ReTRACe: Revocable and Traceable Blockchain Rewrites using Attribute-based Cryptosystems

Gaurav Panwar[*]
New Mexico State University
Las Cruces, NM, USA
gpanwar@nmsu.edu

Roopa Vishwanathan[*]
New Mexico State University
Las Cruces, NM, USA
roopav@nmsu.edu

Satyajayant Misra
New Mexico State University
Las Cruces, NM, USA
misra@cs.nmsu.edu

ABSTRACT

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 being coarse-grained, inability to expunge data, absence of revocation mechanisms, lack of user anonymity, and inefficiency. We present ReTRACe, an efficient framework for transaction-level blockchain rewrites, that is fine-grained and supports revocation. ReTRACe is designed by composing a novel revocable chameleon hash with ephemeral trapdoor scheme, a novel revocable fast attribute based encryption scheme, and a dynamic group signature scheme. We discuss ReTRACe, and its constituent primitives in detail, along with their security analyses, and present experimental results to demonstrate scalability.

CCS CONCEPTS

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

KEYWORDS

Blockchain rewrite; chameleon hash; anonymity; attribute-based encryption

ACM Reference Format:
Gaurav Panwar, Roopa Vishwanathan, and Satyajayant Misra. 2021. ReTRACe: Revocable and Traceable Blockchain Rewrites using Attribute-based Cryptosystems. In *Proceedings of the 26th ACM Symposium on Access Control Models and Technologies (SACMAT '21), June 16–18, 2021, Virtual Event, Spain.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3450569.3463565

1 INTRODUCTION AND RELATED WORK

In access control techniques, revocation is a problem that is easily motivated from a practical standpoint. We present ReTRACe, a system for performing access control including revocation for transaction rewrites in blockchains. A blockchain is an append-only ledger to which entities can post messages in a decentralized manner. A message could be a financial transaction, a smart contract, or any data that needs to be shared among several users, but whose provenance needs to be verified. At a high level, a *block* is a collection of multiple messages (also called transactions), and their hash digests. Usually, once a message has been written to the blockchain, the message is considered immutable and cannot be edited.

While blockchain edits or rewrites are not required in all applications, there is an important class of applications where editing messages written onto a blockchain is essential. For instance, in the European Union (EU), the general data protection regulation (GDPR), Chapter 2, Article 17: *Right to erasure* and Article 19: *Notification obligation regarding rectification or erasure of personal data or restriction of processing*, gives users or *data subjects* the right to request that their personal data be erased/edited by the person or entity collecting or publishing their data, per their request. The U.S. state of California passed the California Consumer Privacy Act (CCPA) in 2018 [19], which has codified similar privacy rights, as described in Article 2. 999.308 (c) (2).

Blockchain technology has widespread applications in healthcare, regulatory compliance and audit, record management, Internet of Things, and more [21]. It is easy to envisage examples where a user's sensitive data is part of the committed blockchain transactions, and at a later point needs to be erased. For example, a global consortium of banks is currently using a blockchain platform (R3 [28]) to manage financial agreements, securities trading, etc. These transaction records will include clients' information and could potentially contain personally identifiable information.

As per the individual's "right to forget" (e.g., in GDPR), a user can request a purge of their identification data from the blockchain, (true even with encrypted data), and should be able to independently verify the said purge. Also, when a blockchain is used for record keeping and auditing the actions of a set of mutually untrusting parties, there may be situations where a non-monetary record needs to be expunged from the blockchain, e.g., offensive content, leaked personal information/encryption keys, etc., and companies have prototyped editable blockchains for addressing this [33]. The U.S. Department of Homeland Security in a recent report on the use of blockchains in government, has judged permissioned blockchains to be useful for maintaining government records, supply chain monitoring, and government approval chain processes [17], which will encourage blockchain adoption. In all such applications, there might arise need for correcting/updating transactions. *Motivated by this, we concentrate on a permissioned blockchain.*

Recently, some solutions have been proposed to enable modifications on data posted to a blockchain [15]. One method is a *hard fork*,

[*]Authors contributed equally.

SACMAT '21, June 16–18, 2021, Virtual Event, Spain
© 2021 Association for Computing Machinery.
ACM ISBN 978-1-4503-8365-3/21/06…$15.00
https://doi.org/10.1145/3450569.3463565

which involves diverging the blockchain at the point where a message needs to be changed, creating a new forked blockchain, and invalidating all subsequent blocks in the old blockchain. Another technique is to modify a message m to m', post m' pointing to m on the blockchain. In addition to being inefficient, these techniques do not expunge the old message from the chain, and do not enable fine-grained control over who can modify messages.

Ateniese *et al.* [2, 33] proposed a solution for blockchain edits by rewriting entire blocks, using *chameleon hash functions*. Deuber *et al.* [16] and Reparo [30] proposed mechanisms for block-level rewrites where any user can propose a redaction or an edit of a block, which is then voted upon by other users, and the edit is accepted if it wins a majority vote. Coarse block-level solutions result in needlessly rewriting an entire block, when only say, a single transaction in a block needs to be expunged, nor do they provide fine-grained transaction level control by helping us set permissions about who can rewrite individual transactions. In this paper, we focus on *transaction-level rewriting*. Our intent is to build a system where a transaction can be updated, if needed, at which point only the updated transaction will be visible on the blockchain.

1.1 Related Work

Recently, chameleon hash functions have been proposed to enable blockchain rewrites. Chameleon hash functions [23] enable a user to find collisions in the domain of a hash function, such that several pre-images can be created that map to the same hash digest. The process of finding a collision on a given message, termed *adapting* the message, is done using a *trapdoor* associated with the digest. For increased security and flexibility, Camenisch *et al.* [10] proposed the idea of two trapdoors being associated with a digest, a permanent *long-term trapdoor*, and an *ephemeral trapdoor*, which is chosen per message; a message cannot be adapted without knowing *both* trapdoors. Derler *et al.* [15] presented an application of [10] to transaction-level blockchain rewrites, where the ephemeral trapdoor associated with a digest of a message posted on the blockchain is encrypted using ciphertext-policy attribute-based encryption (CPABE), and only users possessing enough attributes that satisfy the policy can decrypt the ephemeral trapdoor and adapt a message.

