μchain: How to Forget without Hard Forks

Ivan Puddu
ETH Zurich
pudduivan@me.com

Alexandra Dmitrienko
ETH Zurich
alexandra.dmitrienko@inf.ethz.ch

Srdjan Capkun
ETH Zurich
srdjan.capkun@inf.ethz.ch

***Abstract*—In this paper, we explore an idea of making (proof-of-work) blockchains mutable. We propose and implement μchain, a mutable blockchain, that enables modifications of blockchain history. Blockchains are, by common definition, distributed and immutable data structures that store a history of events, such as transactions in a digital currency system. While the very idea of mutable event history may seem controversial at a first glance, we show that μchain does not undermine security guarantees provided by immutable blockchains. In particular, all mutations in our system are controlled by fiat, enforced by consensus and are verifiable in the same way as regular transactions. At the same time, μchain provides a solution to a number of challenging problems, such as the patching of vulnerable smart contracts and removal of abusive content from blockchain history. It also gives rise to new blockchain applications that were not possible with immutable blockchains. For instance, governments and companies could now maintain registers of citizens and customers, while preserving their legislated rights to be forgotten. Banks could consider consolidation of cryptocurrency with traditional payments, which is hard to achieve without the ability to revert transactions. To further illustrate the power of μchain on more concrete examples, we present two new applications, the collaborative recommendation system with the ability to censor inappropriate content, and a time-lock encryption mechanism that provides a method to decrypt messages after a certain deadline has passed.**

I. Introduction

Blockchain is an emerging technology underpinning digital currencies like bitcoin. The success of Bitcoin[1] [1] has created interest in other applications that could benefit from decentralization and elimination of trusted third parties (TTPs). As a result, new innovative applications using blockchain technology emerged, such as decentralized time-stamping services [2], [3], distributed file storage [4], identity management solutions [5], [6], and smart financial contracts [7], [8], to name just a few. A number of open source projects associated with blockchain appeared [9], [10], [11], [7], [12], [13], [14], and a billion dollars in venture capital has led to more than 140 blockchain-related startups [15].

Blockchains are append-only distributed and replicated databases, which maintain an ever-growing list of immutable and tamper-resistant data records (e.g., financial transactions and account balances). In contrast to regular databases, blockchains do not rely on centralized trusted third parties, but rather leverage a network of participants (or validators) who replicate the database and use a group consensus protocol in order to synchronize the ledger. Blockchain validators integrate data records into blocks and chain them together in an append-only manner. The more blocks have been subsequently appended to the data record in question, the harder it gets to remove or modify the record, and not only for an attacker, but also for the legitimate network. This is especially true for blockchains powered by Proof-of-Work (PoW) consensus algorithms [16], which require validators to invest substantial amount of computational power when building a blockchain in a process called *mining*.

Immutability and tamper-resistance of blockchains stem from their append-only property and are paramount to security of blockchain applications. In a context of digital currency and payments, these properties ensure that all the parties have access to a single history of payment transactions and that such a history cannot be modified. In smart contract systems that additionally store executable code on a blockchain, the underlying blockchain can guarantee that conditions recorded in a smart contract are not to be modified since have been written and published. In other use cases, immutable blockchains could serve as an official registry for official documents such as diplomas, certificates and qualifications; business and governmental registries could log information about customers and citizens and physical and digital assets owned by them (land titles, vehicles, intellectual property rights); while individuals could build trustworthy user profiles which would include true facts from their biography such as birthday, date of marriage and graduation, working experience, etc.

While immutability and tamper-resistance of blockchains are essential to provide important security guarantees, the very same properties can hinder future of the technology. For instance, blockchains don't operate in isolation and sometimes external events will aim to annul existing contracts and transactions that exist on the chain. Furthermore, the data stored on the chain might be such that its distribution is illegal. For instance, Bitcoin blockchain is already hosting child sexual abuse images [17] and other content, such as wikileaks files and leaked cryptographic keys [18]. Moreover, smart contracts may have vulnerabilities and flaws in the same way as any other software programs [19], however their immutability prohibits vulnerability patching. In practice, this led to severe consequences for Ethereum network and DAO [20] – the contract that was exploited in June 2016. Attackers have stolen 3,641,694 Ether worth of about 79 million of US dollars [21]. Remarkably, it was neither possible to stop the attacker from draining DAO accounts even after the attack had been discovered, nor to protect remaining funds by any other means, e.g., by transferring them to a safer place. The problem was "solved" by deploying a hard fork – a

[1]As usual, we use capitalized names to denote system and lower case to refer to monetary currency.

manual intervention of blockchain operation orchestrated by a notable minority, the team of Ethereum core developers. Such a solution undermined essential trust assumptions, resulted in splitting of Ethereum into two blockchains (ETH/ETC) and brought the entire future of Ethereum into a question.

Even if used as a database, blockchains often need the mutability property. In particular, governmental databases may need to be mutable in order to remove records about citizens being convicted of a crime in order to preserve their legislated rights to be forgotten [22], or to change information regarding witnesses involved into a witness protection program. In a use case of trustworthy user profiles we mentioned above, users may want to selectively show and hide information, e.g., to display their graduation certificates when applying for a job, while hiding them when creating an *airbnb*[2] profile. While such modifications can easily be introduced into centralized systems by their (trusted) administrators, they are impossible with blockchain-based counterparts. As a matter of fact, state-of-the-art blockchains do not provide any mechanisms for removal or modification of data records, which, we believe, hinders further technology progress.

In our work, we aim to overcome limitations of immutable blockchains and explore an idea of a *blockchain that can forget without hard forks*. In particular, our contributions are as follows.

Contributions We present μchain, the mutable blockchain, which incorporates mechanisms for removal of records from the blockchain and their modifications. An important property of μchain is that it provides the same tamper resistance guarantees as immutable blockchains, and specifically in blockchains powered by PoW consensus. In μchain, all mutations are controlled by fiat, enforced by consensus and are verifiable in the same way as regular transactions. Hence, any unauthorized modifications are as hard to apply to the history of records as in immutable blockchains. We use the hack of the DAO smart contract as our motivation example and show, how its vulnerability could be patched without hard forks and without any disruption of blockchain operation. We further show how to deal with challenges associated with transaction mutations, such as negative account balances which can occur due to transaction reversions, as well as with the problem of unexpected currency withdrawal. To illustrate the power of μchain on concrete examples, we present two applications for μchain, which leverage its mutability properties: (i) a collaborative recommendation system with censorship, and (ii) a time-lock encryption mechanism. We prototyped μchain using Hyperledger open source project. The code is open sourced and is available at

https://gitlab.inf.ethz.ch/puddui/fabric

Outline The remainder of the paper is structured as follows. We begin by providing background information on blockchain technology in Section II. Next, in Section III we present our core contribution, the mutable blockchain. Further, in Section IV we discuss associated challenges and propose solutions to identified problems. Following, in Section V, we outline the complexity and describe optimizations for specific configurations of μchain. In Section VI, we introduce our applications, the collaborative recommendation system with censorship and the time-lock encryption. In Section VII, we describe in details our prototype implementation. In Section VIII we overview the related work, and then conclude the paper in Section IX.

[2]https://www.airbnb.com/

II. Background

In this section we present a generic blockchain architecture and describe actors, operational modes and core blockchain components. In the following sections we will rely on this background information and show how μchain design affects these generic building blocks.

A. Actors

We distinguish two different types of actors in the blockchain ecosystem: Regular users[3] and validators. Validators are also often referred as *miners* in blockchains powered by Proof-of-Work consensus algorithms, however we will always use term *validators* throughout the paper for consistency reasons. Actors own accounts, which are represented as public key pairs identifiable via hashes of corresponding public keys, referred as *account addresses*. Accounts of regular users are managed by individuals or business entities or can even be managed by computer programs. Validators are system administrators responsible for maintaining a blockchain, who can sometimes also act as regular users, however, regular users do not typically play the role of validators.

B. Blockchain

The blockchain is a data structure which serves as a distributed ledger and records information in a network-wide agreed sequence. Information may include data records of different types, such as cryptocurrency transactions, smart contracts and account balances. In the following, we discuss different types of data records in more details.

1) Transactions: Transactions transfer assets between accounts. While most often used to transfer digital currency, transactions are not limited to financial transfers, but in a general case may also represent the creation or transfer of physical assets, shareholdings, certifications, digital rights, intellectual property or even votes. Normally, transactions can transfer assets between several accounts at once, i.e., one transaction can involve several sender and destination account addresses.

[3]One can further distinguish various types of regular user accounts, e.g., Ripple [12] differentiates users who make/receive payments and market makers – entities providing trading services.

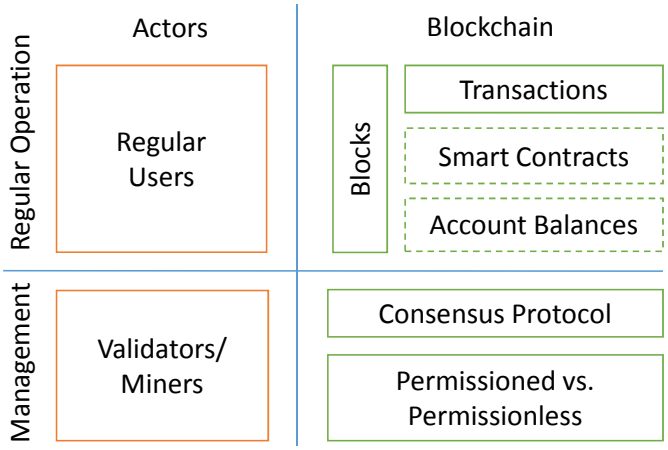

Figure 1: Generic Blockchain Architecture. Dashed lines indicate that the component is optional

2) Smart Contracts: Smart contracts are computer programs that have been previously created by regular users (or other smart contracts) and stored in a blockchain. They typically include code written in a some high-level language (e.g., Solidity [23] for Ethereum and Go [24] for Hyperledger) and data on which smart contracts operate. They can themselves act as regular users: Own regular user accounts holding cryptocurrency, send and receive transactions, or even control other smart contracts. Smart contracts are executed by validators – the distributed network of computing nodes – in a replicated manner, which ensures that an agreement encoded in a contract will be enforced when predetermined conditions are met.

While smart contracts are supported by some blockchains (e.g., Ethereum [7] and Hyperledger [10]), they may be absent in others (e.g., Bitcoin [1]), which we show by dashed lines in Figure 1.

3) Account Balances: Account balances indicate how much cryptocurrency is held within the given account. This information can be either included explicitly by integrating the balance of every account into blockchain's state (e.g., in Ethereum and Hyperledger), or be present implicitly (e.g., in Bitcoin). Implicit account balances are not explicitly recorded in the blockchain, however their integrity can always be verified by following the sequence of transactions from the very first block in the blockchain, the *genesis* block. To reflect this, we indicate account balances as optional in Figure 1.

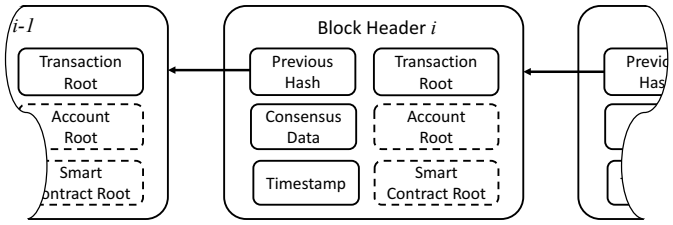

Figure 2: Blockchain Structure

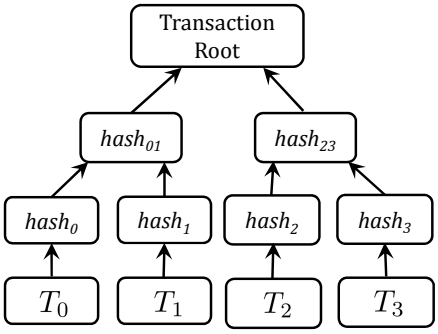

Figure 3: Transaction Merkle Tree

4) Blocks: Information in a blockchain is divided into blocks, where each block is typically associated with a fixed time window. Each block consists of data records (e.g., transactions and account balances) and block headers. Blocks are chained together by referencing previous blocks via inclusion of the hash of the previous block header into the header of the current block (cf. Figure 2). Additionally, block headers include a timestamp which corresponds to the time window of the current block, and integrity measurements of all data structures included into the block. Integrity measurements are typically constructed by using a Merkle tree [25] – a data structure that organizes data records in a tree, such that all the leaf nodes contain the hash of a different data record and all the other nodes of the tree contain the hash of the two nodes below them. The Merkle root can then be used to verify integrity of the data records, as it changes if any of the data records included into the tree change.

We depict an example of a transaction Merkle tree in Figure 3. Similar roots are generated for data structures of other types, such as account balances and smart contracts. Alternatively, different data types can be mixed in a single Merkle tree[4]. Another important value which is included into the block is a consensus data, which is typically a nonce for PoW consensus algorithms, or could be a signature when other forms of consensus are used. We will explain the role of this field in more detail in Section II-D.

C. Regular Operation vs. Management

Generally, actors can interact with a blockchain in two modes: Regular operation and management. When compared to databases, the regular mode is equivalent to read/write operations performed by database users. In this mode, regular users and smart contracts can send transactions, which is equivalent to write operation, and query the blockchain, which corresponds to read operation. In a management mode, which can be seen as an equivalent of database administration, the validators integrate transactions sent by regular users and smart contracts into the blockchain.

[4]For instance, Ethereum blockchain integrates information about smart contracts and account balances in a single tree which root is referred as *state root*.

