', \Gamma'_{\text{pubinfo}}, \Gamma'_{\text{privinfo}}), \perp\}$: This algorithm modifies the trapdoor to an existing chameleon hash for a message m as follows:

1. Decrypt $a \leftarrow \Pi.\text{Decrypt}(SK, C')$, check $a \stackrel{?}{\leftarrow} H_k(m)$. Check $b \stackrel{?}{=} \frac{h^{\beta} \cdot h'^{a}}{D \cdot \Delta}$. If any check fails, return $\perp$.
2. Check $h \stackrel{?}{=} g^x$, $p \stackrel{?}{=} g^{xr}$, $h' \stackrel{?}{=} g^{etd}$, $D \stackrel{?}{=} g^d$, and $\Delta \stackrel{?}{=} g^{\delta}$. If checks fail, return $\perp$.
3. Decrypt $r \leftarrow \Pi.\text{Decrypt}(SK, C)$, if $r = \perp$, return $\perp$. Compute $a' \leftarrow H_k(m')$.
4. Pick $d', \delta' \leftarrow \mathbb{Z}_q^*$. Compute $D' \leftarrow g^{d'}, \Delta' \leftarrow g^{\delta'}$, do $\pi_{D'} \leftarrow \text{NIZKPoK}\{d' : D' = g^{d'}\}$, $\pi_{\Delta'} \leftarrow \text{NIZKPoK}\{\delta' : \Delta' = g^{\delta'}\}$.
5. Set $r' \leftarrow (\frac{rx + a \cdot \text{etd} - a' \cdot \text{etd} + \delta'}{x})$, $p' \leftarrow h^{r'}$. Compute $\beta' \leftarrow (r' + \frac{d'}{x})$, $\pi_{p'} \leftarrow \text{NIZKPoK}\{r' : p' = h^{r'}\}$.
6. Prove knowledge of DDH tuple $(g, g^{\delta}, g^d, g^{\delta d})$. Set and output $\text{rand}' = (\beta', p', \pi_{p'})$, $\Gamma'_{\text{pubinfo}} = (\Delta', \pi_{\Delta'}, D', \pi_{D'})$, and $\Gamma'_{\text{privinfo}} = (\delta', d', \text{etd})$.

Fig. 1. Construction of Revocable Chameleon Hash with Ephemeral Trapdoors (RCHET), continued

and all parties contribute secret randomness to it. Another idea is to choose the crs using the methods of Bellare et al. [4] that guarantee security even when the crs is maliciously chosen.

5.1 RCHET Security Properties

The properties of indistinguishability, public and private collision resistance were introduced by Camenisch et al. [13] for CHET schemes. Derler et al. [21] retained the three properties, but strengthened their security definition by giving the adversary access to an oracle for adapting messages in the private collision resistance game, while [13] only gave adversary access to a hash oracle. We *further strengthen* the security properties by: 1) introducing the notion of *revocation collision resistance*, which any RCHET scheme must provide, and 2) giving the adversary oracle access for *both*, adapting messages and adapting trapdoors.

Informally, *indistinguishability* requires that an outsider, given a random string, rand, cannot tell if rand was obtained by hashing the original message, a message update, or a trapdoor update. *Public collision resistance* requires that a user who possesses neither the long-term nor the ephemeral trapdoor, cannot find collisions by themselves. *Private collision resistance* requires that even the holder of the long-term trapdoor cannot find collisions, as long as the ephemeral trapdoor is unknown to them. *Revocation collision resistance* requires that a user that knows both, the long-term trapdoor and ephemeral trapdoor, cannot find collisions after the ephemeral trapdoor has been updated, as long as the new ephemeral trapdoor is unknown to them. We formalize these security properties in the following theorem and prove the theorem.

Theorem 5.1. *If the discrete log assumption and DDH assumption hold in $\mathbb{G}$, H is collision resistant, Π is IND-CCA2 secure, and the NIZKPoKs satisfy completeness, simulation soundness, extractability and zero knowledge, then our revocable chameleon hash with ephemeral trapdoors scheme,* RCHET *shown in Figure 1 is secure.*

5.2 RCHET Security Properties and Proof

Definition 5.2. *(Security of* RCHET*) An* RCHET *scheme is said to be secure if it possesses the following properties:*

1) *Correctness*: *We require that for all $\lambda \in \mathbb{N}$, for all* RCHET.systemSetup $(1^{\lambda}) \rightarrow$ (pubpar), *for all* RCHET.userKeySetup (pubpar) $\rightarrow (sk_{ch}, pk_{ch})$, *for all* RCHET.cHash(sk_{ch}, pk_{ch}, m) $\rightarrow$ (digest, rand, Γ_{pubinfo}, Γ_{privinfo}), *we have that* RCHET. verifyTransaction(pk_{ch}, m, digest, rand, Γ_{pubinfo}) $\rightarrow 1$. *We also require that for all* RCHET. adaptMessage(sk_{ch}, m, m', digest, rand, Γ_{pubinfo}, Γ_{privinfo}) $\rightarrow$ rand′, *we have that* RCHET.verifyTransaction(pk_{ch}, m, digest, rand′, Γ'_{pubinfo}) $\rightarrow 1$. *Furthermore, we require that for all*

$\mathsf{RCHET.adaptTrapdoor}(sk_{ch}, m, m', \text{digest}, \text{rand}, \Gamma_{\text{pubinfo}}, \Gamma_{\text{privinfo}}) \rightarrow (\text{rand}', \Gamma'_{\text{pubinfo}}, \Gamma'_{\text{privinfo}})$, *we have that* $\mathsf{RCHET.verifyTransaction}(pk_{ch}, m, \text{digest}, \text{rand}', \Gamma'_{\text{pubinfo}}) \rightarrow 1$. *Here* $m, m' \in \mathcal{M}$, $\mathcal{M}$ *is a message space.*

2*) Indistinguishability: Let the advantage of an adversary, $\mathcal{A}$, in the indistinguishability game given in Figure 2 be defined as:* $\mathsf{Adv}^{\mathcal{A}}_{\mathsf{RCHET.Indistinguishability}}(\lambda) = Pr\left[\mathsf{Indistinguishability}^{\mathcal{A}}_{\mathsf{RCHET}}(\lambda) = 1\right]$. *An* RCHET *scheme provides indistinguishability, if* $\mathsf{Adv}^{\mathcal{A}}_{\mathsf{RCHET.Indistinguishability}}(\lambda)$ *is a negligible function in* λ *for all PPT adversaries,* $\mathcal{A}$.

3*) Public collision-resistance: Let the advantage of an adversary, $\mathcal{A}$, in the public collision resistance game given in Figure 3 be defined as:* $\mathsf{Adv}^{\mathcal{A}}_{\mathsf{RCHET.PublicCollRes}}(\lambda) = Pr\left[\mathsf{PublicCollRes}^{\mathcal{A}}_{\mathsf{RCHET}}(\lambda) = 1\right]$. *An* RCHET *scheme provides public collision resistance, if* $\mathsf{Adv}^{\mathcal{A}}_{\mathsf{RCHET.PublicCollRes}}(\lambda)$ *is a negligible function in* λ *for all PPT adversaries,* $\mathcal{A}$.

4*) Private collision-resistance: Let the advantage of an adversary, $\mathcal{A}$, in the private collision resistance game given in Figure 4 be defined as:* $\mathsf{Adv}^{\mathcal{A}}_{\mathsf{RCHET.PrivateCollRes}}(\lambda) = Pr\left[\mathsf{PrivateCollRes}^{\mathcal{A}}_{\mathsf{RCHET}}(\lambda) = 1\right]$. *An* RCHET *scheme provides private collision resistance, if* $\mathsf{Adv}^{\mathcal{A}}_{\mathsf{RCHET.PrivateCollRes}}(\lambda)$ *is a negligible function in* λ *for all PPT adversaries,* $\mathcal{A}$.

5*) Revocation collision resistance: Let the advantage of an adversary, $\mathcal{A}$, in the revocation collision resistance game given in Figure 5 be defined as:* $\mathsf{Adv}^{\mathcal{A}}_{\mathsf{RCHET.RevocationCollRes}}(\lambda)$
$= Pr\left[\mathsf{RevocationCollRes}^{\mathcal{A}}_{\mathsf{RCHET}}(\lambda) = 1\right]$. *An* RCHET *scheme provides revocation collision resistance, if* $\mathsf{Adv}^{\mathcal{A}}_{\mathsf{RCHET.RevocationCollRes}}(\lambda)$ *is a negligible function in* λ *for all PPT adversaries,* $\mathcal{A}$.

Game $\mathsf{Indistinguishability}^{\mathcal{A}}_{\mathsf{RCHET}}(\lambda)$

1. $\mathsf{systemSetup}(1^{\lambda}) \rightarrow (\text{pubpar})$
2. $\mathsf{userKeySetup}(\text{pubpar}) \rightarrow (sk_{ch}, pk_{ch})$
3. $i \leftarrow \{0, 1, 2\}$
4. $\mathcal{A}^{\mathsf{HashOrAdapt}(sk_{ch},\cdot,\cdot,\cdot,\cdot,\cdot,\cdot)}(pk_{ch}) \rightarrow i'$
   where oracle HashOrAdapt on input (sk_{ch}, m, m') does:
   4.1. Do $\mathsf{cHash}(sk_{ch}, pk_{ch}, m) \rightarrow (\text{digest}, \text{rand}_0, \Gamma_{\text{pubinfo}}, \Gamma_{\text{privinfo}})$
   4.2. If $i = 0$, Set $t_0 = (\text{digest}, \text{rand}_0, \Gamma_{\text{pubinfo}}, \Gamma_{\text{privinfo}})$
   4.3. If $i = 1$, do $\mathsf{adaptMessage}(sk_{ch}, m, m', \text{digest}, \text{rand}_0, \Gamma_{\text{pubinfo}}, \Gamma_{\text{privinfo}}) \rightarrow \text{rand}_1$
   Set $t_1 = (\text{digest}, \text{rand}_1, \Gamma_{\text{pubinfo}}, \Gamma_{\text{privinfo}})$
   4.4. If $i = 2$, do $\mathsf{adaptTrapdoor}(sk_{ch}, m, m', \text{digest}, \text{rand}_0, \Gamma_{\text{pubinfo}}, \Gamma_{\text{privinfo}}) \rightarrow (\text{rand}_2, \Gamma'_{\text{pubinfo}}, \Gamma'_{\text{privinfo}})$
   Set $t_2 = (\text{digest}, \text{rand}_2, \Gamma'_{\text{pubinfo}}, \Gamma'_{\text{privinfo}})$
   4.5. Verify output values, if any are $\perp$, return $\perp$
   4.6. Return t_i
5. Return 1 if $(i' = i)$, else return 0.

Fig. 2. RCHET indistinguishability game

5.2.1 High-level Description of Games. Indistinguishability: We recall that the indistinguishability property requires that given an output, the adversary, $\mathcal{A}$, cannot tell if the output was a result of a cHash or adaptMessage or adaptTrapdoor. In our indistinguishability game in Figure 2, the adversary is given access to a HashOrAdapt oracle which takes as input a message m from $\mathcal{A}$, picks an integer $i \in \{0, 1, 2\}$, and returns the output of cHash or adaptMessage or adaptTrapdoor, respectively, depending on the value of i. $\mathcal{A}$ wins the game if it can guess i with probability greater than random guessing.