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. Ideally, we would like to revoke users, such that upon revocation of a user u who possesses the ephemeral trapdoor, the ephemeral trapdoor is swiftly updated to a value unknown to u. We use chameleon hashes, attribute-based encryption (ABE) and a dynamic group signature scheme (GSS) to achieve our goal.

1.2 Challenges in the State-of-the-art

Motivated by the problem of performing transaction rewrites on a blockchain, we seek to answer the following questions: Who gets to erase or overwrite transactions/messages? What if a user is allowed to update a given message only for a short period of time? Once a message has been modified, should the identity of the modifier be revealed, and if yes, to whom? These questions need to be addressed to make editable transactions usable, which is our goal in this paper. We focus on three important challenges: 1) *Revoking access to ephemeral trapdoors*: a revoked user cannot update a message even if she has a local copy of the long-term and ephemeral trapdoors. 2) *Revoking attributes from users*: a user who no longer has access to an attribute secret key cannot update a message and/or trapdoor. 3) *Traceability*: a user who anonymously rewrote a particular message, but violated the rewriting policy can be identified. For addressing the first two challenges, we need to design new cryptographic primitives, since existing ones do not provide the properties we require.

1.3 Our Contributions

Our contributions in this paper include:
1) Design of a new *revocable chameleon hash with ephemeral trapdoor scheme*, RCHET, which guarantees that a revoked user cannot adapt a blockchain message or trapdoor.
2) Design of a new, efficient *revocable ABE scheme*, RFAME, to control access to the ephemeral trapdoor.
3) Design of a *revocable and traceable blockchain rewriting framework*, ReTRACe, using our novel RCHET and RFAME schemes, 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, but their identities can be unmasked by legitimate oversight authorities if needed.
4) Implementation of RCHET, RFAME, and ReTRACe to demonstrate scalability, and their security analyses.

We note that ReTRACe does not modify the way miners communicate with one another and how the blocks are mined in a permissioned blockchain adapted with ReTRACe's functionality (except transaction verification). Both the ReTRACe messages and non-ReTRACe messages can co-exist in the underlying permissioned blockchain, i.e., after integrating ReTRACe into a permissioned blockchain, the resultant 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 the constituents of our system model, and cover preliminaries and assumptions. In Section 3, we give a short technical overview of ReTRACe. In Sections 4 and 5, we introduce RCHET and RFAME, their definitions, security analyses, and constructions. In Section 6 we briefly discuss dynamic GSS schemes, and in Section 7, we give the definition, security analysis, and construction of ReTRACe. In Section 8 we present our experimental analysis, and Section 9 concludes the paper.
Since ReTRACe has several building blocks, we need to make careful choices on what is included in the main paper and our full version [26]. We present our new ideas, constructions, and experiments in the main paper, and include formal definitions and proofs of our constructions in the full version.

2 SYSTEM MODEL AND THREAT MODEL

ReTRACe is constructed using three primitives: 1) a revocable chameleon hash scheme, RCHET, which provides a dynamic and mutable ephemeral trapdoor required for updating a message posted on the BC, 2) a revocable attribute-based encryption scheme, RFAME 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. In this section, we discuss the parties involved in ReTRACe, and the policy structures that control their ability to update messages.

Naming Conventions: In our discussions, we refer to the blockchain as BC, we use *message* to mean a pre-image of the chameleon hash function, which is posted on the BC as a transaction, and when we say "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.

2.1 Policies

ReTRACe has four kinds of policies, all represented as Boolean predicates: 1) A trapdoor associated with the digest of a given message is encrypted under an ABE policy, Υ_{ABE}. 2) The policy Υ_{GS}, spells out certain DGSS groups from which users can anonymously sign a message update, and post it on the BC. 3) For revoking access to the trapdoor, we define a policy, $\Upsilon_{ABE\text{admin}}$, that sets forth who can create an updated ephemeral trapdoor, and encrypt it under a new policy, Υ'_{ABE}. 4) Finally, $\Upsilon_{GS\text{admin}}$ governs who can change Υ_{GS}, and thus exclude users of certain DGSS groups from producing valid signatures.

For simplicity of exposition, we assume that the $\Upsilon_{ABE\text{admin}}$ and $\Upsilon_{GS\text{admin}}$ policies, once set, 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 immutable; our system allows for the *set of people* satisfying them to be dynamic and ever-changing.

Unlike ABE schemes, DGSS do not have the concept of policies built into them; we introduce this notion for ReTRACe. We define a DGSS policy as a publicly-known Boolean predicate linked to a message m that specifies which groups can produce valid signatures on m, e.g., (m, Υ_{GS} = "Admin" AND "Payroll"), indicates users belonging to 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, 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.

2.2 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, Υ_{ABE}, Υ_{GS}, $\Upsilon_{ABE\text{admin}}$, and $\Upsilon_{GS\text{admin}}$ that regulate future message updates.

2) Set of Authorized Users, AuthU: The set of users who can create a collision on a message posted on the BC by the originator (**AuthU** could include the originator) as long as they possess enough attributes that satisfy 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 can modify the trapdoor, as well as create a collision on 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 group satisfying $\Upsilon_{GS\text{admin}}$.

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. For presentation clarity, we use a single AIA and GM, but in practice, more can be used easily in ReTRACe if design warrants.

2.3 Blockchain Operations

A ReTRACe transaction consists of a mutable part (plaintext message, policies, trapdoor, ciphertexts, signatures, etc.) and an immutable part (digest). 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 of the given block instead of the hash of the whole transaction. This makes it possible to update a ReTRACe transaction in the future as long as the mutable part of the updated transaction verifies with the immutable digest on the BC. ReTRACe is independent of the kind of underlying BC system including the consensus mechanism, e.g., Proof of Work/Stake, as long as the security requirements for ReTRACe defined in Section 2.4 are met. For a user to be able to post any ReTRACe messages in the BC, the user needs to be previously onboarded with a AIA and GM in the given ReTRACe system.