D. Consensus Protocol

Consensus protocol is a mechanism that governs blockchain management. It is executed by validators and allows them to achieve an agreement on a single history of a blockchain. There are two major classes of consensus protocols in use: Based on Proof-of-work (PoW) and Proof-of-stake (PoS).

a) Proof-of-work (PoW): Proof of work algorithms are based on cryptographic puzzles which require significant amount of work to find a solution. Once found, the solution can easily be verified by other parties. While different PoW functions can be used to build such a cryptographic puzzle [26], the most common function used in blockchains is a partial hash inversion which requires to brute-force a nonce which, when hashed together with the block, would result in a value lower than a given threshold (referred as a target value). The target value is a security parameter which regulates difficulty of solving the puzzle.

When applied in a context of blockchain, PoW makes the process of appending a block to the blockchain extremely difficult. For instance, at the current hashrate of the Bitcon network of 1,592,411,223 GH/s, it takes on average 2^{66} hashes to find a valid nonce. Such a nonce is then written in the consensus data field of the block header, so that anyone can verify the solution to the PoW puzzle. The more follow-up blocks are appended to the block, the harder it gets to modify it, as it requires redoing the work for finding the current and all the subsequent blocks.

While in a collaborative effort of finding valid blocks a race condition may happen resulting in blockchain forks, these forks are resolved over time, as all the validators work on a longest chain and discard the others. Hence, it may be said that all the validators vote with their computing power to recognize the longest chain and consider its history of data records as valid.

b) Proof-of-stake (PoS): While Proof-of-Work algorithms require validators to provide a proof of computational resources spent to 'mint' a next block of a blockchain, proof-of-stake asks validators to prove ownership of a certain amount of currency (their "stake" in the currency). The more currency is owned by the validator, the higher is the probability for him to be selected to mint the next block. Some systems also extend the selection algorithm with the concept of "coin age", meaning that the longer period of time the stake have been held by the stakeholder, the higher is the probability to be chosen. Each PoS block must be signed by its creator. The signature of the block represents consensus data stored in the block header.

There are also hybrid designs which combine both approaches, the proof of work and proof of stake, to get the best of the two worlds. For instance, PPcoin [27] uses PoW to solve a cryptographic puzzle over limited search space combined with PoS-based checkpoint mechanism which is applied a few times daily. Moreover, Bysantine Fault Tolerance (BFT) protocols [28] were reconsidered in the context of blockchain consensus, which promise better scalability.

E. Permissioned vs. Permissionless blockchains

An important aspect of blockchain systems is the nature of participation. Here, one can distinguish permissioned and permissionless blockchains. Permissionless systems have open participation and allow anyone to become a validator or send transactions without any restrictions. Examples are Bitcoin and Ethereum blockchains. In contrast, permissioned blockchains are operated by vetted players whose identity need to be verified by an identity service, which binds information about business entities to cryptographic keys. For instance, BankCoin blockchain [29] is operated by banks and ensures that only the banks and the official authorities have access to it.

III. Mutable Blockchain

In this section, we present our core contribution, the mutable blockchain μchain. We first provide a high-level overview of our solution, then present a motivating example followed by detailed design description.

A. High-level Overview

Major properties of μchain are its ability to maintain alternative versions of data records, use consensus to agree upon a (currently) valid history, and its capability to hide alternative history versions. The first property is achieved through introduction of mutable transactions, which are represented as transaction sets containing possible transaction versions. In a transaction set, only one of the transactions is specified as *active*, while all the others are inactive alternatives. Transaction sets can be extended at a later time to add new transaction versions. Furthermore, every mutable transaction set includes the so-called $nope$ transaction, which is equivalent to "no operation" action that does not modify the state of the blockchian. Once selected as active, a $nope$ transaction effectively removes a mutable transaction from the history.

Blockchain history can be extended and modified by issuing transactions of special type, *meta*-transactions, which can introduce additional versions of data records and trigger mutations. Meta-transactions are treated in the same way as regular ones – they are issued by users or smart contracts, verified by validators, and once accepted, they are recorded in the history of the blockchain and can be verified by third parties. All mutations in μchain are subject to access control policies, which are specified by transaction senders and are attached to mutable transactions. These policies can define who and in which context is allowed to trigger mutations or add additional versions of data records, and their conditions are verified by validators when meta-transactions are processed.

Certain use cases which we aim to address may require confidentiality of alternative data records. For instance, when aiming to prevent distribution of child pornography and other illegal content via the blockchain, it is necessary to prevent access of users to the affected records. To hide alternative history versions, μchain encrypts all the possible history versions, and makes decryption keys available only for active records. It supports two key management schemes that

achieve confidentiality towards regular users only or towards both, regular users and validators. Once the active transaction becomes inactive due to mutation, its decryption key is not served anymore by validators. While local copies of keys for inactive transactions might be stored locally by clients, it is not a concern in the context of use cases which we aim to address, e.g., as long as distribution of illegal content through the blockchain is prevented. Other use cases, such as patching vulnerabilities of smart contracts, may not at all require confidentiality of alternative data records, and, hence, can be instantiated without using encryption.

B. Motivating Example

We consider the vulnerability in the DAO contract, that resulted in a loss of a few million dollars worth of cryptocurrency, as our motivating example.

The DAO takes its name from a distributed autonomous organization. It was designed to receive investments from the participants, and to distribute the cryptocurrency to other Ethereum-based startups and projects. Participants in return for supporting the project receive dividends. The DAO decides which projects will actually get funded using a voting procedure. Voting rights are given to DAO participants by means of a digital token, which is issued in return to an investment made.

The DAO has a built-in update mechanism called "newContract" [30]. In a nutshell, the proposal to upgrade to a new version of DAO is treated in the same way as any other DAO contract proposal – DAO token holders need to vote whether to approve the upgrade or not, and 53 % of votes is needed for the update to succeed. However, it appeared that by using this mechanism it was not possible to recover the internal state of the DAO in the new contract instance – in particular, an account balance called the "extraBalance" of a few millions worth would be lost if such an upgrade was performed [31]. It explains why the DAO was not updated using its "newContract" feature, although the recursive call bug that was exploited by an attacker was known to the community before the attack has happened [32].

In the following, we will introduce the key components of μchain design, and then show how they can be used to build a solution for patching the DAO vulnerability.

C. Mutable Transactions

Mutable transactions in μchain are represented as triplets $(\mathcal{T}, a, P)$, where $\mathcal{T}$ is a transaction set, a identifies an active transaction, and P specifies a mutability policy.

Transaction set A transaction set includes plurality of transactions, where each transaction in the set corresponds to one possible transaction version. In particular, a transaction set is defined as $\mathcal{T} = (\tau_1, \ldots, \tau_k)$, where any transaction $\tau_i \in \mathcal{T}$ may have three different types:

- Classical (*C*): these are traditional transactions signed by a specific sender S addressed to a given recipient R, transferring a certain amount of cryptocurrency from S to R and optionally including a data field.
- Deploying (*D*): transactions of this type are used to deploy smart contracts on a blockchain. They are signed by their sender S and are addressed to all the validators. In their data field they include the code to be deployed.
- Nope (*N*): these are transactions with no sender, no recipient and an empty data field.

Any mutable transaction must include at least one transaction of either classical or deploying type and a $nope$ transaction in its set. Further, all the transactions of classical and deploying type in the same transaction set must have the same sender S and the same recipient R[5]. Hence, it may be said that S is the sender of $\mathcal{T}$ and R is its recipient.

Active transaction The sender S specifies a default active transaction τ_a in the transaction set $\mathcal{T}$ by identifying an index $a \in \{1, ..., k\}$.

Mutability policy The transaction set is accompanied by a policy P which defines conditions for changing an active transaction. In particular, it specifies a party that is allowed to mutate the active transaction, which could be the sender S of the transaction set $\mathcal{T}$, its recipient R, or any other regular user or smart contract (or plurality of them). Furthermore, it may specify that the transaction set is extendable in the future, and to define who is allowed to add new transaction versions. Finally, the policy may specify a time window Δt within which the transaction remains mutable or extendable and becomes immutable/non-extendable afterwards.

Potentially[6], mutable transactions can co-exist with regular (legacy) transactions, which do not specify alternative transactions and do not include policies. To distinguish regular and mutable transactions, we denote them T and $\hat{T}$, respectively.

D. Modifying Transaction History

A default active transaction τ_a can be replaced at a later time with any $\tau_{a'} \in \mathcal{T}$ by issuing a transaction of special type, so-called *mutant transaction*.

- Mutant (*M*): This is a new type of transaction that has a sender, but no specific recipient – similarly to deploying transactions, it is implicitly addressed to all the validators in the blockchain. The mutant transaction references the transaction to be mutated $\hat{T}$ and defines a new active transaction $\tau_{a'}$ by specifying a new index $a' = \{1, \ldots, k\}$, $a' \neq a$. The mutant transaction needs to be signed by its sender and can only be sent by a legitimate party specified in the transaction policy P.

We denote mutant transactions as $T' = (ref_{\hat{T}}, a')$. In order to integrate a mutant transaction into *μchain*, validators ensure that the transaction policy $P \in \hat{T}$ is fulfilled and that the

[5]This implies that classical and deploying transactions cannot appear in the same set, as they have different recipients.

[6]However, when aiming to prevent distribution of illegal content, all the transactions should be made mutable to $nope$ transactions.

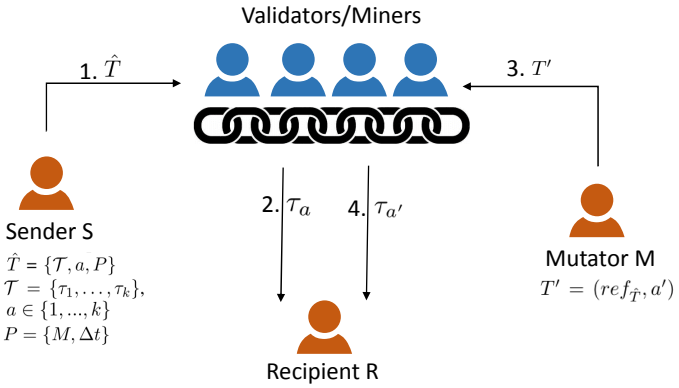

Figure 4: Transaction Mutation

new active transaction is different from the previous one (i.e., $a' \neq a$).

Note that the new active transaction defined with a mutant transaction might not be final. As long as it is allowed by the policies one can issue several mutant transactions for the same $\hat{T}$ and the last one will determine the value of the current active transaction of $\hat{T}$.

The mutation does not happen in the blockchain itself, but rather changes the view the validators have on the transactions recorded in the blockchain. Meaning: It *does not* change the content of the block containing the mutated transaction. It *does not* change that or other blocks' hash. It *does not* break the integrity of the chain and, as a consequence, *does not* require to perform again the PoW[7] for *any* block, no matter how many blocks where added on top of the one containing the mutated transaction.

Mutations in μchain are retroactive, therefore once a transaction has been mutated, validators read it as it always was in this state. This has implications on how the state of the system is calculated. Once mutated, the previous version of a transaction is as good as it never existed, and effects of the the new version will manifest in the most recent state of the blockchain.

In Figure 4 we illustrate how a mutable transaction $\hat{T}$ can be issued, mutated and received by the recipient. In step 1, the sender S sends the mutable transaction $\hat{T}$, which defines τ_a as an active transaction and specifies that it may be mutated by the mutator M within the time window Δt. Once accepted, the active transaction τ_a becomes available to regular users as well as to its recipient R. In step 3, the mutator M sends the mutant transaction T' to mutate the state of $\hat{T}$ to $\tau_{a'}$. If sent within Δt time window, it will take effect and replace τ_a with $\tau_{a'}$ in the mutable transaction $\hat{T}$, so that the recipient would now receive $\tau_{a'}$ instead of τ_a (step 4).

E. Transaction Set Extensions

A mutable transaction $\hat{T}$ may be extended at a later time by an extender E by sending the transaction of a special type, a so-called extending transaction.

[7]Or any other consensus mechanism used

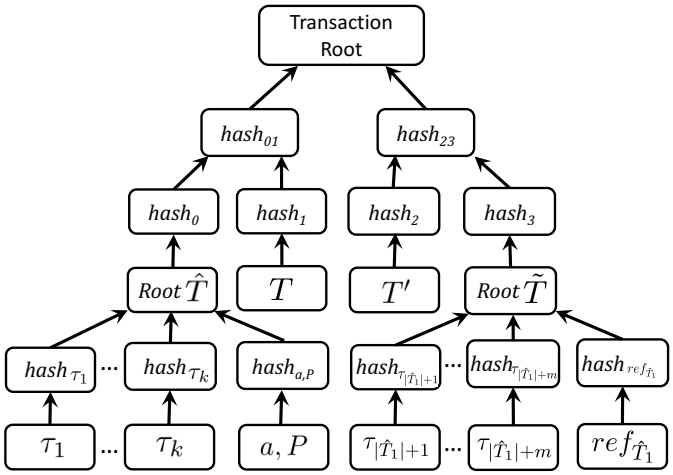

Figure 5: μchain transaction tree structure. The depicted example illustrates integration of four transactions: a mutable transaction $\hat{T}$, a regular transactions T, a mutant transaction T' and an extending transaction $\tilde{T}$.

- Extending (X): This is a new type of a transaction, which has a sender, but no specific recipient, similar to deploying transactions. The extending transaction references the mutable transaction $\hat{T}$ and specifies an extension set $\mathcal{T}_x = \{\tau_{k+1}, \ldots, \tau_m\}$, which includes $m - k$ new transaction alternatives. The extending transaction needs to be signed by its sender and can only be sent by a legitimate extender specified in the transaction policy $P \in \hat{T}$. Furthermore, all the transactions τ_i, $k < i \leq m$ must have the same sender S and the same recipient R as those of $\hat{T}$.