Game $\mathsf{PublicCollRes}^{\mathcal{A}}_{\mathsf{RCHET}}(\lambda)$

1. $\mathsf{systemSetup}(1^{\lambda}) \rightarrow (\mathsf{pubpar})$
2. $\mathsf{userKeySetup}(\mathsf{pubpar}) \rightarrow (sk_{ch}, pk_{ch})$
3. $Q \leftarrow \emptyset$, $\mathcal{A}$ picks $b \leftarrow \{0,1\}$
4. $\mathcal{A}^{\mathsf{cHash}(sk_{ch},\cdot,\cdot)}(pk_{ch})$
   where oracle cHash on input (sk_{ch}, pk_{ch}, m) does
   4.1. $\mathsf{cHash}(sk_{ch}, pk_{ch}, m) \rightarrow (\mathsf{digest}, \mathsf{rand}, \Gamma_{\mathrm{pubinfo}}, \Gamma_{\mathrm{privinfo}})$
   4.2. return $(\mathsf{digest}, \mathsf{rand}, \Gamma_{\mathrm{pubinfo}})$
5. $\mathcal{A}^{\mathsf{Adapt}(sk_{ch},\cdot,\cdot,\cdot,\cdot,\cdot,\cdot),\mathsf{MsgOrTrap}(\cdot)}(pk_{ch}) \rightarrow (m^*, \mathsf{rand}^*, \Gamma^*_{\mathrm{pubinfo}}, m^{*\prime}, \mathsf{rand}^{*\prime}, \Gamma^{*\prime}_{\mathrm{pubinfo}}, \mathsf{digest}^*)$
   where oracle Adapt on input (sk_{ch}, m, m'), and oracle MsgOrTrap on input b do:
   5.1. If $b = 0$, do $\mathsf{adaptMessage}(sk_{ch}, m, m', \mathsf{digest}, \mathsf{rand}, \Gamma_{\mathrm{pubinfo}}, \Gamma_{\mathrm{privinfo}}) \rightarrow \mathsf{rand}_0$
   Set $t_0 = (\mathsf{rand}_0, \Gamma_{\mathrm{pubinfo}})$
   5.2. If $b = 1$, do $\mathsf{adaptTrapdoor}(sk_{ch}, m, m', \mathsf{digest}, \mathsf{rand}, \Gamma_{\mathrm{pubinfo}}, \Gamma_{\mathrm{privinfo}}) \rightarrow (\mathsf{rand}_1, \Gamma'_{\mathrm{pubinfo}}, \Gamma'_{\mathrm{privinfo}})$
   Set $t_1 = (\mathsf{rand}_1, \Gamma'_{\mathrm{pubinfo}})$
   5.3. Verify output values, if any are $\perp$, return $\perp$
   5.4. $Q \leftarrow Q \cup \{m, m'\}$
   5.5. Return rand, t_b
6. Return 1 if $((\mathsf{verifyTransaction}(pk_{ch}, m^*, \mathsf{digest}^*, \mathsf{rand}^*, \Gamma^*_{\mathrm{pubinfo}}) \rightarrow 1)$
   $\wedge(\mathsf{verifyTransaction}(pk_{ch}, m^{*\prime}, \mathsf{digest}^*, \mathsf{rand}^{*\prime}, \Gamma^*_{\mathrm{pubinfo}}) \rightarrow 1) \vee$
   $(\mathsf{verifyTransaction}(pk_{ch}, m^*, \mathsf{digest}^*, \mathsf{rand}^*, \Gamma^*_{\mathrm{pubinfo}}) \rightarrow 1) \wedge$
   $(\mathsf{verifyTransaction}(pk_{ch}, m^{*\prime}, \mathsf{digest}^*, \mathsf{rand}^{*\prime}, \Gamma^{*\prime}_{\mathrm{pubinfo}}) \rightarrow 1)) \wedge$
   $(m^{*\prime} \notin Q) \wedge (m^* \neq m^{*\prime}) \wedge (\Gamma^*_{\mathrm{pubinfo}} \neq \Gamma^{*\prime}_{\mathrm{pubinfo}})$. Else return 0.

Fig. 3. RCHET public collision resistance game

Public collision resistance: The public collision resistance property requires that an adversary who has neither the long-term trapdoor nor ephemeral trapdoor cannot successfully create collisions. In Figure 3, we need to give the adversary oracle access to both, the adaptMessage and adaptTrapdoor functionalities. We provide $\mathcal{A}$ access to a MsgOrTrap oracle, which takes an input bit b chosen by $\mathcal{A}$, and passes them on to an Adapt oracle, which either does an adaptMessage or adaptTrapdoor, depending on the value of b. The output of cHash and adaptMessage or adaptTrapdoor is returned to $\mathcal{A}$. $\mathcal{A}$ wins if it can successfully either adapt a message or adapt a trapdoor, where "successful" means that the output passes verification.
Private collision resistance: The private collision resistance property requires that even the holder of the long term trapdoor cannot find collisions, as long as the ephemeral trapdoor is unknown to them. In Figure 4, $\mathcal{A}$ picks the long term trapdoor and public key, the Adapt oracle creates digests, and either adapts a message or adapts the ephemeral trapdoor, per $\mathcal{A}$'s choice (governed by bit b). $\mathcal{A}$ is deemed to have won if $\mathcal{A}$ can successfully either adapt a message or adapt the trapdoor, and in both cases, produce a pre-image that maps on to one of the digests returned by the oracle.
Revocation collision resistance: Revocation collision resistance requires that someone who knows *both*, the long term and ephemeral trapdoors cannot successfully create collisions, after the ephemeral trapdoor has been updated, as long as the new trapdoor is unknown to them. In Figure 5, $\mathcal{A}$ creates a digest of a message, can adapt it and can also adapt the ephemeral trapdoor. The oracle Adapt is then invoked for updating the ephemeral

Game $\mathsf{PrivateCollRes}^{\mathcal{A}}_{\mathsf{RCHET}}(\lambda)$

1. $\mathsf{systemSetup}(1^\lambda) \rightarrow (\mathsf{pubpar})$
2. $\mathcal{A}(\mathsf{pubpar}) \rightarrow (sk^*_{ch}, pk^*_{ch})$
3. $Q \leftarrow \emptyset$, $\mathcal{A}$ picks $b \leftarrow \{0,1\}$
4. $\mathcal{A}^{\mathsf{cHash}(\cdot,\cdot,\cdot)}(pk_{ch})$
   where oracle cHash on input $(sk^*_{ch}, pk^*_{ch}, m)$ does
   4.1. $\mathsf{cHash}(sk^*_{ch}, pk^*_{ch}, m) \rightarrow (\mathsf{digest}, \mathsf{rand}, \Gamma_{\text{pubinfo}}, \Gamma_{\text{privinfo}})$
   4.2. Return $(\mathsf{digest}, \mathsf{rand}, \Gamma_{\text{pubinfo}})$
5. $\mathcal{A}^{\mathsf{Adapt}(\cdot,\cdot,\cdot,\cdot,\cdot,\cdot,\cdot),\mathsf{MsgOrTrap}(\cdot)}(pk^*_{ch}) \rightarrow (m^*, \mathsf{rand}^*, \Gamma^*_{\text{pubinfo}}, m^{*\prime}, \mathsf{rand}^{*\prime}, \Gamma^{*\prime}_{\text{pubinfo}}, \mathsf{digest}^*)$
   where oracle Adapt on input (sk^*_{ch}, m, m'), and oracle MsgOrTrap on input b do:
   5.1. If $b = 0$, do $\mathsf{adaptMessage}(sk^*_{ch}, m, m', \mathsf{digest}, \mathsf{rand}, \Gamma_{\text{pubinfo}}, \Gamma_{\text{privinfo}}) \rightarrow \mathsf{rand}_0$
   Set $t_0 = (\mathsf{rand}_0, \Gamma_{\text{pubinfo}})$
   5.2. If $b = 1$, do $\mathsf{adaptTrapdoor}(sk^*_{ch}, m, m', \mathsf{digest}, \mathsf{rand}, \Gamma_{\text{pubinfo}}, \Gamma_{\text{privinfo}}) \rightarrow (\mathsf{rand}_1, \Gamma'_{\text{pubinfo}}, \Gamma'_{\text{privinfo}})$
   Set $t_1 = (\mathsf{rand}_1, \Gamma'_{\text{pubinfo}})$.
   5.3. Verify output values, if any are $\perp$, return $\perp$
   5.4. $Q \leftarrow Q \cup \{\mathsf{digest}, m\}$
   5.5. Return $(\mathsf{digest}, \mathsf{rand}, t_b)$
6. Return 1 if $((\mathsf{verifyTransaction}(pk^*_{ch}, m^*, \mathsf{digest}^*, \mathsf{rand}^*, \Gamma^*_{\text{pubinfo}}) \rightarrow 1) \wedge$
   $(\mathsf{verifyTransaction}(pk^*_{ch}, m^{*\prime}, \mathsf{digest}^*, \mathsf{rand}^{*\prime}, \Gamma^*_{\text{pubinfo}}) \rightarrow 1) \vee$
   $(\mathsf{verifyTransaction}(pk^*_{ch}, m^*, \mathsf{digest}^*, \mathsf{rand}^*, \Gamma^*_{\text{pubinfo}}) \rightarrow 1) \wedge$
   $(\mathsf{verifyTransaction}(pk^*_{ch}, m^{*\prime}, \mathsf{digest}^*, \mathsf{rand}^{*\prime}, \Gamma^{*\prime}_{\text{pubinfo}}) \rightarrow 1)) \wedge$
   $(\mathsf{digest}^*, m^{*\prime} \notin Q) \wedge (\mathsf{digest}^*, \cdot) \in Q \wedge (\Gamma^*_{\text{pubinfo}} \neq \Gamma^{*\prime}_{\text{pubinfo}}))$ else return 0.

Fig. 4. RCHET private collision resistance game

trapdoor. After the update, $\mathcal{A}$ is tasked with either correctly updating the message or trapdoor. $\mathcal{A}$ wins if can successfully either adapt a message or adapt the trapdoor, and in both cases, produce a pre-image that maps on to one of the digests returned by the oracle.

5.2.2 RCHET *Proof.*

Proof. We need to prove our RCHET scheme depicted in Figure 1 provides indistinguishability, public collision resistance, private collision resistance and revocation collision resistance. We prove each property separately, and assume all communication between parties takes place via secure and authenticated channels. We need two additional games from [13], the zero knowledge game and simulation-sound extractability game, given in Figure 6a, and Figure 6b, respectively. We recollect that a function $\nu : \mathbb{N} \rightarrow \mathbb{R}_{\geq 0}$ is negligible, if it vanishes faster than every inverse polynomial, i.e., $\forall k \in \mathbb{N}, \exists n_0 \in \mathbb{N}$ such that $\nu(n) \leq n^{-k}, \forall n > n_0$.

Indistinguishability: Trivial, as an adversary will either see a hash or its adapted version (either as a result of an adaptTrapdoor or an adaptMessage), but will never see both, a hash and its adapted version at the same time.

Public Collision Resistance: Let us consider a sequence of games:

Game 0 (G0): The original public collision-resistance game.

Game 1 (G1): Same as **G0** but upon setup we obtain $(crs, \tau) \leftarrow S_1(1^\lambda)$, store τ and henceforth simulate all proofs using $S_2(crs, \tau, \cdot)$.