To use ReTRACe with a given BC system, the hash verification function of the BC needs to be modified to perform the ReTRACe messages' verification (miner verification, and public verification of the ReTRACe messages). But ReTRACe does not leak sensitive information to the miners hence the miners are not onboarded by the AIA and GM. Adding or removing miners in the system using ReTRACe works in the same as any other BC system; the miners have a few extra steps when verifying updates to ReTRACe transactions, but none of those steps involve the miners learning privileged information about the messages posted on the BC.

Each block on a ReTRACe-modified BC could have mutable as well as immutable transactions and the resultant hash of the block will always be immutable as per the rules of traditional BCs. The verification algorithm can verify mutable as well as immutable transactions, thus ReTRACe can be implemented on a blockchain which supports both, mutable and immutable transactions. Note that once a transaction is written to the BC as immutable, it cannot be modified, only transactions which were originally posted as ReTRACe transactions with a trapdoor and access policy are updatable. The AIA and GM are publicly available to all users to sign up with, if any user needs to post mutable transactions on the BC, user do not need to sign up with the AIA/GM if they do not want to post immutable transactions.

2.4 Trust Assumptions and Threat Model

We make a few modest trust assumptions in ReTRACe. As in most BC-enabled systems, we assume the consensus protocol being used ensures honest operation by miners. We assume the AIA will honestly generate attributes and secret keys; one could relax this assumption by using multi-AIA techniques of Chase [11], where components of users' secret keys are generated by multiple AIAs. Similarly for the DGSS scheme, we assume that the GM for a group will issue signing keys properly, and trace users honestly. We could dilute this assumption by using techniques of Bootle *et al.* [9], by introducing a *tracing manager*, and separating the roles of group and tracing managers. These relaxations could be applied on as as-needed basis, we do not discuss them for narrative simplicity.

Threat Model and Security Goals: Our main goal is to protect against adversaries that have no access to the long-term trapdoor and/or current version of the ephemeral trapdoor for a given message, m, posted on the BC, do not satisfy the policies associated with m, yet who try to update m or its trapdoors. A second goal is to protect the privacy of, and provide anonymity to, the individuals who post a message or update a message and/or its trapdoor on the BC. The only way for an adversary to violate our goals is to break the security of our cryptographic constructs. We do not consider network attacks (e.g., eclipse attacks, traffic analysis, etc.) in this paper, prior works [6] that focus on them can be used in conjunction with ReTRACe.

2.5 Computational Assumptions

The security of ReTRACe is derived from time-tested, well-regarded assumptions based on the Discrete Log problem, the Decision Linear problem (DLIN), and the Decisional Diffie-Hellman (DDH) problem, defined in [26]. We model 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, as is common in the literature.

3 ReTRACe TECHNICAL 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 $\mathsf{mpk}_{\mathsf{ABE}}$ and gpk denote the public keys of the ABE and DGSS schemes 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), creates a policy, Υ_{ABE}, which defines a set of authorized users, **AuthU** that are allowed to modify m. User u encrypts τ with $\mathsf{mpk}_{\mathsf{ABE}}$ (using our novel revocable ABE scheme, RFAME), $E_{\mathsf{mpk}_{\mathsf{ABE}}}(\tau, \Upsilon_{\mathsf{ABE}}) \rightarrow X$. 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 creates the $\Upsilon_{ABE\mathrm{admin}}$ policy that defines the set of authorized user administrator(s), **AuthUAdmins** who are allowed to update τ, and computes $E_{\mathsf{mpk}_{\mathsf{ABE}}}(r, \Upsilon_{ABE\mathrm{admin}}) \rightarrow X_r$. A user in **AuthU** updates m during a message adaptation, a user in **AuthUAdmins** updates m, X and X_r during a trapdoor update.

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 satisfies Υ_{GS}, and post m' and the signature on the BC.

The digest 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 X, and only members of **AuthUAdmins** have permission to update m, X and X_r. The miner's in ReTRACe do not need any extra or privileged information other than what is already available as part of the public parameters of ReTRACe, hence they *do not* need to sign up 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, in which case Υ_{ABE} is unchanged. 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 $\Upsilon_{ABE\mathrm{admin}}$, 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 $E_{\mathsf{mpk}_{\mathsf{ABE}}}(\tau', \cdot) \rightarrow X'$, j will prove knowledge of r to the miner thus proving it can satisfy $\Upsilon_{ABE\mathrm{admin}}$, and is a member of **AuthUAdmins**. User j then picks $r' \leftarrow\!\!\$\, \mathbb{Z}_q$, computes $E_{\mathsf{mpk}_{\mathsf{ABE}}}(r', \cdot) \rightarrow X'_r$, to replace r. Then j will sign and post m', X' and X'_r on the BC. The DGSS signature will be produced using j's signing keys, sk_j, that satisfy $\Upsilon_{GS\mathrm{admin}}$.

We discuss the details and subtleties of ReTRACe in Section 7; in what follows, we describe the building primitives.

4 REVOCABLE CHAMELEON HASH WITH EPHEMERAL TRAPDOORS

We envisioned 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.

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.

4.1 RCHET Security Properties

The properties of indistinguishability, public and private collision resistance were introduced by Camenisch *et al.* [10] for CHET schemes. Derler *et al.* [15] 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

<u>RCHET Algorithms</u>

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

1. Check $\mathsf{verify}(\pi_{pk}, h) \overset{?}{=} 1$, 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, $a \leftarrow H_k(m)$ 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 $\mathsf{digest} = (b, h', \pi_t, C, C')$, $\mathsf{rand} = (\beta, p, \pi_p)$, $\Gamma_{\mathsf{pubinfo}} = (\Delta, \pi_\Delta, D, \pi_D)$, $\Gamma_{\mathsf{privinfo}} = (\delta, d, \mathsf{etd})$.