We denote an extending transaction as $\tilde{T} = (ref_{\hat{T}}, \mathcal{T}_x)$. Note that $\tilde{T}$ does not have its own policy – once accepted, it will be a subject to policy conditions specified in the original mutable transaction $\hat{T}$. Furthermore, it does not specify any active transactions in $\mathcal{T}_x$, because all of them will be inactive by default, but could be activated afterwards by mutating $\hat{T}$.

F. μchain Blocks

The structure of the μchain blocks is along the lines with the generic structure depicted in Figure 2: It includes such values as a hash of a previous block, consensus data, time stamp, and roots of data structures storing transactions, account balances and smart contracts. Our changes affect data structures for storing transactions. Further, we introduce a new data structure which we call *μchain version*. Its purpose is to manage the history of the blockchain by recording last active transactions within transaction sets.

Transaction Data Structure New transaction types are integrated into the Merkle transaction tree along with regular transactions. The updated structure of the transaction tree is depicted in Figure 5, which shows an example where one immutable, one mutable, one mutant and one extending transaction are integrated into the Merkle tree. In particular, the

mutant transactions (which are immutable) get integrated into the tree in the same way as regular transactions. In contrast, mutable and extending transactions are first compressed in their own tree structures, and their root values are then integrated into the transaction tree of the block along with regular transactions and mutant transactions.

Blockchain Version Data Structure To keep track of active transactions within transaction sets, we introduce a new data structure which contains information about mutable transactions and stores the last active value for each of them. To keep this information, we use a *trie* data structure[8], where each *(key, value)* pair is a node that can be addressed by its hash.

Account Balances As we already mentioned in Section II, account balances can either be explicitly written to the blockchain, or be inferred implicitly by following the sequence of transactions from the genesis block. In case of μchain, it is more reasonable to follow implicit approach due to the fact that mutant transactions may modify amount of cryptocurrency transferred between accounts. If account balances would be written explicitly, transaction mutations could invalidate account states stored in past blocks. Furthermore, such invalidated account balances can leak information about mutated transactions. For instance, it would be possible to figure out the amount of cryptocurrency previously transferred by a deactivated transaction by comparing the implicit state with the explicitly written (and already invalidated) account balance. All these drawbacks can be eliminated by using an implicit approach, which is also used by Bitcoin.

G. Hiding Alternative Data Records

To hide alternative data records, all the transactions in mutable transaction sets are encrypted using transaction-specific keys. We have chosen not to encrypt $nope$ transactions, because every transaction set must include one $nope$ transaction, and, hence, its encryption would not hide the very fact of its existence. Moreover, $nope$ transactions do not transfer any cryptocurrency and do not include any data, hence, their encryption does not seem to have any benefit, but would result in additional keys to manage. Furthermore, if the $nope$ transaction was encrypted, more often than not, validators would have no way to check whether or not the set includes it.

For the sake of simplicity, we also opted not to encrypt code deploying transactions, as we do not see any use case at the moment where this would be useful. However, if such a use case is to be discovered in the future, our design can be trivially adjusted to additionally encrypt code deployment transactions.

For the key management, we propose two design options: (i) simple encryption, and (ii) encryption with secret sharing.

[8]Trie is a term used in the Ethereum community [33] to refer to a combination of a Radix tree data structure, also known as Patricia tree [34], and a Merkle tree.

1) Simple Encryption: In the simple encryption approach, we encrypt all the transactions of classical type in mutable transaction sets with transaction-specific keys. Network validators are then responsible to reveal decryption keys for active transactions, while keeping keys for inactive transactions in secret. This approach is more suitable for use cases where it is reasonable to assume that validators would not reveal keys for inactive transactions on purpose. For instance, such an assumption is quite reasonable for private blockchains or for the recommendation system with censorship, which we describe in Section VI-A.

2) Encryption with Secret Sharing (ESS): In our second approach, the transaction-specific keys are split into shares using a secret sharing scheme [35]. The resulting shares are then distributed among the validators, which can only reconstruct the entire key if a sufficient number of shares (supplied by honest validators) is collected. Throughout the rest of the paper we refer to this configuration as *ESS*. The ESS option does not put any additional trust assumptions on validators beyond assumptions typically made for blockchain ecosystems. However, the price to pay is more significant performance and communication overhead. In Section V we formalize the ESS protocol and provide solutions for improving its communication complexity and storage requirements, so to make this key management feasible regardless of the number of validators securing the blockchain.

Note that Proactive Secret Sharing schemes [36], [37] can deal with proactive adversaries that can corrupt all the involved parties over a long period of time, but no more than a threshold t during a certain time window (called the refresh period). Furthermore, Dynamic Proactive Secret Sharing (DPSS) [38] schemes provide similar properties for dynamic groups, where the number of parties can vary during the course of the execution. Using a DPSS scheme in ESS allows us not only to support a dynamic number of validators, but also makes it sufficient for transaction senders to distribute key shares to a limited number of directly reachable validators, which then can, in turn, re-share the keys among all the online validators.

The main point in favor of using ESS is that regular users *and validators* alike would not be able to read inactive transactions. While necessary in some settings, this property can be a liability to the enforcement of some transaction set properties. For instance, as described in the previous sections, all the transactions in a set must have the same sender and recipient. This clearly cannot be checked if validators do not have access to all the transactions in a set. As a consequence, it is impossible to enforce it before the transaction set is included in the blockchain. Therefore checks depended on hidden transactions are to be performed at mutation time.

H. Patching Vulnerability of the DAO using μchain

By now we introduced all the building blocks necessary for showing how μchain could be used to patch the vulnerability of the DAO smart contract. In Figure 6, we demonstrate the necessary steps that need to be performed.

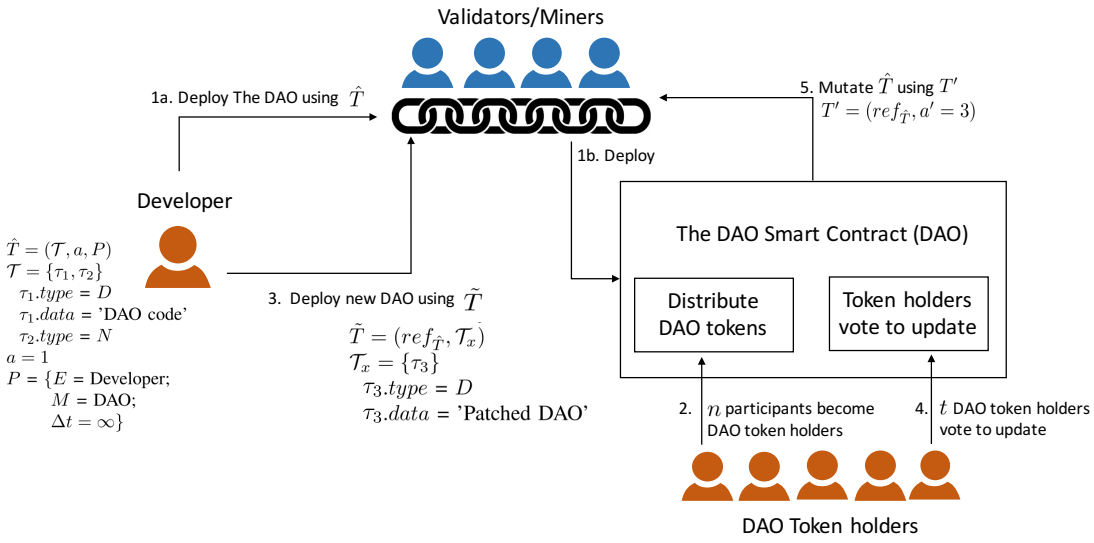

Figure 6: Patching the vulnerability of the DAO Smart Contract

First of all, the DAO developer needs to deploy the DAO code on the blockchain using the mutable transaction $\hat{T}$ (step 1a). Its transaction set $\mathcal{T}$ includes two transactions, τ_1 and τ_2, where the first transaction is a code deploying transaction of type (D) and includes the DAO code. The second transaction τ_2 is a $nope$ transaction that is always included into mutable transaction sets. τ_1 is selected as an active transaction (a = 1), and the policy P states that the transaction can be extended by the DAO developer and mutated by the DAO contract within unlimited time window. Once the code deployment transaction $\hat{T}$ is processed by the validators, the DAO smart contract is deployed on a blockchain (step 1b).

While operating, the contract enrolls n DAO token holders (step 2). Whenever a vulnerability is discovered in the smart contract and the patch is issued, the developer extends the mutable transaction $\hat{T}$ (issued at step 1a) by using extending transaction $\tilde{T}$ (step 3). $\tilde{T}$ includes the reference $ref_{\hat{T}}$ to the original code deploying transaction and carries out a single transaction τ_3 of type (D), which includes the patched code of the DAO contract in its data field. Once the new code becomes available on the blockchain, the DAO token holders can vote to accept or reject the update (step 4). Whenever t votes are collected, the DAO contract issues the mutant transaction T' (step 5), which makes the code deploying transaction τ_3 with the patched code active.

The validators will accept the mutant transaction, as long as it is issued by the legitimate mutator, the DAO contract. As a result, all the transactions which have been ever received by the DAO contract will be re-processed using the new code, resulting in all the stolen funds being returned to the contract.

Note that, in contrast to the "newContract" feature of the DAO, our approach does not require migration of the contract state to a new instance. Instead, the code of the old instance is replaced with the patched code – an approach which eliminates possible errors during state migration. Furthermore, using our approach it is possible to patch the vulnerability even *after* the attack has happened. When patched, the contract internal state would return to the state as if the attack had never happened.

IV. Challenges and Solutions

In this section we discuss a number of challenges which need to be tackled before taking advantage of the mutable blockchain becomes possible, and propose solutions to identified problems.

A. Problem of Negative Account Balances

Recall that, as explained in Section III-D, mutations are retroactive. Therefore, some transactions that were valid before a mutation, could now move more money than available in the balance of their sender at the moment of their emission, and as a consequence become invalid. We refer to these transactions as dependent transactions. In other words, mutable transactions may result in negative account balances, once the incoming transaction is replaced with the $nope$ transaction at the time when the account does not have sufficient funds (which might happen if, e.g., the received funds were already spent). We illustrate this situation in Figure 7. In the depicted example, we have accounts A, B and C with account balances of 10, 10 and 0, respectively. In a first step, A sends 10 units of cryptocurrency to B in a mutable transaction. As a result, the account balance of A changes from 10 to 0. In a second step, B sends 15 units to C. Third, the first transaction is mutated by replacing the first transaction with $nope$ transaction. As a result, the account balance of B becomes -5. This final state is a problem especially for permissionless blockchains where accounts are operated anonymously, and hence there is no way to force any entity to compensate the missing funds.

To address this challenge we suggest to compensate the missing amount by cancelling transactions that spent insufficient funds. When a transaction is mutated, dependent trans-

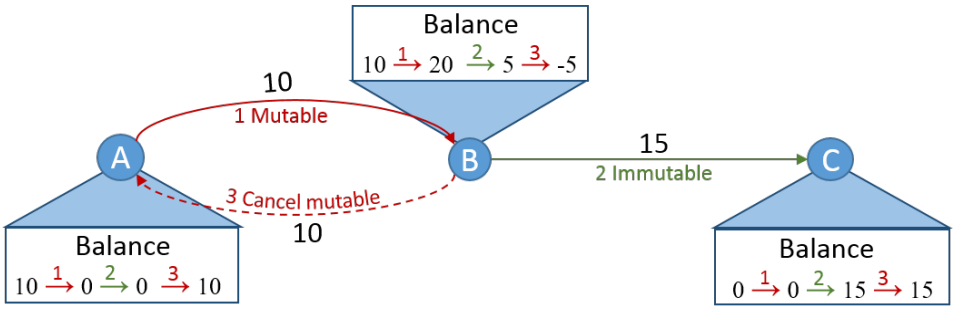

Figure 7: Problem of negative account balances

actions are also mutated to compensate the missing amount of cryptocurrency in the account. In particular, we mutate them to *nope* as a side effect of the first mutation. This process ensures that only valid transactions are left in the blockchain after a mutation. Further, this implies that a balance can *never* go below zero, not even if there exist another subsequent transaction that causes the balance to be restored.

In general the mechanism we described can be enforced only if transactions always have a time window (cf. Section III-C) that makes them immutable at the same time or after the transactions they depend on. If this was not the case we could be in the situation where a transaction is mutated to *nope*, but its depended transactions are already immutable and therefore cannot be reverted. In blockchains that follow an implicit approach (cf. Section II-B3), the relevant time window for transactions can be trivially obtained. In such blockchains usually each transaction is characterized by some "input" transactions and some "output" transactions, with the condition that the output transactions cannot transfer more cryptocurrency than the amount provided with the input transactions. To avoid negative balances, validators must check if all the output transactions have a time window in their policies that makes them immutable at the same time or after all the input transactions they depend on.

In blockchains that follow an explicit approach solving this problem is more involved since balances are simply an aggregate of the money they contain at any given moment. However to enforce our solution, for each balance we would need to keep track of each incoming transaction and relate each outgoing transaction to them, therefore losing the efficiency gained by aggregating the transactions in the first place. In such systems we suggest to split account balances into mutable and immutable parts. Both balances can be spent using mutable transactions, but only immutable balances can be spent using immutable transactions. Any amount of cryptocurrency contained in the mutable balance can be moved in the immutable balance as soon as it becomes immutable (as specified in the transaction policy). Note that validators still need to make sure that transactions issued from the mutable balance do not become immutable too early.