Game $\mathsf{RevocationCollRes}^{\mathcal{A}}_{\mathsf{RCHET}}(\lambda)$

1. $\mathsf{systemSetup}(1^\lambda) \rightarrow \mathsf{pubpar}$
2. $\mathcal{A}(\mathsf{pubpar}) \rightarrow (sk^*_{ch}, pk^*_{ch})$
3. $\mathcal{A}$ does $\mathsf{cHash}(sk^*_{ch}, pk^*_{ch}, m^*) \rightarrow (\mathsf{digest}^*, \mathsf{rand}^*, \Gamma^*_{\mathrm{pubinfo}}, \Gamma^*_{\mathrm{privinfo}})$
4. $\mathcal{A}^{\mathsf{Adapt}(\cdot,\cdot,\cdot,\cdot,\cdot,\cdot)}(pk^*_{ch}) \rightarrow (m^{*\prime\prime}, \mathsf{rand}^{*\prime\prime}, \{\Gamma^{*\prime\prime}_{\mathrm{pubinfo}}, \perp\}, \mathsf{digest}^*)$
   where oracle Adapt on input $(sk^*_{ch}, m^*, m^{*\prime}, \mathsf{digest}^*, \mathsf{rand}^*, \Gamma^*_{\mathrm{pubinfo}}, \Gamma^*_{\mathrm{privinfo}})$ does:
   4.1. If $\mathsf{verifyTransaction}(pk^*_{ch}, m^*, \mathsf{digest}^*, \mathsf{rand}^*, \Gamma^*_{\mathrm{pubinfo}}) \rightarrow 0$ return $\perp$
   4.2. Do $\mathsf{adaptTrapdoor}(sk^*_{ch}, m^*, m^{*\prime}, \mathsf{digest}^*, \mathsf{rand}^*, \Gamma^*_{\mathrm{pubinfo}}, \Gamma^*_{\mathrm{privinfo}}) \rightarrow (\mathsf{rand}^{*\prime}, \Gamma^{*\prime}_{\mathrm{pubinfo}}, \Gamma^{*\prime}_{\mathrm{privinfo}})$
   4.3. Verify output values, if any are $\perp$, return $\perp$
   4.4. Return $(m^{*\prime}, \mathsf{rand}^{*\prime}, \Gamma^{*\prime}_{\mathrm{pubinfo}})$
5. Return 1 if $((\mathsf{verifyTransaction}(pk^*_{ch}, m^*, \mathsf{digest}^*, \mathsf{rand}^*, \Gamma^*_{\mathrm{pubinfo}}) \rightarrow 1) \wedge$
   $(\mathsf{verifyTransaction}(pk^*_{ch}, m^{*\prime\prime}, \mathsf{digest}^*, \mathsf{rand}^{*\prime\prime}, \Gamma^{*\prime}_{\mathrm{pubinfo}}) \rightarrow 1) \vee$
   $(\mathsf{verifyTransaction}(pk^*_{ch}, m^*, \mathsf{digest}^*, \mathsf{rand}^*, \Gamma^*_{\mathrm{pubinfo}}) \rightarrow 1) \wedge$
   $(\mathsf{verifyTransaction}(pk^*_{ch}, m^{*\prime\prime}, \mathsf{digest}^*, \mathsf{rand}^{*\prime\prime}, \Gamma^{*\prime\prime}_{\mathrm{pubinfo}}) \rightarrow 1))$. Else return 0.

Fig. 5. RCHET revocation collision resistance game

Game $\mathsf{Zero\text{-}Knowledge}^{\mathsf{NIZKPoK}}_{\mathcal{A}}(\lambda)$

$b \leftarrow \{0,1\}$
$(crs, \tau) \leftarrow S_1(1^\lambda)$
$a \leftarrow \mathcal{A}^{P_b(\cdot,\cdot)}(crs)$
  where oracle P_0 on input (x, w) :
    return $\pi \leftarrow \{(w) : R(x, w) = 1\}$, if $(x, w) \in L$
    return $\perp$
  and oracle P_1 on input (x, w) :
    return $\pi \leftarrow S_2(crs, \tau, x)$, if $(x, w) \in L$
    return $\perp$
return 1, if $a = b$
return 0

(a) Zero knowledge game

Game $\mathsf{SimSoundExt}^{\mathsf{NIZKPoK}}_{\mathcal{A},E}(\lambda)$

$(crs, \tau, \xi) \leftarrow S(1^\lambda)$
$(x, \pi) \leftarrow \mathcal{A}^{Sim(\cdot)}(crs)$
  where oracle Sim on input x :
    obtain $\pi \leftarrow S_2(crs, \tau, x)$
    set $Q^{Sim} \leftarrow Q^{Sim} \cup \{(x, \pi)\}$
$w \leftarrow \mathcal{E}(crs, \xi, x, \pi)$
return 1, if $\mathsf{verify}(x, \pi) =$ **true** $\wedge$
    $(x, w) \notin R \wedge (x, \pi) \notin Q^{Sim}$
return 0

(b) Simulation sound extractability game

Fig. 6. Additional games for RCHET proof [13]

Transition - Game 0 → Game 1: A distinguisher between **G0** and **G1** zero-knowledge distinguisher, i.e., $| Pr[\mathbf{G0}] - Pr[\mathbf{G1}] | \leq \nu_{zk}(\lambda)$.
Game 2 (G2): Same as **G1**, but upon setup we obtain $(crs, \tau, \chi) \leftarrow S_1(\lambda)$, and additionally store χ.
Transition - Game 1 → Game 2: Under simulation sound extractability, this change is conceptual, i.e., $Pr[\mathbf{G1}] = Pr[\mathbf{G2}]$.
Game 3 (G3): Same as **G2**, but we simulate the Adapt oracle thus: to find a collision w.r.t. m, m', digest

(b, h', π_t, C, C'), randomness (β, p, π_p), and trapdoor information $\Gamma_{\text{pubinfo}} = (\Delta, \pi_\Delta, D, \pi_D)$, $\Gamma_{\text{privinfo}} = (\delta, d, \mathsf{etd})$, and $sk_{ch} = (SK, x)$, if $\mathsf{rand} = (\beta, p, \cdot)$ corresponds to a previous Adapt query, set $AD = \top$ and $AD = \bot$ otherwise. If only p corresponds to a previous query, return $\bot$.

Transition - Game 2 → Game 3: This change is conceptual, i.e., $Pr[\mathbf{G2}] = Pr[\mathbf{G3}]$ (observe that p is unconditionally binding, and, thus, modifying β implies that the check $b \stackrel{?}{=} \frac{h^\beta \cdot h'^a}{D \cdot \Delta}$ which is performed within Adapt fails, and the oracle would abort anyway).

Game 4 (G4): Same as **G3** but we further change the Adapt oracle thus: to find a collision w.r.t. m, m', digest (b, h', π_t, C, C'), randomness (β, p, π_p), and trapdoor information $\Gamma_{\text{pubinfo}} = (\Delta, \pi_\Delta, D, \pi_D)$, $\Gamma_{\text{privinfo}} = (\delta, d, \mathsf{etd})$, and $sk_{ch} = (SK, x)$ do:

(1) If $\mathsf{rand} = (\beta, p, \cdot)$ corresponds to a previous Adapt query, set $AD = \top$ and $AD = \bot$ otherwise. If only p corresponds to a previous query, return $\bot$.

(2) Decrypt $a \leftarrow \Pi.\mathsf{Decrypt}(SK, C')$, check if $a \stackrel{?}{\leftarrow} H_k(m)$. $\boxed{\text{If } AD = \bot,}$ check if $b \stackrel{?}{=} \frac{h^\beta \cdot h'^a}{D \cdot \Delta}$. If checks fail, return $\bot$.

...

If $\mathcal{A}$ chose $b = 0$:

(4) $\boxed{\text{If } AD = \bot,}$ Decrypt $r \leftarrow \Pi.\mathsf{Decrypt}(SK, C)$ and if $r = \bot$, return $\bot$.

else

(4) $\boxed{\text{If } AD = \bot,}$ Decrypt $r \leftarrow \Pi.\mathsf{Decrypt}(SK, C)$ and if $r = \bot$, return $\bot$. Compute $a' \leftarrow H_k(m')$.

Transition - Game 3 → Game 4: This change is conceptual, i.e., $Pr[\mathbf{G3}] = Pr[\mathbf{G4}]$ (the checks are only omitted if we know that they would not yield to an abort).

Game 5 (G5): Same as **G4**, but we further change the Adapt oracle as follows:

(1) If $\mathsf{rand} = (\beta, p, \cdot)$ corresponds to a previous Adapt query, set $AD = \top$ and $AD = \bot$ otherwise. If only p corresponds to a previous query, return $\bot$.

(2) Decrypt $a \leftarrow \Pi.\mathsf{Decrypt}(SK, C')$, check if $a \stackrel{?}{\leftarrow} H_k(m)$.

$\boxed{\text{If } AD = \bot,}$ check if $b \stackrel{?}{=} \frac{h^\beta \cdot h'^a}{D \cdot \Delta}$. If checks fail, return $\bot$.

...

If $\mathcal{A}$ chose $b = 0$:

(4) $\boxed{\text{If } AD = \bot,}$ Decrypt $r \leftarrow \Pi.\mathsf{Decrypt}(SK, C)$ and if $r = \bot$, return $\bot$.

(5) Compute $a' \leftarrow H_k(m')$. $\boxed{\text{Compute } r' = garbage}$.

(6) $\boxed{\text{Set } p' = h^{r'} \text{ and } \pi_{p'} \leftarrow \mathsf{NIZKPoK}\{r' : p' = h^{r'}\}.}$

$\boxed{\text{Compute } \beta' \leftarrow (r + \frac{a.etd}{x} - \frac{a'.etd}{x} + \frac{\delta}{x} + \frac{d}{x}).}$

else

(4) $\boxed{\text{If } AD = \bot,}$ Decrypt $r \leftarrow \Pi.\mathsf{Decrypt}(SK, C)$ and if $r = \bot$, return $\bot$. Compute $a' \leftarrow H_k(m')$.

(6) $\boxed{\text{Set } r' = \bot, p' \leftarrow h^{r'}, \beta' \leftarrow (r + \frac{a.etd}{x} - \frac{a'.etd}{x} + \frac{\delta}{x} +}$

$\boxed{\frac{d'}{x}), \pi_{p'} \leftarrow \mathsf{NIZKPoK}\{r' : p' = h^{r'}\}}$

Transition - Game 4 → Game 5:A distinguisher between **G4** and **G5** is an IND-CCA2 distinguisher for Π, i.e., $| Pr[\mathbf{G4}] - Pr[\mathbf{G5}] | \leq \nu_c(\lambda)$.

Game 6 (G6): Same as **G5** but for every query to Adapt, we store (β, p, π_p) if π_p was not previously simulated within Adapt in $R[(b, h', \pi_t, C, C')] \leftarrow (\beta, p, \pi_p)$. Now, for every forgery either both rand^* or $\mathsf{rand}^{*\prime}$ are fresh, or one of them contains a proof π_p (resp. π'_p) which we previously simulated in the Adapt oracle. If one of them contains such a proof, we replace the respective randomness tuple (β, p, π_p) by $R[\mathsf{digest}^*]$.

Transition - Game 5 → Game 6: This change is conceptual, i.e., $Pr[\mathbf{G5}] = Pr[\mathbf{G6}]$. Observe that the fact that a proof stems from a tuple returned by Adapt implies that a query with a tuple (β, p, π_p) where π_p was not simulated must once have happened. Further, the modified forgery is still a valid public collision freeness forgery.

Game 7 (G7): Same as **G6** but for the modified forgery we extract both rand and rand′ from π_p and π'_p contained in $\mathsf{rand}^* = (\beta, p, \pi_p)$ and $\mathsf{rand}^{*\prime} = (\beta', p', \pi'_p)$ If the extraction fails, we abort.
Transition - Game 6 → Game 7: Both games proceed identically, unless we have to abort, i.e., $\mid Pr[\mathbf{G6}] - Pr[\mathbf{G7}] \mid \leq 2\nu_e(\lambda)$.
Game 8 (G8): Same as **G7** but for π_t contained in digest^* we extract the etd and abort if the extraction fails.
Transition - Game 7 → Game 8: Both games proceed identically, unless we have to abort, i.e., $\mid Pr[\mathbf{G7}] - Pr[\mathbf{G8}] \mid \leq \nu_e(\lambda)$.
Game 9 (G9): Same as **G8** but we obtain a DL-challenge $(\mathbb{G}, g, q, g^x)$, perform the setup with respect to $(\mathbb{G}, g, q)$ and embed g^x into pk_{ch}.
Transition - Game 8 → Game 9: This change is conceptual, i.e., $\mid Pr[\mathbf{G8}] = Pr[\mathbf{G9}] \mid$.