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

1. Check $\mathsf{verify}(\pi_{pk}, h) \overset{?}{=} 1$, $\mathsf{verify}(\pi_p, p) \overset{?}{=} 1$, $\mathsf{verify}(\pi_t, h') \overset{?}{=} 1$, $\mathsf{verify}(\pi_D, D) \overset{?}{=} 1$, and $\mathsf{verify}(\pi_\Delta, \Delta) \overset{?}{=} 1$.

2. Check $b \overset{?}{=} \frac{h^\beta \cdot h'^a}{D \cdot \Delta}$, where $a \leftarrow H_k(m)$. If check passes, return 1, else return 0.

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

1. Decrypt $a \leftarrow \Pi.\mathsf{Decrypt}(SK, C')$, check $a \overset{?}{\leftarrow} H_k(m)$. Check $b \overset{?}{=} \frac{h^\beta \cdot h'^a}{D \cdot \Delta}$. If checks fail, return $\perp$.

2. Check $h \overset{?}{=} g^x, p \overset{?}{=} g^{xr}, h' \overset{?}{=} g^{etd}, D \overset{?}{=} g^d$, and $\Delta \overset{?}{=} g^\delta$. 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 $\mathsf{rand}' = (\beta', p', \pi_{p'})$.

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

1. Decrypt $a \leftarrow \Pi.\mathsf{Decrypt}(SK, C')$, check $a \overset{?}{\leftarrow} H_k(m)$. Check $b \overset{?}{=} \frac{h^\beta \cdot h'^a}{D \cdot \Delta}$. If checks fail, return $\perp$.

2. Check $h \overset{?}{=} g^x, p \overset{?}{=} g^{xr}, h' \overset{?}{=} g^{etd}, D \overset{?}{=} g^d$, and $\Delta \overset{?}{=} g^\delta$. If checks fail, return $\perp$.

3. Decrypt $r \leftarrow \Pi.\mathsf{Decrypt}(SK, C)$, 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 $\pi_{D'} \leftarrow \mathsf{NIZKPoK}\{d' : D' = g^{d'}\}$, $\pi_{\Delta'} \leftarrow \mathsf{NIZKPoK}\{\delta' : \Delta' = g^{\delta'}\}$.

5. Set $r' \leftarrow (\frac{rx + a \cdot \mathsf{etd} - a' \cdot \mathsf{etd} + \delta'}{x}), p' \leftarrow h^{r'}$. Compute $\beta' \leftarrow (r' + \frac{d'}{x})$, $\pi_{p'} \leftarrow \mathsf{NIZKPoK}\{r' : p' = h^{r'}\}$.

6. Prove knowledge of DDH tuple $(g, g^\delta, g^d, g^{\delta d})$. Set and output $\mathsf{rand}' = (\beta', p', \pi_{p'})$, $\Gamma'_{\mathsf{pubinfo}} = (\Delta', \pi_{\Delta'}, D', \pi_{D'})$, and $\Gamma'_{\mathsf{privinfo}} = (\delta', d', \mathsf{etd})$.

Figure 1: Construction of Revocable Chameleon Hash with Ephemeral Trapdoors (RCHET)

game, while [10] 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 access to oracles 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 ephemeral trapdoor, cannot find collisions by himself. *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, and give the proof of the following theorem in [26].

THEOREM 4.1. *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 REVOCABLE ATTRIBUTE-BASED ENCRYPTION

In this section, we describe our revocable ciphertext policy attribute-based encryption (CPABE) scheme, RFAME, and discuss its efficiency and security properties. Most of the benchmark schemes in the ABE literature do not consider attribute revocation [20, 31, 32]. FAME [1] is a state-of-the-art efficient ABE scheme that performs better in terms of qualitative and quantitative metrics, compared to prior works, and provides full IND-CPA security, although it does not discuss revocation. We use FAME as a starting point for designing an efficient *revocable CPABE scheme*, RFAME.

Revocation model: In ABE, there are broadly speaking, two possible kinds of revocation. One is *policy-level revocation*, where revocation entails deleting a clause from an ABE policy, e.g., Υ = "CS students" and "EE staff" can be updated to a more restrictive policy, Υ' = "CS students". The other is the more fine-grained *user-level revocation* which calls for revoking decryption rights of individual users, e.g., if we do not wish to update Υ, but revoke access of a member of staff from EE. We consider user-level revocation in this paper (although our scheme also supports modifiable policies, if needed).

Most of the benchmark ABE schemes proposed in the literature [1, 20, 24, 32] do not support revocation. The ones that do are either based on non-standard assumptions [7, 13, 27, 29], offer only selective security [3, 8, 34], are application-specific [13, 34], do only policy-level revocation [7, 29], do not support Type III pairings [3, 7, 8, 13, 14, 27, 29, 34], need revocation lists which do not scale well [3, 8, 13, 14, 29, 34], or perform inefficient revocation [12]. We design an efficient revocable ABE scheme, RFAME, to revoke access to ephemeral trapdoors that provides our desired properties.

5.1 Construction

We give our construction of RFAME in Figure 2, whose security is based on the DLIN assumption. We walk the reader through a few initial steps of the decryption algorithm that will help in verifying correctness in [26]. We now describe our intuitive ideas behind RFAME and its efficiency.

Efficient Rekeying in RFAME: Let us consider an AIA of an organization that issues four kinds of attributes, "Admin", "Payroll", "Benefits", and "Accounts". Let us assume there are y unique users possessing each attribute–a total of $4y$ users in the system, and that there are three messages in the system encrypted under different policies: *i)* Msg_1 is encrypted under $\Upsilon_{\mathsf{Msg}_1}$ = ("Admin" OR "Payroll"); *ii)* Msg_2 is encrypted under $\Upsilon_{\mathsf{Msg}_2}$ = ("Payroll" OR "Benefits"); *iii)* Msg_3 is encrypted under $\Upsilon_{\mathsf{Msg}_3}$ = ("Benefits" OR "Accounts").