Figure 8 illustrates how such a mechanism helps to avoid negative balances by applying it to the example shown in Figure 7 and to a blockchain that explicitly stores account balances. The first step stays the same, namely A uses a mutable transaction to send 10 units of cryptocurrency to B. However, in Figure 8, B cannot spend 15 units in step 2 using an immutable transaction, because it does not have sufficient funds in its immutable balance. Hence, it splits its transaction into two, one mutable and one immutable, which send 5 and 10 units, respectively (denoted 2a and 2b in the figure). When the third step occurs, i.e., the transaction previously sent from A to B is mutated, this event also automatically triggers the mutation of transaction 2a to return missing (and mutable) funds from C to B.

B. Problem of Unexpected Currency Withdrawal

Now, when μchain has the ability to recursively withdraw the money from mutable accounts, the possibility of unexpected currency withdrawal may have negative impact on applications. To solve this problem, we propose a mechanism which enables account owners to define *account policies* which specify conditions for incoming transactions. For instance, the account holder C on Figure 8 could specify that his account only receives immutable transactions, or transactions which can be mutated within a specified period of time.

Given such account policies in place, the blockchain validators will reject transactions if the policy of a destination account is not fulfilled. For instance, if the owner of the account C specified that his account can receive only immutable transactions, the transaction sent from B to C at step 2a would be rejected.

C. Code Patching by Validators

In Section III we showed how μchain could help to patch the vulnerability of the DAO contract. The shown approach relies on the DAO token holders to make a decision if the new code version is to be accepted or not. In a more generic scenario, however, there might be no community similar to DAO holders which could be made responsible for making such a decision. Hence, the question remains how to patch vulnerabilities in contracts, which do not have stakeholders similar to the DAO.

We believe, that similar to legal contracts, in many cases smart contracts could be made overwritable by legal authorities

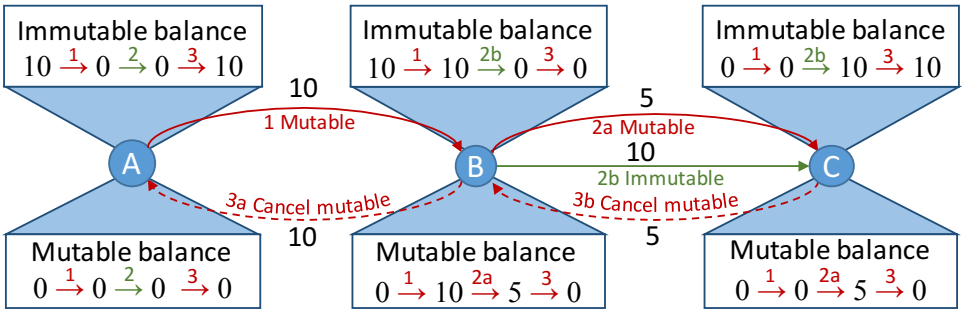

Figure 8: Solving the problem of negative account balances

like courts and alternative dispute resolution (ADR) authorities. In the same way as one would specify, e.g., an ADR authority for dispute resolution in a legal contract, smart contract creators could define similar parties as possible contract extenders and mutators at the time of contract deployment.

Alternatively, one could think of relying on validators for voting and making a decision instead of stakeholders. While straightforwardly achievable in the case of permissioned blockchains, where validators are vetted and typically own some sort of identity, this cannot be easily adapted to permissionless blockchains. Voting in permissionless blockchains is known as challenging problem [39], because validators there have no inherent identities or external PKI of trust. A malicious validator can therefore create a large set of Sybil identities [40] and vote multiple times in order to manipulate voting results.

To tackle this challenge, we outline a solution which enables validators in PoW blockchains to vote with their mining power. In particular, we can introduce a new sort of meta-transactions, which would originate from validators rather than from the specific sender S. Because validators (as a group) do not have any particular sender address and, hence, no corresponding signing key, technically such transactions could be realized in the same way as coinbase transactions[9] in Bitcoin, which do not have any senders. Once issued, these transactions are to be processed by all the validators, who then need to individually decide if they would accept or reject them. Depending on their decision, these transactions would either be included or excluded into the blockchain they are working on. The blockchain which accumulates more computing power will eventually win, thus recording the positive voting result in the blockchain history. Note that similar approach could be used to mutate otherwise immutable transactions, or overrule mutation policies attached to mutable transaction sets.

As of today, such a validator-based voting could be applied only in exceptional cases, because the decision to make (to vote or not to vote) would require some sort of manual input – a solution which cannot be applied on a regular basis. Furthermore, there is a potential for a denial of service attack, in which an attacker could issue bogus code extending and mutant transactions, forcing the validations to spend manual effort in order to decide if the proposal needs to be accepted or rejected. However, we see a potential of solving this obstacle in the future – there are already first efforts [41] in automated bug identification in smart contracts. We believe, this line of research could be further extended to identify methods to automatically issue vulnerability patches. Such solutions could then facilitate decision logic during voting process. While beyond the scope of this paper, we would like to investigate this approach in our future work.

[9]Coinbase transactions collect and spend any transaction fees paid by transactions included into the block

V. Complexity Analysis and Optimizations

In this section we present an enhanced version of the ESS protocol (cf. Section III-G2) called Enhanced-ESS or EESS for short, which is optimized in terms of communication complexity. We first provide background information on Dynamic Proactive Secret Sharing (DPSS) and then use the introduced primitives to formalize ESS. We then present details of the EESS protocol and compare the asymptotic communication complexity of the two protocols.

A. Background information on DPSS

As briefly introduced in Section III-G2, a Dynamic Proactive Secret Sharing (DPSS) scheme is the most suitable choice when dealing with secret shared keys in a permissionless blockchain, mostly because it allows to keep a secret confidential even if the set of parties holding its shares changes over time.

A DPSS scheme [38] generally consists of three protocols: Share, Redistribute, and Open. In the following we describe them in more details and report their associated complexity.

- Share allows a *dealer*[10] to share a secret, s, among n parties such that the secret remains secure against an adversary that controls up to t parties, whilst allowing any group of at least $t + 1$ or parties to reconstruct the secret. The current most efficient implementation of this protocol is RobustShare from [42] which can share $O(n)$ secrets in the same "batch" to n parties with $O(n)$ communication complexity, hence achieving $O(1)$ amortized communication complexity per secret.

[10]A dealer is a party that posses some input to be fed to the protocol. In our case it would be a transaction sender.

- Redistribute allows transferring the set of secrets from one group of parties to another and change the threshold. This protocol also allows to "proactivize" the shares, as shares gathered before and after this protocol takes place cannot be used together to reconstruct the secret. Currently the most efficient implementation is Redistribute from [38], which, in a synchronous network, achieves $O(1)$ per secret amortized communication complexity.
- Open allows to reveal a secret that was previously secret shared. Reco from [42] is the current most efficient instantiation of Open, it lets a set of parties reconstruct a secret toward a specific party with constant amortized communication complexity per secret. The same protocol can be used to reconstruct towards all parties at the same time and with the same communication complexity by broadcasting every value rather than sending it to only a specifc party.

Regarding the threshold t, current cutting edge DPSS [38] offer different values depending on the security guarantees desired. If one expects around half of the validators to be malicious, then the DPSS scheme can tolerate $t < \frac{n}{2} - \epsilon$ malicious parties, with statistical security. This means that t parties might still be able to break the protocol with a probability dependent on ϵ, the bigger ϵ is, the smaller the probability is they can break the scheme. On the other hand the scheme can be made perfectly secure if one is willing to tolerate $t < \frac{n}{3} - \epsilon$ corrupted parties. Various proactive secret sharing schemes provide different tolerance threshold [43], and some of them may even provide security against dishonest majority [44], however these schemes have complexities that are not even close to the asymptotic communication complexity offered by [38].

When a secret is shared using a secret sharing scheme, it gets embedded at a point x of a polynomial h of degree d. The shares of the secret that the validators hold are simply values of h at points different from x. Open works by obtaining enough (i.e. at least $d + 1$) points to interpolate h and then returns the value of $h(x)$, while Redistribute works by changing the polynomial that embeds the secret.

The communication complexity of the current state of the art protocols can be amortized per-secret, because, among other mechanisms, secrets can be packed together and, as a consequence, each subprotocol of a SS scheme can be performed with a multitude of secrets at the same time. This technique has been introduced by Franklin and Yung in [45] and works by embedding different secrets at different points of the *same* polynomial. From this follows that by giving shares of a single polynomial one is actually sharing multiple secrets at the same time, and in general any operation performed to the sharing polynomial affects all the secrets it contains. However, this also means that all the secrets are necessarily opened together.

Further observe that the more secrets one adds to the same polynomial, the greater the degree, d, of the polynomial must necessarily be. This has implications on t, as the bigger the degree is, the more honest parties are needed to correctly reconstruct the polynomial and thus the secrets embedded therein. In the current state of the art schemes [42] the maximum number of secrets that can be shared in a single batch is $\ell = n - 3t$ for a perfectly secure DPSS scheme, which, as mentioned above, has threshold of $t < \frac{n}{3} - \epsilon$. For a statistically secure DPSS scheme on the other hand we have $\ell = 2^{\lfloor \log_2 \frac{n}{4} \rfloor}$.

B. Complexity Optimizations

Note that the DPSS schemes discussed above are efficient only if one can actually pack secrets together, as sharing a secret alone has the same complexity of sharing ℓ secrets at the same time, which is $O(n)$. However we observe that it is not trivial to take advantage of the amortized communication complexity offered by the DPSS schemes to implement the ESS protocol (cf. SectionIII-G2). This primarily follows from the fact that senders only have a couple of secrets to share at the same time – namely the keys related to the transactions in the set they are about to send – which, in most μchain use cases, we don't expect to be large enough to influence the complexity of the sharing schemes. Moreover, as pointed out in Section V-A, when multiple secrets are shared together, they are also opened together, which negates the only advantage of using the ESS configuration over the simple key encryption (cf. Section III-G1), that is to hide inactive transactions not only from regular users, but also from validators.

The previous limitations could be almost completely circumvented if one was able to "pack" and "unpack" secrets when necessary, and if one would pack secrets per block rather than per transaction set. In more detail, given these two primitives, one could pack all the keys related to a μchain block after those keys have been, one at a time, shared by the users. Since blocks should contain a significant number of transactions, this process would allow to take advantage of the amortization offered by the DPSS scheme, while still leaving the flexibility to open a single secret by unpacking it when necessary. The techniques to pack and unpack already shared secrets are presented among other considerations in [45] where it is shown how to join several "*singly-shared*" secrets into one polynomial and how to split "*multi-shared*" secrets into several polynomials. This two operations are what we refer to with subprotocols MergeShares and DivideShares respectively. Both algorithms have a communication complexity[11] of $O(n)$.

Protocol 1, which we name Enhanced-ESS or EESS for short, shows how key management can be done by leveraging a packed DPSS scheme in conjunction with MergeShares and DivideShares. The correct execution of the protocol by more

[11]In more detail, as mentioned in [45], MergeShares and DivideShares have the same communication complexity of a "2-ary multiplication" of a MPC protocol. Note that the DPSS scheme that we described in Section V-A is used in [38] to construct a Dynamic Proactive MPC scheme whose 2-ary multiplication has linear, in the number of parties, communication complexity. Hence, for what concerns our setting, both MergeShares and DivideShares can be instantiated with $O(n)$ communication complexity.

Protocol 1 Enhanced-ESS Key Management

Input: The decryption key of each encrypted transaction in the transaction set $\mathcal{T}$ of a mutable transaction $\hat{T}$

Output: The key of the active transaction k_a of $\hat{T}$.

1: The sender S of $\hat{T}$ broadcasts k_a, the key of the default active transaction of $\hat{T}$ to the validators.
2: S shares in the same batch each decryption key related to $\hat{T}$ to validators it can contact directly, by running Share.
3: If the validators reject the first active transaction of $\hat{T}$, they discard every share related to $\hat{T}$ and the protocol aborts. Otherwise it continues.
4: The set of validators, who received their shares from S directly, execute Redistribute to re-share each key related to $\hat{T}$ with every online validator.
5: As soon as a new block, B, is confirmed, each validator runs MergeShares to merge every share of the keys of every mutable transaction issued in B. Let m be the number of decryption keys related to transactions in B, validators repeat MergeShares $\lceil \frac{m}{\ell} \rceil$ times, with ℓ different keys in each iteration (padding with random values as neccesary).
6: At fixed time intervals the validators redistribute the shares merged at step 5 by running Redistribute for each batch of secrets in their possession.
7: Whenever a mutant transaction is issued regarding $\hat{T}$, the validators extract the shares of $k_{a'}$ from the batch where it was shared using DivideShares.
8: Each validator uses the shares obtained with DivideShares to run, toward every validator, Open on $k_{a'}$. As a result, each validator knows the new $k_{a'}$.
9: Each validator deletes all the shares obtained with DivideShares at step 7.
10: If the new active transaction of $\hat{T}$ does not execute successfully then each validator discards the $k_{a'}$ for $\hat{T}$, otherwise each validator discards k_a by assigning the new value to it, which is $k_a = k_{a'}$.

than t honest validators[12] ensures that each honest validator always knows the key necessary to decrypt the current active transaction of every transaction set. At the beginning of the protocol, the sender S of a mutable transaction $\hat{T}$ with associated transaction set $\mathcal{T}$ holds the keys to decrypt the encrypted transactions in $\mathcal{T}$. The protocol is initiated by S, who broadcasts k_a – the decryption key of the default active transaction of $\hat{T}$ (step 1). Subsequently S secret shares all the decryption keys of $\hat{T}$ in the same batch to the validators it can contact directly (step 2). At the same time validators decrypt the default active transaction of $\hat{T}$ with the key received from step 1 and start executing it. If the transaction is rejected, every information related to $\hat{T}$ is deleted and validators abort the protocol, since managing keys for that mutable transaction is no longer necessary (step 3).