Private Collision Resistance:
Game 0 (G0): The original private collision-resistance game.
Game 1 (G1): Same as **G0**, but upon setup we obtain $(crs, \tau) \leftarrow S_1(1^\lambda)$ upon setup, store τ and henceforth simulate all proofs using $S_2(crs, \tau, \cdot)$.
Transition - Game 0 → Game 1: A distinguisher between **G0** and **G1** is a zero-knowledge distinguisher, i.e., $\mid Pr[\mathbf{G0}] - Pr[\mathbf{G1}] \mid \leq \nu_{zk}(\lambda)$.
Game 2 (G2): Same as **G1** but upon setup we obtain $(crs, \tau, \chi) \leftarrow S_1(\lambda)$, and additionally store χ.
Transition - Game 1 → Game 2: Under simulation sound extractability, this change is conceptual, i.e., $Pr[\mathbf{G1}] = Pr[\mathbf{G2}]$.
Game 3 (G3): Same as **G2** but we modify the Adapt oracle so that it no longer draws etd uniformly at random but directly draws h' uniformly at random from G^*. To hash m w.r.t. $pk_{ch} = (PK, h, \pi_{pk})$ do $\boxed{h' \leftarrow G^*}$, $D \leftarrow g^d$, and $\cdots$
Game 4 (G4): Same as **G3** but for every π_p returned by cHash we record the value β so that $\beta = r$ in $R[\beta] \leftarrow r$:
Transition - Game 3 → Game 4: This change is conceptual, i.e., $Pr[\mathbf{G3}] = Pr[\mathbf{G4}]$
Game 5 (G5): Same as **G4** but pk^* output by the adversary. We extract x so that $g^x = h$. If the extraction fails, we abort.
Transition - Game 4 → Game 5: Both games proceed identically, unless we have to abort, i.e., $\mid Pr[\mathbf{G4}] - Pr[\mathbf{G5}] \mid \leq \nu_e(\lambda)$.
Game 6: Same as **G5**, but we obtain a DL instance (G, g, q, g^t), perform the setup with respect to (G, g, q) and further modify cHash thus: to hash m w.r.t. $pk_{ch} = (PK, h, \pi_{pk})$, do $\boxed{\text{Set } s \leftarrow \mathbb{Z}_q^*, h' \leftarrow (g^t)^s}$, $D \leftarrow g^d$, and $\cdots$. Furthermore, we record $S[h'] \leftarrow s$.
Transition - Game 5 → Game 6: This change is conceptual, i.e., $Pr[\mathbf{G5}] = Pr[\mathbf{G6}]$.
Game 7: Same as **G6**, but if π_p or π'_p contained in $\mathsf{rand}* = (\beta, p, \pi_p)$ and $\mathsf{rand}'* = (\beta', p', \pi'_p)$ do not correspond to a cHash answer we algebraically obtain r from $\beta \leftarrow (r + \frac{\delta}{x} + \frac{d}{x})$ and r' from $r' \leftarrow (\frac{rx + a \cdot \mathsf{etd} - a' \cdot \mathsf{etd} + \delta}{x})$, set $R[\beta] \leftarrow r$ or $R[\beta'] \leftarrow r'$.
Transition - Game 6 → Game 7: Both games proceed identically, i.e., $\mid Pr[\mathbf{G6}] = Pr[\mathbf{G7}] \mid$.

Revocation Collision Resistance:
Game 0 (G0): The original revocation collision-resistance game.
Game 1 (G1): Same as Game 0, but after the Adapt oracle returns $(m^{*\prime}, \mathsf{rand}^{*\prime}, \Gamma^{*\prime}_{\text{pubinfo}})$, $\mathcal{A}$ runs RCHET.adaptTrapdoor, and at Step (6), $\mathcal{A}$ will return an incorrect result.
If $\mathcal{A}$ is successful in Step (6), RCHET.adaptTrapdoor, it proved knowledge of $g^{\delta d}$, and it was able to extract δ and d values from Δ and D, respectively, thus breaking the DL assumption.

The advantage of the adversary for the entire RCHET scheme is given by the following equation:

$$\mathsf{Adv}^{\mathcal{A}}_{\mathsf{RCHET}}(\lambda) \leq \Big[\mathsf{Adv}^{\mathcal{A}}_{\mathsf{NIZKPoK}}(\lambda) + \mathsf{Adv}^{\mathcal{A}}_{\mathsf{H}^{k}_{\mathbb{Z}^{*}_{q}}.\mathsf{CollisionResistance}}(\lambda) + \mathsf{Adv}^{\mathcal{A}}_{DL}(\lambda) + \mathsf{Adv}^{\mathcal{A}}_{DDH}(\lambda) + \mathsf{Adv}^{\mathcal{A}}_{\Pi.CCA}(\lambda)\Big]$$

□

6 ATTRIBUTE-BASED ENCRYPTION AND DYNAMIC GROUP SIGNATURE SCHEMES

We need a revocable attribute-based encryption scheme, and a dynamic group signature scheme in ReTRACe. The revocable attribute-based encryption scheme is needed for controlling access to the ephemeral trapdoor of RCHET, including updating the trapdoor. We use the revocable ciphertext policy attribute-based encryption scheme proposed by us in [37], called RFAME, which does user-level revocation, in addition to policy-level revocation. Using RFAME, we ensure that only authorized users in **AuthU** can update Υ_{ABE}, and only authorized users in **AuthUAdmins** can update $\Upsilon_{ABE\text{admin}}$. The RFAME scheme presented in [37] provides full IND-CPA security under the DLIN assumption in the random oracle model for Type III pairings; we give its definition below.

Definition 6.1. *(Revocable ciphertext policy attribute-based encryption (*RFAME*) scheme)*
1) $\mathsf{RFAME.Setup}(1^{\lambda}, \mathbb{U}) \rightarrow (mpk, msk)$*: This algorithm takes as input the security parameter, the attributes in the universe,* $\mathbb{U}$*, and generates the master public and secret keys.*
2) $\mathsf{RFAME.KeyGen}(msk, \mathbb{S}) \rightarrow (sk_1, sk_2, \ldots, sk_{|\mathbb{S}|})$*: This algorithm takes in the master secret key, a set of attributes* $\mathbb{S}$*, and outputs secret keys for each attribute in* $\mathbb{S}$*.*
3) $\mathsf{RFAME.Encrypt}(mpk, m, \Upsilon) \rightarrow C$*: The encrypt algorithm takes in the master public key, a message to be encrypted, and an access policy,* Υ*. It outputs a ciphertext* C*.*
4) $\mathsf{RFAME.Decrypt}(sk_1, \ldots, sk_{|\mathbb{S}|}, C, \Upsilon) \rightarrow \{m, \perp\}$*: The decryption algorithm takes in the set of signing keys, a ciphertext tagged with a policy* Υ*, and outputs the message* m *if decryption is successful, else outputs* $\perp$*.*
5) $\mathsf{RFAME.Revoke}(mpk, msk, uid, v) \rightarrow (mpk', msk', sk_v)$*: This algorithm takes in the master public and secret keys, a user uid with attribute* v*, who needs to be revoked. It returns the new master public and secret keys to AIA, and the new secret key,* sk_v*, for the non-revoked users possessing* v*.*

For a detailed security analysis of RFAME and its proof of security, we refer the reader to [37].

We use a group signature scheme for providing privacy and anonymity to users posting messages on the BC, yet retaining the ability to trace them if necessary. The group signature scheme can be easily replaced with a regular signature scheme in ReTRACe if anonymity is not required in the system. Group signature schemes are based on three kinds of groups: static, semi-dynamic, and dynamic groups. Static groups do not support user addition or revocation [5], semi-dynamic groups support addition but not revocation [7], and dynamic groups allow addition and revocation [10]. We use a dynamic group signature scheme (DGSS) in ReTRACe, we do not construct a DGSS, as existing constructions [10, 34] provide the properties we need. ReTRACe is independent of the specifics of any DGSS construction.

At a high level in a GSS, there is a group manager (GM), who administers group membership, and a set of users who get their signing keys from the GM. There are eight algorithms: the probabilistic GSetup, GKGen, UpdateGroup, Sign, UserTrace are used for setting up group parameters, GM's keys, users' keys, signing a message, and tracing the signer of a message, respectively. The deterministic IsActive, VerifySignature, Judge are used to check if a user is an active group member, to verify signatures, and to judge if the tracing procedure has been run correctly by the GM. Additionally, a DGSS also has an interactive Join protocol run between the GM and users.

A DGSS is said to be secure if it is: 1) mathematically *correct*, 2) provides *anonymity* (does not reveal the identity of a group member who produced signatures), 3) provides *traceability* (a group manager can trace all valid signatures to corresponding active members of the group), and 4) provides *non-frameability* (even if the rest of the group collude, they cannot generate a signature that is falsely attributed to a honest user who did not produce it).

7 THE ReTRACe FRAMEWORK

We first present the definition of ReTRACe, which describes, at an abstract level, the purpose of each constituent algorithm.

Definition 7.1. *(*ReTRACe *scheme)*
(1) ReTRACe.Keygen$(1^\lambda) \rightarrow$ (SecPar, PubPar)*: This algorithm takes in a security parameter, and outputs the secret and public parameters of the* ReTRACe *system.*
(2) ReTRACe.UserSetup(SecPar, PubPar) $\rightarrow$ key*: This algorithm takes as input the secret and public parameters, and initializes each user with a tuple,* key*, consisting of their group signing keys,* RFAME *keys, and* RCHET *long-term trapdoor.*
(3) ReTRACe.Sign$(\mathbb{GSK}, m, \Upsilon) \rightarrow \xi_m$*: This algorithm takes in a set of signing keys,* $\mathbb{GSK}$*, a message, a* DGSS *policy,* Υ*, and outputs a set of signatures,* ξ_m *satisfying* Υ*.*
(4) ReTRACe.CreateMessage(key, PubPar, m) $\rightarrow$ (msg, ξ_{msg})*: This algorithm is run by the originator, takes in a tuple,* key*, public parameters, a message, and outputs a tuple, msg (consisting of* RCHET*,* RFAME *and* DGSS *parameters for m) and a set of signatures on msg,* ξ_{msg}*.*
(5) ReTRACe.AdaptMessage(key, PubPar, m', msg, ξ_{msg}) $\rightarrow$ (msg', $\xi_{msg'}$)*: This algorithm, run by members of* ***AuthU*** *takes as input a tuple,* key*, the public parameters, a message, a tuple, msg, and a set of signatures on msg,* ξ_{msg}*. If the* RCHET*,* RFAME*,* DGSS *parameters contained in msg pass verification, it updates the m contained in msg to* m'*, returns an updated message tuple,* msg' *with new* RCHET*,* RFAME *and* DGSS *parameters, and a set of signatures,* $\xi_{msg'}$ *on* msg'*.*
(6) ReTRACe.Verify(PubPar, msg, ξ_{msg}) $\rightarrow \{0, 1\}$*: This algorithm takes in* PubPar*, msg, and set of signatures on msg,* ξ_{msg}*. If the* RCHET*,* RFAME*, and* DGSS *parameters contained in msg pass verification, it returns 1.*
(7) ReTRACe.VerifyMiner(PubPar, msg, ξ_{msg}, ς) $\rightarrow \{0, 1\}$*: This algorithm is run by the miners, and takes in the public parameters, a msg tuple, a set of signatures on msg,* ξ_{msg}*, and a variable* ς*, which conveys whether the person that submitted msg was a member of* ***AuthU*** *or* ***AuthUAdmins****. If the* RCHET*,* RFAME *and DGSS parameters in msg pass verification, the algorithm returns 1, and the msg will be posted on the BC.*
(8) ReTRACe.RevokeUser(key, PubPar, m', msg, ξ_{msg}) $\rightarrow$ (msg', $\xi_{msg'}$)*: This algorithm is run by members of* ***AuthUAdmins*** *who want to revoke users either by updating the Boolean policies associated with* RFAME/DGSS *contained in msg, or in response to the AIA/GM revoking users. It outputs an updated tuple,* msg' *containing new* RCHET*,* RFAME*,* DGSS *parameters and set of signatures,* $\xi_{msg'}$ *on* msg'*.*

Remark 7.1. *An* originator *of a message could possibly create malformed policies, e.g., policies containing bogus or non-existent attributes. We assume the miner has knowledge of all the (public) attributes in the universe and in this case would reject malformed policies. An originator of a message, m, could create a bogus trapdoor, which would not be discovered until someone attempts to update m. Note that the miner cannot check if the encrypted trapdoor is correct or not, since the miner likely will not be part of the* ***AuthU****, or* ***AuthUAdmins*** *sets. Solutions to this problem include having the originator do a verifiable encryption [14, 43] of the trapdoor, while submitting m to the miner, or have the originator include an NIZK proof along with the message. We leave the construction of a scheme that incorporates these ideas as future work.*

ReTRACe.Keygen(1^λ)

1: GSetup(1^λ) $\rightarrow$ pubpar
2: GKGen(pubpar) $\rightarrow$ (out_{GM}, st_{GM}), where $out_{GM} = (mpk, \text{info}_0)$
3: Set gpk = (pubpar, mpk), st_{GM} is GM's state
4: RFAME.SetupABE($1^\lambda, U$) $\rightarrow$ $(\text{mpk}_{\text{ABE}}, \text{msk}_{\text{ABE}})$
5: RCHET.cSetup(1^λ) $\rightarrow$ param
6: RCHET.userKeySetup(param) $\rightarrow$ (sk_{ch}, pk_{ch})
7: PubPar = $(\mathbb{G}, g, q, pk_{ch}, \text{mpk}_{\text{ABE}}, gpk)$
8: SecPar = $(\text{msk}_{\text{ABE}}, st_{GM})$
9: **return** (SecPar, PubPar, sk_{ch})

(a) ReTRACe: AIA/GM setup.

