TÉCNICO
LISBOA

# Recoverable Token

Recovering from Intrusions against Digital Assets in Ethereum

**Filipe Miguel Fernandes Martins**

Thesis to obtain the Master of Science Degree in

## Information Systems and Computer Engineering

Supervisor(s): Prof. Miguel Filipe Leitão Pardal
Prof. Miguel Nuno Dias Alves Pupo Correia

## Examination Committee

Chairperson: Prof. Paolo Romano
Supervisor: Prof. Miguel Nuno Dias Alves Pupo Correia
Member of the Committee: Dr. Fábio André Castanheira Luís Coelho

**January 2021**

In loving memory of my father...

## Acknowledgments

I would like to begin by thanking my advisors, Professor Miguel Pardal and Professor Miguel Correia, for their continuous support and guidance in the course of this dissertation. They are, without a doubt, the main reason as for why I was able to get to this point. Also, a special thanks to Doctor David Matos for accompanying this work, his knowledge and expertise were truly appreciated.

This definitely was one of the biggest challenges in my life so far. It required a lot of resilience, fortitude, and of course, hard work. In the five years that led to this moment, there were a lot of times where I doubted the path I was taking. However, looking back, I realize that the experiences I had – either good or bad – taught me lessons that I would not have learned anywhere else. All the friends I made, the all-nighters we went through to meet deadlines, the events we attended; these are things that I will always fondly remember.

Last but not least, I would like to thank my family for always believing in me and for pushing me to better myself every single day. They always helped me however they could and for that, there are no words that could even begin to express my gratitude. To you all – thank you.

# Resumo

Os sistemas de *blockchain* permitem armazenar bens digitais de uma forma descentralizada num registo infalsificável, imutável e com um mecanismo de consenso onde nenhum participante tem controlo total. Uma *blockchain* é um registo imutável que permite anexar novas transações. Infelizmente, em alguns casos existe a necessidade de reverter transações que são consequência de intrusões, por exemplo, quando as chaves privadas de uma carteira são roubadas, quando um dos participantes não cumpre com o que foi acordado, ou quando vulnerabilidades em *smart contracts* são exploradas por atacantes. Para além disso, existem cenários acidentais como p. ex., perda das chaves privadas que deixam os bens digitais que lhe estão associados – permanentemente inacessíveis. Embora existam algumas propostas que permitem modificar a *blockchain*, elas quebram garantias básicas que são esperadas por sistemas suportados por ela. Neste trabalho, propomos uma nova abordagem para permitir a utilizadores recuperar de ataques ou perda acidental dos seus bens digitais e que ao mesmo tempo assegura que as propriedades fundamentais da *blockchain* não são comprometidas. Este mecanismo foi implementado e avaliado na plataforma Ethereum / EVM, tendo sido concluído que é possível fazer a recuperação dos bens digitais por um preço relativamente baixo, quando se considera que até agora, não era possível fazer este tipo de operações de uma forma rápida e incontroversa.

**Palavras-chave:** recuperação de intrusões, blockchain, smart contracts, criptomoedas, tokens, Ethereum

## Abstract

*Blockchain* systems allow storing digital assets in a tamper-proof, consensus-based, append-only ledger in a decentralized fashion, where no single party has full control. A *blockchain* is an immutable, append-only, log of transactions. Unfortunately, in some cases it is necessary to *undo* transactions that result from intrusions, e.g., when the private keys of a wallet are stolen, when one of the transaction participants does not comply with what was agreed upon, or when smart contract vulnerabilities are exploited by attackers. There are also accidental scenarios, e.g., when private keys are lost leaving the associated digital assets inaccessible. Although there have been a few proposals which allow modifications to the *blockchain*, they break the basic guarantees they are supposed to provide. We propose an approach to allow wallet owners to recover from attacks against their digital assets and accidental loss, while still assuring fundamental properties of the *blockchain* technology. We implemented and evaluated the mechanism for Ethereum / EVM, showing that it is possible to perform these types of operations in a fast and noncontroversial manner.



**Keywords:** intrusion recovery, blockchain, smart contracts, cryptocurrency, tokens, Ethereum

# Contents


Acknowledgments . . . v

Resumo . . . vii

Abstract . . . ix

List of Tables . . . xiii

List of Figures . . . xv

**1 Introduction** **1**

1.1 Blockchain wallet challenges . . . 2

1.2 Recoverable tokens . . . 3

1.3 Objectives . . . 3

1.4 Thesis Outline . . . 4

**2 Related Work** **5**

2.1 Intrusion Recovery . . . 5

2.2 Blockchain . . . 9

2.2.1 Blockchain . . . 10

2.2.2 Decentralization and consensus . . . 11

2.2.3 Cryptocurrencies . . . 11

2.2.4 Transaction Lifecycle . . . 13

2.2.5 Summary . . . 14

2.3 Blockchain Recovery . . . 15

2.3.1 Forking . . . 15

2.3.2 Data Redaction . . . 17

2.3.3 Transaction Reversion . . . 18

2.3.4 Summary . . . 20

2.4 Ethereum Platform . . . 21

2.4.1 Ethereum Virtual Machine . . . 21

2.4.2 Smart contracts . . . 22

2.4.3 ERC-20 and ERC-721 Tokens . . . 22
2.4.4 Initial Coin Offerings . . . 23
2.4.5 ERC-1080 . . . 25
2.4.6 Dispute Resolution . . . 26
2.5 Summary . . . 27

**3 Recoverable Token** **29**
3.1 Recoverable Token Architecture . . . 29
3.1.1 Attack Model . . . 29
3.2 Recovery process . . . 31
3.2.1 Initial configuration . . . 32
3.2.2 Submitting a claim . . . 33
3.2.3 Dispute resolution . . . 36
3.2.4 Token recovery . . . 37
3.3 Recoverable Token Application . . . 38
3.4 Implementation . . . 39
3.5 Summary . . . 41

**4 Evaluation** **43**
4.1 Evaluation Methodology . . . 43
4.2 Source code . . . 44
4.3 Gas consumption . . . 46
4.4 Time consumption . . . 48
4.5 Summary . . . 50

**5 Conclusions** **51**
5.1 Achievements . . . 51
5.2 Future Work . . . 51
5.2.1 Privacy . . . 51
5.2.2 Decentralization . . . 52
5.3 Final remarks . . . 52

**Bibliography** **53**

# List of Tables

2.1 Intrusion Recovery systems . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 9
2.2 Blockchain Recovery systems . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 20

3.1 Public methods of the RCVToken, Claims and Profiles contracts. . . . . . . . . . . 31

4.1 Recoverable Token contracts' source code metrics. . . . . . . . . . . . . . . . . . . 44
4.2 Gas breakdown for the Loss, Theft and Chargeback dispute resolution scenarios. 47
4.3 Time breakdown for the Loss, Theft and Chargeback dispute resolution scenarios. 49

# List of Figures

2.1 Bitcoin block structure . . . . . 10
2.2 Blockchain transaction lifecycle . . . . . 13
2.3 Fork taxonomy . . . . . 15

3.1 Recoverable Token architecture . . . . . 30
3.2 Loss claim sequence diagram . . . . . 33
3.3 Theft claim sequence diagram . . . . . 34
3.4 Chargeback claim sequence diagram . . . . . 35
3.5 RCV App . . . . . 38
3.6 Smart contracts class diagram . . . . . 39

4.1 Average gas contribution . . . . . 46
4.2 Average time contribution . . . . . 48

# Chapter 1

# Introduction

Blockchain technology has been gaining popularity in the last decade with the rise of interest in cryptocurrencies such as *Bitcoin* [1] and *Ether* [2]. The initial goal was to have fast and cheap monetary transactions without involving a trusted third party, a role that is currently performed by banks and other financial institutions. As the technology matured, more use cases were discovered in several fields such as healthcare [3], business processes [4], and educational certificates [5].

The Ethereum Virtual Machine (EVM) allows running low-level machine code in the form of EVM bytecode. Developers can program *smart contracts* using high-level languages such as Solidity[1] or Vyper[2], compile them into bytecode and deploy them on the Ethereum blockchain. These smart contracts are analogous to objects in object-oriented programming, as they have attributes that define their state and methods that allow changing that state. Essentially they are immutable programs that run deterministically in an EVM context and its execution is triggered through *transactions* (akin to method calls).

In the *Ethereum* blockchain, wallets store keys that provide access to accounts. These accounts are associated with *Ether*, Ethereum's intrinsic cryptocurrency, handled at the protocol level, and optionally to *tokens* [6, 7, 8] that are handled at the smart contract level.

Tokens are frequently used to represent private currencies or value (e.g., capital stock), although they may also serve other purposes, e.g., representing voting rights, collectibles, identity, ownership of resources or other types of digital assets. As of December 2020, the top 10 tokens implemented over the Ethereum blockchain hold a market capitalization of over $38 billion (USD)[3]. There are a set of standard interfaces for tokens that can be used in many contexts, e.g., ERC-20 and ERC-721 [6, 7]. ERC-20 defines a standard interface for *fungible* tokens while ERC-

[1] Solidity documentation – `https://solidity.readthedocs.io/en/latest/`
[2] Vyper documentation – `https://vyper.readthedocs.io/en/latest`
[3] CoinMarketCap Top 100 Tokens by Market Capitalization – `https://coinmarketcap.com/tokens/`

721 targets *non-fungible* tokens. With such large amounts of value being exchanged, security and recovery mechanisms are indispensable for token management.

## 1.1 Blockchain wallet challenges

The learning curve for using the *blockchain* technology is still relatively high for non tech-savvy users. A minimum requirement is to have a set of asymmetric key pairs – each key pair has a public and a private key – which are used to sign and verify transactions that are then submitted into the blockchain. The *private key* is an especially sensitive piece of data that requires strong protection since it gives the holder of the key full access to the corresponding blockchain account. Digital wallets are used to store and manage these key pairs. That is to say, having access to a wallet will result in having access to all the digital assets associated with its accounts. Handling such keys may be complicated but wallet software is improving daily, so is accessibility and usability. This allows more and more users to make use of blockchain to perform money transfers and store both information and value.

In regards to Ethereum, when creating an account using a wallet software, an Ethereum address is derived from the generated public key. This Ethereum address is what uniquely identifies the wallet owner in the Ethereum blockchain. In order to interact with the Ethereum network, the *private* key is used to digitally *sign* transactions and the *public* key is used to *verify* the integrity and authenticity of those transactions. Unfortunately, if the *private key* is *lost* then it is no longer possible to interact with the network using that specific account and all the resources linked to it such as *Ether* or *tokens* become *permanently inaccessible*. This may happen for several reasons, from laptop/smartphone loss or theft, to ransomware that denies access to the user's files.

Most wallet software generates deterministic wallets [9] meaning that the key pairs are derived from *seed phrases*: lists of 12 to 24 words that allow users to recreate both the public and private keys. However, the user may never store their seed phrases in paper or digitally, so they may be unavailable when they are needed for recovery purposes.

Recovery mechanisms are still scarce in the blockchain domain and if there are no fallbacks, users can be led to avert joining and using the blockchain along with all the features it can provide.

## 1.2 Recoverable tokens

The work presented in this dissertation addresses the issue of the current lack of recovery mechanisms present in the blockchain domain. More specifically, recovering digital assets in the form of Ethereum tokens.

With this in mind, we present the first full solution for *recovering tokens* implemented in Ethereum. Ethereum smart contracts allow implementing arbitrary applications that may have different forms of operating and interacting with the external world e.g., using clients that are not wallets [5]). Therefore, we do not aim to recover arbitrary applications, but tokens. Notice that although we often refer to Ethereum, our solution applies to other blockchains that run EVM, e.g., Ethereum Classic, TRON, Cardano, Ropsten; and private blockchains based on clients that run EVM, e.g., Quorum, Hyperledger Besu and Pantheon. Our intrusion recovery approach is even more generic and applies to many other blockchains.

The approach taken involves a blockchain-based *dispute resolution mechanism* used to determine if a recovery request should be executed. To perform a *recovery action*, a claimant first has to submit a claim that becomes a dispute. If the claim is supported, the recovery action is executed. Our solution does not require any changes to the underlying blockchain protocol and it does not involve changes to the chain of blocks, so it does not break immutability. Instead, the recovery happens in a smart contract that manages the balance of tokens for each account.

## 1.3 Objectives

The major objective of this work is to *improve the usability of the blockchain technology by providing a way of creating an opportunity for its users to be able to recover their tokens.* When considering the ever increasing number of token transactions [4] being performed on top of the Ethereum blockchain, we believe this to be a necessary step for increasing the adoption of the blockchain technology.

The *Recoverable Token* we propose allows users to recover their Ethereum tokens in the following scenarios:

- *S1: account loss* – user lost the access to the private key of an account and/or corresponding seed phrases and can no longer recover them;
- *S2: account theft* – there is reasonable proof to believe that an account has been compromised;

[4]ERC-20 Daily Token Transfer Chart – `https://etherscan.io/chart/tokenerc-20txns`

- *S3: chargebacks* – payment is made for any good or service and the payer believes that it did not receive what was agreed upon.