Using current revocable CPABE schemes (e.g., [12]), if one user possessing the *Admin* attribute terminates employment, their secret key is revoked by the AIA, and the other $y - 1$ *Admin* users get rekeyed. Msg_1 needs to be re-encrypted to prevent the revoked user from decrypting it. Since *Payroll* is part of the $\Upsilon_{\mathsf{Msg}_1}$ policy, all y users holding *Payroll* get rekeyed, as Msg_1 got re-encrypted. *Payroll* users getting rekeyed results in Msg_2 needing to be re-encrypted. Consequently, users holding *Benefits* and *Accounts* attributes need to be rekeyed, and Msg_3 needs to get re-encrypted. In total, we need to perform three re-encryptions and rekey $4y - 1$ users, for a *single* user revocation. Our goal is to avoid such a domino effect.

At a high level, our idea is to associate each attribute, attr with some unique randomness, r, and embed r into the secret keys of all users who possess attr. When a user possessing attr needs to be revoked, we update the randomness to r' and reissue new secret keys with r' embedded in them only to the non-revoked users holding attr, and re-encrypt all ciphertexts whose policies involve attr. For facilitating this, the AIA can maintain a compact local table identifying which users possess a given attribute—a small storage cost in exchange for avoiding system-wide rekeying of users.

Furthermore, non-revoked users possessing attributes other than attr can still use their *current keys* to decrypt re-encrypted ciphertexts, which significantly reduces the number of users that need to be rekeyed, and the ciphertexts that need to be re-encrypted. (note that in [29], the authors propose a scheme that does not require re-encryptions, for *policy-level* revocation, with the restriction that a ciphertext can only be re-encrypted to a *more restrictive* policy. Our work is significantly more flexible, in that we perform *user-level* revocation, and do not impose any restrictions on policies).

Thus, RFAME handles revocation more efficiently; when a user possessing *Admin* gets revoked, only the other $y - 1$ users in *Admin* are rekeyed, and only Msg_1 needs to be re-encrypted to prevent the revoked user from decrypting it. The price we pay for this is that RFAME is a small-universe revocable CPABE scheme.

In summary, in the worst case, if x is the number of unique attributes in a system, y the number of users per attribute, then the overhead, using state-of-the-art revocation methods is $O(x)$ re-encryptions and $O(xy)$ rekeyings. In RFAME, the overhead is $\Theta(1)$ re-encryptions and $\Theta(y)$ rekeyings. We first prove RFAME CPA-secure, we later turn this into a CCA-secure scheme for ReTRACe. We give the IND-CPA game for RFAME and the proof of the following theorem in [26].

Theorem 5.1. RFAME *is fully IND-CPA secure under the DLIN assumption on Type III pairings in the random oracle model.*

6 DYNAMIC GROUP SIGNATURE SCHEMES

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 [4], semi-dynamic groups support addition but not revocation [5], and dynamic groups allow addition and revocation [9]. We use a dynamic group signature scheme (DGSS) in ReTRACe. We do not construct a DGSS, as existing constructions [9, 25] provide the properties we need. ReTRACe is independent of any specific construction.

7 THE ReTRACe FRAMEWORK

We now give the detailed construction of the ReTRACe framework comprising of eight algorithms in Figure 3, Figure 4, Figure 6, and Figure 5. In the algorithms, $\mathbf{M}$ denotes the monotone span program representing an ABE policy, and ρ represents a mapping function that maps rows of $\mathbf{M}$ onto attributes. We use BC.write to denote a blockchain write operation. The Keygen, UserSetup, Sign, Verify,

RFAME Algorithms

a) RFAME.SetupABE$(1^\lambda, U) \rightarrow (\text{mpk}_{\text{ABE}}, \text{msk}_{\text{ABE}})$: The algorithm first generates the group parameters $(q, \mathbb{G}, \mathbb{H}, \mathbb{G}_T, e, g, h)$, picks $a_1, a_2, b_1, b_2, p_1, p_2 \leftarrow_{\$} \mathbb{Z}_q^*$, $d_1, d_2, d_3 \leftarrow_{\$} \mathbb{Z}_q$. It picks $\alpha_y \leftarrow_{\$} \mathbb{Z}_q^*$, and computes h^{α_y} for each $y \in U$. It sets $\text{mpk}_{\text{ABE}} = (h, H_1 = h^{a_1}, H_2 = h^{a_2}, T_1 = e(g,h)^{p_1 d_1 a_1 + d_3}, T_2 = e(g,h)^{p_2 d_2 a_2 + d_3}, h^{\alpha_{y_1}}, \ldots, h^{\alpha_{y_{|U|}}})$, and sets $\text{msk}_{\text{ABE}} = (g, h, a_1, a_2, b_1, b_2, p_1, p_2, g^{d_1}, g^{d_2}, g^{d_3}, \alpha_{y_1} \ldots \alpha_{y_{|U|}})$.