ReTRACe.Sign($\mathbb{GSK}, m, \Upsilon_{\text{GS}}$)

1: Pick $\mathbb{K}$ *s.t.*, $\mathbb{K} \subseteq \mathbb{GSK}, \Upsilon_{\text{GS}}(\mathbb{K}) = 1$
2: **for** $gsk^i \in \mathbb{K}$, where $i \in 1 \cdots | \mathbb{GSK} |$ **do**
3: DGSS.Sign($gsk^i, i.\text{info}, m$) $\rightarrow$ $\{\sigma_i, \perp\}$
4: $\xi = (\sigma_i, i.gpk) \cup \xi$
5: **return** ξ

(b) ReTRACe: Signing a message.

ReTRACe.UserSetup(SecPar, PubPar)

1: Get $\mathbb{L}$, the list of all groups in DGSS that current user needs to join, set $\mathbb{GSK} = \emptyset$
2: For each group in $\mathbb{L}$, Run DGSS.Join get gsk, $\mathbb{GSK} = \mathbb{GSK} \cup gsk$
3: RFAME.KeyGenABE($\text{mpk}_{\text{ABE}}, \text{msk}_{\text{ABE}}, y_1, \ldots, y_{|U|}$) $\rightarrow$ SK
4: Retrieve sk_{ch}
5: **return** key = $(\mathbb{GSK}, SK, sk_{ch})$

(c) ReTRACe: System setup for user.

ReTRACe.Verify(PubPar, msg, ξ_{msg})

1: **for** $(\sigma_l, l.gpk) \in \xi_{msg}$ **do**
2: **if** DGSS.VerifySignature($l.gpk, l.\text{info}, msg, \sigma_l$) $\stackrel{?}{=} 0$, **return** 0
3: **for** $(\sigma_l, l.gpk) \in \xi_{\Upsilon_{\text{info}}}$ **do**
4: **if** DGSS.VerifySignature($l.gpk, l.\text{info}, \Upsilon_{\text{info}}, \sigma_l$) $\stackrel{?}{=} 0$, **return** 0
5: **if** RCHET.verifyTransaction(pk_{ch}, m, digest, rand, Γ_{pubinfo}) $\stackrel{?}{=} 1$
6: **return** 1.

(d) ReTRACe: Verifying a message.

Fig. 7. ReTRACe algorithms for system setup and signing/verifying messages.

7.1 ReTRACe Construction

We now give the detailed construction of the ReTRACe framework comprising of eight algorithms in Figure 7, Figure 8, Figure 10, and Figure 9. In the algorithms, **M** denotes the monotone span program representing an ABE policy, and ρ represents a mapping function that maps rows of **M** onto attributes. We use BC.write to

denote a blockchain write operation. The Keygen, UserSetup, Sign, Verify, AdaptMessage algorithms are fairly self-explanatory. We now describe some of the salient features of the CreateMessage, VerifyMiner, and RevokeUser algorithms, which are more involved. We assume a given implementation will use standard techniques like nonces and timestamps to prevent against replay attacks.

ReTRACe.CreateMessage(key, PubPar, m)

1 : RCHET.cHash$(sk_{ch}, pk_{ch}, m) \rightarrow (\text{digest}, \text{rand}, \Gamma_{\text{pubinfo}}, \Gamma_{\text{privinfo}})$
2 : RFAME.Encrypt$(\text{mpk}_{\text{ABE}}, \Gamma_{\text{privinfo}}, (\mathbf{M}_{\Upsilon_{\text{ABE}}}, \rho_{\Upsilon_{\text{ABE}}})) \rightarrow X$
3 : Create Υ_{GS}. Set $\Upsilon_{\text{info}} = (\Upsilon_{\text{ABE}}, \Upsilon_{\text{GS}})$
4 : $r \leftarrow\$ \mathbb{Z}_q^*, \omega = g^r, \pi_\omega \leftarrow \text{NIZKPoK}\{r : \omega = g^r\}$
5 : RFAME.Encrypt$(\text{mpk}_{\text{ABE}}, r, (\mathbf{M}_{\Upsilon_{ABE\text{admin}}}, \rho_{\Upsilon_{ABE\text{admin}}})) \rightarrow X_r$
6 : Create $\Upsilon_{GS\text{admin}}$
7 : Set$\Upsilon_{\text{admin}} = (\Upsilon_{ABE\text{admin}}, \Upsilon_{GS\text{admin}}, X_r, \omega, \pi_\omega)$
8 : ReTRACe.Sign$(\mathbb{GSK}, \Upsilon_{\text{info}}, \Upsilon_{GS\text{admin}}) \rightarrow \xi_{\Upsilon_{\text{info}}}$
9 : $msg = (m, \text{digest}, \text{rand}, \Gamma_{\text{pubinfo}}, X, \Upsilon_{\text{info}}, \Upsilon_{\text{admin}}, \xi_{\Upsilon_{\text{info}}})$
10 : ReTRACe.Sign$(\mathbb{GSK}, msg, \Upsilon_{\text{GS}}) \rightarrow \xi_{msg}$
11 : Call ReTRACe.VerifyMiner(PubPar, $msg, \xi_{msg}, \pi_\omega$)
12 : **return** (msg, ξ_{msg})

(a) ReTRACe: Creating a message.

ReTRACe.AdaptMessage(key, PubPar, m', msg, ξ_{msg})

1 : **if** ReTRACe.Verify(PubPar, $msg, \xi_{msg}) \stackrel{?}{=} 0$
return $\perp$
2 : RFAME.Decrypt$(SK, X, (\mathbf{M}_{\Upsilon_{\text{ABE}}}, \rho_{\Upsilon_{\text{ABE}}})) \rightarrow \Gamma_{\text{privinfo}}$
3 : RCHET.adaptMessage$(sk_{ch}, m, m', \text{digest}, \text{rand}, \Gamma_{\text{pubinfo}}, \Gamma_{\text{privinfo}}) \rightarrow \text{rand}'$
4 : $msg' = (m', \text{digest}, \text{rand}', \Gamma_{\text{pubinfo}}, X, \Upsilon_{\text{info}}, \Upsilon_{\text{admin}}, \xi_{\Upsilon_{\text{info}}})$
5 : ReTRACe.Sign$(\mathbb{GSK}, msg', \Upsilon_{\text{GS}}) \rightarrow \xi_{msg'}$
6 : **if** ReTRACe.VerifyMiner(PubPar, $msg', \xi_{msg'}, \perp) \stackrel{?}{=} 0$
return $\perp$
7 : **return** $(msg', \xi_{msg'})$

(b) ReTRACe: Updating a message.

Fig. 8. ReTRACe algorithms for creating and updating a message.

ReTRACe.CreateMessage: This algorithm (Figure 8a), is run by the originator who first runs RCHET to create a digest and trapdoors for a message m. The originator sets Υ_{ABE}, Υ_{GS} (for members of **AuthU**), and $\Upsilon_{ABE\text{admin}}$ and $\Upsilon_{GS\text{admin}}$ (for members of **AuthUAdmins**). The ephemeral trapdoor is encrypted under Υ_{ABE} to obtain X. The originator then picks an $r \leftarrow\$ \mathbb{Z}_q^*$ and encrypts it under $\Upsilon_{ABE\text{admin}}$ to obtain X_r. This ensures that only members of **AuthUAdmins** can decrypt r, and modify Υ_{ABE} and Υ_{GS}. The originator creates a tuple, msg, with X and **AuthU**, **AuthUAdmins** policy information, signs msg using her signing key(s) that satisfy Υ_{GS}, and creates a set of signatures, $\xi_{\Upsilon_{\text{info}}}$, with each signature bundled with its corresponding verification key. Finally, the originator

signs Υ_{info} and sends the signature, along with msg, ξ_{msg}, $\xi_{\Upsilon_{\text{info}}}$ to the miner.

ReTRACe.VerifyMiner: This algorithm (Figure 10) is run only by miners to verify a message before posting it on the BC. If a message is being adapted ($\varsigma = \perp$), the miner does not do NIZK verifications. If a trapdoor is being adapted ($\varsigma = \pi_\omega$), the tuple submitted to the miner is an update to a pre-existing msg on the BC, and the ω used to verify the NIZK π_ω is obtained from the current msg on the BC. If a new message msg is being created ($\varsigma = \pi_\omega$), then the ω used to verify the NIZK π_ω is obtained from the msg tuple itself. In all cases, the miner checks if all signatures in ξ_{msg} pass verification w.r.t. Υ_{GS} contained in the tuple msg, checks if all signatures in $\xi_{\Upsilon_{\text{info}}}$ pass verification w.r.t. $\Upsilon_{GS\text{admin}}$, and the digest of m is checked. If all checks pass, the msg tuple, along with the list of signatures on it is written to the BC. Note that if ReTRACe is deployed in a BC that hosts both mutable and immutable transactions, then for immutable transactions, the miner verification process is the same as in current BC systems.

ReTRACe.RevokeUser: This algorithm (Figure 9a, Figure 9b) is called by a member of **AuthUAdmins** either when they want to revoke clauses from the ABE policy, Υ_{ABE}, or when the ephemeral trapdoor, Γ_{privinfo}, needs to be re-encrypted in response to the AIA revoking a user. Both cases are handled differently:

Case 1: Revoking a clause from Υ_{ABE}: This type of revocation is similar to direct revocation, as defined in the ABE literature [3], where the revocation happens via a policy update by a user. This algorithm (Figure 9a) is run by an user $v \in$ **AuthUAdmins** who wants to modify an Υ_{ABE} associated with msg. The AIA/GM are not involved, and no algorithm from RFAME is called. User v first decrypts the trapdoor, Γ_{privinfo}, using her RFAME secret keys,

v picks an r', and encrypts r' under $\Upsilon_{ABE\text{admin}}$. This is to ensure that only non-revoked members of **AuthUAdmins** can decrypt r' and adapt the ephemeral trapdoor in the future. Next, v adapts the ephemeral trapdoor. The new message and trapdoor are encrypted under a new policy, Υ'_{ABE}, which is a low cost operation and involves no re-keying operations. We have not depicted the Υ_{GS} getting updated, for clarity of presentation. There are four cases:

1) If Υ_{ABE} changes to a more inclusive Υ'_{ABE}, the new user groups need to be present in the Υ_{GS} as well.
2) If Υ_{ABE} changes to a more restrictive Υ'_{ABE}, the revoked users cannot decrypt the trapdoor and successfully adapt the message, so Υ_{GS} does not need to change.
3) If Υ_{GS} changes to a more restrictive Υ'_{GS}, such that the users satisfying Υ_{GS} were also part of Υ_{ABE}, Υ_{ABE} needs to change too, revoking the said users from the ABE scheme.
4) If Υ_{GS} changes to a more inclusive Υ'_{GS}, such that the users satisfying Υ'_{GS} are not part of Υ_{ABE}, the new users cannot decrypt the trapdoor and successfully adapt the message, so Υ_{ABE} does not need to change. User v then signs the new Υ'_{info} using their signing keys that satisfy $\Upsilon_{GS\text{admin}}$; the signature set is denoted as $\xi_{\Upsilon'_{\text{info}}}$. A new msg' is created and signed using a set of keys that satisfy Υ_{GS}, and the resulting signature set is denoted by $\xi_{msg'}$. Finally, msg' and $\xi_{msg'}$ are given to the miner who verifies it before posting on the BC.