While it may seem a trivial problem to solve, the fact that this feature is implemented in a decentralized domain brings several issues which need to be addressed:

- Who should be able to request token recoveries?
- How can we ensure that all participating entities agree to any recovery action being performed?
- How can we recover the tokens belonging to an account if it was lost or stolen?
- How can we validate the authenticity of a recovery request?

Our contribution is the design and implementation of a system that allows smart contract developers to add such features while also giving a solution to the above-mentioned issues.

## 1.4 Thesis Outline

The remainder of the document is structured as follows. Chapter 2 introduces key concepts and previous research in topics related to intrusion recovery and blockchains. The next chapter describes the type of problems addressed and the proposed solution along with the core concepts of its architecture and implementation. The methodology used to evaluate the solution is outlined in Chapter 4, along with a presentation of the obtained results. Finally, Chapter 5 provides a summary and concludes suggesting future improvements that could be applied to our work.

# Chapter 2

# Related Work

In this chapter we present work from different domains that is relevant for our proposal while also giving a brief introduction to each of them. We start by providing an overview of *Intrusion Recovery* in Section 2.1. In Section 2.2 we briefly introduce *Blockchain* technology and explain its major features and limitations. Then, in Section 2.3, we give insight on how the previous concepts can be connected for *Blockchain Recovery*. Finally, in Section 2.4, we outline the features that differentiate *Ethereum* from other blockchains and why it is the platform of choice for our solution.

## 2.1 Intrusion Recovery

Every system has vulnerabilities which are a part of a system's attack surface. The attack surface is the sum of the attack vectors, i.e., the points in a system where it can be compromised. The fact that systems can be compromised creates the need for techniques to recover from intentional attacks or in other words, *intrusions*. Intrusion Recovery mechanisms are a set of procedures that are applied to a system after an intrusion was detected to remove its malicious effects.

There are three main fault tolerance techniques regarding error handling, which can be adapted to intrusion recovery scenarios, according to Avizienis et al. [10]:

- **Rollback**: Bring the system back to a previous recorded state, preferably to a time before the intrusion ever ocurred. This is usually achieved using *snapshot* or *checkpoint* mechanisms.

- **Rollforward**: From a state in which the intrusion was detected, fix the error and allow the system to make *forward progress* and bring it to a state in where the intrusion never occurred.

- **Compensation**: Mask the intrusion by executing a set of *compensating actions* that undo its effects.

These techniques are not mutually exclusive, in fact it is desirable to be able to combine them in order to recover from an incorrect system state.

One of the first works related to this topic that combines the previously mentioned techniques is Undo for Operators [11]. Its main goal was to build a system that allows system administrators to recover from accidental mistakes or data corruption without suffering legitimate data loss and without modifying the target application's source code. An email store was used as an example of software that is able to benefit from the use of the tool. The approach used is based on the *three R's model: Rewind, Repair, Replay.* Essentially, a log of user interactions is recorded and when a system recovery is necessary, the system's state is physically rewound to a previous state through the use of a *snapshot* mechanism. The administrator then performs the necessary repairs followed by a replay phase in which only the legitimate interactions are re-executed, therefore allowing the system to recover from the error while also minimizing data loss. The drawbacks of this approach are that, since it relies on a proxy to intercept and log user interactions, a set of operations have to be previously defined causing it to be moderately application dependent. Furthermore, it can only recover from well-known predictable intrusions as the proxy needs to be configured in order to intercept them. However, one of the most relevant contributions is the approach taken to deal with the issue of *external inconsistencies* through the use of application-specific *compensating actions.* In the email store implementation, most of the compensating actions consist of messages to the user explaining why the inconsistencies were observed and what actions were taken.

Taser [12] also has similar goals, but the solution is more generic and uses a technique known as *taint analysis* to record dependencies between file-system operations. Additionally, it implemented automatic conflict resolution through the use of *resolution policies.* A standard recovery process using Taser begins by having a system administrator or an Intrusion Detection System point out a set of objects that were potential sources of the intrusion or were affected by it. These objects are fed into a *tracing algorithm* that outputs a set of objects from which the administrator then selects the ones he deems to be the source of the intrusion. As a consequence, the selected objects become *tainted.* Next comes the *propagation phase* where a causal dependency graph for the tainted objects is computed. A set of previously defined dependency rules are used to propagate the tainted status to other objects. These rules are operations performed between objects. For instance, if a tainted object performs a write operation on another object, that object is marked as tainted. Objects added to the dependency graph are repeatedly analyzed

until the tainted status does not propagate any further. Afterwards, by taking into account the dependency graph, a filesystem snapshot and an audit log, a *selective redo* algorithm is executed to revert the malicious operations. The algorithm then only replays the legitimate operations included in the log that occurred on the tainted objects.

Although Taser allows for a greater range of recovery scenarios and, when compared to Undo for Operators it reduces manual effort from system administrators, one of its drawbacks is that it can incur in both false positives (removing legitimate data) and false negatives (not removing malicious data). To minimize the issue, *policies* that gradually relax the dependency rules between objects were created. This allows testing the same scenarios using different policies. In some cases there are policies that able to reduce the amount of false positives and false negatives to none.

Kim et al. proposed a tool called RETRO [13] that "repairs a desktop or server after an adversary compromises it, by undoing the adversary's changes while preserving legitimate user actions, with minimal user involvement". It uses what they call an *action history graph* which is a detailed dependency graph used to describe the system's execution history. It also uses a technique named *refinement* that allows describing objects in the dependency graph in various levels of abstraction, thus allowing to reach the required levels of precision when necessary. Furthermore, RETRO leverages *predicates* to *selectively re-execute* only the actions whose dependencies are different after repair. This minimizes the problem of cascading re-execution, i.e. executing operations that only accessed legitimate data despite having interacted with tampered objects. A standard recovery cycle using RETRO begins with the administrator or an IDS (Intrusion Detection System) identifying the intrusion point. Then, the system is rebooted so that non-persistent state is discarded. Next, RETRO's repair controller undoes the unwanted action and the objects modified by it by rolling them back to a previous checkpoint and replacing the unwanted action in the action graph with a no-op. Then, using the action history graph, the actions that were potentially influenced by the unwanted one are identified and the objects they depended on are rolled back and re-executed with corrected arguments. This process is repeated for all actions that are influenced by the previously re-executed actions. To deal with externalized state RETRO uses compensating actions when applicable, and if none are available, the administrator has to manually decide on what action to take.

DARE [14] is a tool based on RETRO but with the additional feature of being able to perform recovery with selective re-execution on *distributed systems*. The authors had to solve several challenges such as tracking dependencies across machines, repairing network connections, minimizing distributed repair and dealing with long-running daemon processes. The solution

for the cross-machine dependency identification is to assign a random token to every connection so as to uniquely identify it. To repair network connections, an API is exposed allowing the controllers of different machines to request rollbacks on socket data objects to each other. To reduce distributed repairs, predicate checking is used just like in RETRO, with the addition of having a *proxy predicate checker* that compares the bytes sent by a process during repair with the ones sent during original execution and, if they matched, no repair was initiated in the affected machines. Finally, to deal with long-running daemon processes such as sshd (secure shell daemon), the fact that these kind of processes often enter a *quiescient state* (which is equivalent to them being restarted) is used so that these processes are considered short-lived, thus enabling DARE to only re-execute the operations from the last recorded quiescient state.

More recent work such as Rectify [15] takes a more specific approach and targets intrusion recovery for PaaS (Platform as a Service) hosted applications. PaaS is a cloud computing model that allows customers to develop and deploy applications without having to manage the hardware and middleware that support them. Rectify aims to have a generic approach by treating the target applications as black-boxes. In practice, this means that it does not require any modification to the application's source code nor any application-specific implementation of the tool. It only requires the configuration of proxies for web and database requests. One of the novel ideas of Rectify is to map application requests to database statements by using *supervised machine learning* algorithms that produce classifiers capable of performing the mapping in runtime. Eventually the malicious database statements are collected and then undone by calculating and executing *compensation operations* that remove the effects of the unwanted statements in the database.

Table 2.1 shows a comparison between the works mentioned in this section. For each work, it names the type of mechanism used to rollback and rollforward state, the practical applications used to evaluate them, relevant techniques used to perform recovery and finally if they have some form of automatic resolution for external consistency issues. The most common type of rollback and rollforward mechanisms are snapshot and log replays, respectively. Dependency graphs are also a common technique used to compute the set of objects that have been affected by intrusions. In regards to external consistency, four out of the five works have a way to deal with those issues. This implies that it is a common feature in Intrusion Recovery systems.

Taking into account the works we have mentioned, it is possible to notice similarities between their goals and how they would preferably be achieved. The point of Intrusion Recovery tools is to enable systems to revert undesirable actions. This should be accomplished without having to modify the target system's source code and with minimal administrator manual effort. Most

Table 2.1: Comparative evaluation of Intrusion Recovery systems.

| System | Rollback Type | Rollforward Type | Practical Applications | Relevant Techniques | Automated External Consistency |
|---|---|---|---|---|---|
| Undo for Operators [11] | Snapshot | Log Replay | Services with well defined interfaces | Service Proxying and Compensating Actions | ✗ |
| Taser [12] | Snapshot | Log Replay | Server Applications | Taint Analysis & Dependency Graphs | ✓ |
| RETRO [13] | Snapshot (Selective) | Log Replay | Filesystems | Dependency Graphs & Selective Re-execution | ✓ |
| DARE [14] | Snapshot (Selective) | Log Replay | Distributed Systems | Dependency Graphs Selective Re-execution | ✓ |
| Rectify [15] | None | Compensating Transactions | PaaS Applications | Supervised Machine Learning | ✓ |

of the tools' recovery cycle can be abstracted into the three R's model [11]. For the Rewind portion the most common techniques are based on *snapshot rollback* mechanisms and for the Repair and Replay portions *log analysis* and *replay*, respectively.

Two of the most desired features of Intrusion Recovery systems are *external consistency* and *valid data preservation*, thus their effectiveness has major impact when it comes to comparing these systems and deciding which performs better. Undo for Operators [11] is one of the first works able to implement both features. Taser [12] improved on it by making a system that can be applied to a broader range of applications while also reducing manual effort in providing external consistency. RETRO [13] was directly compared to Taser as it able to recover from the scenarios that were presented by Taser without any false positives, thus improving on the *valid data preservation* feature. DARE [14] extended RETRO and solved the problem of identifying dependencies between different machines thus allowing recovery to be executed in distributed systems. Finally, Rectify [15] focused on application development platforms and deviated from the standard approaches by leveraging *machine learning* techniques in order to perform intrusion recovery without modifying the original source code.

## 2.2 Blockchain

The *Blockchain* as we know it today was first made popular by *Bitcoin* [1] and it is one of its fundamental underlying technologies. However, the concept of a blockchain is much older,

dating back to 1991 when Haber and Stornetta published a paper [16] where they proposed a system for timestamping documents using an append-only data structure where hashes of the documents are stored in blocks, which are then cryptographically linked together. In practice this means that the blocks are *timestamped* and *digitally signed*. The sudden rise in popularity of Blockchain results from the fact that Nakamoto added *proof-of-work consensus* and an *incentive layer* to the blockchain data-structure which makes it fully decentralized and also solves the *double-spending*[1] problem which makes cryptocurrencies like Bitcoin [1] and Ether [2] possible.

### 2.2.1 Blockchain

In a few words, a blockchain can be described as a consensus-based replicated append-only digital ledger that is *decentralized*, *persistent*, *immutable* and provides both *anonymity* and *auditability*. A blockchain consists of a sequence of blocks that are *cryptographically* chained together. Each block contains, in its header, the hash of the previous block – also called the *parent block*. The first block of a blockchain is called the *genesis block*.

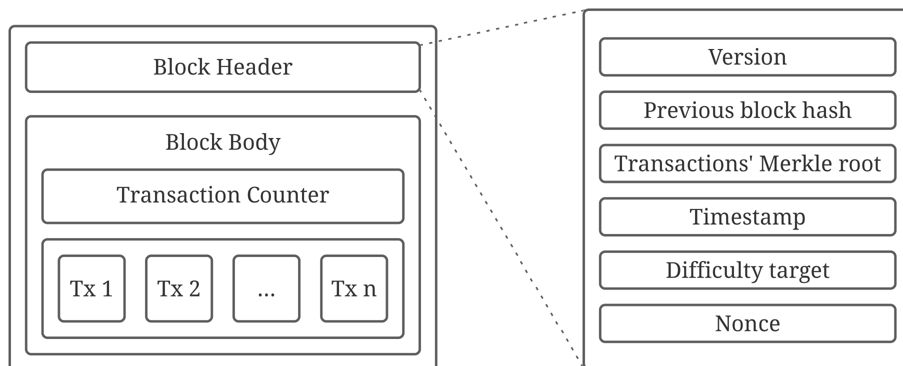


Figure 2.1: Components of a Bitcoin block header.