b) RFAME.KeyGenABE$(\text{mpk}_{\text{ABE}}, \text{msk}_{\text{ABE}}, y_1, \ldots, y_{|U|}) \rightarrow SK$: The algorithm generates the secret keys for all attributes $y \in U$. Pick $r_1, r_2 \leftarrow_{\$} \mathbb{Z}_q$. Compute $\text{sk}_0 = (h^{b_1 r_1}, h^{b_2 r_2}, h^{r_1 + r_2})$.
For all $y \in U$ and $t \in \{1, 2\}$, pick $\sigma_y, \sigma' \leftarrow_{\$} \mathbb{Z}_q$, and compute:

$$\text{sk}_{y,t} = \mathcal{H}(y1t)^{\frac{b_1 r_1}{a_t + \alpha_y}} \cdot \mathcal{H}(y2t)^{\frac{b_2 r_2}{a_t + \alpha_y}} \cdot \mathcal{H}(y3t)^{\frac{r_1 + r_2}{a_t + \alpha_y}} \cdot g^{\frac{\sigma_y}{a_t + \alpha_y}} \cdot g^{\frac{\alpha_y}{a_t + \alpha_y}}; \text{sk}_{y,3} = (g^{-\alpha_y} \cdot g^{-\sigma_y})$$

$$\text{sk}'_t = \mathcal{H}(011t)^{\frac{b_1 r_1}{a_t}} \cdot \mathcal{H}(012t)^{\frac{b_2 r_2}{a_t}} \cdot \mathcal{H}(013t)^{\frac{r_1 + r_2}{a_t}} \cdot g^{\frac{\sigma'}{a_t}}; \text{sk}'_3 = (g^{d_3} \cdot g^{-\sigma'}), \text{sk}'' = g^{d_t p_t}$$

$$\text{Set and return } SK = (\text{sk}_0, \text{sk}_{y,1}, \text{sk}_{y,2}, \text{sk}_{y,3}, \text{sk}'_1, \text{sk}'_2, \text{sk}'_3, \text{sk}'')$$

c) RFAME.Encrypt$(\text{mpk}_{\text{ABE}}, msg, (\mathbf{M}, \rho)) \rightarrow C$. Pick $s_1, s_2 \leftarrow_{\$} \mathbb{Z}_q$. Let $\rho(i)$ denote a mapping to the attributes $i \in I$ that satisfy a given policy. Compute:

$$\text{ct}_{\rho(i),1} = H_1^{s_1} \cdot (h^{\alpha_{\rho(i)}})^{s_1} = h^{s_1(a_1 + \alpha_{\rho(i)})} \text{ and similarly } \text{ct}_{\rho(i),2} = h^{s_2(a_2 + \alpha_{\rho(i)})}, \text{ and set}$$

$$\text{ct}_0 = (\text{ct}_{0,1} = H_1^{s_1}, \text{ct}_{0,2} = H_2^{s_2}, \text{ct}_{\rho(i),1}, \text{ct}_{\rho(i),2}, \text{ct}_{0,3} = h^{s_1 + s_2})$$

Assume $\mathbf{M}$ has n_1 rows and n_2 columns. Then, for each row, $i \in [1..n_1]$ and $l = 1, 2, 3$, compute:

$$\text{ct}_{i,l} = \mathcal{H}(\rho(i)l1)^{s_1} \cdot \mathcal{H}(\rho(i)l2)^{s_2} \cdot \prod_{j=1}^{n_2} [\mathcal{H}(0jl1)^{s_1} \cdot \mathcal{H}(0jl2)^{s_2}]^{(\mathbf{M})_{i,j}}; \text{Set } \text{ct}' = (T_1^{s_1} \cdot T_2^{s_2} \cdot msg)$$

$$\text{Set and output } C = (\text{ct}_0, \text{ct}_{i,l} \,\forall\, i \in [1..n_1], l \in \{1, 2, 3\}, \text{ct}')$$

d) RFAME.Decrypt$(SK, C, (\mathbf{M}, \rho)) \rightarrow \{msg, \perp\}$: Parse C as $(\text{ct}_0, \text{ct}_{i,l} \,\forall\, i \in [1..n_1], l \in \{1, 2, 3\}, \text{ct}')$. For each row $\text{ct}_{i,l} \in \mathbf{M}$, pick coefficients $\gamma_i \in \{0, 1\}$ such that $\sum_{i \in I} \gamma_i (\mathbf{M})_i = [1, 0, \ldots, 0]$.

$$\text{num} = \text{ct}' \cdot e(\prod_{i \in I} \text{ct}_{i,1}^{\gamma_i}, \text{sk}_{0,1}) \cdot e(\prod_{i \in I} \text{ct}_{i,2}^{\gamma_i}, \text{sk}_{0,2}) \cdot e(\prod_{i \in I} \text{ct}_{i,3}^{\gamma_i}, \text{sk}_{0,3})$$

$$\text{den} = \prod_{i \in I} e(\text{sk}_{\rho(i),1}^{\gamma_i}, \text{ct}_{\rho(i),1}) \cdot \prod_{i \in I} e(\text{sk}_{\rho(i),2}^{\gamma_i}, \text{ct}_{\rho(i),2}) \cdot e(\text{sk}'_3 \cdot \prod_{i \in I} \text{sk}_{\rho(i),3}^{\gamma_i}, \text{ct}_{0,3}) \cdot \prod_{t \in \{1,2\}} e(\text{sk}'_t \cdot \text{sk}''_t, \text{ct}_{0,t})$$

e) RFAME.Revoke$(\text{mpk}_{\text{ABE}}, \text{msk}_{\text{ABE}}, v) \rightarrow (\text{mpk}_{\text{ABE}}', \text{msk}_{\text{ABE}}', SK')$: Let a user holding attribute $v \in U$ be revoked by the *AIA*. This algorithm is run by the *AIA* which generates new parameters for the non-revoked users of attribute group v, and updates its mpk_{ABE} and msk_{ABE}. It picks $\beta_v \leftarrow \mathbb{Z}_q^*$, and computes h^{β_v}. It updates $\text{mpk}_{\text{ABE}}' = (h, H_1, H_2, T_1, T_2, h^{\alpha_{y_1}}, \ldots, h^{\alpha_{y_{|U|-1}}}, h^{\beta_v})$. The msk_{ABE} remains the same except the α_v gets replaced with β_v. It then generates (a component of) the secret key for all *non-revoked* users possessing attribute v as follows:

$$\text{sk}_{v,t} = \mathcal{H}(v1t)^{\frac{b_1 r_1}{a_t + \beta_v}} \cdot \mathcal{H}(v2t)^{\frac{b_2 r_2}{a_t + \beta_v}} \cdot \mathcal{H}(v3t)^{\frac{r_1 + r_2}{a_t + \beta_v}} \cdot g^{\frac{\sigma_v}{a_t + \beta_v}} \cdot g^{\frac{\beta_v}{a_t + \beta_v}}; \text{sk}_{v,3} = (g^{-\beta_v} \cdot g^{-\sigma_v})$$

where $t \in \{1, 2\}$, and all other variables are as defined in the SetupABE and KeyGenABE algorithms. Set $SK' = (\text{sk}_0, \text{sk}_{v,t}, \text{sk}_{v,3}, \text{sk}'_1, \text{sk}'_2, \text{sk}'_3, \text{sk}'')$. SK' is distributed only to the non-revoked users who possess attribute v.