Case 2: AIA revoking a user: This type of revocation is similar to indirect revocation [3], where an AIA revokes a user's access by renewing the keys of the non-revoked users. This algorithm (Figure 9b) is run by a user $v \in$ **AuthUAdmins** as soon as the AIA revokes a user holding attribute y (which appears in either Υ_{ABE} or $\Upsilon_{ABE\text{admin}}$). First, the AIA updates its own public key from mpk_{ABE} to $\text{mpk}_{\text{ABE}}{}'$ (which results in PubPar getting updated to PubPar′), and then issues new signing keys, SK', only to the *non-revoked* users holding attribute y. User v then proceeds to adapt the ephemeral trapdoor, Γ_{privinfo}, to prevent the revoked user from being able to perform any future message adaptations. User v then generates a new r', encrypts it, etc., the rest of the steps are similar to Case 1.

ReTRACe.RevokeUser(key, PubPar, m', msg, ξ_{msg})

1 : **if** ReTRACe.Verify(PubPar, msg, ξ_{msg}) $\stackrel{?}{=} 0$ **return** $\perp$
2 : RFAME.Decrypt($SK, X_r, (\mathbf{M}_{\Upsilon_{ABE\text{admin}}}, \rho_{\Upsilon_{ABE\text{admin}}})) \rightarrow r$
3 : $r' \leftarrow\$ \mathbb{Z}_q^*, \omega' = g^{r'}$.Set $\pi_{\omega'} \leftarrow$ NIZKPoK$\{r' : \omega' = g^{r'}\}$
4 : RFAME.Encrypt($\text{mpk}_{\text{ABE}}, r', (\mathbf{M}_{\Upsilon_{ABE\text{admin}}}, \rho_{\Upsilon_{ABE\text{admin}}})) \rightarrow X_{r'}$
5 : $\Upsilon'_{\text{admin}} = (\Upsilon_{ABE\text{admin}}, \Upsilon_{GS\text{admin}}, X_{r'}, \omega', \pi_{\omega'})$
6 : RFAME.Decrypt($SK, X, (\mathbf{M}_{\Upsilon_{\text{ABE}}}, \rho_{\Upsilon_{\text{ABE}}})) \rightarrow \Gamma_{\text{privinfo}}$
7 : RCHET.adaptTrapdoor(sk_{ch}, m, m', digest, rand, $\Gamma_{\text{pubinfo}}, \Gamma_{\text{privinfo}}) \rightarrow$ (rand$', \Gamma'_{\text{pubinfo}}, \Gamma'_{\text{privinfo}}$)
8 : RFAME.Encrypt($\text{mpk}_{\text{ABE}}, \Gamma'_{\text{privinfo}}, (\mathbf{M}_{\Upsilon'_{\text{ABE}}}, \rho_{\Upsilon'_{\text{ABE}}})) \rightarrow X'$
9 : $\Upsilon'_{\text{info}} = (\Upsilon'_{\text{ABE}}, \Upsilon_{\text{GS}})$
10 : ReTRACe.Sign($\mathbb{GSK}, \Upsilon'_{\text{info}}, \Upsilon_{GS\text{admin}}) \rightarrow \xi_{\Upsilon'_{\text{info}}}$
11 : $msg' = (m'$, digest, rand$', \Gamma'_{\text{pubinfo}}, X', \Upsilon'_{\text{info}}, \Upsilon'_{\text{admin}}, \xi_{\Upsilon'_{\text{info}}})$
12 : ReTRACe.Sign($\mathbb{GSK}, msg', \Upsilon_{\text{GS}}) \rightarrow \xi_{msg'}$
13 : Call VerifyMiner(PubPar, $msg', \xi_{msg'}, \pi_{\omega'}$)
14 : **return** $(msg', \xi_{msg'})$

(a) ReTRACe: Revoke Case 1: Revoke users by updating policies.

ReTRACe.RevokeUser(key, PubPar$'$, m', msg, ξ_{msg})

1 : **if** ReTRACe.Verify(PubPar$', msg, \xi_{msg}) \stackrel{?}{=} 0$
return $\perp$
2 : RFAME.Decrypt($SK, X_r, (\mathbf{M}_{\Upsilon_{ABE\text{admin}}}, \rho_{\Upsilon_{ABE\text{admin}}})) \rightarrow r$
3 : $r' \leftarrow\$ \mathbb{Z}_q^*, \omega' = g^{r'}$.Set $\pi_{\omega'} \leftarrow$ NIZKPoK$\{r' : \omega' = g^{r'}\}$
4 : RFAME.Encrypt($\text{mpk}_{\text{ABE}}', r', (\mathbf{M}_{\Upsilon_{ABE\text{admin}}}, \rho_{\Upsilon_{ABE\text{admin}}})) \rightarrow X_{r'}$
5 : $\Upsilon'_{\text{admin}} = (\Upsilon_{ABE\text{admin}}, \Upsilon_{GS\text{admin}}, X_{r'}, \omega', \pi_{\omega'})$
6 : RFAME.Decrypt($SK, X, (\mathbf{M}_{\Upsilon_{\text{ABE}}}, \rho_{\Upsilon_{\text{ABE}}})) \rightarrow \Gamma_{\text{privinfo}}$
7 : RCHET.adaptTrapdoor(sk_{ch}, m, m', digest, rand, $\Gamma_{\text{pubinfo}}, \Gamma_{\text{privinfo}}) \rightarrow$ (rand$', \Gamma'_{\text{pubinfo}}, \Gamma'_{\text{privinfo}}$)
8 : RFAME.Encrypt($\text{mpk}_{\text{ABE}}', \Gamma'_{\text{privinfo}}, (\mathbf{M}_{\Upsilon_{\text{ABE}}}, \rho_{\Upsilon_{\text{ABE}}})) \rightarrow X'$
9 : $msg' = (m'$, digest, rand$', \Gamma'_{\text{pubinfo}}, X', \Upsilon_{\text{info}}, \Upsilon'_{\text{admin}}, \xi_{\Upsilon_{\text{info}}})$
10 : ReTRACe.Sign($\mathbb{GSK}, msg', \Upsilon_{\text{GS}}) \rightarrow \xi_{msg'}$
11 : Call VerifyMiner(PubPar$', msg', \xi_{msg'}, \pi_{\omega'}$)
return $(msg', \xi_{msg'})$

(b) ReTRACe: Revoke Case 2: AIA revoking a single user.

Fig. 9. ReTRACe algorithms for revoking users.

7.2 ReTRACe Security Properties

We now informally discuss the security properties of ReTRACe: indistinguishability, public, private, and revocation collision resistance. The first three properties were first introduced by Derler et al. [21] for any policy-based chameleon hash scheme. We define revocation collision resistance, and strengthen the first three properties, by giving the adversary the ability to adapt messages and revoke messages. Indistinguishability requires that it should be computationally infeasible for an adversary to distinguish whether the randomness associated with a given message was generated as a result of a CreateMessage, AdaptMessage or RevokeUser. Public collision resistance requires that an adversary who knows neither the long-term nor the ephemeral trapdoor cannot produce valid collisions even after seeing past adaptations of messages and trapdoors, even with access to some attributes, but not the complete attribute set that can decrypt the ephemeral trapdoor.

ReTRACe.VerifyMiner(PubPar, $msg, \xi_{msg}, \varsigma$)

1 : **for** $(\sigma_l, l.gpk) \in \xi_{msg}$ **do**
2 : **if** DGSS.VerifySignature$(l.gpk, l.\text{info}, msg, \sigma_l) \stackrel{?}{=} 0$
3 : **return** 0
4 : **for** $(\sigma_l, l.gpk) \in \xi_{\Upsilon_{\text{info}}}$ **do**
5 : **if** DGSS.VerifySignature$(l.gpk, l.\text{info}, \Upsilon_{\text{info}}, \sigma_l) \stackrel{?}{=} 0$
6 : **return** 0
7 : **if** $\varsigma = \pi_\omega$
8 : **if** verify$(\omega, \pi_\omega) \neq 1$, **return** 0
9 : **if** RCHET.verifyTransaction$(pk_{ch}, m, \text{digest}, \text{rand}, \Gamma_{\text{pubinfo}}) \stackrel{?}{=} 1$
10 : BC.write$(msg, \xi_{msg},)$ **return** 1
11 : **return** 0

Fig. 10. ReTRACe: Miner verifying a message.

Private collision resistance requires that an adversary that knows the long-term trapdoor, but not ephemeral trapdoor of the RCHET scheme, cannot produce valid collisions, even with knowledge of past message and trapdoor adaptations. This property should hold even if she has access to a subset of attributes, but not the complete set of attributes, needed to decrypt the current trapdoor. Revocation collision resistance requires that an adversary, who knows the long-term and ephemeral trapdoors, and has valid attributes to decrypt the ephemeral trapdoor, cannot produce valid collisions, if, either the RFAME policy changed to exclude her, or the AIA revoked some or all of her attributes necessary to decrypt the trapdoor. We have proven the IND-CPA security of RFAME; we apply the Fujisaki-Okamoto transform [24] to convert RFAME to an IND-CCA2 secure scheme to accomplish the proof. The formalization of the security properties and the proof of the following theorem are in the full version [38].

Theorem 7.1. *If* RCHET *is secure,* RFAME *is fully IND-CCA2 secure, and* DGSS *is a secure dynamic group signature scheme then* ReTRACe *provides the properties of indistinguishability, public, private and revocation collision resistance.*

7.3 Complexity Analysis

Complexity Analysis: We compute the total number of operations in RCHET in Table 2 in terms of the number of exponentiations (Exp), multiplications (Mul), encryptions (Enc), decryptions (Dec), hash computations (Hash),

pairings (Pair), NIZK proof generation (NIZK_G), NIZK proof verification (NIZK_V). We denote the total number of attributes in the system as N, the number of attributes in Υ_{GS} as x, and the number of attributes in $\Upsilon_{GS\text{admin}}$ as y. We assume conjunctive access policies of the form Attr_1 AND Attr_2 AND Attr_3 AND ... AND Attr_N. The users cardinality in each attribute group is denoted by U_g, and we assume all attribute groups have the same users cardinality.

Table 2. Complexity of RCHET.

Algorithm	Exp	Mul	Hash	NIZK_G	NIZK_V	Enc	Dec
cSetup	1	-	-	1	-	-	-
cHash	5	3	1	4	-	1	-
verifyTransaction	2	3	1	-	5	-	-
adaptTrapdoor	10	8	2	3	-	1	2
adaptMessage	8	8	2	1	-	-	2

Table 3. Complexity of ReTRACe (N is # of total attributes, U_g is # of users in the revoked attribute, x is # of satisfied attributes in the Υ_{GS}, y is # of satisfied attributes in the $\Upsilon_{GS\text{admin}}$).

Algorithm	Exp	Mul	Hash	Pair	NIZK_G	NIZK_V	SIG_S
Keygen	N+7	4	-	2	1	-	-
UserSetup	12N+14	19N+17	3N+3	-	-	-	-
CreateMessage	$16N+12n_2+6x+6y+11$	$16N+6n_2+7$	$12N+12n_2+1$	-	x+y+5	-	x+y
Sign	6(x|y)	-	-	-	x|y	-	x|y
Verify	2	3	1	-	-	x+y+5	-
VerifyMiner	2	3	1	-	-	x+y+6	-
AdaptMessage	6x+12	6y+18	4	2y+6	x+1	x+y+11	x
RevokeUser Case 1	$16N+12n_2+6x+6y+19$	$16N+6n_2+12y+23$	$12N+12n_2+3$	4y+12	x+y+4	x+y+5	x+y
RevokeUser Case 2	$16N+12n_2+6x+19$	$16N+6n_2+12y+23$	$12N+12n_2+3$	4y+12	x+4	x+y+5	x

The analysis in Table 2 shows that the computation complexity of each algorithm of RCHET is similar to CHET (for results refer Table 5). In Table 3, we analyze the asymptotic cost of the ReTRACe algorithms. The cost of ReTRACe includes the cost of the respective RCHET, RFAME, and DGSS functions called within ReTRACe algorithms. We use Bellare *et al.* [7] to quantify the cost of DGSS operations in ReTRACe. Due to the modular nature of our framework, this scheme can easily be replaced with another scheme, such as [10], for different security guarantees. The DGSS scheme involves regular signature operations signified by SIG_S. Since [7] needs a CCA-secure encryption scheme, we instantiate it using the Cramer-Shoup cryptosystem [18].