A *block* in a blockchain is composed of a *block header* and a *body*. As shown in Figure 2.1, the fields inside of the *header* of a *Bitcoin* block are the following: version of the Bitcoin block; hash of the previous block header; Merkle root of the transactions in the block; date of creation; difficulty target for the block; the nonce that was the solution for the block.

The *body* of the block contains a transaction counter and a list of all transactions inside the block. To provide both persistency and a high degree of anonymity the use of *digital signatures* is essential. Every participant in the network owns a pair of keys, a private key and a public key. The private key is used to sign transactions so that others can verify (using the corresponding

[1]A double-spend happens when one user is able to spend the same digital asset more than once. In blockchain systems, this happens when one transaction uses the same input as another transaction that was already validated in the network.

public key) that the funds included in the transactions are owned by it.

### 2.2.2 Decentralization and consensus

*Decentralization* is achieved through the use of consensus algorithms which allows to preserve data consistency in a distributed environment without having to rely on a trusted third party. One of the most fundamental characteristics that differentiates blockchains is what consensus algorithm it uses. There are many possible strategies such as PoW (proof-of-work), PoS (proof-of-stake), PBFT [17] (Practical Byzantine Fault Tolerance), BFT-SMaRt (used to develop a ordering service for Hyperledger Fabric) [18], among others.

Another important characteristic of blockchains is in regards to what entities are allowed to participate in the consensus process. Blockchains can be divided into two categories: *permissionless* and *permissioned*. Permissionless blockchains allow any entity to participate in the consensus process as long as they strictly follow the consensus rules. Public blockchains like *Bitcoin* [1] and *Ethereum* [2] are examples of permissionless blockchains. Permissioned blockchains only allow authenticated entities to participate in the consensus process. This breaks the *decentralization* characteristic of the blockchain as now consensus is determined and controlled by a known group. Consortium blockchains and private blockchains belong to the permissioned blockchain category and they are mostly used in corporate environments.

### 2.2.3 Cryptocurrencies

There are a multitude of different cryptocurrencies but the ones that hold the bigger market share and are the most relevant are *Bitcoin* [1] and *Ether* [2].

The Bitcoin blockchain is public and fully decentralized. It uses what is called *Nakamoto Consensus* as its consensus algorithm which is considered to be the reason behind the success of Bitcoin. Nakamoto Consensus is a proof-of-work consensus algorithm that requires *miners*, i.e. nodes that want to extend the blockchain by appending a block to it, to have performed enough computational work so that the block is considered valid and accepted into the chain. The way this work is performed is by solving a *cryptographic puzzle* in which a counter in the block header is incremented until the hash of the block is lower than a pre-defined value (requires a certain number of most significant zero bits defined by the *difficulty target*). This process is known as *hashing* or *mining*. All miners compete to solve this puzzle and once a solution is found it is propagated through the network so that the other miners know that they can stop and start working on finding the solution for the next block. Also, by finding a solution the mining node obtains a reward which is an agreed-upon value by every participant in the network along

with the sum of the fees included in all the transactions contained in the block. This creates an incentive for miners to participate whether they are able to solve the crypto-puzzle or not. More often than not, miner nodes do not have enough hashing power by themselves to compete with others and at the same time make the process profitable. The probability for a miner to solve a block by themselves is too low nowadays. To illustrate this point, let us calculate the average time it would take to find a solution for a block considering the current difficulty target of the network – approximately 18 trillion – and one of the current top ASICs for mining – an AntMiner S19 Pro [2] – which has an average hash rate of 110 trillion hashes per second. The formula to calculate the average time it would take to solve a block based on the difficulty target and hash rate is:

$$averagetime = difficulty \times 2^{32} / hashrate$$

By replacing the values in the formula, the result is that the average time to solve a block is over *22 years*. To solve the issue, miners join clusters of other miner nodes, i.e. *mining pools*. These mining pools combine the computational resources of the participating nodes so that, whenever a block is added to the chain and is attributed to a pool, the rewards are split between the nodes in that pool with the given value being proportional to their contribution.

Implicitly, the *longest chain* is considered to be the *valid chain* by all participants as it is the one with the most amount of computational power expended into it, therefore aiding in solving the *double-spending* problem which was something that made previous attempts of creating digital currencies difficult.

Double-spending is not an issue when dealing with physical currencies as you cannot use the same bill twice in different transactions unless you physically steal it. When performing digital transactions, they first have to be broadcasted to all the nodes in the network which in turn have to validate and confirm them. The problem is that there was no method that prevented someone from performing a digital transaction and, before it is confirmed in the network, perform another with the digital currency that was previously used. In theory this allowed a single entity to use the same digital currency in two separate transactions. The Bitcoin blockchain solves this problem due to the fact that blocks are timestamped and since the blocks are chained together, an immutable order of transactions is created. Since all the nodes in the network have to agree in a global view of the blockchain, which is accomplished due to the use of the Nakamoto consensus PoW algorithm, that combined with the ordered validated transactions makes it practically impossible for someone to successfully execute a double-spending attack.

Ethereum [2] is another public blockchain that was announced in 2014. Even though it also

[2]https://www.asicminervalue.com/miners/bitmain/antminer-s19-pro-110th

has a cryptocurrency – *Ether* – associated with it, the main goal of Ethereum is to provide an open-ended decentralized platform that enables the development and use of *smart contracts* and *decentralized applications* with built-in economic functions. In contrast to Bitcoin which has a limited scripting language, Ethereum is designed to be a *programmable blockchain* that runs a virtual machine and has a programming language that is *Turing complete* meaning that it can function as a general-purpose computer. Ethereum is further described in Section 2.4

### 2.2.4 Transaction Lifecycle

A transactions' lifecycle corresponds to the stages it goes through starting from its creation and ending with its confirmation in the blockchain. It is important to note that different blockchains have different protocols and as such the lifecycle of a transaction may not be exactly the same, although they should not deviate much from the one depicted in Figure 2.2.

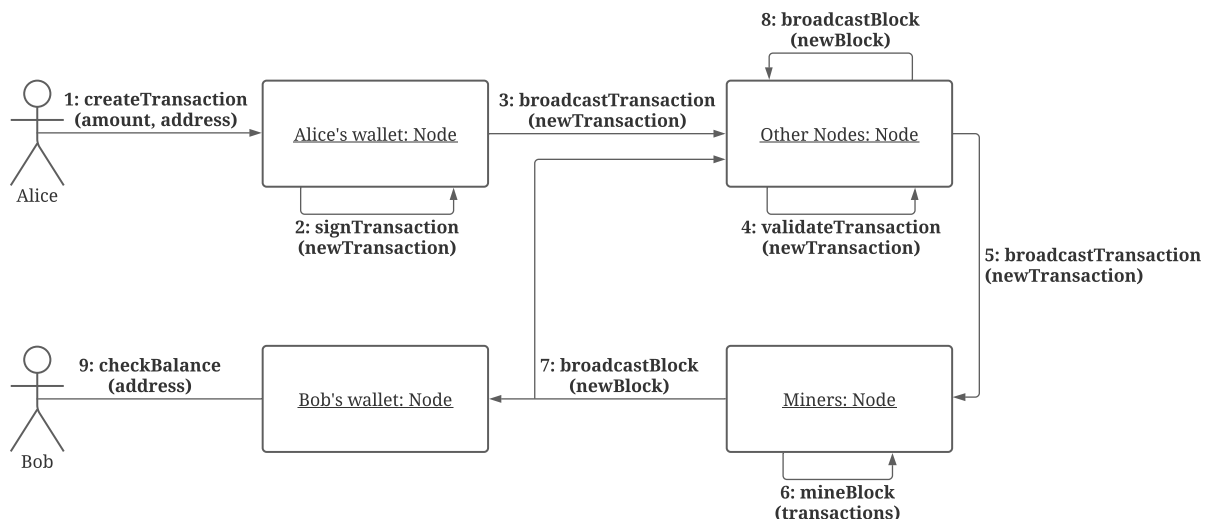


Figure 2.2: UML collaboration diagram for the lifecycle of a blockchain transaction corresponding to a coin transfer from Alice to Bob.

The following is an overview of the steps required for a transaction that transfers coins from Alice to Bob to be confirmed in the blockchain:

1. Alice creates a new transaction containing Bob's address as its destination and the amount to send;
2. Alice signs the new transaction using her private key;
3. The new transaction is broadcast to neighboring nodes;
4. Every client that receives the new transaction has to validate it by:
   4.1. Verifying the signature;

4.2. Checking for errors in the transaction;

4.3. Verifying its not a double-spend attempt;

If successful then the transaction is stored by the verifying client, otherwise it is ignored.

5. All the clients that know about the new transaction and, upon successful validation, will also broadcast it to their peers;

6. Eventually the transaction will reach some mining pools as well as its recipient. The latter will see it and store a copy of it indefinitely, although with zero confirmations at that time. The mining pools will notice it is a new transaction and so will include it (if the fee is adequate) to the block they are trying to create and pass it on to the workers so that they start to try and solve it.

7. After a while the transaction will be included in a block that gets solved, which will in turn be broadcast through the network and everyone keeps a note of it. At this point in time the transaction will have one confirmation;

8. When the other nodes receive the new block and validate it, they will also broadcast it to their peers;

9. Eventually the new block will be received by Bob's node and he will be able to verify that the new transaction is included in it, meaning that the balance associated with his address is updated.

The block creation process will continue, and as more blocks are built atop the block which includes the new transaction, it will gain more confirmations. As the number of confirmations grows, so does the guarantee that the block which contains the new transaction will remain in the blockchain.

### 2.2.5 Summary

In this section we discussed how the concept of blockchain was born and the fundamental characteristics that define one. Then we introduced Bitcoin as the original blockchain and briefly explained how it solved the double-spend problem. This allowed the development of cryptocurrencies and the ability to perform transactions between two entities without requiring a trusted third party. Next came Ethereum, another public blockchain that aimed to provide a platform for the development of decentralized applications, going beyond Bitcoin which was designed to be a substitute for regular currency.

## 2.3 Blockchain Recovery

Applying intrusion recovery techniques to blockchains might be seen as counterintuitive because of the *immutability* guarantee. Nevertheless there are cases where the ability to alter the state of the blockchain is desirable.

Imagine a scenario where an attacker gains access to a user's private key and proceeds to perform a transaction to transfer all the funds to an account he owns. Once this transaction is confirmed in the blockchain there is (currently) no way to reverse it. There are also cases where data that is either personal, leaked or even illegal, is stored in the blockchain where it cannot be deleted and if we also take into account the 2016/769 GDPR regulations [20], these situations become even more complicated as the concept of blockchain is not fully compatible with them.

These issues may be split into two different categories: *data redaction* and *transaction reversion*. Data redaction refers to the issue of *removing data* stored on the blockchain while transaction reversion has a similar purpose to what can be considered a blockchain rollback, where the goal is to either undo specific actions or moving the state of the blockchain to a previous point in time. Therefore, having the ability to, in exceptional cases, perform alterations on the blockchain is one method to make blockchains GDPR compliant and address complications that are the result of human errors.

### 2.3.1 Forking

One way of dealing with these issues would be by resorting to *intentional forking*. Before explaining why forking is not the most optimal approach it is necessary to clarify the meaning of a fork in the context of a blockchain.

Forks, albeit rare, are a natural occurrence in a blockchain and exist due to the fact that, in order for the blockchain to grow, there is a need for the participating nodes to agree on its state, in other words, reach *consensus*.

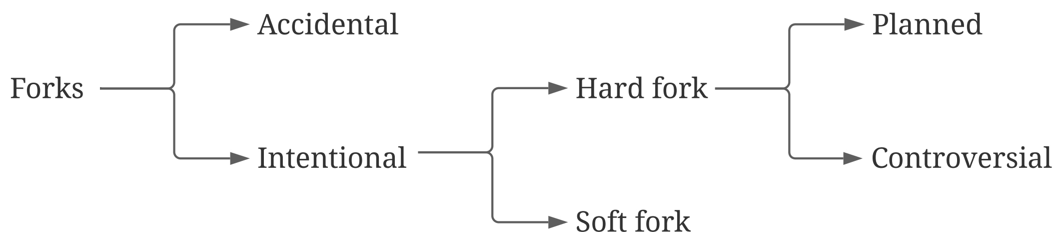


Figure 2.3: Fork taxonomy in the blockchain domain.

As depicted in Figure 2.3, there are two main types of forks: *accidental* and *intentional*.

*Accidental forks* occur when independent nodes receive different blocks to include on the chain. In order to fix these disagreements, in blockchains that use proof-of-work consensus, the *longest chain rule* decides the chain that nodes should adopt. When these types of forks arise, the chain is split into two parallel chains. Nodes can decide which chain they adopt, but the longest chain rule ensures that nodes always choose to adopt the chain with the most computational work expended into it which typically is the chain containing the most blocks. The blocks on the discarded chains are called *orphaned blocks* and their transactions are put back into the pool of transactions that miner nodes use to choose the ones to add to their next block.