Figure 2: Construction of Revocable Fast Attribute Based Encryption (RFAME)

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: This algorithm (Figure 4a), 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 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 6) 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

```
ReTRACe.Keygen($1^\lambda$)
1 : GSetup($1^\lambda$) $\rightarrow$ 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($1^\lambda, U$) $\rightarrow$ (mpk$_{ABE}$, msk$_{ABE}$)
5 : RCHET.cSetup($1^\lambda$) $\rightarrow$ param
6 : RCHET.userKeySetup(param) $\rightarrow (sk_{ch}, pk_{ch})$
7 : PubPar = ($\mathbb{G}, g, q, pk_{ch}$, mpk$_{ABE}$, $gpk$)
8 : SecPar = (msk$_{ABE}$, $st_{GM}$)
9 : return (SecPar, PubPar, $sk_{ch}$)
```

(a) ReTRACe: **AIA/GM setup**

```
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(mpk$_{ABE}$, msk$_{ABE}$, $y_1, \ldots,$
    $y_{|U|}) \rightarrow SK$
4 : Retrieve $sk_{ch}$
5 : return key = ($\mathbb{GSK}, SK, sk_{ch}$)
```

(b) ReTRACe: **System setup for user**

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

(c) ReTRACe: **Signing a message**

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

(d) ReTRACe: **Verifying a message**

Figure 3: ReTRACe **algorithms for system setup and signing/verifying messages**

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

(a) ReTRACe: **Creating a message**

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

(b) ReTRACe: **Updating a message**

Figure 4: ReTRACe **algorithms for creating and updating a message**

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 $\xi_{\Upsilon_{info}}$ pass verification w.r.t. $\Upsilon_{GSadmin}$, 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 5a, Figure 5b) 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,

```
ReTRACe.RevokeUser(key, PubPar, $m'$, $msg$, $\xi_{msg}$)
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(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(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$'$, $m'$, $msg$, $\xi_{msg}$)
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(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(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**

Figure 5: ReTRACe **algorithms for revoking users**

Γ_{privinfo}, needs to be re-encrypted in response to the AIA revoking a user. Both cases are handled differently:

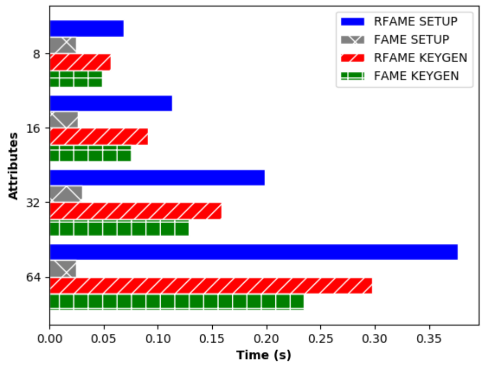
Case 1: Revoking a clause from Υ_{ABE}: This algorithm (Figure 5a) is run by an user $v \in$ **AuthUAdmins** who wants to modify an Υ_{ABE} associated with *msg*. The AIA/GM are not involved, and no algorithm from RFAME is called. User v first decrypts the trapdoor, Γ_{privinfo}, using her RFAME secret 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 and posts them.

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

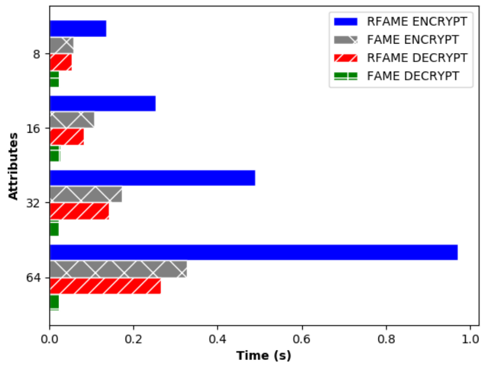
Figure 6: ReTRACe: **Miner verifying a message**

Case 2: AIA revoking a user: This algorithm (Figure 5b) 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$_{\text{ABE}}$ to 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.

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 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.



(a) Key generation and Setup



(b) Encryption and Decryption

Figure 7: Timings for RFAME vs. FAME [1] (80 users per attribute)

7.1 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.* [15] 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.

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 a subset of her attributes necessary to decrypt the trapdoor. We have proven the IND-CPA security of RFAME; we apply the Fujisaki-Okamoto transform [18] 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 [26].

Theorem 7.1. *If* RCHET *is secure,* RFAME *is fully IND-CCA2 secure, and* DGSS *is a secure dynamic group signature scheme then* ReTRACe *is secure.*

8 IMPLEMENTATION AND RESULTS

We implemented RFAME, RCHET, and ReTRACe in Python 3, and used Charm [22] 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 (see Section 5). Using RCHET and RFAME we implement ReTRACe. Note that ReTRACe is the first system that provides transaction-level revocable blockchain rewrites, there is no equivalent state-of-the-art scheme to compare with.

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 7 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 ($\mathrm{mpk}_{\mathrm{ABE}}$) and master secret key ($\mathrm{msk}_{\mathrm{ABE}}$) during setup; and the growth of the public key size in RFAME is linear in the number of attributes (small-universe property).

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, accounting for the difference in their encryption and decryption timings. For decryption, the number of pairing operations in RFAME is 6 + 2×(number of attributes satisfying a given policy), as compared to 6 pairing operations for FAME.

Table 1: 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

Table 1 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.

As mentioned before in Section 5, 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 the revoked user is in a single attribute group.

RCHET Results: Table 2 compares the running times for CHET and RCHET. In RCHET, when compared to CHET, 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. 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 2: Comparison of RCHET **vs. CHET [10] (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

Table 3 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 testing 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 since it includes the AIA's operations for revoking a user from one attribute group and rekeying of the rest of the users in the same group, whereas Case 1 just changes the message policy and updates the message trapdoor.