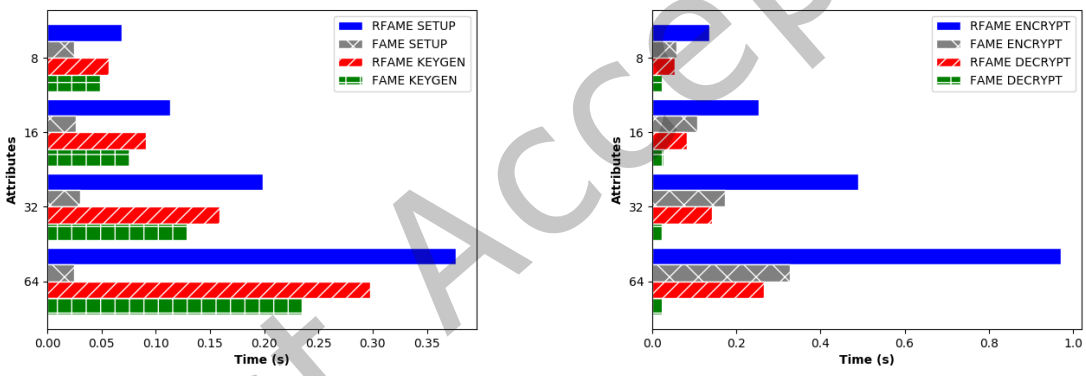
Security Analysis: ReTRACe provides security from a malicious originator of *msg* getting revoked from the system and locking out any other user in the system from updating the *msg*. Malicious regular users cannot update the *msg* and send bogus verification metadata for the transaction since the miner will deny the transaction. A malicious regular user or a set of colluding malicious regular users cannot update trapdoor and hence cannot lock out other regular users from updating *msg*. We provide security from a single or a set of malicious admin users locking out other authorized users from updating the *msg*, or locking out honest admin users from updating the trapdoor.

8 IMPLEMENTATION AND RESULTS

We implemented RFAME, RCHET, and ReTRACe in Python 3, and used Charm [15] for cryptographic modules. All the experiments were carried out on a machine with 64 GB RAM and an Intel(R) Core(TM) i7-6700K CPU clocked at 4.00 GHz. We implemented RCHET and RFAME to compare their performance against CHET and

FAME, respectively, to quantify the price of adding revocation. We do not compare RFAME quantitatively with other revocable ABE schemes, since they do not provide the properties that RFAME provides. Using RCHET and RFAME we implement ReTRACe. Note that ReTRACe is the first system that provides transaction-level revocable blockchain rewrites without the need of trusted entities or other infrastructure, such as revocation lists. All other contending schemes in the state-of-the-art require revocation lists, or some kind of trusted entity/entities, or have limiting assumptions (refer Table 1). As ReTRACe is – capability-wise – in a class of its own, a quantitative comparison with the other approaches would not be accurate nor appropriate. Hence, in what follows, we present the quantitative results of ReTRACe. All operations in ReTRACe other than UserSetup and Keygen took less than 1 second for different attribute and user configurations which demonstrates its applicability in the real world.

RFAME Results: We set our ABE policies to contain a total of 8, 16, 32 and 64 attributes, and all our policies have two equisized conjunctive clauses separated by a single disjunction. In each run, 10, 20, 40, or 80 users signed up with the AIA for each attribute. The computation time increases linearly with the number of users, so for brevity, in Figure 11 we show results for RFAME and FAME for 80 users per attribute only. The setup times for RFAME are higher than for FAME because of the extra operations involved in computing the master public key (mpk_{ABE}) and master secret key (msk_{ABE}) during setup; and the growth of the public key size in RFAME is linear in the number of attributes (small-universe property).



(a) Key generation and Setup

(b) Encryption and Decryption

Fig. 11. Timings for RFAME vs. FAME [1] (80 users per attribute).

In FAME, the size of one of the components of the ciphertext increases linearly in the number of attributes satisfying the given policy, whereas for RFAME there are two components whose size increases linearly, which accounts for the difference in the timings of the encryption and decryption operations. For decryption, the number of pairing operations is 6 + 2×(number of attributes satisfying a given policy) for RFAME, as compared to 6 pairing operations for FAME.

Table 4 shows the time taken when revoking one user from each attribute group with 10, 20, 40, and 80 users each, which results in the rekeying of the remaining users. The results are linear as expected because in each case 9, 19, 39, and 79 users got new keys, respectively. We expect this trend to continue as the number of users per attribute increases.

Table 4. Timing for the RFAME.Revoke (time in secs).

10 Users per attribute	0.115
20 Users per attribute	0.2
40 Users per attribute	0.364
80 Users per attribute	0.714

Previous schemes do not provide efficient revocation. To carry out a user revocation under previous schemes, the entire system would have to be rekeyed using the Setup and Keygen functions, and all ciphertext re-encrypted regardless of whether the revoked user had access to the message or not. Thus, the cost in rekeying the users would be significantly lower in RFAME, especially if a user is in a single attribute group and is revoked.

Table 5. Comparison of RCHET vs. CHET (time in secs).

Algorithm	CHET	RCHET
Setup	0.537	0.5369
Chash	0.0216	0.0234
Verify	0.000697	0.000967
Adapt Message	0.0414	0.0415
Adapt Trapdoor	-	0.04305

RCHET Results: In RCHET, when compared to CHET [13], we have added one extra encryption and decryption, two NIZKPoK generation and verifications, and three modular exponentiations to all functions, except systemSetup and userKeySetup. Despite this, RCHET does not display a significant increase in latency, at the same time, providing the ability to adapt the trapdoor of a message digest. Table 5 compares the running times for CHET and RCHET. The time difference between RCHET and CHET algorithms is in the order of milliseconds and this is a minimal trade-off for the added functionality that RCHET provides.

ReTRACe Results: ReTRACe was implemented with the DGSS policies being the same as the ABE policies, and containing 20 users per attribute for 8 and 16 attributes. Except for the RFAME revocation component, whose running time is proportional to the number of users, the rest of the cryptographic primitives, i.e., DGSS, RCHET, and other RFAME algorithms, are independent of the number of users in the system. The running time for operations in ReTRACe would increase linearly with the number of users per attribute, as is evident from the RFAME results.

Table 6 shows the timings of ReTRACe, with 20 users/attribute and messages with policies containing 8 and 16 attributes respectively. UserSetup and Keygen take significantly more time than the other functions as expected; both these functions involve all users in the system and are run only once at the beginning during system and users' setup. CreateMessage, Sign, Verify, and VerifyMiner would be run more frequently, and all have sub-second timings. For implementing Case 1 of ReTRACe.RevokeUser, we eliminate one attribute from Υ_{ABE}, and in Case 2 we revoke one user from the AIA that held an attribute in Υ_{ABE}. Case 2 takes longer because it includes the AIA's operations for revoking a user from a single attribute group and rekeying of the rest of the users in the same group, whereas Case 1 updates the message policy and the message trapdoor.

Implementation in Ethereum: ReTRACe can be plugged into an existing blockchain (e.g., Ethereum) by updating relevant cryptographic operations with equivalent ones in ReTRACe. For instance, in Ethereum the signature algorithm in the module "crypto/crypto.go" needs to be modified to use ReTRACe.Sign; "trie/trie.go" to use the digest of rewritable transactions at the leaves of the blocks' Merkle trees; ReTRACe.AdaptMessage and ReTRACe.RevokeUser need to be added to the "ethclient" module and ReTRACe.VerifyMiner to the "miner/miner.go" module. We are porting these modifications to Ethereum.

Table 6. ReTRACe running time, 20 users/attribute (secs).

ReTRACe Algorithms	8 Attr	16 Attr
UserSetup and Keygen (for 20 users)	2.997	4.694
CreateMessage	0.473	0.963
Sign	0.0904	0.180
Verify	0.114	0.232
VerifyMiner	0.225	0.460
AdaptMessage	0.0928	0.152
RevokeUser (Case 1)	0.545	1.015
RevokeUser (Case 2) (for 19 users)	0.676	1.049

With ReTRACe-adapted Ethereum, an authorized user updates a transaction using the chameleon hash and then submits it to the transaction pool. In our design, the transactions will be updated with a binary flag ('0' ← new; '1' ← updated). A miner that picks up an updated transaction verifies the transaction using ReTRACe.Verify, updates the transaction in the block–the remaining transactions are untouched– and propagates the block for consensus. At each node storing the BC, the block with the updated transaction replaces the old block post transaction-verification.

The cost of ReTRACe operations in Ethereum (in gas) would be proportional to their computational cost shown in this section. The exact gas cost is dynamic, varying based on many factors (number of pending transactions, minimum cost, etc.). At the base computation level, ReTRACe scales linearly with increasing number of attributes and users—highly desirable.

9 CONCLUSION

We present ReTRACe, a blockchain transaction rewriting framework building on a novel revocable chameleon hash with ephemeral trapdoor scheme and a novel revocable CP-ABE scheme. We discuss ReTRACe's functionalities that provide efficient and authorized transaction rewrites in blockchains, in addition to revocability and traceability of the users updating the transactions(s). We have performed rigorous security and experimental analyses to demonstrate ReTRACe's scalability.

ACKNOWLEDGMENTS

Research supported by NSF awards #2417062, #2148358 (RINGS), #1914635, US Department of Energy award #DE-SC0023392, and Intel Labs. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the federal government and Intel Inc.

REFERENCES

[1] Shashank Agrawal and Melissa Chase. 2017. FAME: Fast Attribute-based Message Encryption. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS*. 665–682.

[2] Giuseppe Ateniese, Bernardo Magri, Daniele Venturi, and Ewerton Andrade. 2017. Redactable blockchain–or–rewriting history in bitcoin and friends. In *2017 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 111–126.

[3] Nuttapong Attrapadung and Hideki Imai. 2009. Attribute-based encryption supporting direct/indirect revocation modes. In *Cryptography and Coding: 12th IMA International Conference, Cryptography and Coding 2009, Cirencester, UK, December 15-17, 2009. Proceedings 12*. Springer, 278–300.

[4] Mihir Bellare, Georg Fuchsbauer, and Alessandra Scafuro. 2016. NIZKs with an Untrusted CRS: Security in the Face of Parameter Subversion. In *Advances in Cryptology - ASIACRYPT, Proceedings, Part II*. 777–804.

[5] Mihir Bellare, Daniele Micciancio, and Bogdan Warinschi. 2003. Foundations of Group Signatures: Formal Definitions, Simplified Requirements, and a Construction Based on General Assumptions. In *Advances in Cryptology - EUROCRYPT, Proceedings*. 614–629.

[6] Mihir Bellare and Todor Ristov. 2014. A Characterization of Chameleon Hash Functions and New, Efficient Designs. *J. Cryptology* 27, 4 (2014), 799–823.

[7] Mihir Bellare, Haixia Shi, and Chong Zhang. 2005. Foundations of Group Signatures: The Case of Dynamic Groups. In *Topics in Cryptology - CT-RSA, Proceedings*. 136–153.

[8] Eli Ben-Sasson, Alessandro Chiesa, Christina Garman, Matthew Green, Ian Miers, Eran Tromer, and Madars Virza. 2014. Zerocash: Decentralized Anonymous Payments from Bitcoin. In *2014 IEEE Symposium on Security and Privacy, SP*. 459–474.

[9] Iddo Bentov, Yan Ji, Fan Zhang, Lorenz Breidenbach, Philip Daian, and Ari Juels. 2019. Tesseract: Real-Time Cryptocurrency Exchange Using Trusted Hardware. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS*. 1521–1538.

[10] Jonathan Bootle, Andrea Cerulli, Pyrros Chaidos, Essam Ghadafi, and Jens Groth. 2016. Foundations of Fully Dynamic Group Signatures. *IACR Cryptol. ePrint Arch.* 2016 (2016), 368. http://eprint.iacr.org/2016/368

[11] Christina Brzuska, Marc Fischlin, Tobias Freudenreich, Anja Lehmann, Marcus Page, Jakob Schelbert, Dominique Schröder, and Florian Volk. 2009. Security of Sanitizable Signatures Revisited. In *Public Key Cryptography - PKC 2009, Proceedings*. 317–336.