*Intentional forks* have a more human component linked to it since they do not exist naturally as a result of the blockchain protocol unlike accidental forks. They can be split into two categories: *soft forks* and *hard forks*.

A *soft fork* is a *backward-compatible* upgrade to a blockchain that allows nodes that do not upgrade to participate as long as they do not break the new protocol rules. An example of a soft fork would be the implementation of a rule that reduced the maximum network block size from 4MB to 2MB. Nodes that do not update will still be able to process incoming transactions but they will not be able to contribute to the growth of the blockchain as the blocks created by them do not abide by the new rules.

A *hard fork* is a change in the protocol that is *incompatible* with previous versions. Therefore, nodes that do not upgrade to the new version will not be able to contribute to the updated chain. Hard forks can either be *planned* or *controversial*. Planned hard forks usually converge to a state in which the majority will upgrade to the new version and the nodes that do not upgrade will be on the old chain that eventually becomes stale since not many nodes are contributing to it. Controversial hard forks are usually the result of a disagreement within the community which results in two incompatible blockchains. The split of the Ethereum blockchain into Ethereum and Ethereum Classic is one example of a situation that is the result of a controversial hard fork.

As a matter of fact, the fork that led to the split of Ethereum is a great example of its use to recover from intrusions as it happened because of the infamous *The DAO hack* [21]. DAO stands for decentralized autonomous organization and it was a form of venture capital funding implemented on top of the Ethereum blockchain. In essence, a vulnerability in the *smart contract* that implemented *The DAO* was exploited by an attacker that ended up collecting around 3.6 million ETH – worth around $50 million (USD) at the time and 10 times more nowadays with

the current exchange rate. The hard fork proposal to recover the funds essentially placed the funds that were in The DAO and all its children in a new refund contract account which the stakeholders could call to withdraw their funds. No transactions in the block where the hard fork ocurred (block #1,920,000) reflect these changes as they were implemented directly into the Ethereum node client software.

### 2.3.2 Data Redaction

One of the first works that focuses on *data redaction* is Redactable Blockchain [22]. It proposes a blockchain which replaces the usual cryptographic hash functions with a *chameleon hash function* containing a known trapdoor that allowed the computing of hash collisions. Cryptographic hash functions allow the mapping of arbitrary strings into a fixed-size number. These functions are one-way in the sense that, given the output, it is unfeasible to find the corresponding input. Another useful property of these functions is their *collision resistance*. In essence, given an input and its corresponding output, it is unfeasible to find a different input that maps to the same output. In regards to chameleon hash functions, knowledge of the trapdoor key allows collisions to be generated. This is a gross simplification of the way chameleon hash functions behave and we refer the reader to the paper for a more profound and complete explanation of the process.

The use of chameleon hash functions in this scenario enables the possibility of editing blocks. When the data inside a block is edited, the resulting hash is different from the original. However, by using a chameleon hash function, it is possible to compute an hash collision so that the hash of the edited block remains the same as the hash of the original block. In this way, since the hash of the blocks remain unchanged, the links between them are not severed and the whole chain is still valid.

This approach works mostly for permissioned blockchains as the trapdoor key must be owned by a central authority that is responsible for the editing of blocks. Although the authors claim that there are ways to adapt it to permissionless blockchains by either distributing the trapdoor key among all participants – which seems impractical – or only distributing it between a set of known parties – which is questionable.

Deuber et al. developed a Redactable Blockchain for the *Permissionless* Setting [23] that does not rely on heavy cryptographic tools or trust assumptions in order to remove data from the blockchain, but instead uses a consensus-based voting mechanism which is dictated by policies that set the requirements for a redaction to be accepted. Furthermore, any participant can make requests for a redaction while also allowing *public verifiability* and *accountability* of the redacted chain.

The adjustments made to enable all these features are: extending the block structure so that there is an extra field that contains a copy of the original transactions' Merkle root, and the creation of an editing policy that defines the constraints and requirements for an edit request to be approved.

To perform an edit in the chain a user has to first propose an edit request by creating and submitting a special transaction which, apart from other information, contains the index of the block to be edited and the new candidate block that is going to potentially replace it. Next, miners receive the request and verify it by checking if it contains the correct information about the previous block, if it solves the crypto-puzzle, and if it does not invalidate the next block in the chain. If all these conditions are fulfilled then the next step is for miners to vote for the request to be accepted by, during a pre-determined voting period, including the hash of the request in the next block they mine. After the voting period finishes, all participants can verify if the request was accepted or not by checking the number of votes it received and see if it matches the requirements set by the defined policies.

The authors mention that it is important to note that chain validation is performed similarly as it is in a regular blockchain, with the exception that when validating an edited block it is necessary to consider the hash of the original unedited block. This hash is computed by resorting to the old value of the transaction's Merkle root, as well as checking if the block was accepted according to the editing policy.

Once again, this type of work is useful to hide or delete unwanted information in the blockchain, hence being more appropriate for data redaction scenarios as opposed to transaction reversion. Although, in both of the previously mentioned works, it is theoretically possible to revert a transaction – and all its dependencies – by proposing a redaction to remove all the transactions from their respective blocks. This would end up eliminating any records of it ever happening in the blockchain, if the redaction request is accepted.

### 2.3.3 Transaction Reversion

There have been proposals for systems in which transaction reversion in blockchains was supported. Reversecoin [24] is a solution in which a user has access to two different types of accounts: a *standard account* and a *vault account*. *Standard accounts* are to be used as a regular account for day-to-day transactions, so ideally the user would never hold large amounts of coin in them. *Vault accounts* should be considered as an equivalent to a savings account – where large amounts of coin are held – and they provide the highest security. They are safer due to the fact that they are backed by two key pairs – one online and one offline – as well as a configurable timeout from

which transactions made using the online keys stay pending in the blockchain. Offline keys allows users to perform transactions without being constrained by the timeout period. This means they have the ability to perform *immediate transactions* and *override pending transactions* that are made using the online keys.

However, this mechanism only works for transactions that are not yet confirmed in the blockchain. Ideally, one would combine this approach with a monitoring tool that alerts the user any time a transaction is be performed from his accounts. This would give an opportunity for the user to use the offline keys to revert any undesired transactions.

Unconfirmed transaction replacement in the blockchain domain is a topic that has always been discussed. In fact, in the first release of Bitcoin [1], there was a transaction replacement feature called Replace-By-Fee which was later disabled [25] due to the fact that it could be exploited for denial-of-service attacks. This feature was later re-implemented as described in BIP 125 [26] and named Opt-in Replace-By-Fee with it being introduced in Bitcoin Core 0.12.0. The main difference between both implementations is that replacement transactions now have to pay extra fees thus discouraging denial-of-service attacks. Essentially, it allows spenders to mark a transaction as replaceable in one of two ways:

- **Explicit Signaling**: Assign to the transaction a sequence number lower than 0xffffffff-1
- **Inherited Signaling**: Transactions which have ancestors that are both signaled for replacement and are unconfirmed are also replaceable

The requirements, as stated in BIP 125, that need to be fulfilled in order to replace one or more transactions are the following:

1. The original transactions has to signal replaceability;
2. The replacement transaction may only include an unconfirmed input if that input is included in one of the original transactions;
3. The replacement transaction pays an absolute fee of at least the sum paid by the original transactions;
4. The replacement transaction must also pay for its own bandwidth at or above the rate set by the minimum relay fee setting;
5. The number of original transactions to be replaced and their descendant transactions must not exceed a total of 100.

There are several potential use cases for this feature. Some examples are: to increase the fee of an unconfirmed transaction so that miners are more likely to include it in their next block, merge several transactions so as to reduce the total amount of fees paid, or even correct human errors.

### 2.3.4 Summary

Table 2.2 shows a comparison between the works mentioned in this Section. The comparison considers the types of blockchains the works are better suited for – permissioned or permissionless – and the forms of recovery they are able to do. *Data redaction* refers to being able to remove or hide data from the blockchain, *unconfirmed transaction replacement* covers the use case of being able to stop a transaction that was already broadcasted from being confirmed, and *linked transaction reversion* means being able to revert a transaction and all the transactions whose validity depends on it.

Table 2.2: Comparative evaluation of Blockchain Recovery mechanisms.

| System | Permissioned/ Permissionless | Data Redaction | Unconfirmed Transaction Replacement | Linked Transaction Reversion |
|---|---|---|---|---|
| Rewriting History in Bitcoin and Friends [22] | ✓/ ✗ | ✓ | ✗ | ✗ |
| Redactable Blockchain [23] | ✓/ ✓ | ✓ | ✗ | ✗ |
| Reversecoin [24] | ✓/ ✓ | ✗ | ✓ | ✗ |
| Opt-in RBF [26] | ✓/ ✓ | ✗ | ✓ | ✗ |

In summary, Redactable Blockchain [22] works well for permissioned blockchains because the trapdoor key for the chameleon hash algorithm has to be split among a set of entities in order to allow block editing. For permissionless blockchains, Redactable Blockchain in the Permissionless Setting [23] proposes a consensus-based solution that is controlled by editing policies where miners can accept or deny an edit request through a voting mechanism. Both of these solutions work in regards to removing or altering data without invalidating the next blocks in the chain. If there is a need to revert a transaction that transfers cryptocurrencies between any number of parties, these solutions no longer work as it is necessary to keep track of the dependencies related to the coins involved, and they offer no mechanism to do so.

Reversecoin [24] increases the level of security for cryptocurrency holders. Although it can be used to fix mistakes and some types of stealing attempts, it is not able to revert transactions

that are stored in blocks already confirmed in the blockchain. Opt-in Replace by Fee [26] is a feature currently present in Bitcoin and it allows participants to signal transactions so that they become replaceable until they are not confirmed in the blockchain. It can be useful in cases where the user makes a mistake in a transaction, but not when an attacker has access to his wallet due to the fact that since the feature is "opt-in" the attacker can simply choose not to signal the transaction as replaceable.

One feature that is lacking in these works is the ability to revert a transaction that is already confirmed in the blockchain and has other transactions that depend on it.

## 2.4 Ethereum Platform

Although a brief introduction of Ethereum is given in Section 2.2, a more detailed description of some of its components and features follows since it is the platform of choice for this work.

### 2.4.1 Ethereum Virtual Machine

The Ethereum Virtual Machine (EVM) [2] is similar to the Java Virtual Machine (JVM) [27], in the sense that both provide an abstraction of a *computing environment* – computation and storage – and each have their own instruction set for program execution [8]. To allow the EVM to be *Turing-complete*, all instructions (opcodes) are assigned a set *gas* cost.

Gas is a unit used in Ethereum to measure the computational effort of executing a transaction. When creating a transaction it is necessary to pay for the amount of gas that it will consume using Ether. The sender of the transaction offers a value for each unit of gas. It is possible to estimate how much gas a transaction will spend since every instruction has a set gas cost.[3] While the amount of gas a transaction will require is mostly predictable, the price to pay for each unit of gas is not. It depends on different factors such as the number of pending transactions and the number of active miners and how fast you want it to be confirmed in the blockchain [28]. To determine how much the price will be in terms of a fiat currency (e.g., euros or dollars) the formula is:

$$gasPrice \times gasCost \times etherCost$$

where *gasPrice* is the price of each unit of gas in Ether, *gasCost* is the amount of gas the transaction requires and *etherCost* is the conversion rate from Ether to the desired fiat currency. As the gas price offered by the sender increases so does the likelihood of a miner adding that specific transaction to the block it is mining because the reward he gets from doing so also

[3]https://github.com/crytic/evm-opcodes

increases.

There are a variety of Ethereum node client software [29, 30, 31], implemented in various programming languages, which naturally contain their own EVM implementation. That said, they all have to follow the formal specification of the Ethereum blockchain [2]. When a node is running, it is actively downloading and verifying all blocks – and their transactions – to secure the network. Every state change on the Ethereum blockchain is executed by transactions which are handled by the EVM, including smart contract executions.

### 2.4.2 Smart contracts

In the context Ethereum, a *smart contract* is an immutable, deterministic computer program that runs on top of the Ethereum blockchain. High-level languages [32, 33] are used to write smart contracts that are compiled into EVM bytecode which is interpreted by the EVM. There are two types of accounts in Ethereum: *externally owned accounts* (EOA) and *contract accounts* (CA). Both types can receive, hold and send Ether and tokens as well as interact with deployed smart contracts. What makes contract accounts different from externally owned accounts is that: creating a contract account has a cost (because it uses network storage to store code); they cannot initiate transactions (only EOAs can); and transactions sent from an EOA to a CA can trigger code execution that may lead to state changes (transactions between EOAs can only be ETH transfers).

Smart contracts have many possible use cases such as: being the backend for DApps (applications that are mostly or entirely decentralized); provide a way to enforce intellectual property rights [34, 35]; buy and sell property without intermediaries [36]; or even fundraising through *Initial Coin Offerings* [37]. Furthermore, it is very common in the aforementioned use cases to make use of a *digital token* [38]. This naturally led to the necessity of having a standard interface so that tokens would be easier to integrate in a variety of wallet clients and applications.