[12] t is the maximum number of allowed malicious parties for a DPSS scheme as defined in Section V-A

In case $\hat{T}$ has been accepted, the shares received directly by validators from the sender are re-shared with all the validators (currently online) using the Redistribute protocol of the DPSS scheme (step 4). Next, validators wait until the block, B, of μchain containing $\hat{T}$ is included in the blockchain with high certainty[13]. Then they execute MergeShares to merge all the shares of the keys related to mutable transactions contained in B into as few polynomials as possible (step 5). Note that they may need to be merged into several polynomials because, as explained before, there is a maximum number of secrets, ℓ, that can be packed together in the same batch. Validators re-share the merged secrets by executing Redistribute at fixed time intervals (step 6). This step is intended to provide to all the validators shares of secrets issued while they were offline, and to increase the resilience of the system against a mobile adversary. If, for instance, this step is performed once per day, all the shares the attacker managed to gather yesterday become useless, provided that they were less than $t+1$ shares. Note that similar remarks apply also to step 4, although, rather than to refresh the secrets, it is performed mainly to increase the corruption threshold t. Since t is directly proportional to the number of validators, by redistributing the secret from a small set of validators to every online validator, not only we make the leak of the secrets more difficult – as more validators would need to be compromised or collude in order to obtain it – but we also increase their resilience to failures, as more validators would need to fail simultaneously in order to render the secrets unrecoverable[14].

All the validators keep k_a, the decryption key of the current active transaction of a mutable transaction, in their local memory for as long as the transaction related to that key is active. Once a mutant transaction T' is issued changing the state of a transaction set $\hat{T}$, the validators begin to compute the key of the new active transaction. They first run DivideShares to obtain shares of a polynomial that only contains the key of the new active transaction of $\hat{T}$ (step 7). They then open the new key $k_{a'}$ to each other in step 8 and, at step 9, they discard all the shares obtained in step 7. As soon as validators get to know $k_{a'}$, they start executing its related transaction. If the new active transaction is rejected, validators keep the old k_a and discard $k_{a'}$, otherwise they store $k_{a'}$ in place of k_a.

Note that protocol 1 can be used, with minor changes, also to manage keys of transaction extensions. In particular, extensions are treated as normal transactions by the protocol, except that, since every transaction they introduce is inactive, step 3 is not performed for them.

C. Complexity Analysis

In the following we are going to compare the difference in complexity communication between the protocol presented in Section V-B and the ESS protocol. In order to do so we first formalize the latter in protocol 2 using the concepts presented in Section V-A.

[13] In Bitcoin, for instance, by convention this happens as soon as at least six blocks have been appended to it.

[14] If t or less shares remain in the network, the secrets are practically lost

Protocol 2 ESS Key Management

Input: The decryption key of each encrypted transaction in the transaction set $\mathcal{T}$ of a mutable transaction $\hat{T}$

Output: The key of the active transaction k_a of $\hat{T}$.

1: The sender S of $\hat{T}$ broadcasts k_a, the key of the default active transaction of $\hat{T}$ to the validators.
2: For each encrypted transaction $\tau_i \in \mathcal{T}$, S runs Share(k_i) toward the validators it can contact directly.
3: If the validators reject the first active transaction of $\hat{T}$, they discard every share related to $\hat{T}$ and the protocol aborts. Otherwise it continues.
4: The set of validators, who received their shares from S directly, execute Redistribute to re-share each key related to $\hat{T}$ with every online validator.
5: At fixed time intervals each validator runs Redistribute for each secret in its possession.
6: Whenever a mutant transaction is issued regarding $\hat{T}$, the validators run, toward every validator, Open on $k_{a'}$. As a result each validator knows the new $k_{a'}$.
7: If the new active transaction of $\hat{T}$ does not execute successfully then each validator discards the $k_{a'}$ for $\hat{T}$, otherwise each validator discards k_a by assigning the new value to it, which is $k_a = k_{a'}$.

To analyze the difference in complexity between the two protocols it is helpful to split them into three phases:

1) In the first phase the sender sends the keys of a new transaction set to the validators. This includes steps 1 to 5 for protocol EESS and steps 1 to 4 for protcol ESS.
2) In the middle phase secrets are refreshed at fixed intervals. This is performed with step 6 in protocol EESS and with step 5 in protocol ESS.
3) The last phase captures the diffusion of an updated key as a consequence of a mutation. It happens from step 7 to 10 in protocol EESS and from step 6 to 7 in protocol ESS.

The complexity of the first phase of the ESS protocol is dominated by step 4. It has a communication complexity of $O(xn)$, where x is the number of encrypted transactions in the transaction set, whose keys have been shared, one by one, during step 2. Note that, as already mentioned, in most use cases we expect x to be negligible in respect to n, in these cases then the asymptotic communication complexity of this phase can be approximated to $O(n)$. Similarly the complexity of this phase for protocol EESS is dominated by step 4 and step 5. Step 4 has a complexity of $O(n)$, while step 5 has complexity $O(\frac{m}{\ell} \cdot n)$, where ℓ is the maximum number of secrets allowed in the same batch, while m is the number of decryption keys related to transactions contained in a given μchain block. The given complexity follows from the fact that step 5 executes MergeShares $\lceil \frac{m}{\ell} \rceil$ times and each execution takes $O(n)$ communication complexity. From these considerations we can conclude that the complexity of the first phase of protocol EESS is $O(\frac{mn}{\ell})$.

The middle phase depends, for both protocols, on Redistribute. Protocol ESS runs it once for every key the validators are supposed to hold. This number is roughly the sum of the number of different versions possible for each mutable transaction in μchain, let us name it v. The total complexity for the middle phase of protocol ESS can then be expressed as $O(vn)$. For what concerns protocol EESS, each key is merged in as few batches as possible during the first phase, and, since batches are what ultimately gets redistributed, its complexity on this phase is $O(\frac{v}{\ell} \cdot n)$. The middle phase depends, for both protocols, on Redistribute. Protocol ESS runs it once for every key the validators are supposed to hold. This number is roughly the sum of the number of different versions possible for each mutable transaction in μchain, let us name it v. The total complexity for the middle phase of protocol ESS can then be expressed as $O(vn)$. For what concerns protocol EESS, each key is merged in as few batches as possible during the first phase, and, since batches are what ultimately gets redistributed, its complexity on this phase is $O(\frac{v}{\ell} \cdot n)$.

The last phase of protocol ESS can be seen to have communication complexity $O(n)$, as the only communication happens with the broadcasts performed to run Open at step 6. For analogous reasons the last phase of protocol EESS is influenced solely by step 7 and 8. Both steps have complexity $O(n)$ therefore making it also the complexity of the phase at which they belong to.

To summarize, the last phase in both protocols has the same complexity. It does not favor one over the other and, hence, can be put aside. For what concerns the first phase, it is $O(xn)$ for protocol ESS, while it is $O(\frac{m}{\ell} \cdot n)$ for protocol EESS. However this comparison is not entirely fair, as protocol ESS requires several iterations of its first phase to reach the same result as the one achieved by protocol EESS with its first phase. This is because the first phase of protocol EESS actually handles all the keys related to transactions belonging to a block of μchain, while the first phase of protocol ESS only considers the keys of a single mutable transaction. If we repeat the first phase of protocol ESS for each transaction in a block, the complexity is then $O(mn)$. Therefore, in aggregate, the first phase is more expensive to be performed with protocol ESS than with protocol EESS. Finally the middle phase of the protocol EESS is more efficient than that of ESS, having complexity $O(\frac{v}{\ell} \cdot n)$, as opposed to $O(vn)$.

To conclude, protocol EESS outperforms the protocol ESS by a factor of ℓ. Remember that, based on the security guarantees desired, ℓ can either be $n - 3t$ or $\leq \frac{n}{4}$, either way we have $O(\ell) = O(n)$. By using this fact we can express the complexity of the first phase of protocol EESS as $O(m)$ and the second phase as $O(v)$, meaning that we actually achieve linear amortized communication complexity per key shared. This result is asymptotically optimal, and makes the optimized protocol, contrary to the trivial protocol, a feasible alternative for permissioned as well as *permissionless* blockchains of any size. Note however that the third phase, the one that produces a new key after a mutation, remains relatively expensive in both protocols, as each of them requires $O(n)$ communication

complexity. We leave the optimization of the last phase of the key management protocol for future work, but as of now this just adds itself to the list of reasons for which mutations should be used only if strictly necessary in *open* blockchains.

VI. Applications on Top of μchain

The mutable blockchain, whose design we presented in previous sections, gives rise to new blockchain applications that were not possible with immutable blockchains. To illustrate flexibility of μchain as an application platform, we present a collaborative recommendation system with censorship and a time-lock encryption application.

A. Collaborative Recommendation System with Censorship

The recommendation system is a key component of many today's online services, such as online market places and hotel booking web-sites. They help customers to get the best product or service for the lowest price and enable service providers to get recognition for good services. As of today, such recommendation systems are typically implemented using a centralized trusted party, that is trusted not to unfairly push their favored products at the top of the recommendation lists. However, given a strong inclination among customers towards considering the best-rated offers, there is a clear monetary incentive to sellers, and in turn to service providers, to manipulate recommendations.

Using μchain, it is possible to realize a blockchain-based collaborative recommendation system, which can be used by service providers like hotels, restaurants and online market places and replace systems like Tripadvisor [46], HRS [47], Yelp [48] and Amazon [49], traditionally managed by trusted third parties. Recommendation systems are typically censored in order to clean up reviews containing inappropriate content, like rude expressions, or inappropriate statements about groups based on race, gender, or religion. This, in turn, requires the majority being able to remove information (to be censored) from the blockchain history, which is impossible with immutable blockchains, but becomes possible with μchain.

Entities We name our recommendation system Justice, as it is designed to fairly treat all the involved parties. The system model of Justice includes two types of users: (i) Clients and (ii) Endorsers. Clients are regular users of the recommendation system who write and read reviews. Hence, we may also distinguish them as readers and writers. In contrast, endorsers are service providers, who collaboratively maintain the recommendation system for rating their services. They are responsible for censoring reviews and filtering out only inappropriate content. Endorsers may be blockchain validators – the approach that is more suitable for private permissioned blockchains, or be just regular blockchain users. Both, the clients and the endorsers own regular user accounts on the blockchain and, hence, are capable of sending and receiving transactions.

We denote a group of n endorsers as $\mathcal{E}$. Furthermore, we denote readers and writers as R and W, respectively.

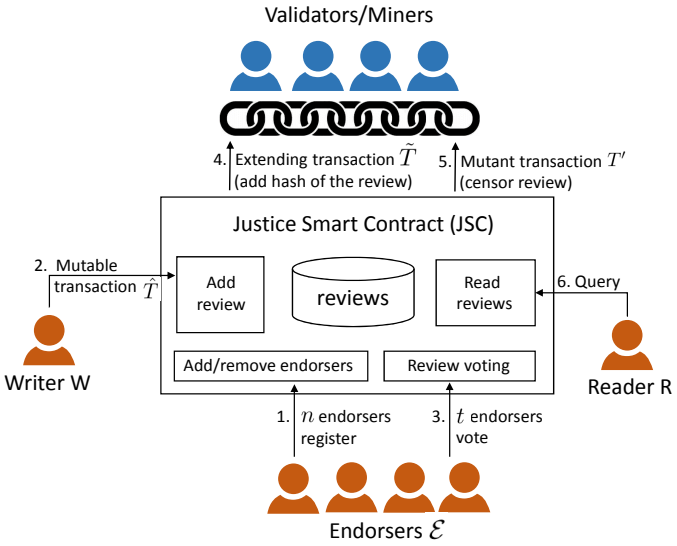

Figure 9: Justice review system with censorship: Architecture

Architecture The architecture of Justice system is depicted in Figure 9. Its core functionality is realized by means of a Justice smart contract (JSC) that is hosted by μchain. The smart contract implements a number of functions that can be invoked by means of transactions or queries[15]. It includes a database of user reviews, which can be written using mutable transactions and read by querying the contract. Further, the smart contract itself is able to mutate mutable transactions, whenever the state of the review need to be changed from published to unpublished and vice versa.

After the first deployment, the Justice smart contract needs to enroll n endorsers to the system (step 1 in Figure 9). We do not specify any exact conditions which need to be fulfilled by regular users to become endorsers, as those may vary in different applications. For instance, in case if μchain is instantiated as permissioned, the smart contract could validate user permissions. In permissionless blockchains it might be more reasonable to keep registration open to any parties who would pay a participation fee (or, a stake).

Once the contract has enough endorsers in the system, it can accept transactions from review writers. Hence, the writer W sends a mutable transaction $\hat{T}$ with the review to the smart contract (step 2). We recall that mutable transactions have the following structure: $\hat{T} = \{\mathcal{T}, a, P\}$, where $\mathcal{T}$ is a transaction set, a specifies active transaction and P defines mutability conditions. In this particular case, $\mathcal{T}$ consists of two transactions $\{\tau_1, \tau_2\}$, where τ_1 is of classical (C) type and includes the review m in the data field, while τ_2 is of type $nope$, as required by μchain design (cf. Section III-C). The writer W also defines the transaction τ_1 as an active transaction by specifying $a = 1$. Furthermore, the policy P states that $\hat{T}$ is extendable and mutable by JSC contract and within unlimited period of time.

[15]In contrast to transactions, queries do not change the state of the blockchain.