Table 3: 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

Implementation in Ethereum: ReTRACe can be plugged into existing blockchains (e.g., Ethereum) by updating 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.

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' $\leftarrow$ new; '1' $\leftarrow$ 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 cost of operations 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 contributions and functionalities that provide efficient and authorized transaction rewrites in blockchains, in addition to revocability and traceability of the users updating the transaction(s). We have performed rigorous security and experimental analyses to demonstrate ReTRACe's scalability.

ACKNOWLEDGMENTS

Research supported by NSF awards #1800088, #2028797, #1914635, Intel Labs, and the Federal Aviation Administration (FAA). 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 NSF, FAA, 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 *12th IMA International Conference, Cryptography and Coding, Proceedings*. 278–300.

[4] 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.

[5] 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.

[6] 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.

[7] John Bethencourt, Amit Sahai, and Brent Waters. 2007. Ciphertext-Policy Attribute-Based Encryption. In *2007 IEEE Symposium on Security and Privacy (S&P 2007)*. 321–334.

[8] Alexandra Boldyreva, Vipul Goyal, and Virendra Kumar. 2008. Identity-based encryption with efficient revocation. In *Proceedings of the 2008 ACM CCS*. 417–426.

[9] 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

[10] 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.

[11] 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.

[12] Sherman S. M. Chow. 2016. A Framework of Multi-Authority Attribute-Based Encryption with Outsourcing and Revocation. In *Proceedings of the 21st ACM on Symposium on Access Control Models and Technologies, SACMAT*. 215–226.

[13] Hui Cui, Robert H. Deng, Yingjiu Li, and Baodong Qin. 2016. Server-Aided Revocable Attribute-Based Encryption. In *Computer Security - ESORICS, Proceedings, Part II*. 570–587.

[14] Pratish Datta, Ratna Dutta, and Sourav Mukhopadhyay. 2015. Fully Secure Unbounded Revocable Attribute-Based Encryption in Prime Order Bilinear Groups via Subset Difference Method. *IACR Cryptology ePrint Archive* (2015). http://eprint.iacr.org/2015/293

[15] 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*.

[16] Dominic Deuber, Bernardo Magri, and Sri Aravinda Krishnan Thyagarajan. 2019. Redactable Blockchain in the Permissionless Setting. In *2019 IEEE Symposium on Security and Privacy, SP 2019, San Francisco, CA, USA, May 19-23, 2019*. 124–138.

[17] DHS. 2018. Department of Homeland Security: Blockchain and Suitability for Government Applciations. https://www.dhs.gov/sites/default/files/publications/2018_AEP_Blockchain_and_Suitability_for_Government_Applications.pdf.

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

[19] California Gov. 2018. California Consumer Privacy Act. https://oag.ca.gov/privacy/ccpa.

[20] Vipul Goyal, Omkant Pandey, Amit Sahai, and Brent Waters. 2006. Attribute-based encryption for fine-grained access control of encrypted data. In *Proceedings of the 13th ACM CCS*. 89–98.

[21] Business Insider. 2020. The growing list of applications and use cases of blockchain technology in business and life. https://www.businessinsider.com/blockchain-technology-applications-use-cases.

[22] JHUISI. 2012. Charm: A tool for rapid cryptographic prototyping. http://charm-crypto.io.

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

[24] Allison B. Lewko, Tatsuaki Okamoto, Amit Sahai, Katsuyuki Takashima, and Brent Waters. 2010. Fully Secure Functional Encryption: Attribute-Based Encryption and (Hierarchical) Inner Product Encryption. In *Advances in Cryptology - EUROCRYPT, Proceedings*. 62–91.

[25] Benoît Libert, Thomas Peters, and Moti Yung. 2012. Scalable Group Signatures with Revocation. In *Advances in Cryptology - EUROCRYPT, Proceedings*. 609–627.

[26] Gaurav Panwar, Roopa Vishwanathan, and Satyajayant Misra. 2021. ReTRACe: Revocable and Traceable Blockchain Rewrites using Attribute-based Cryptosystems. Cryptology ePrint Archive, Report 2021/568. https://eprint.iacr.org/2021/568.

[27] Matthew Pirretti, Patrick Traynor, Patrick D. McDaniel, and Brent Waters. 2006. Secure attribute-based systems. In *Proceedings of the 13th ACM Conference on Computer and Communications Security, CCS*. 99–112.

[28] Reuters. 2018. 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.

[29] Amit Sahai, Hakan Seyalioglu, and Brent Waters. 2012. Dynamic Credentials and Ciphertext Delegation for Attribute-Based Encryption. In *Advances in Cryptology - CRYPTO. Proceedings*. 199–217.

[30] 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

[31] Junichi Tomida, Yuto Kawahara, and Ryo Nishimaki. 2020. Fast, compact, and expressive attribute-based encryption. In *IACR International Conference on Public-Key Cryptography*. Springer, 3–33.

[32] Brent Waters. 2011. Ciphertext-Policy Attribute-Based Encryption: An Expressive, Efficient, and Provably Secure Realization. In *Public Key Cryptography - PKC, Proceedings*. 53–70.

[33] 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.

[34] Shucheng Yu, Cong Wang, Kui Ren, and Wenjing Lou. 2010. Attribute based data sharing with attribute revocation. In *Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security, ASIACCS*. 261–270.