### 2.4.3 ERC-20 and ERC-721 Tokens

The first token standard was introduced as an Ethereum Request for Comments (ERC), and it is named ERC-20 [6]. It defines an interface for implementing a token such that all ERC-20 tokens can be accessed and used using the same methods. The ERC-20 is currently the most widely used standard for *fungible tokens* which means that different units of the same token are interchangeable and hold the same value. A common example used to explain fungibility is *money*. For instance, a genuine $5 (USD) bill has the same value than all other genuine $5 bills even if they are not in exactly the same condition (one might be more rugged than the other).

Additionally, it is interchangeable with five $1 coins or any other combination of coins that add up to $5. In contrast, *non-fungible tokens* can be used to represent items that are unique and therefore not interchangeable. To give an idea, most rare collectibles are non-fungible items. For illustration purposes, imagine a system that uses tokens to represent ownership of rare collectibles. In that system, a token that is used to represent a Fabergé egg and another to represent Leonardo Da Vinci's "Mona Lisa" are not equivalent, and thus not interchangeable. In Ethereum, the standard for *non-fungible tokens* is ERC-721 [7].

These kinds of tokens are handled at the smart contract level as opposed to what can be considered protocol level tokens, e.g. Ether and Bitcoin, meaning that the Ethereum protocol does not have the ability to manage them. That task is the responsibility of the smart contract that implements the token. Therefore, actions such as ownership, transfers and access rights to tokens are handled by their respective smart contracts.

### 2.4.4 Initial Coin Offerings

Initial Coin Offerings, or ICOs for short, are a mechanism for fundraising that allows investors to trade their currency, e.g. cryptocurrencies such as Bitcoin and Ether or fiat money such as Dollars and Euros, for another currency which in most cases is a token created by the company/startup that is doing the fundraising. People who decide to invest in ICOs believe that the project will become successful causing the tokens to raise in value. If the tokens become more valuable compared to when they were first acquired then they can be sold for a profit. Ethereum is actually an example of what can be considered a successful ICO since it was able to raise the amount of funds necessary to continue with the development of the project. As a matter of fact, the use of smart contracts in the Ethereum blockchain allows ICOs to be held in the blockchain itself. This means that one can develop a smart contract in order to create their own token and start an ICO.

#### Exploits

Since smart contracts are developed by programmers, no matter the amount of effort put into auditing and designing their implementation, there is always the possibility for vulnerabilities to arise. The ICO market had over $14 billion (USD) being raised in 2018 alone [39]. This creates a very good incentive for attackers to exploit these contracts in order to steal funds. There have been a few documented cases of programming errors being taken advantage of by attackers.

BeautyChain (BEC) was the target of one of those attacks, more specifically, a common case of an integer overflow vulnerability that allowed the attacker to gather a large amount of tokens.

```
function batchTransfer(address[] _receivers, uint256 _value)
  public
  whenNotPaused
  returns (bool)
{
  uint cnt = _receivers.length;
  uint256 amount = uint256(cnt) * _value;
  require(cnt > 0 && cnt <= 20);
  require(_value > 0 && balances[msg.sender] >= amount);

  balances[msg.sender] = balances[msg.sender].sub(amount);
  for (uint i = 0; i < cnt; i++) {
      balances[_receivers[i]] = balances[_receivers[i]].add(_value);
      Transfer(msg.sender, _receivers[i], _value);
  }
  return true;
}
```

Listing 2.1: batchTransfer function of the BeautyChain smart contract.

As shown in Listing 2.1, the variable *amount* is calculated as the product of *cnt* and *_value*. The type of the variable *_value* is uint256 which corresponds to an arbitrary 256 bit unsigned integer. The attacker set the *_value* to eight vigintillion (an eight followed by 63 zeroes) causing an overflow in the *amount* variable setting it to zero. With *amount* set to zero, the sanity checks are fulfilled, the subtraction to the balance of the sender has no effect on its balance (as it is effectively subtracting zero), and to the balance of the *_receivers* the large value of *_value* is being added without deducting any balance of the *msg.sender*.

Another instance of an attack to smart contracts is the Parity multi-sig hack [40]. The attacker was able to steal over 150,000 ETH – worth approximately $30 million (USD) at the time – and would have stolen many more if not for a group of white-hat hackers that quickly organized and in order to minimize the damage they used the same exploit on the remaining vulnerable accounts so that the attacker could not have access to it and ended up saving over $179 million (USD).

```
// constructor - just pass on the owner array to the multiowned and
// the limit to daylimit
function initWallet(address[] _owners, uint _required, uint _daylimit) {
  initDaylimit(_daylimit);
  initMultiowned(_owners, _required);
}
```

Listing 2.2: initWallet function of WalletLibrary contract.

In order to gain access to each account the attacker had to send two transactions to each of

the affected wallet contracts: one to obtain exclusive access to the contract and the second to move all its funds to an account he owns. The first transaction was a call to a function named *initWallet* (shown in Listing 2.2) that had the role to extract the wallet's constructor logic into a separate library. Since the wallet forwards all unmatched function calls to that library, this allows public functions of that library to be called by anyone – including the *initWallet* function.

Due to the fact that there were no checks to prevent an attacker from calling the function after a contract had already been initialized, the attacker only had to call the *initWallet* function and make himself the sole owner of the account as well as only requiring one confirmation to perform transactions. Afterwards the attacker invoked the *execute* function to move the funds of the exploited account to an account it owners.

The issue with this approach is mainly the use of the delegatecall method as a catch-all forwarding mechanism as opposed to explicitly defining what library functions can be invoked externally. Listing 2.3 shows the catch-all (unnamed) function. For the attacker to call the *execute* function it had to send a transaction with no value ($msg.value = 0$) and with non-empty *msg.data*. The contents of *msg.data* will match the *execute* function of the *_walletLibrary* contract which trigger its execution, thus completing the attack.

```
// gets called when no other function matches
function() payable {
  // just being sent some cash?
  if (msg.value > 0)
    Deposit(msg.sender, msg.value);
  else if (msg.data.length > 0)
    _walletLibrary.delegatecall(msg.data);
}
```

Listing 2.3: Forwarding of unmatched function calls through the delegatecall method.

The solution was not optimal, and alternatives such as soft or hards forks so that the funds could be returned to the respective owners would be difficult to both justify and enforce.

Many other cases of exploited vulnerabilities in smart contracts due to human programming errors [41] would benefit from transaction reversion mechanisms, as an alternative to forking or even performing no action at all.

### 2.4.5 ERC-1080

In spite of all these vulnerabilities and the risks associated with them, new applications that rely on these digital tokens are constantly being developed. Moreover, considering the large amounts of currency being exchanged, there is a need for users to have methods of securing their assets

in case of loss, theft or fraud. The usual practice when setting up an Ethereum wallet is to have an asymmetrical key pair generated based of a mnemonic seed which commonly has 12 to 24 words. If somehow access to the private key is lost then one can still use the mnemonic seed to recover and regain access to the wallet. In the case of losing both the private key and the mnemonic seed words then it is practically impossible to recover the wallet, meaning that all assets associated with that address are going to be unusable.

Supposing that the wallet contained both Ether and a variety of tokens, then we can safely assume that the Ether is unrecoverable since it is handled at the Ethereum protocol layer, unless a hard fork is performed to move the funds to another address. However, tokens are handled at the smart contract layer and the point of centralization are the smart contracts that implement them. The smart contract that manages each particular token may have the authority to move them from one address to another. If, hypothetically, there is a way to prove that the lost address belonged to that user, then there could be a mechanism in the smart contract that would transfer the tokens from the old (lost) wallet to the new one. In the case of theft, a similar mechanism could be implemented. As long as the thief does not exchange the tokens for a currency not controlled by the smart contract then it would be possible to retrieve them.

Regarding this topic, a new standard was proposed as an ERC, named Recoverable Token [42]. According to its author, it is a "token standard that allows for users to dispute transfers, report and recover lost accounts, and find appropriate resolution in the case of account theft" [43]. As of right now there are no known implementations of this standard, as it is currently defined as an interface that is still up for discussion.

This approach, despite being less broad in the sense that it is limited to the handling of tokens, is a good start into developing recovery mechanisms on top of the blockchain since it is implemented at the smart contract layer and not the Ethereum protocol layer, thus being an opt-in approach for account security.

### 2.4.6 Dispute Resolution

The above-mentioned interface proposal – ERC-1080 – suggests the usage of a dispute resolution mechanism. In the legal system there are several types of dispute resolution mechanisms such as lawsuits, mediation and *arbitration*. Arbitration is a method of resolving disputes outside the court of law, commonly used in commercial and consumer disputes [44, 45]. Generically the process consists in two parties that have a dispute to agree upon a group of arbitrators who ideally constitutes, as a group, an unbiased third party. Then those arbitrators, after analyzing the arguments and evidence provided by both parties, will make a decision in favor of one party

or neither. A dispute involves the payment of a *fee* that is used to reward the arbitrators for their work, i.e., as an incentive for them to do their job.

The idea to bring dispute resolution to the blockchain is being explored [46]. Kleros [47] is a decentralized arbitration application in which crowd-sourced arbitrators give rulings on disputes. The arbitrators are expected to rule correctly and fairly due to game theoretical incentives.

Aragon [48], a software used to create and govern organizations on the Ethereum blockchain, has a component named Aragon Court that works similarly to Kleros but is limited to organizations in the Aragon Network.

Mattereum [49] is a an infrastructure to build a layer to manage property or assets on-chain. As it is the case with several types of transfers, disputes may arise, thus a dispute resolution mechanism has to exist. The approach taken is akin to the standard used in common arbitration courts. When a dispute is raised, either both parties have a predetermined agreement where the arbitrator had already been chosen, or alternatively and by default, an arbitrator is appointed from a panel of arbitrators.

In addition to those systems, there is a proposal for standard interfaces for both arbitration and evidence submission, ERC-792 and ERC-1497 [50, 51] respectively.

Having dispute resolution systems in the blockchain gives them all the benefits of the blockchain technology: public, immutable and decentralized dispute resolution.

## 2.5 Summary

This Chapter presented past literature of the main areas related to our work. It started with a brief introduction of the area of *intrusion recovery* followed by a chronological review of previous works. Next, it described the *blockchain* technology, namely: its characteristics, the problems it solves, the most relevant blockchains, and an overview of how they operate. Then, it presented the concept of *blockchain recovery*, the reasons behind its existence as well as its most recent developments. The Chapter concluded with a description of the *Ethereum* platform along with some of the features it provides that are going to be used in our proposal, presented in the next Chapter.

# Chapter 3

# Recoverable Token

This chapter presents our approach in greater detail. We will give an overview of the components that make up our recovery system.

## 3.1 Recoverable Token Architecture

The architecture of the system is shown in Figure 3.1, i.e., of the Ethereum blockchain with the Recoverable Token smart contracts and client (RCV App), users, and a storage service (IPFS). To interact with the system, users need *private keys*. The tokens owned by an account are stored and managed by the smart contracts deployed on the blockchain. These smart contracts are extended with the functionality of those that implement our recovery mechanism: *RCVToken*, *Claims*, and *Profiles*. Users also need to have access to an Ethereum node to send transactions (value transfers or smart contract calls) and propagate them to other nodes so that they may be validated and appended to the chain. Storing data on the blockchain is costly and as such it is necessary to use an off-chain storage system. To that end, the system will make use of a decentralized, tamper-resistant, content-addressable, peer-to-peer storage network. In the architecture we consider that system to be the Inter-Planetary File System (IPFS) [52] that is commonly-used and peer-to-peer. Users need access to an IPFS node so that they are able to store and access data using the IPFS network. However, other decentralized storage systems [53, 54] may be used, each with their own trade-offs.