Once the transaction $\hat{T}$ is received by the smart contract, the review is added to the reviews' database. From now on, endorsers $\mathcal{E}$ may vote to censor the review, if they find its content inappropriate (step 3). Whenever the votes by endorsers reach the threshold t, the smart contract issues an extending transaction $\tilde{T}$ to extend $\hat{T}$ with a new transaction τ_3, which includes the hash of the review in its data field (step 4). Next, it mutates the state of the mutable transaction $\hat{T}$ by issuing a mutant transaction T' and specifying τ_3 as active (step 5). This action results in a full review being substituted with its hash value, therefore hiding the review itself[16].

To read the reviews, the reader R sends a query request to the smart contract (step 6). The smart contract returns the data stored in its review database, including full reviews and hashes of censored reviews. Note that the writer W can also play a role of a reader R and query the smart contract. By observing the hash value instead of a review, W can infer information that the review was received by the system, however it does not appear online due to censoring.

Additional considerations If the system requires support for review withdrawals by their writers, such a functionality can be realized by adding a policy rule into P of $\hat{T}$ to allow the original writer W to activate *nope* transaction. This will allow W to issue a mutant transaction and activate τ_2 in the transaction set of $\hat{T}$.

For backward compatibility reasons, it is beneficial to realize communication between clients and the smart contract using a web-server as a proxy. This would enable clients to use their regular web-browsers for the communication with the recommendation system. On another hand, if compromised, such a server could manipulate real reviews of clients, e.g., substitute them with bogus ones. However, such attacks are detectable, e.g., clients may independently verify if there is a hash of their review in the smart contract in case they do not see their review online. Furthermore, although a proxy web-server is capable of writing and publishing arbitrary bogus reviews, it is equivalent to becoming a review writer who can always write arbitrary reviews. This attack vector, in turn, can be mitigated, e.g., as suggested by Kerschbaum [50], by accepting reviews only from clients who can prove they have paid for the service they are going to review, and only can write one review per payment.

B. Time-lock Encryption

Our second application provides functionality of the time-lock encryption. The concept of time-lock encryption [51] enables decryption of a message or a file at some point in the future – the primitive which is useful in a number of real-world applications such as electronic auctions, scheduled payment methods, sealed-bid auctions, and lotteries.

In the following, we show how it is possible to achieve time-lock encryption functionality using μchain.

[16]Because the review messages (and especially insulting words) may have low entropy, it is advisable to use a hash function salted with a random value

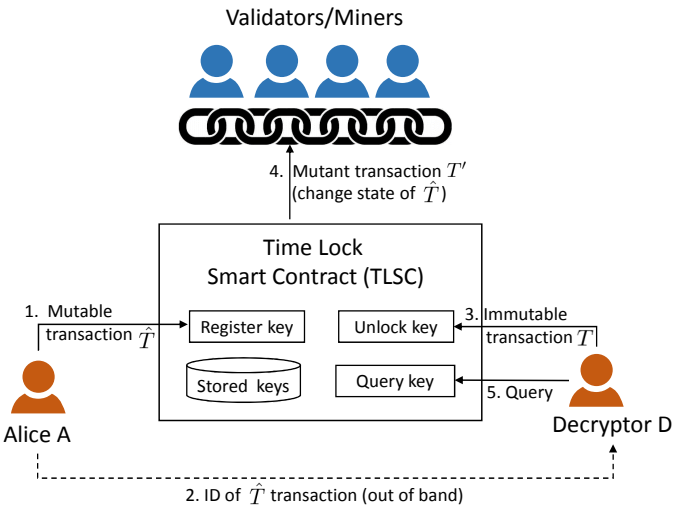

Figure 10: Time Lock Encryption: Architecture

Use Case Scenario We borrow a scenario from Liu et al. [52] who considered a use case very relevant to the research community. Assume that Alice would like to publish a paper disclosing a security vulnerability, however the vendor of the product tries to inhibit her. Instead, Alice can publish an encrypted version of her paper using a time-release encryption scheme. Her paper will only be decrypted when the deadline passes. During this waiting period, Alice cannot change her paper and the vendor cannot force her to change it, either.

Entities Our solution is realized by means of Time Lock Smart Contract (TLSC) that is residing within the μchain and can receive transactions and queries from regular blockchain users. We assume that there are two parties communicating with the contract: Alice A and the decryptor D. Alice encrypted her paper with the decryption key and would like to turn her key into the time-locked key. Decryptor is any other user of the μchain. For instance, he could be a program committee member of a conference where Alice submitted her paper, a conference participant, or even the product vendor.

Architecture The system architecture is depicted in Figure 10. To turn her decryption key K into the time-locked key, Alice sends the mutable transaction $\hat{T}$ to the Time Lock Smart Contract (step 1). The structure of the transaction is as follows: $\hat{T} = \{\mathcal{T}, a, P\}$, where $\mathcal{T}$ is a transaction set, a specifies active transaction and P defines mutability conditions. Here, $\mathcal{T}$ consists of three transactions$\{\tau_1, \tau_2, \tau_3\}$, where τ_1 and τ_2 are of classical (C) type and τ_3 is a *nope* transaction. τ_1 includes time t of key unlock in its data field, while the data field of τ_2 is populated by the decryption key K. Alice selects τ_1 as an active transaction by specifying $a = 1$. In the policy, she states that the transaction may be mutated by the TLSC contract, and the mutation is possible at any time after the deadline t. When processed by the contract's function that registers keys, transaction's identifier and the deadline t are recorded in the database of stored keys.

In a second step, Alice informs the public that the document she encrypted can be decrypted with the time-locked key. She also tells how to identify her key within the TLSC contract, by providing the key's identifier (or, more precisely, the identifier of the transaction set associated with the key). This information can be made public in the same way as her encrypted paper.

In a third step, decryptor D sends a regular (immutable) transaction T to the smart contract, which triggers a function to check if the deadline t has passed. If positive, the contract will issue a mutant transaction T' to mutate the active state of $\hat{T}$ to $a = 2$ (step 4). This effectively replaces the deadline t stored in contract's key database with the actual decryption key K. Since now on, decryptor D can query the key database to receive the actual key (step 5). Note, that if D would query the contract before the deadline t passes, he would receive the deadline t instead.

In the use case above, we demonstrated how μchain can be used to achieve typical time-locking functionality, e.g., to make a decryption key available after a certain deadline. Beyond that, our system can provide more advanced features, e.g., make the decryption key available only if requested within a certain time window, or enforce additional access control upon unlocking keys.

VII. Implementation

We developed a proof of concept of μchain using the Hyperledger Fabric [10], version *0.6.0-preview*. The Hyperledger Fabric is an open source project, mainly written in the Go programming language [24], that implements blockchain technology in a modular manner. Various modules can be combined in a lego style to achieve different properties of the blockchain, which gives the opportunity to blockchain deployers to select the configuration that matches their needs, e.g., to choose various plugins for consensus protocol or decide if the blockchain needs to be permissioned or permissionless. Also, it can be easily extended with new modules, which can originate from Hyperledger developers or be developed by third parties. Note however that Hyperledger Fabric does not have any associated cryptocurrency and the only transactions it natively supports are from users to smart contracts. Nevertheless its modularity and extendability made Hyperledger a very attractive platform for our prototype.

In the following, we first present details of our implementation and then discuss selected implementation aspects which needed special consideration during implementation.

A. Implementation Details

In figure 11 we show the overall architecture of Hyperledger Fabric (as presented by the Hyperledger project [53]) and highlight in orange[17] modules which were extended or modified in our implementation. Our implementation consists of 9066 lines of code, which extends the overall codebase of Hyperledger by 1.54%. Below we explain our modifications in more details.

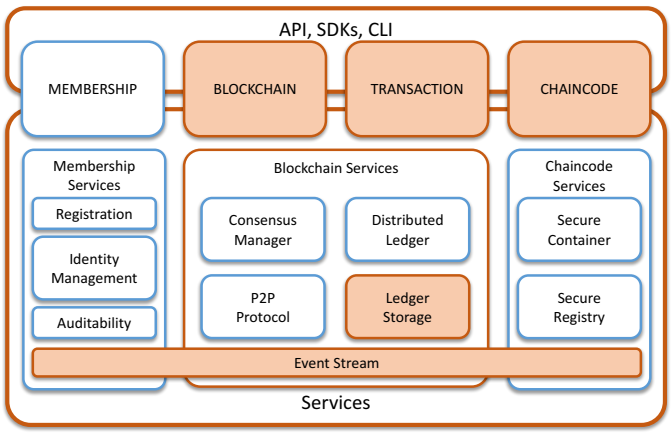

Figure 11: Modular breakdown of the Hyperledger Fabric, credit: [53]. The components affected by the implementation of μchain have been marked in orange[17].

Transaction The *Transaction* module was modified to introduce the notions of transaction sets and mutant transactions. In particular, two new transaction data structures have been defined for μchain: `TransactionSet` and `Mutant`, where `TransactionSet` has a type associated with it that could either be `Creation` or `Extension`. Since various interfaces of Hyperledger require a single type of transaction to be given as input, another data structure called `InBlockTransaction` was created to group all the possible transaction types. We do not specify a dedicated type for legacy (immutable) transactions. However, they can be issued by encapsulating them into a `TransactionSet` that only contains one transaction.

Chaincode Chaincode is the synonym used for "Smart Contract" in Hyperledger. Transaction mutations have been made possible by changing the `exectransaction.go` file in the chaincode module. We added two new functions called `GetAffectedTxs` and `ApplyMutations`. `GetAffectedTxs` returns the list of dependent transactions that are affected by the mutated transaction and, hence, need to be re-processed. `ApplyMutations` gets such a list as an input and starting from the oldest affected transaction, processes them again in their respective blocks, updating the state of each affected block as the execution goes on. The Hyperledger's `ExecuteTransactions` function is usually called to process a batch of transactions. We modified it in such a way that it first selects and executes all the mutant transactions from the given batch, and then calls `ApplyMutations` to bring into play their changes. Finally, it executes all the other transactions from the batch.

In Hyperledger, smart contracts are run by validators in

[17]Or in gray, when the paper is printed as grayscale.

a docker[18] virtualized environment. Potentially they can be written in any language as long as for that language there exists a layer that allows it to be interfaced with the blockchain. The operations that chaincode can perform on the blockchain are defined in the `ChaincodeStubInterface`. We added several new functions to this interface: `GetCurrentTxID`, `GetTransactionByID`, `Mutate` and `Extend`. `GetCurrentTxID` allows smart contracts to obtain the ID of the transaction that resulted in their invocation. This is useful as, at some later point, contracts might want to mutate (extend) one of the transactions they received: this can be done by calling `Mutate` (`Extend`) and providing the ID of the transaction to mutate (extend) as input. Similarly, `GetTransactionByID` is intended to allow chaincode to get information about transactions related to other smart contracts that they wish to mutate or extend.

Hyperledger Fabric allows to deploy a so called "System Chaincode" which is a smart contract designed to perform blockchain management operations. In our implementation we leveraged this type of smart contract to allow entities to associate an account policy P_a (cf. Section IV-B) to their address. Whenever a transaction set is to be executed, first the relevant access rights are checked in the system chaincode. Note that in Hyperledger only smart contracts can be at the receiving end of transactions. Hence, what we store in the system chaincode is just a policy and a mapping to a smart contract identifier. The System Chaincode can express conditions at the granularity of smart contract functions, so that it is possible to, e.g., specify that one function of the smart contract accepts mutable transactions, while another one only deals with immutable ones.

Event Stream The event stream component is responsible for the communication between various modules. We modified event stream to add new types of messages by modifying the definition of the `ChaincodeMessage` message type. This message is exchanged between the chaincode and the ledger storage. For instance it can be used in order to modify the ledger or get some information from it. The new types we introduced in `ChaincodeMessage` are: `MUTATE`, `EXTEND` and `OBTAIN_TRANSACTION`, which are related to the new functionalities `Mutate`, `Extend`, and `GetTransactionByID`, respectively, that we introduced in the chaincode module. Note that it was not necessary to define a new event for `GetCurrentTxID` function, as the transaction ID of the current transaction can be directly obtained by the chaincode from the chaincode stub.

Ledger Storage The ledger storage was extended to store information about current active transactions. For each mutable transaction issued we keep track of the current active index and at which block every mutant transaction was created and extended. This information forms the state of the *blockchain version*, this data structure is stored separately from the chaincode state and its hash is included into the block header. To manage operations related to the *blockchain version* state, we created an interface called `HashableTxSetState`, with similar properties to the Hyperledger's `HashableState`. The `HashableState` interface is implemented by various Hyperledger's packages that take care of organizing the `State` of the chaincode into the database. The new packages that implement the `HashableTxSetState` interface manage the *μchain version* deltas and allow the ledger to persist the changes to the validator's database.

Because μchain needs to work with current active transactions, we added a new method `GetCurrentDefault` in the `Ledger` class to retrieve the current active transaction of a transaction set. The method works by first querying the *μchain version* to acquire the active index of a transaction set, then fetches the correct transaction from the blockchain and returns a decrypted copy of it. Almost every part of the Fabric codebase that was previously dealing with transactions now handles transaction sets and has been modified to first get the current active transaction from the `Ledger` and then perform its operations using the active transaction.

To assist the `ApplyMutations` function, which we introduced in the chaincode module, the ledger storage provides new methods to enter in what we call *reset mode*. `ApplyMutations` can request the ledger to start resetting. When this happens, the ledger reverts the current blockchain state to a given block. `ApplyMutations` then can execute transactions from the current resetting block and when it finishes it can instruct the ledger to apply the changes from all the other transactions in that block. This process continues until `ApplyMutations` has re-executed all the affected transactions. Note that before the reset starts the ledger makes a snapshot of the current state of the blockchain, so that, if any irreversible error occurs while the mutations are being applied, the state of the blockchain can be easily rolled-back to the last valid state.