[12] ca18 [n. d.]. California Consumer Privacy Act. https://oag.ca.gov/privacy/ccpa.

[13] Jan Camenisch, David Derler, Stephan Krenn, Henrich C. Pöhls, Kai Samelin, and Daniel Slamanig. 2017. Chameleon-Hashes with Ephemeral Trapdoors - And Applications to Invisible Sanitizable Signatures. In *Public-Key Cryptography - PKC, Proceedings, Part II*. 152–182.

[14] Jan Camenisch and Victor Shoup. 2003. Practical Verifiable Encryption and Decryption of Discrete Logarithms. In *Advances in Cryptology - CRYPTO, Proceedings*. 126–144.

[15] charm [n. d.]. Charm: A tool for rapid cryptographic prototyping. http://charm-crypto.io.

[16] Melissa Chase. 2007. Multi-authority Attribute Based Encryption. In *Theory of Cryptography, 4th Theory of Cryptography Conference, TCC, Proceedings*, Salil P. Vadhan (Ed.). 515–534.

[17] Xiaofeng Chen, Fangguo Zhang, Willy Susilo, and Yi Mu. 2007. Efficient Generic On-Line/Off-Line Signatures Without Key Exposure. In *Applied Cryptography and Network Security, 5th International Conference, Proceedings*. 18–30.

[18] Ronald Cramer and Victor Shoup. 1998. A practical public key cryptosystem provably secure against adaptive chosen ciphertext attack. In *Advances in Cryptology—CRYPTO'98: 18th Annual International Cryptology Conference Santa Barbara, California, USA August 23–27, 1998 Proceedings 18*. Springer, 13–25.

[19] Poulami Das, Lisa Eckey, Tommaso Frassetto, David Gens, Kristina Hostáková, Patrick Jauernig, Sebastian Faust, and Ahmad-Reza Sadeghi. 2019. Fastkitten: Practical smart contracts on bitcoin. In *28th {USENIX} Security Symposium ({USENIX} Security 19)*. 801–818.

[20] Sourav Das, Vinay Joseph Ribeiro, and Abhijeet Anand. 2018. Yoda: Enabling computationally intensive contracts on blockchains with byzantine and selfish nodes. *arXiv preprint arXiv:1811.03265* (2018).

[21] David Derler, Kai Samelin, Daniel Slamanig, and Christoph Striecks. 2019. Fine-Grained and Controlled Rewriting in Blockchains: Chameleon-Hashing Gone Attribute-Based. In *26th Annual Network and Distributed System Security Symposium, NDSS*.

[22] dhs [n. d.]. Department of Homeland Security: Blockchain and Suitability for Government Applciations. https://www.dhs.gov/sites/default/files/publications2018_AEP_Blockchain_and_Suitability_for_Government_Applications.pdf.

[23] Marc Fischlin. 2001. *Trapdoor commitment schemes and their applications*. Ph. D. Dissertation. Goethe University Frankfurt, Frankfurt am Main, Germany. http://zaurak.tm.informatik.uni-frankfurt.de/diss/data/src/00000229/00000229.pdf.gz

[24] Eiichiro Fujisaki and Tatsuaki Okamoto. 1999. Secure Integration of Asymmetric and Symmetric Encryption Schemes. In *Advances in Cryptology - CRYPTO, Proceedings*. 537–554.

[25] Jens Groth, Markulf Kohlweiss, Mary Maller, Sarah Meiklejohn, and Ian Miers. 2018. Updatable and Universal Common Reference Strings with Applications to zk-SNARKs. In *Advances in Cryptology - CRYPTO Proceedings, Part III*. 698–728.

[26] Yanxue Jia, Shifeng Sun, Yi Zhang, Zhiqiang Liu, and Dawu Gu. 2021. Redactable Blockchain Supporting Supervision and Self-Management. In *ASIA CCS '21: ACM Asia Conference on Computer and Communications Security, Virtual Event, Hong Kong, June 7-11, 2021*, Jiannong Cao, Man Ho Au, Zhiqiang Lin, and Moti Yung (Eds.). ACM, 844–858.

[27] Sukrit Kalra, Seep Goel, Mohan Dhawan, and Subodh Sharma. 2018. ZEUS: Analyzing Safety of Smart Contracts.. In *Ndss*. 1–12.

[28] Mauricio Karchmer and Avi Wigderson. 1993. On Span Programs. In *Proceedings of the Eigth Annual Structure in Complexity Theory Conference*. 102–111.

[29] Ahmed E. Kosba, Andrew Miller, Elaine Shi, Zikai Wen, and Charalampos Papamanthou. 2016. Hawk: The Blockchain Model of Cryptography and Privacy-Preserving Smart Contracts. In *IEEE Symposium on Security and Privacy, SP*. 839–858.

[30] Hugo Krawczyk and Tal Rabin. 2000. Chameleon Signatures. In *Proceedings of the Network and Distributed System Security Symposium, NDSS*.

[31] Stephan Krenn, Henrich C. Pöhls, Kai Samelin, and Daniel Slamanig. 2018. Chameleon-Hashes with Dual Long-Term Trapdoors and Their Applications. In *Progress in Cryptology - AFRICACRYPT, Proceedings*. 11–32.

[32] Russell W. F. Lai, Tao Zhang, Sherman S. M. Chow, and Dominique Schröder. 2016. Efficient Sanitizable Signatures Without Random Oracles. In *Computer Security - ESORICS, Proceedings, Part I*, Ioannis G. Askoxylakis, Sotiris Ioannidis, Sokratis K. Katsikas, and Catherine A. Meadows (Eds.). 363–380.

[33] Allison B. Lewko and Brent Waters. 2011. Decentralizing Attribute-Based Encryption. In *Advances in Cryptology - EUROCRYPT, Proceedings*, Kenneth G. Paterson (Ed.). 568–588.
[34] Benoît Libert, Thomas Peters, and Moti Yung. 2012. Scalable Group Signatures with Revocation. In *Advances in Cryptology - EUROCRYPT, Proceedings*. 609–627.
[35] Jinhua Ma, Shengmin Xu, Jianting Ning, Xinyi Huang, and Robert H. Deng. 2022. Redactable Blockchain in Decentralized Setting. *IEEE Transactions on Information Forensics and Security* 17 (2022), 1227–1242. https://doi.org/10.1109/TIFS.2022.3156808
[36] Medium. [n. d.]. Hundreds of Millions of Dollars Locked at Ethereum 0x0 Address and Smart Contracts' Addresses. https://medium.com/@maltabba/hundreds-of-millions-of-dollars-locked-at-ethereum-0x0-address-and-smart-contracts-addresses-how-4144dbe3458a.
[37] Gaurav Panwar, Roopa Vishwanathan, and Satyajayant Misra. 2021. ReTRACe: Revocable and Traceable Blockchain Rewrites using Attribute-based Cryptosystems. In *SACMAT '21: The 26th ACM Symposium on Access Control Models and Technologies, Virtual Event, Spain, June 16-18, 2021*, Jorge Lobo, Roberto Di Pietro, Omar Chowdhury, and Hongxin Hu (Eds.). ACM, 103–114.
[38] Gaurav Panwar, Roopa Vishwanathan, and Satyajayant Misra. 2021. ReTRACe: Revocable and Traceable Blockchain Rewrites using Attribute-based Cryptosystems. *IACR Cryptol. ePrint Arch.* (2021), 568. https://eprint.iacr.org/2021/568
[39] Ania M. Piotrowska, Jamie Hayes, Tariq Elahi, Sebastian Meiser, and George Danezis. 2017. The Loopix Anonymity System. In *26th USENIX Security Symposium, USENIX Security*. 1199–1216.
[40] r3 [n. d.]. Banks complete 25 million euros securities transaction on blockchain platform. https://uk.reuters.com/article/uk-blockchain-securities/banks-complete-25-million-euros-securities-\transaction-on-blockchain-platform-\idUKKCN1GD4DW.
[41] Kai Samelin and Daniel Slamanig. 2020. Policy-Based Sanitizable Signatures. In *Topics in Cryptology - CT-RSA 2020, Proceedings*. 538–563.
[42] Wei Shao, Jinpeng Wang, Lianhai Wang, Chunfu Jia, Shujiang Xu, and Shuhui Zhang. 2023. Auditable Blockchain Rewriting in Permissioned Setting with Mandatory Revocability for IoT. *IEEE Internet of Things Journal* (2023).
[43] Stephen R. Tate and Roopa Vishwanathan. 2009. Improving Cut-and-Choose in Verifiable Encryption and Fair Exchange Protocols Using Trusted Computing Technology. In *Data and Applications Security XXIII, 23rd Annual IFIP WG 11.3 Working Conference, Montreal, Canada, July 12-15, 2009. Proceedings (Lecture Notes in Computer Science, Vol. 5645)*, Ehud Gudes and Jaideep Vaidya (Eds.). Springer, 252–267.
[44] Sri Aravinda Krishnan Thyagarajan, Adithya Bhat, Bernardo Magri, Daniel Tschudi, and Aniket Kate. 2020. Reparo: Publicly Verifiable Layer to Repair Blockchains. *CoRR* abs/2001.00486 (2020). http://arxiv.org/abs/2001.00486
[45] Yangguang Tian, Nan Li, Yingjiu Li, Pawel Szalachowski, and Jianying Zhou. 2020. Policy-based Chameleon Hash for Blockchain Rewriting with Black-box Accountability. In *ACSAC '20: Annual Computer Security Applications Conference, Virtual Event / Austin, TX, USA, 7-11 December, 2020*. ACM, 813–828.
[46] Yangguang Tian, Atsuko Miyaji, Koki Matsubara, Hui Cui, and Nan Li. 2022. Revocable policy-based chameleon hash for blockchain rewriting. *Comput. J.* (2 July 2022).
[47] Guangbo Wang and Jianhua Wang. 2017. Research on Ciphertext-Policy Attribute-Based Encryption with Attribute Level User Revocation in Cloud Storage. *Mathematical Problems in Engineering* 2017, 1 (2017), 4070616.
[48] Business Wire. 2016. Accenture Editable Blockchain. https://www.businesswire.com/news/home/20160920005551/en/Accenture-Debuts-Prototype-of-%E2%80%98Editable%E2%80%99-Blockchain-for-Enterprise-and-Permissioned-Systems.
[49] Shengmin Xu, Xinyi Huang, Jiaming Yuan, Yingjiu Li, and Robert H. Deng. 2023. Accountable and Fine-Grained Controllable Rewriting in Blockchains. *IEEE Trans. Inf. Forensics Secur.* 18 (2023), 101–116.
[50] Shengmin Xu, Jianting Ning, Jinhua Ma, Guowen Xu, Jiaming Yuan, and Robert H. Deng. 2021. Revocable Policy-Based Chameleon Hash. In *Computer Security - ESORICS 2021 - 26th European Symposium on Research in Computer Security, Darmstadt, Germany, October 4-8, 2021, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 12972)*, Elisa Bertino, Haya Shulman, and Michael Waidner (Eds.). Springer, 327–347.
[51] Zhiqian Xu and Keith M Martin. 2012. Dynamic user revocation and key refreshing for attribute-based encryption in cloud storage. In *2012 IEEE 11th International Conference on Trust, Security and Privacy in Computing and Communications*. IEEE, 844–849.
[52] Rui Zhang. 2007. Tweaking TBE/IBE to PKE Transforms with Chameleon Hash Functions. In *Applied Cryptography and Network Security, 5th International Conference, ACNS 2007, Proceedings*, Jonathan Katz and Moti Yung (Eds.). 323–339.
[53] Yuqing Zhang, Zhaofeng Ma, Shoushan Luo, and Pengfei Duan. 2023. Dynamic Trust-Based Redactable Blockchain Supporting Update and Traceability. *IEEE Transactions on Information Forensics and Security* (2023).
[54] ZhiShuo Zhang, Wei Zhang, and ZhiGuang Qin. 2020. Multi-authority CP-ABE with dynamical revocation in space-air-ground integrated network. In *2020 International Conference on Space-Air-Ground Computing (SAGC)*. IEEE, 76–81.