### 3.1.1 Attack Model

A *user* can be a *regular* user or an *arbitrator*. The regular users (that we often designate simply as users) are those entities that use the tokens provided by smart contracts. They own the tokens and can perform any actions just as they would if there was no recovery mechanism in

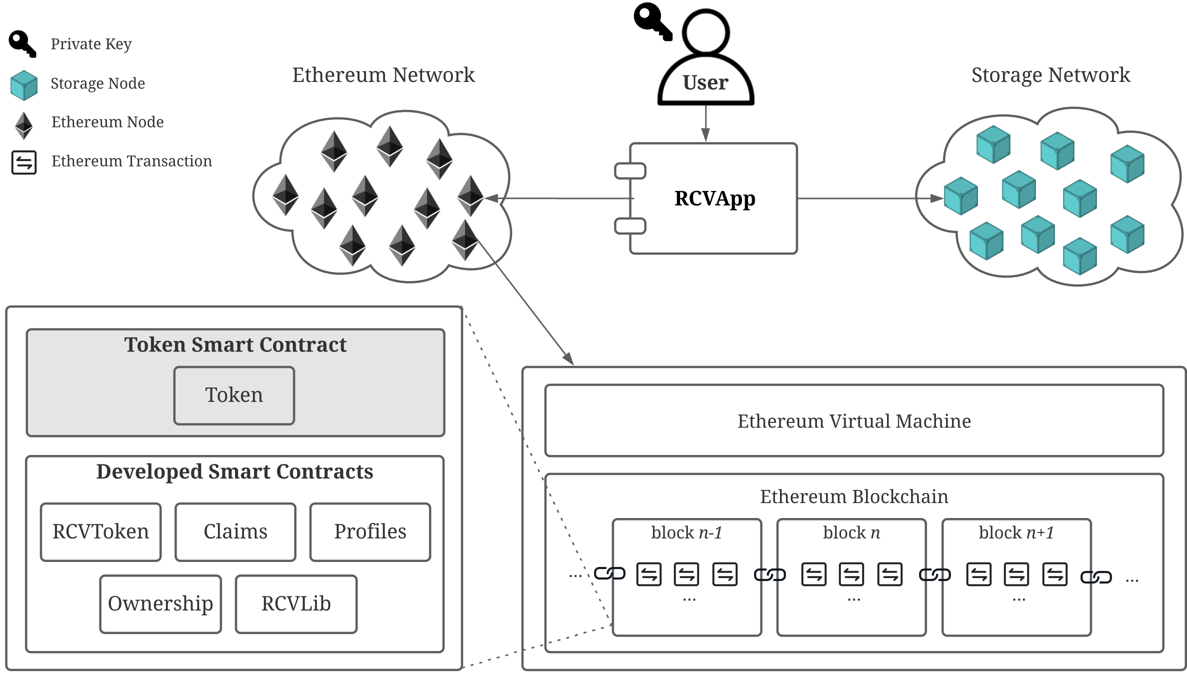


Figure 3.1: Recoverable Token system architecture. A user which owns a private key is able to use the RCVApp to connect to a node of the Ethereum network and interact with the storage network. Each Ethereum node runs a local copy of an EVM and maintains a copy of the chain. The smart contracts that comprise the token recovery mechanism and the token which is to be recovered (shown in a grey background) are deployed on the blockchain.

the system. The recovery mechanism allows these users to submit claims that may escalate to disputes and in turn lead to account recoveries being performed. Arbitrators are special users who have permission to rule disputes.

Both regular users and arbitrators may positively contribute to the system or play against it. A user or arbitrator that does what it is supposed to is said to be *correct*, whereas one that deviates from that behavior is said to be *malicious*. Some possible attacks that users may try to perform against our mechanism include: submitting false claims; exploiting bugs; colluding with arbitrators. Arbitrators have the ability to rule on disputes, they may give dishonest rulings or none at all. We assume that less than a third ($f$) of the total amount of arbitrators ($n$) participating in a dispute are malicious, i.e., $n \geq 3f + 1$ (the same proportion as in common Byzantine fault-tolerant consensus algorithms [55]).

We do the usual assumptions that Ethereum works as expected and that cryptography is not compromised (e.g., no transactions can be issued on behalf of a user without his private keys).

## 3.2 Recovery process

In this section we explain in detail from start to finish the necessary steps that need to be executed in order to perform any *recovery actions*. Throughout this section we will be referring to the methods specified in Table 3.1.

Table 3.1: Public methods of the RCVToken, Claims and Profiles contracts.

| Method Name | Description |
|---|---|
| **RCVToken contract** | |
| claimLost(lostAccount) | Report the *lostAccount* address as lost. |
| cancelLossClaim() | Report the address as not being lost. |
| reportStolen() | Report the address as being stolen. |
| chargeback(pendingTransferNumber) | Request a transfer chargeback. |
| getPendingTransferTime(account) | Get the time an account has to chargeback a transfer. |
| setPendingTransferTime(account) | Set the time an account has to chargeback a transfer. |
| getLostAccountRecoveryTime(account) | Get the time account has to wait before a lost account dispute can start. |
| setLostAccountRecoveryTime(account) | Set the time account has to wait before a lost account dispute can start. |
| submitMetaEvidence(claimID, metaEvidence) | Link a meta evidence URI to a claim. |
| submitEvidence(claimID, evidence) | Link an evidence URI to a claim. |
| signUp(transferTime, recoveryTime) | Sign up an account. |
| addRecoveryInfo(recovery, proof, identity) | Submit proof of ownership. |
| **Claims contract** | |
| createLossClaim(claimant, lostAccount) | Create a loss claim. |
| createTheftClaim(claimant) | Create a theft claim. |
| createChargebackClaim(claimant,transferID) | Create a chargeback claim |
| voteOnClaim(claimID, vote) | Vote on claim number *claimID*. |
| giveRuling(claimID) | Commit to a final ruling. |
| rule(claimID, ruling) | Enforce *ruling* on claim number *claimID*. |
| **Profiles contract** | |
| appeal(disputeID, extraData) | Request an appeal for a dispute. |
| appealCost() | Return the cost of requesting an appeal. |
| appealPeriod() | Return the time window for appeals. |
| arbitrationCost(extraData) | Return the cost of submitting a claim. |
| currentRuling(disputeID) | Return the current ruling of a dispute. |
| disputeStatus(disputeID) | Return the status of a dispute. |
| isPendingTransfer(transferID) | Check whether transfer status is pending. |
| enforceRuling(disputeID) | Enforce ruling of a dispute. |

### 3.2.1 Initial configuration

To be able to trigger any recovery actions, the user has to perform an initial account configuration with the contract that holds the digital assets to be recovered. The first fundamental configuration is performed by calling the *signUp* method of the RCVToken contract. This method requires two arguments: *transferTime* and *recoveryTime* which correspond to the time – in block units – that the user will have to chargeback a transfer or to cancel a loss claim. The need for this is to prevent abuse of these two features. For *chargebacks* it ensures that if *transferTime* number of blocks are mined after performing a transfer then it will no longer be reversible. In case of *loss* claims, it ensures that if a fake claim is performed then the owner of the account has *recoveryTime* number of blocks to cancel that claim and prevent a dispute from starting.

However, this only allows for performing chargeback claims since it is the only case where the token will return to the claimant's account. To be able to perform both loss or theft claims an additional step is required. This additional step consists in linking a secondary account that will be the recovery account where the tokens will be transferred to in the case of either of those claims resulting in a successful recovery. To link two accounts it is necessary to generate what we refer to as *proof of ownership*. It essentially is the *digital signature* of a message that contains three elements: an *identity* and the *addresses* of the *account* and its corresponding *recovery account*. The *keccak256* [56] hash of the proof of ownership message is digitally signed with the private key of the recovery account and then submitted and stored in the blockchain using the account that is being protected. To be more precise, the message components are ABI-encoded [1] and then tightly packed, i.e. concatenated without padding. Next, the result is prefixed with the string "\x19EthereumSignedMessage:\n" and the length of the hash of the proof of ownership message. Finally, all that data is keccak256 hashed and then signed. So, in the end, what is submitted into the blockchain is the digital signature of:

$$keccak256("\backslash x19EthereumSignedMessage : \backslash n", keccak256(msg).length, keccak256(msg))$$

This is to ensure that *at least* at this point in time one person has control over both accounts. After generating the *proof of ownership*, then a call to the *addRecoveryInfo* method has to be made. It requires three arguments: *recoveryAccount*, *proof* and *identity* which are respectively the address of the recovery account, the proof of ownership and the identity that was recorded in it. Only after all these previous steps are executed will the user have permission to submit any type of claim using their account.

[1] https://docs.soliditylang.org/en/develop/abi-spec.html

### 3.2.2 Submitting a claim

From here on out we assume that the user has performed the initial configuration steps mentioned in the previous section. This implies that now the user is able to submit all types of claims. Recall that there are three types of claims: *loss*, *theft* and *chargeback* claims. The process for submitting a claim is very similar for all types barring some particularities of each one. Nevertheless, we will go through each one in detail.

#### Loss claim

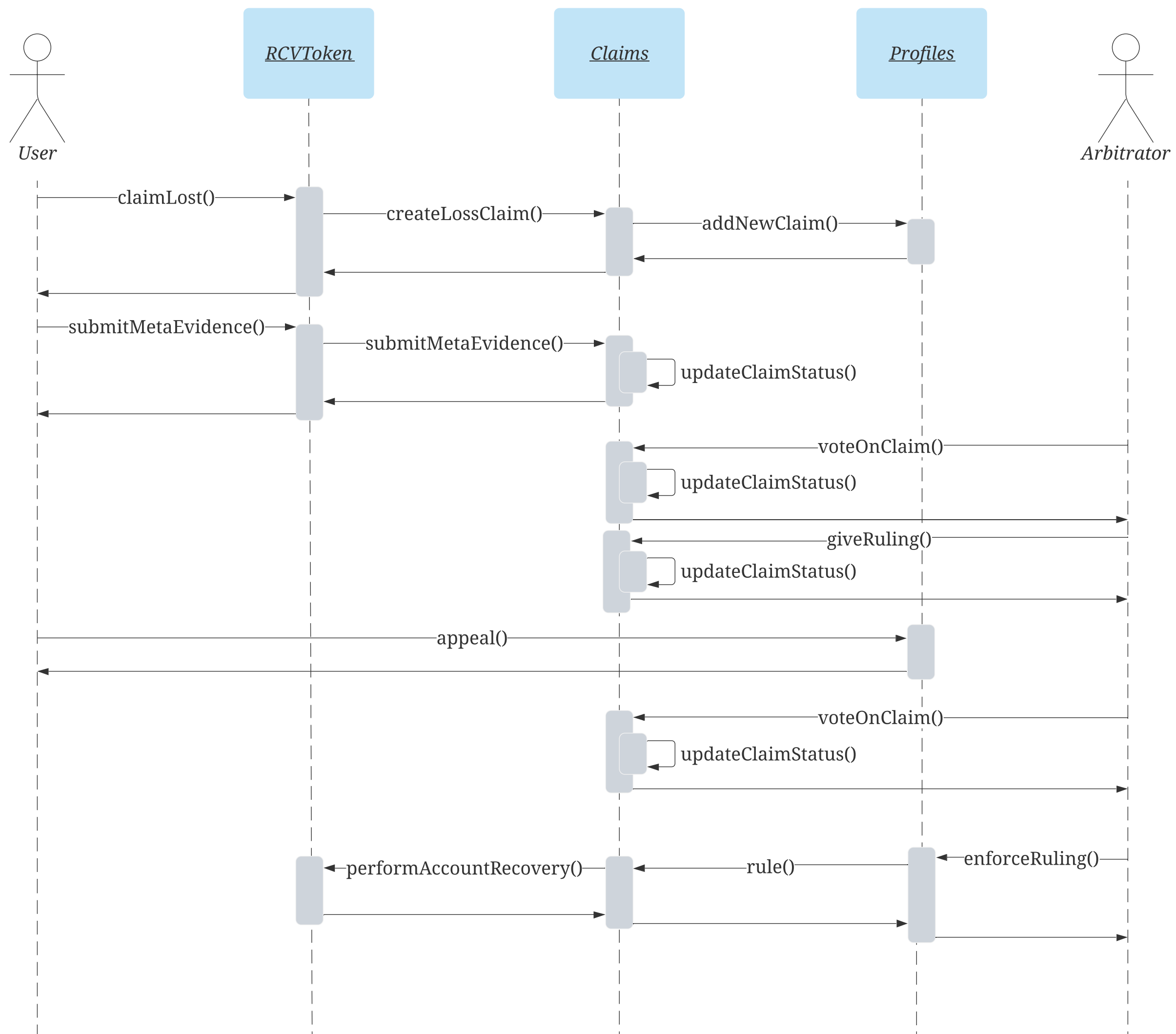


Figure 3.2: Example of a successful loss claim.

To submit a *loss* claim the user has to call the *claimLost* method of the *RCVToken* contract using the corresponding *recovery* account. This requires a fee to be paid and its value can be discovered by calling the *arbitrationCost* method in the *Profiles* contract. The *claimLost* call will trigger other contract interactions: first the *createLossClaim* method is called which creates

a new loss claim followed by a call to *addNewClaim* od the *Profiles* contract which links it to the claimant's profile. Then, depending on the specific account configuration there is a time window in which this claim can be cancelled (by calling the *cancelLossClaim* method using the account that is claimed to be lost). This is necessary in case of someone falsely claiming that an account is lost, by gaining access to its recovery account instead. This time window allows the owner of the address to deny the claim therefore proving that the account was in fact not lost.