Blockchain In Hyperledger, every transaction is indexed so that it can be easily retrieved by querying it by its ID. In particular, a record is kept that maps transactions to their position in their issuing block. However in μchain sets can be extended, and as a consequence it could happen that transactions at different blocks belong to the same transaction set. Therefore, we modified the information stored by `addIndexDataForPersistence` in order to maintain, for every transaction set ID, a list of blocks that include extending transactions and their position within blocks.

The ledger class interfaces itself with the blockchain for multiple operations. Particularly, it can query transactions from it, or retrieve the current height of the blockchain. Some of the behaviors of the blockchain class had to be modified to account for the *reset mode* of the ledger. The height of the blockchain used to simply be the length of the current chain, in μchain on the other hand its behavior is different when mutations are being applied. This is necessary since some parts of Hyperledger need to know the current height of the

[18]https://www.docker.com/

chain. When resetting, in order to compute the right value, they should operate as if the blockchain is currently as long as the last mutated block, rather than the real length of the chain. For this purpose whenever the ledger is in *reset mode*, it communicates with the blockchain to let it be aware of the current reset status so that the blockchain can always provide the correct value.

Crypto The crypto module is not depicted in the Hyperledger architecture in Figure 11, yet it is present in its implementation as a separate module. We extended it to provide new functions in order to encrypt each transaction of a transaction set as discussed in Section III. We encrypt transactions by first serializing them and then encrypting the resulting bytes using AES in CBC mode and PKCS7 padding with a 256 bits long key. This process is done client-side for each transaction belonging to a set. Subsequently the transaction set is sent to the validators together with its accompanying keys.

We did not yet implement secret sharing in the crypto module, therefore currently our implementation only allows *simple* key management.

B. Special Aspects of Implementation

In this subsection we discuss special aspects of implementation, and in particular some problems which arose during implementation and our approach to tackle them.

a) Mutations and Verifiability: The first problem is related to the scenario where smart contracts mutate themselves (e.g., in vulnerability patching scenario), which, as we explain, may cause problems with the verifiability of the blockchain.

Mutations caused by smart contracts might yield an invalid *μchain version* state for clients, in the sense that they might end up with a different view of it from the validators. As an example, consider a smart contract A that upon receiving a transaction $\hat{T}$ induces a chain of transactions and invocations that ultimately causes the deployment transaction of A to be changed into the $nope$ transaction. Clients that try to parse the blockchain in the future have no way to verify that the current active transaction of the deployment transaction of A should be the $nope$ transaction because to do so they would need to execute $\hat{T}$ in A, but A does not exist anymore.

This is just a sample of a bigger problem: similar complications ensue whenever a smart contract tries to modify itself, in total or in part. Note that solving this by just forbidding smart contracts to modify themselves is not desirable. Furthermore, determining whether a mutation issued by a smart contract causes changes to the smart contract itself might not be possible a priori. Resources would need to be invested in executing the transaction just to determine whether or not it violates the rule, drastically diminishing the potential throughput of transactions as a consequence. In addition, this might limit the expressiveness of mutations performed through smart contracts.

To tackle this problem, we decided to record *all* the mutations caused by smart contracts in the block where they are originated, *without* applying them. Afterwards mutations queued in a block are applied at their succeeding block, before the execution of any other transaction starts. In this way clients parsing μchain can know what the current *μchain version* is, since they can always observe the changes made to the *blockchain version* state.

b) Recursive Mutations: The second problem, which we would like to highlight, is related to the situation where mutant transactions issued by a smart contract may lead to recursive mutations resulting in endless loops of re-execution.

To explain how a cycle of mutations can form, assume that A receives a mutant transaction $\hat{T}$ receiving 5 units of cryptocurrency. At some later point in time, it checks the balance, and if it equals or larger than 3, it mutates $\hat{T}$ to alternative version, which modifies the amount of received cryptocurrency to 5. Making such a mutation would lead to a new mutant transaction issued by the contract which would mutate $\hat{T}$ to an alternative version (so that the amount of the received cryptocurrency is 3 again), because the condition that the balance is equal or larger than 3 would still be satisfied. In this way, mutation would trigger the execution of an endless loop while in reset mode.

To avoid such loops, we prevent any mutations from taking effect while the system is in the *reset mode*. This countermeasure guarantees that execution will always end and the a definite *blockchain version* will always be reached. Moreover it does not hinder the flexibility of mutations, as they can always be triggered from the current state in order to reach the desired version.

VIII. Related Work

The idea of untraceable digital payments was first introduced by the seminal work of Chaum [54], which proposed to realize electronic cash using blind signatures. This cryptographic primitive would allow a bank to keep account balances of clients without being able to trace their money flow. Follow up works improved the system, allowing clients of different banks to interchange money, while keeping both the sender and its bank concealed from the recipient [55], efficiently divide anonymized coins [56], [57], and working even if the bank is offline at various points of the protocol [58], [59], and optimizing the overall complexity of the scheme [60], [61]. While presenting many improvements, all these schemes share a common drawback, a central point of trust which is necessary to guarantee that the blinded coins are not double spent: the bank. Moreover, none of these cryptocurrencies were ever able to reach a significant user base.

The most notable breakthrough in this regard was the introduction of Bitcoin [1]. Bitcoin relies neither on banks nor on any other central authority and seems to be the most successful cryptocurrency that is used for real world payments. It solves the problem of having a central book-keeper by broadcasting every transaction to every participant of a peer-to-peer network. Since everyone knows every transaction, it can be immediately verified whether a given "coin" has been already spent. Within a short time Bitcoin has been

thoroughly analyzed with regards to security [62], [63], [64], privacy issues [65], [66], [67] as well as economic aspects [68], [69]. Moreover, a number of alternative cryptocurrencies (*altcoins*) were proposed that have made substantial changes to initial Bitcoin design with different goals. For instance, Zerocoin [70], Zerocash [71], CryptoNote [72] and Pinnochio-Coin [73] aim at providing advanced anonymity, Litecoin [74] and Dogecoin [75] use "memory-hard" puzzles that prevent specialized hardware to give an edge over general purpose hardware in the mining process, moreover they tweak other parameters to influence the emission rate of new cryptocurrency, Primecoin [76] puts the massive amount of resources used to secure the blockchain to scientific use, as the same effort spent on securing the cryptocurrency also produces a sequence of large prime numbers as a side effect, while Ethereum [7] extends Bitcoin's transaction semantics to enable support for smart contracts. Enigma [77] builds upon Ethereum and provides confidentiality of smart contract's data through the application of Multi Party Computation (MPC).

However, none of the blockchains mentioned above provide any means to modify history of the blockchain or revert transactions. The only altcoin that makes an attempt to revert transactions is Reversecoin [78]. It works by setting accounts with two keys: one of these keys allows to issue regular transactions, that are revertible within a fixed amount of time from their emission with the use of the other key. This mechanism is merely intended to return cryptocurrency stolen by, e.g., malware targeting bitcoin wallets, but not to modify the blockchain history. All the reverted transactions remain untouched in the blockchain history and can be publicly viewed.

Möser et al. [79] implement "covenants" in Bitcoin, that is transactions that are able to pose limitations on the output of transactions that try to spend them. They use said mechanism to simulate the "reversion" of transactions by third parties, through what they call *poisonous transactions*. In analogy to the mechanics of Reversecoin, poisonous transactions work by locking the funds for a predefined time window. Therefore they do not modify the transaction history, but simply allow any third party to cancel transactions while they are locked.

Mutable Blockchains Parallel to our work, Ateniese et al. [81] presented a concept of redactable blockchain (RB) – the blockchain which can be modified by a trusted (potentially distributed) third party (TTP), which knows the secret key. To achieve this goal, RB uses a chameleon hash function during block mining – the crypto primitive which enables a party with the knowledge of the trapdoor key to efficiently calculate hash collisions. The chameleon hash function is applied to chain blocks together, which means that the TTP has the ability to arbitrarily modify the blockchain in non-accountable manner. In contrast, in μchain any blockchain mutation is governed "by fiat", which makes any mutation controllable (e.g., access control can be enforced) and also verifiable by the network. Further, RB strongly relies on confidentiality of a single trapdoor key, while our solution uses per-transaction keys and, hence, has better resilience to key compromise. We, however, believe that μchain could also benefit from using the chameleon hash crypto primitive when using it to hash transactions before they are compressed into the Merkle tree. We plan to explore this design option in our future work.

Blockchain-based recommendation systems Several previous works [82], [83], [84] proposed to instantiate recommendation systems using blockchain technology. Such systems distribute trust among service providers and give guarantees that all the reviews written by customers become public, so that it is not possible to, e.g., hide negative reviews. However, at the same time, reputation systems typically require some sort of censorship to hide inappropriate content like swearwords and inappropriate expressions. In contrast to these works, our system can provide censorship functions and even allow service providers to censor reviews published in the past.

Time-lock encryption From the very beginning, there were two lines of research aiming to solve the problem of time-lock encryption: one based on the computational complexity approach [85], [86], [52] and another one relying on trusted agents [85], [87], [88]. Both of them, however, have limitations. Using trusted agents has the obvious problem of ensuring that the agents are trustworthy, while in a computational approach the obstacle is that the CPU time required to recover a decryption key is typically dependent on the hardware. Therefore, it is challenging to reliably estimate the decryption time.

Recently, several works [89], [52] explored the idea of time-lock encryption using Bitcoin blockchain. Liu et al. [52], as opposed to Jager [89] provide an, albeit inefficient, implementation of their work, while Jager [89] gives formal security definitions and hence makes more rigorous security claims. They show a construction based on witness encryption scheme [90] that makes it possible to leverage the mining power of Bitcoin to compute the decryption key. Because the difficulty of Bitcoin computational puzzles is adjusted to available computational power in the network, it becomes possible to correctly estimate the time at which the decryption key will be generated. However, the current instantiations of witness encryption are far from being practical. While more efficient witness schemes can be discovered in the future, the current time for encryption and decryption is said to be astronomical [52]. Similarly to these works, our time-lock encryption solution can be used to release the decryption key in pre-defined time and without any assumptions made on computational power available in the underlying network. Furthermore, our scheme is efficient, as encryption and decryption can be done using efficient cyphers like Advanced Encryption Standard (AES).

IX. Conclusion

In this paper, we investigated the idea of making blockchain history mutable. In particular, we proposed the design and

implementation of μchain, a mutable blockchain, which integrates new mechanisms enabling the removal and modification of blockchain data records. All modifications in μchain are performed using transactions of a special type, meta transactions, so that unauthorized modifications are rejected in the same way as invalid transactions, while legitimate modifications are verifiable by blockchain operators and regular users. The μchain design yields alternative version histories that are recorded in the blockchain and ensures that only an active history, agreed upon by a consensus, is accessible by users. To hide alternative histories, μchain relies on encryption and supports two variants of key management that achieve confidentiality towards regular users only, or towards both regular users and blockchain operators.

We prototyped μchain using the Hyperledger Fabric open source project (powered by the Linux foundation) and presented details of our implementation. We further showed how μchain could be used to patch vulnerabilities in smart contracts without imposing hard forks and without disrupting of blockchain operations. To demonstrate μchain's flexibility as a platform for blockchain applications, we designed a collaborative recommendation system with censorship and a time-lock encryption – applications which could not be instantiated using immutable blockchains.

REFERENCES

[1] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," Technical Report, 2008, http://www.vsewiki.cz/images/archive/8/89/20110124151146!Bitcoin.pdf.

[2] M. Araoz, "Proof of existence," https://www.proofofexistence.com/.

[3] B. Gipp, N. Meuschke, and A. Gernandti, "Decentralized trusted timestamping using the crypto currency Bitcoin," in *iConference 2015*, 2015.

[4] "Storj. Decentralized cloud storage," https://storj.io/.

[5] "Blockstack. The blockchain application stack," https://blockstack.org/.

[6] "Keybase. Public key crypto for everyone, publicly auditable proofs of identity," https://keybase.io/.

[7] V. Buterin, "A next-generation smart contract and decentralized application platform," http://www.the-blockchain.com/docs/Ethereum_white_paper-a_next_generation_smart_contract_and_decentralized_application_platform-vitalik-buterin.pdf, 2014.

[8] "Rootstock. Smarter Bitcoin," http://www.rsk.co/.

[9] "Multichain. Open source private blockchain platform," http://www.multichain.com/.

[10] "Hyperledger project," https://www.hyperledger.org/.

[11] "ERIS: The smart contract application platform," https://erisindustries.com/.

[12] "Ripple. Join the global settlement network," https://ripple.com/.

[13] A. Lebo, "Implementation of a decentralized, transferable, and open software license system using the Bitcoin protocol," in *Github project*, 2014.

[14] J. Herbert and A. Litchfield, "A novel method for decentralised peer-to-peer software license validation using cryptocurrency blockchain technology," in *Australasian Computer Science Conference*, 2015.

[15] C. Thompson, "The top 10 blockchain startups to watch in 2016: The leaders who are changing the game," https://medium.com/the-intrepid-review/the-top-10-blockchain-startups-to-watch-in-2016-the-leaders-who-are-changing-the-game-6195606b0d70#.kqqetdcmp, 2016.

[16] M. Jakobsson and A. Juels, "Proofs of work and bread pudding protocols," in *IFIP TC6/TC11 Joint Working Conference on Secure Information Networks: Communications and Multimedia Security*, 1999.