### Theft claim

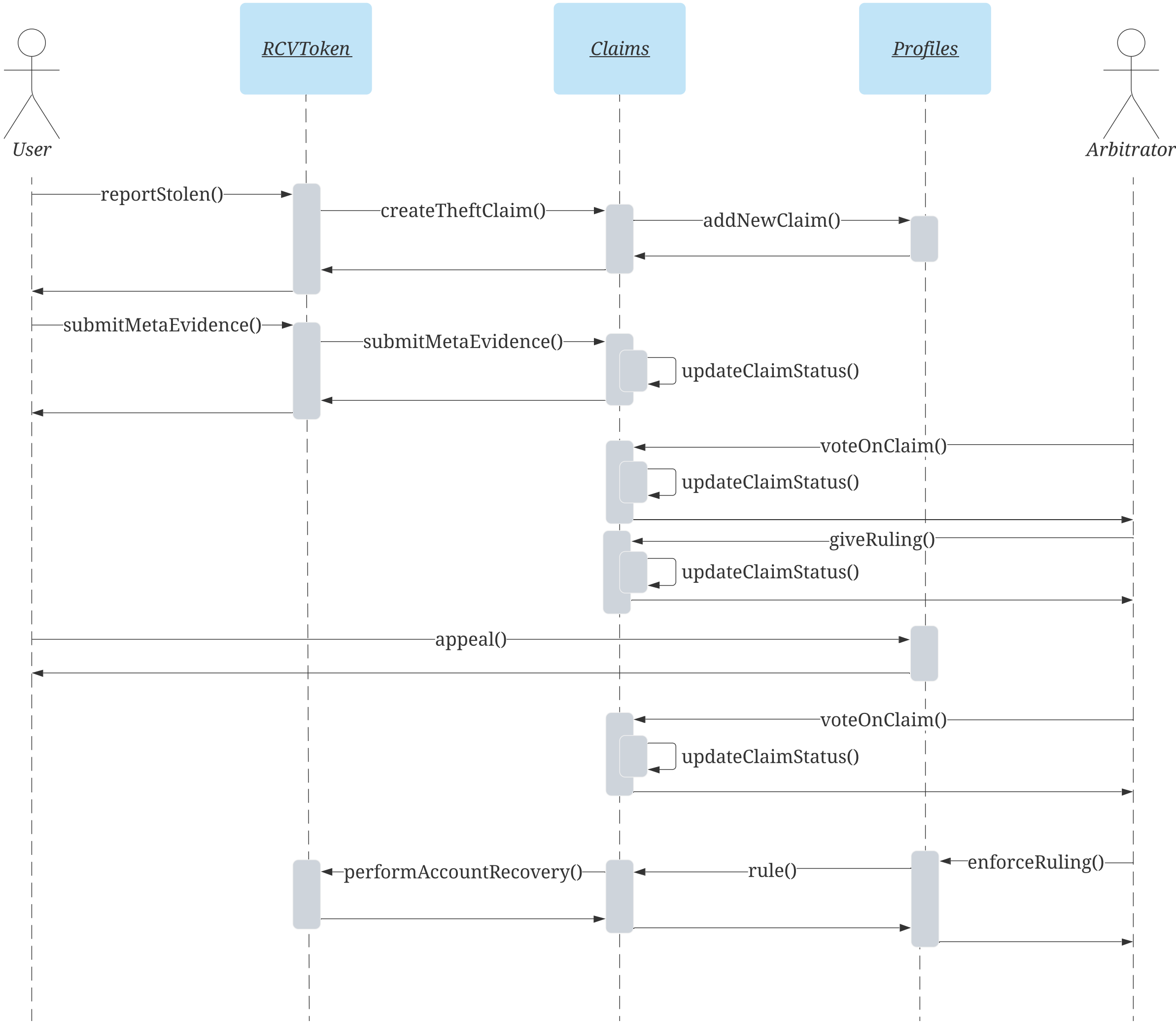


Figure 3.3: Example of a successful theft claim.

In case of a *theft* claim, the process is very similar with only some slight differences. First of all, only accounts that have a *proof of ownership* linked to them can be reported as stolen. To start a theft claim the user has to call the *reportStolen* method of the *RCVToken* contract. Another difference when compared to the *account loss* scenario is that there is no time window in which the claim can be cancelled but instead the tokens owned by the account are *frozen*.

### Chargeback claim

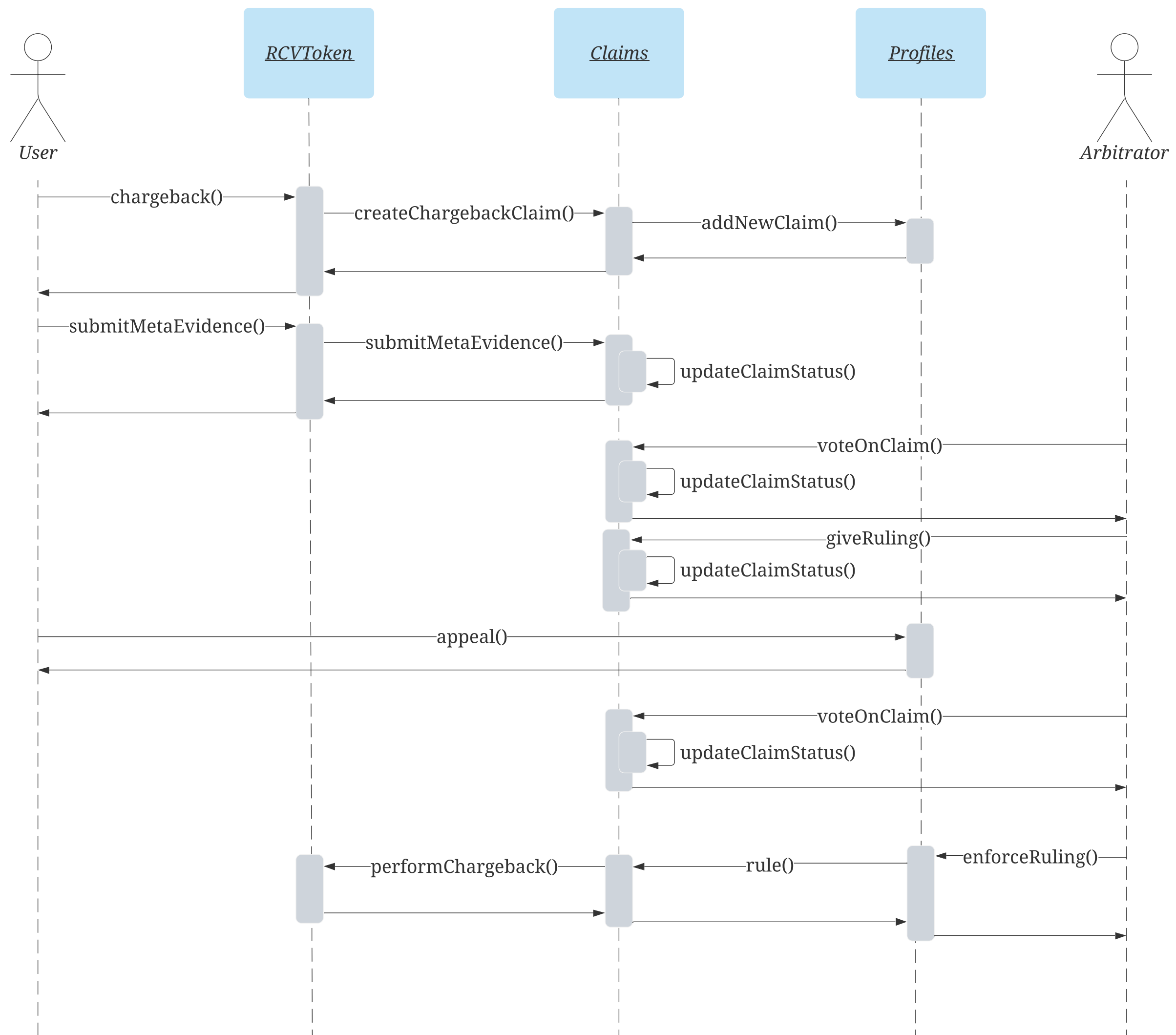


Figure 3.4: Example of a successful chargeback claim.

Finally, when dealing with *chargeback* scenarios, a request is submitted to chargeback a *pending transfer* via the *chargeback* method. A transfer is considered pending when the amount of time specified by the account configuration (which can be discovered by calling *getPendingTransferTime*) has not yet passed since the transfer was performed.

### Meta evidence

The next step is to escalate the claim to a dispute. However, in order to do that, a *meta evidence* file has to be linked to the ongoing claim. This mechanism is supported by an evidence submission standard proposal [51] that enables linking evidence to disputes. That file contains some form of argument towards the resolution of the dispute, i.e., towards the recovery being accepted. The processing of the documents is done by the human arbitrators, so the format of

```
/* MetaEvidence.json */
{
    fileURI:
        "ipfs://QmWRUgLu9iRk...",
    fileHash:
        "QmWRUgLu9iRk...",
    fileTypeExtension: ".txt",
    category: "Loss Claim",
    description: "I lost access to my address.",
    question:
      "Should the tokens be transferred to the specified recovery account?",
    rulingOptions: {
      type: "single-select",
      titles: ["Yes", "No"],
      descriptions: [
        "The account is indeed lost
        Tokens will be transferred to the specified recovery account",
        "There is not enough proof to conclude that the account is lost.
        Tokens will remain in the account.",
      ],
    },
  }
```

Listing 3.1: Example meta evidence file.

the files is opaque to the system. Its purpose is to provide information regarding the context of the dispute and to reference an URI to a file which is the basis of the dispute, e.g. a document showing that the incident was reported to the local police department (an example of the meta evidence file is in Listing 3.1). Without this file claims are unable to escalate to disputes.

If all the conditions are met according to the type of claim then it will escalate to a dispute and we move on to the *dispute resolution* mechanism and a fee is split between the participating arbitrators.

### 3.2.3 Dispute resolution

In order to perform any recovery actions, first a *claim* has to be submitted, then escalated to a *dispute* and finally approved. This decision is the result of the dispute resolution mechanism. The method we decided to use to choose arbitrators relies on address whitelisting. This means that there is a known group of arbitrators who are trusted by the community to resolve disputes by voting. A new arbitrator is able to join the group if all the current members agree on it and the same process is used to remove a member.

The linking between a claim and the file is then performed by executing a smart contract call such as *submitMetaEvidence* which will end up emitting an *event* that, in turn, is stored in the event log of the resulting transaction. Users are then able to query the blockchain for events

emitted by the smart contract and retrieve all evidence related to a particular dispute.

#### Voting

Our dispute resolution mechanism starts with the creation of a new dispute which is triggered after the steps explained in Section 3.2.2 are successfully complete.

At this point, the users who are participating entities in the dispute (e.g. in the case of a *chargeback* dispute the participating entities are the claimant and the user who received the token) are now able to submit additional evidence using the same evidence submission mechanism mentioned in Section 3.2.2 but now resorting to the *submitEvidence* method call instead.

When it comes to the arbitrators job, they are able to analyze all the evidence related to the dispute and eventually commit to a ruling decision. This commitment is made known after an arbitrator makes a call to the *voteOnClaim* method. All disputes have a *voting period* where if there are not sufficient votes, the dispute is *cancelled*. In our system, we define that at least two thirds plus one arbitrators $(2f + 1)$ have to vote. This is the same proportion as in common Byzantine fault-tolerant consensus algorithms [55]; it ensures that there are at most $f$ malicious $(n \geq 3f + 1)$ and a majority of the $2f + 1$ arbitrators are not malicious. A ruling is determined based on the decisions of the participating arbitrators when *giveRuling* is called. At this point in time an appeal period begins.

#### Appeals

After the ruling is acknowledged, the claimant has an opportunity to appeal that decision during a previously defined time period. The request for an appeal is started through a call to the *appeal* method and it will require an additional fee from the appellant (which could be the claimant and additionally, in chargeback claims, the respondent). This fee will be used to pay the new set of arbitrators that will be selected to participate in this new round of voting. The whole voting process is then repeated from the initial voting to the final ruling.

### 3.2.4 Token recovery

We then reach the final step of the whole process and assume that the decision given by the dispute resolution can be one of two: an *approval* or a *denial*. If the dispute resulted in denying the claim then all that is left to do is to distribute the fee between all the participating arbitrators as a reward for their work. However, if the dispute resulted in a claim approval then depending on the type of claim a *recovery action* will be performed. These actions are enforced by final

call to the *enforceRuling* method. For both *loss* and *theft* claims the recovery action is to transfer the tokens from the lost or stolen account to the recovery account. When it comes to *chargebacks*, the recovery action is to perform another transfer that will revert the one being claimed (essentially creating a new one but swapping the source and destination).