[17] J. Mathew, "Bitcoin: Blockchain could become 'safe haven' for hosting child sexual abuse images," http://www.dailydot.com/business/bitcoin-child-porn-transaction-code/, 2015.

[18] K. Shirriff, "Hidden surprises in the bitcoin blockchain and how they are stored: Nelson mandela, wikileaks, photos, and python software," http://www.righto.com/2014/02/ascii-bernanke-wikileaks-photographs.html, 2014.

[19] V. Buterin, "Thinking about smart contract security," https://blog.ethereum.org/2016/06/19/thinking-smart-contract-security/.

[20] C. Jentzsch, "Decentralized autonomous organization to automate governance," https://download.slock.it/public/DAO/WhitePaper.pdf.

[21] J. I. Wong, "A $79 million cryptocurrency heist just happened, and it's threatening the future of blockchains," http://qz.com/710126/a-massive-79-million-heist-just-happened-and-its-threatening-the-future-of-blockchains/.

[22] "The EU general data protection regulation," 2016, http://www.allenovery.com/SiteCollectionDocuments/Radical%20changes%20to%20European%20data%20protection%20legislation.pdf.

[23] "Solidity smart-contract language," https://solidity.readthedocs.io/.

[24] "The Go programming language," https://golang.org/.

[25] R. C. Merkle, "A certified digital signature," in *Advances in Cryptology - CRYPTO. Annual International Cryptology Conference*, 1989.

[26] C. Dwork and M. Naor, "Pricing via processing or combatting junk mail," in *Advances in Cryptology - CRYPTO. Annual International Cryptology Conference*, 1993.

[27] S. King and S. Nadal, "PPCoin: Peer-to-peer crypto-currency with Proof-of-Stake (white paper)," https://peercoin.net/assets/paper/peercoin-paper.pdf, 2012.

[28] M. Castro and B. Liskov, "Practical byzantine fault tolerance," in *Symposium on Operating Systems Design and Implementation*. USENIX Association, 1999.

[29] A. Batlin, H. Jaffrey, C. Murphy, A. Przewloka, and S. Williams, "Building the trust engine. how the blockchain could transform finance (and the world)," https://www.ubs.com/microsites/blockchain-report/en/home/_jcr_content/mainpar/gridcontrol_3/col2/actionbutton_769218921.828358967.file/bGluay9wYXRoPS9jb250ZW50L2RhbS91YnMvbWljcm9zaXRlcy9ibG9ja2NoYWluL3doaXRlcGFwZXItMTkwNTE2LnBkZg==/whitepaper-190516.pdf, A UBS Group Technology, Tech. Rep., 2016.

[30] S. Tual, "Upgrading The DAO to framework 1.1: a step by step guide," https://blog.slock.it/upgrading-the-dao-to-framework-1-1-a-step-by-step-guide-547a58e00137#.vsk8fim4c.

[31] E. G. Sirer, "Thoughts on The DAO hack," http://hackingdistributed.com/2016/06/17/thoughts-on-the-dao-hack/.

[32] S. Tual, "No DAO funds at risk following the Ethereum smart contract 'recursive call' bug discovery," https://blog.slock.it/no-dao-funds-at-risk-following-the-ethereum-smart-contract-recursive-call-bug-discovery-29f482d348b#.d6kxxza9i.

[33] "Merkle - Patricia tree description," https://github.com/ethereum/wiki/wiki/Patricia-Tree.

[34] D. R. Morrison, "PATRICIA – Practical Algorithm To Retrieve Information Coded in Alphanumeric," *J. ACM*, vol. 15, no. 4, 1968.

[35] A. Shamir, "How to share a secret," *Communications of the ACM*, vol. 22, no. 11, 1979.

[36] R. Ostrovsky and M. Yung, "How to withstand mobile virus attacks (extended abstract)," in *Proceedings of the Tenth Annual ACM Symposium on Principles of Distributed Computing*, ser. PODC '91. New York, NY, USA: ACM, 1991, pp. 51–59. [Online]. Available: http://doi.acm.org/10.1145/112600.112605

[37] J. Baron, K. E. Defrawy, J. Lampkins, and R. Ostrovsky, "How to withstand mobile virus attacks, revisited," Cryptology ePrint Archive, Report 2013/529, 2013, http://eprint.iacr.org/2013/529.

[38] ——, "Communication-optimal proactive secret sharing for dynamic groups," Cryptology ePrint Archive, Report 2015/304, 2015, http://eprint.iacr.org/2015/304.

[39] L. Luu, V. Narayanan, C. Zheng, K. Baweja, S. Gilbert, and P. Saxena, "A secure sharding protocol for open blockchains," in *ACM Conference on Computer and Communications Security*. ACM, 2016.

[40] J. R. Douceur, "The sybil attack," in *Revised Papers from the First International Workshop on Peer-to-Peer Systems*. Springer-Verlag, 2002.

[41] Z. A. Wen and A. Miller, "Scanning live Ethereum contracts for the "unchecked-send" bug," http://hackingdistributed.com/2016/06/16/scanning-live-ethereum-contracts-for-bugs/.

[42] I. Damgård, Y. Ishai, M. Krøigaard, J. B. Nielsen, and A. Smith, *Scalable Multiparty Computation with Nearly Optimal Work and Resilience*.

Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 241–261. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-85174-5_14

[43] A. Beimel, "Secret-sharing schemes: A survey," in *International Conference on Coding and Cryptology*, 2011.

[44] S. Dolev, K. ElDefrawy, J. Lampkins, R. Ostrovsky, and M. Yung, "Proactive secret sharing with a dishonest majority," in *Conference on Security and Cryptography for Networks*, 2016.

[45] M. Franklin and M. Yung, "Communication complexity of secure computation (extended abstract)," in *ACM Symposium on Theory of Computing*. ACM, 1992.

[46] "Tripadvisor: Read reveiws, compare & book," https://www.tripadvisor.com/.

[47] "HRS – hotel reservation service," https://www.hrs.com/web.

[48] "Yelp – user reviews and recommendations of top restaurants, shopping, nightlife, entertainment, services and more," https://www.yelp.com/.

[49] "Amazon – electronic commerce and cloud computing," https://www.amazon.com.

[50] F. Kerschbaum, "A verifiable, centralized, coercion-free reputation system," in *ACM Workshop on Privacy in the Electronic Society*. ACM, 2009.

[51] T. May, "Time-release crypto. Manuscript," 1993, https://www.gwern.net/Self-decrypting%20files.

[52] J. Liu, S. A. Kakvi, and B. Warinschai, "Extractable witness encryption and timed-release encryption from Bitcoin," Cryptology ePrint Archive, Report 2015/482, 2015, https://eprint.iacr.org/2015/482.pdf.

[53] "Hyperledger fabric. Protocol specification, architecture," http://hyperledger-fabric.readthedocs.io/en/latest/protocol-spec/#21-architecture.

[54] D. Chaum, "Blind signatures for untraceable payments," in *Advances in Cryptology - CRYPTO. Annual International Cryptology Conference*. Springer US, 1983.

[55] A. Lysyanskaya and Z. Ramzan, *Group blind digital signatures: A scalable solution to electronic cash*. Springer Berlin Heidelberg, 1998.

[56] T. Nakanishi and Y. Sugiyama, *Unlinkable Divisible Electronic Cash*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2000, pp. 121–134. [Online]. Available: http://dx.doi.org/10.1007/3-540-44456-4_10

[57] T. Okamoto and K. Ohta, "Universal electronic cash," in *Advances in Cryptology - CRYPTO. Annual International Cryptology Conference*, J. Feigenbaum, Ed. Springer Berlin Heidelberg, 1992.

[58] D. Chaum, A. Fiat, and M. Naor, "Untraceable electronic cash," in *Proceedings on Advances in cryptology*. Springer-Verlag New York, Inc., 1990, pp. 319–327.

[59] A. de Solages and J. Traoré, "An efficient fair off-line electronic cash system with extensions to checks and wallets with observers," in *Financial Cryptography and Data Security*, 1998.

[60] S. A. Brands, "An efficient off-line electronic cash system based on the representation problem." CWI (Centre for Mathematics and Computer Science), Amsterdam, The Netherlands, Tech. Rep., 1993.

[61] J. Camenisch, S. Hohenberger, and A. Lysyanskaya, "Compact e-cash," in *Advances in Cryptology - EUROCRYPT. International Conference on the Theory and Applications of Cryptographic Techniques*, 2005.

[62] G. O. Karame, E. Androulaki, and S. Capkun, "Double-spending attacks on fast payments in Bitcoin," *ACM Conference on Computer and Communications Security*, 2012.

[63] S. Barber, X. Boyen, E. Shi, and E. Uzun, "Bitter to better – how to make Bitcoin a better currency," in *Financial Cryptography and Data Security*, 2012.

[64] I. Eyal and E. G. Sirer, "Majority is not enough: Bitcoin mining is vulnerable," in *Financial Cryptography and Data Security*, 2014.

[65] E. Androulaki, G. Karame, M. Roeschlin, T. Scherer, and S. Capkun, "Evaluating user privacy in Bitcoin," in *Financial Cryptography and Data Security*, 2013.

[66] D. Ron and A. Shamir, "Quantitative analysis of the full Bitcoin transaction graph," *Financial Cryptography and Data Security*, 2012.

[67] S. Meiklejohn, M. Pomarole, G. Jordan, K. Levchenko, D. McCoy, G. M. Voelker, and S. Savage, "A fistful of Bitcoins: Characterizing payments among men with no names," in *Conference on Internet Measurement Conference*, 2013.

[68] M. Babaioff, S. Dobzinski, S. Oren, and A. Zohar, "On Bitcoin and red balloons," in *ACM Conference on Electronic Commerce*, 2012.

[69] J. A. Kroll, I. C. Davey, and E. W. Felten, "The economics of Bitcoin mining or, bitcoin in the presence of adversaries," *Workshop on the Economics of Information Security*, 2013.

[70] I. Miers, C. Garman, M. Green, and A. D. Rubin, "Zerocoin: Anonymous distributed e-cash from Bitcoin," in *2013 IEEE Symposium on Security and Privacy*, 2013.

[71] E. B. Sasson, A. Chiesa, C. Garman, M. Green, I. Miers, E. Tromer, and M. Virza, "Zerocash: Decentralized anonymous payments from Bitcoin," in *IEEE Symposium on Security and Privacy*, 2014.

[72] N. van Saberhagen, "Cryptonote v 2. 0 (white paper)," https://cryptonote.org/whitepaper.pdf, 2013.

[73] G. Danezis, C. Fournet, M. Kohlweiss, and B. Parno, "PinocchioCoin: building Zerocoin from a succinct pairing-based proof system," in *PETShop*, 2013.

[74] "C. Lee, Litecoin," https://litecoin.org/.

[75] P. Jackson and N. Shibetoshi, "Dogecoin," http://dogecoin.com/.

[76] S. King, "Primecoin: Cryptocurrency with prime number proof-of-work," 2013, http://academictorrents.com/details/d0f9accaec8ac9d538fdf9d675105ae1392ea32b.

[77] G. Zyskind, O. Nathan, and A. Pentland, "Enigma: Decentralized computation platform with guaranteed privacy," *CoRR*, vol. abs/1506.03471, 2015.

[78] O. Challa, "Reversecoin: Worlds first cryptocurrency with reversible transactions (white paper)," https://docs.google.com/document/d/1hMCkEQUYm9oFCQpxtIWFqVpt66pTQn1zCDW8WX0b7hw/edit#.

[79] M. Möser, I. Eyal, and E. G. Sirer, "Bitcoin covenants," in *3rd Workshop on Bitcoin and Blockchain Research*, 2016.

[80] G. Ateniese, B. Magri, D. Venturi, and E. Andrade, "Redactable blockchain, or rewriting history in Bitcoin and friends," *Cryptology ePrint Archive: Report 2016/757*, 2016.

[81] R. Dennis and G. Owen, "Rep on the block: A next generation reputation system based on the blockchain," in *International Conference for Internet Technology and Secured Transactions*, 2015.

[82] A. Schaub, R. Bazin, O. Hasan, and L. Brunie, "A trustless privacy-preserving reputation system," in *IFIP SEC - Privacy*, 2016.

[83] D. Carboni, "Feedback based reputation on top of the Bitcoin blockchain," *CoRR*, vol. abs/1502.01504, 2016.

[84] R. L. Rivest, A. Shamir, and D. A. Wagner, "Time-lock puzzles and timed-release crypto," Massachusetts Institute of Technology, Cambridge, MA, USA, Tech. Rep., 1996.

[85] D. Boneh and M. Naor, "Timed commitments," in *Proceedings of the 20th Annual International Cryptology Conference on Advances in Cryptology*, ser. CRYPTO '00. London, UK, UK: Springer-Verlag, 2000, pp. 236–254. [Online]. Available: http://dl.acm.org/citation.cfm?id=646765.704125

[86] G. Di Crescenzo, R. Ostrovsky, and S. Rajagopalan, "Conditional oblivious transfer and timed-release encryption," in *Advances in Cryptology - EUROCRYPT. International Conference on the Theory and Applications of Cryptographic Techniques*. Springer-Verlag, 1999.

[87] J. Cathalo, B. Libert, and J. Quisquater, "Efficient and non-interactive timed-release encryption," in *International Conference on Information and Communication Systems*, 2005.

[88] T. Jager, "How to build time-lock encryption," Cryptology ePrint Archive, Report 2015/478, 2015, https://eprint.iacr.org/2015/478.pdf.

[89] S. Garg, C. Gentry, A. Sahai, and B. Waters, "Witness encryption and its applications," in *ACM Symposium on Theory of Computing*. ACM, 2013.