## 3.3 Recoverable Token Application

As user interface (UI), we created an application tailored to both regular users and arbitrators (RCV App in Figure 3.1). For a *regular user*, the application allows: signing up to the Recoverable Token system; generating and submitting proof of ownership; submitting new claims; viewing on-going claims and disputes; viewing pending transfers; submitting evidence; appeal the rulings given to disputes. For *arbitrators* the application allows them to: view claims and disputes; access the evidence linked to disputes; rule (or vote) on disputes.

```
/***************\
|    RCV APP    |
\***************/

Swarm listening on /ip4/127.0.0.1/tcp/4002/p2p/Qmd5c4xWVXQ5tRuuQBoF9pH3VXRk43U3gQqqDENz8Q42Uk
Swarm listening on /ip4/192.168.1.66/tcp/4002/p2p/Qmd5c4xWVXQ5tRuuQBoF9pH3VXRk43U3gQqqDENz8Q42Uk
Swarm listening on /ip4/172.19.0.1/tcp/4002/p2p/Qmd5c4xWVXQ5tRuuQBoF9pH3VXRk43U3gQqqDENz8Q42Uk
Swarm listening on /ip4/127.0.0.1/tcp/4003/ws/p2p/Qmd5c4xWVXQ5tRuuQBoF9pH3VXRk43U3gQqqDENz8Q42Uk

? What methods do you need to access? User
? Call a method
  submitMetaEvidence
  appeal
  transfer
❯ signUp
  addRecoveryInfo
  mintCar
  getClaim
(Move up and down to reveal more choices)
```

Figure 3.5: Using the RCV App to call the signUp method of the RCVToken smart contract.

Figure 3.5 shows the RCV App command line interface. When the app is launched, it starts by booting up an IPFS node and then shows a list of the available interfaces, i.e. *User*, *Arbitrator* and *Utility*. Each interface has a set of methods that are expected to be called by their respective types, e.g. the submitMetaEvidence method is expected to be called by a user. When one of the methods is selected the necessary inputs to perform that method are requested. Next, when the inputs are provided then a transaction is created and broadcast to the network. Finally, when the transaction is confirmed or if any error occurs (e.g. invalid inputs, insufficient gas) then the result is shown to the user.

## 3.4 Implementation

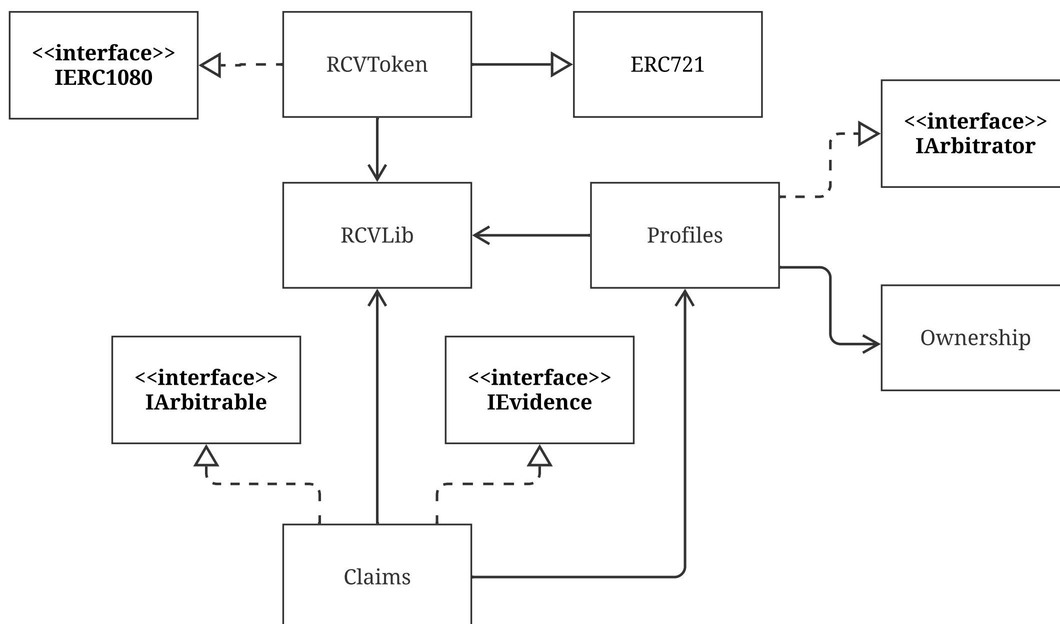


Figure 3.6: Inheritance relationships of the smart contracts.

To implement the bulk of the Recoverable Token functionalities we developed a set of smart contracts written in the Solidity programming language.

Figure 3.6 shows the inheritance relationships between the main types of contracts. *Profiles* is the smart contract that holds information about all the addresses that signed up to use the application and their disputes. It implements the IArbitrator interface of the ERC-792 arbitration standard we follow [50]. Additionally, it references the *Ownership* library which has the necessary functions to validate the *proof of ownership* of an account. Depicted in Listing 3.2 is *verifyProof* function of the *Ownership* library which is used to verify the legitimacy of the *proof of ownership* message. It starts by reconstructing the message (as detailed in Section 3.2) and then computing its *keccak256* hash. Then it extracts the data ($v$, $r$ and $s$ parameters) necessary to verify the signature and using the *ecrecover* function it extracts the signer's address. Finally, if the signer's address matches the *recoveryAccount* then the *proof of ownership* is valid, otherwise it is not.

The role of the *Claims* contract is to hold the information related to any sort of dispute, i.e., loss, theft or chargeback claims. Through it arbitrators can vote on claims and enforce rulings by calling the methods shown in Table 3.1.

Finally, the *RCVToken* contract implements the Recoverable Token interface and extends the token that is the target of recovery, e.g., an ERC-20 or ERC-721 token. A non-arbitrator user may perform the necessary actions to recover the tokens belonging to an account via the

```
/* Ownership.sol */

function verifyProof(
  string memory identity,
  address account,
  address recoveryAccount,
  bytes memory proof
) public returns (bool success) {
  uint8 v;
  bytes32 r;
  bytes32 s;
  bytes32 msgHash = prefixed(
      keccak256(abi.encodePacked(identity, account, recoveryAccount))
  );

  (v, r, s) = splitSignature(proof);

  address signer = ecrecover(msgHash, v, r, s);
  if (signer == recoveryAccount) {
      emit ProofVerified(account, signer);
      return true;
  }
  return false;
}
```

Listing 3.2: Proof of ownership verification function.

methods described in Table 3.1. Depending on the type of token being extended, there is a need to modify the token transfer function so that the account freezing functionality may be added. An example of what modifications are required is seen in Listing 3.3. The modified *transferFrom* function starts by checking if the sender's account is *frozen*. If it is not, then it calls the *transferFrom* function of the parent contract, i.e. the contract of the token it extends, and completes the transfer by adding it to the account's *pending transfers*.

```
/* RCVToken.sol */

function transferFrom(address from, address to, uint256 tokenID) public override {
    RCVLib.Profile memory profile = _profiles.getAccountProfile(from);
    require(!profile.isClaimedStolen, "FROZEN");
    super.transferFrom(from, to, tokenID);
    uint256 transferNumber = _profiles.addTransfer(from, to, tokenID);
    emit PendingTransfer(from, to, tokenID, transferNumber);
}
```

Listing 3.3: Changes to transferFrom function.

All the previously mentioned contracts use a library – *RCVLib* – that stores definitions of structures shared between them.

## 3.5 Summary

This Chapter provided an outline of the design of the proposed solution. It started by presenting the overall architecture and the assumptions made regarding the attack model. Then it explains, from start to finish, the process of performing a recovery of the digital assets linked to an account. Afterwards, a brief description of the application for users to interact with the Recoverable Token system is given. The chapter concludes with implementation details of a proof of concept Recoverable Token system.

# Chapter 4

# Evaluation

In this chapter we discuss the evaluation of our system. Our goal with this evaluation is to ascertain if it is feasible – mainly in terms of cost – to perform an account recovery. We start by describing the evaluation methodology and then proceed to interpret the obtained results.

## 4.1 Evaluation Methodology

For our evaluation we applied the *Recoverable Token* system to an application [1] that makes use of an ERC-721 token. We deployed the smart contracts using *Truffle* [2] on *Ropsten*, a public *test network* for Ethereum that to most extent mimics the Ethereum main network. Using a test network allowed us to request Ether from publicly available faucets for free (although the amount is limited depending on the faucet used). All smart contracts were compiled with solc v0.6.8. In the evaluation we assess source code metrics (Section 4.2) for our smart contracts as well as gas (Section 4.3) and time (Section 4.4) for the methods commonly used by all three main use cases of the system: trying to recover from the scenarios *S1*, *S2* and *S3* (Chapter 1).

For *gas* and *time*, the metrics represented are the result of calculating the average of 10 claims performed in the Ropsten testnet. In every scenario the account to be recovered only holds one token and there are three arbitrators ruling on the claim. All method calls were performed resorting to a modified version of our *RCVApp* which output metrics for each method executed and used *Infura* [3] nodes to connect to the Ethereum network. In the context of a single claim, it also executed each method call immediately after the previous one had been confirmed in the blockchain, i.e. included in a valid block.

Before moving on to the interpretation of the results gathered in our experimental evaluation

[1] `https://github.com/CodinMaster/Crypto-Car-Battle`
[2] `https://www.trufflesuite.com/docs/truffle/overview`
[3] `https://infura.io/docs`

it is necessary to give more insight to what the meaning of the *time consumption* metric is in this context. In EVM-based blockchains, *computational effort* is not measured in CPU cycles nor wall-clock time, but rather in *gas consumption.* Therefore, what we are actually measuring when we refer to *time consumption* is the time it took from creation the transaction locally to it being confirmed in the blockchain. In between those two events are a number of steps that contribute to the total execution time: *a)* making a call to the *Infura* API endpoint responsible for broadcasting the transaction to the Ethereum network; *b)* that same transaction being included in a valid block by a miner node; *c)* the block that includes the transaction being broadcast to the network until it is received by the node that we are interacting with; *d)* receiving the response from the call made in step *a).* At this point we consider the transaction to be completed.

Due to the fact that we do not have control over all entities involved in these steps it is infeasible for us to individually measure the time it took each of these steps to complete. However, in normal conditions, we can safely assume that the ones that most contribute to the total time are *b)* and *c).* The time to complete those two steps is heavily influenced by the current blockchain network conditions, e.g. participating nodes and number of pending transactions. Although knowing the exact value of the total number of nodes contributing to a network is difficult [57], it is reasonable to assume that test networks such as *Ropsten* are more volatile when it comes to the rate at which nodes join and leave the network. Moreover, since Ropsten contains significantly less nodes it means that volatility is more impactful. This means that the exact same transaction might take a largely different amount of time to be executed depending on the network state.

## 4.2 Source code

Table 4.1: Recoverable Token contracts' source code metrics.

| **Contract** | **Deployed Bytecode** | **Gas** | **Cost (USD)** | **LoC** |
|---|---|---|---|---|
| RCVToken | 15703 bytes | 3635595 | $171.02 | 175 lines |
| Claims | 14469 bytes | 3280773 | $154.33 | 345 lines |
| Profiles | 10100 bytes | 2294378 | $107.93 | 305 lines |
| Ownership | 1126 bytes | 295402 | $13.90 | 51 lines |
| Total | 41398 bytes | 9506148 | $447.17 | 876 lines |

To get an idea of the deployment costs and the overhead the system would introduce, four different source code metrics were gathered: *deployed bytecode*, which is the size of bytecode (in

bytes) that is stored on-chain; *gas*, the amount of gas used to deploy the contract; *cost*, how much it would cost (in USD) to deploy each contract; and finally *lines of code*, the number of source code lines – excluding comments – after running a code formatter. To calculate the cost we used the approximate average values of *gas price* ($96 \times 10^{-9} ether$), i.e. the fee paid for each gas unit, and *Ether price* ($490 USD), i.e. the market value for 1 *Ether*, from the month of November [4].

Looking at Table 4.1, we notice that the *RCVToken* contract has the highest bytecode size per lines of code ratio. This is due to the fact that it extends the contract that implements the ERC-721 token which already has a size of 6836 bytes, therefore having an overhead of 8867 bytes. Furthermore, as is to be expected, the amount of gas required to deploy the smart contract increases linearly with the size of the deployed bytecode. Note that the values of the size of the *deployed bytecode* vary with the compiler used as well as the optimizer settings. For reference, the configuration used is shown in Listing 4.1.

In terms of costs, most people would consider them too high. A total of $447.17 (USD) just to deploy the contracts might seem unreasonable, but it is no cause for concern. First, this is a one-time cost since it is only necessary to deploy the contracts once. Second, scalability issues [58] have plagued both the Bitcoin and Ethereum blockchains in the past few years. As their popularity increases the network becomes more congested which causes fees to raise. Solutions for the scalability issues [59, 60] are being proposed and worked on. Third, even if these solutions are disregarded, it is possible to significantly reduce the cost by lowering *gas price*. The trade-off would be that the transaction would take more time to confirm as the ones which offer an higher value would be prioritized by the miners.

```
/* truffle-config.js */

compilers: {
  solc: {
    version: '0.6.8',
    settings: {
      optimizer: {
        enabled: true,
        runs: 1500,
      },
    },
  },
},
```

Listing 4.1: Solidity compiler settings in Truffle.

[4] All historical data was gathered from: `https://www.etherscan.io/`

## 4.3 Gas consumption

The metrics collected for our gas evaluation were: the amount of *gas* used; the amount of gas used relative to the total (in percentage); the *cost* (in USD) of executing the transaction; and the standard deviation ($\sigma$) of the obtained values.

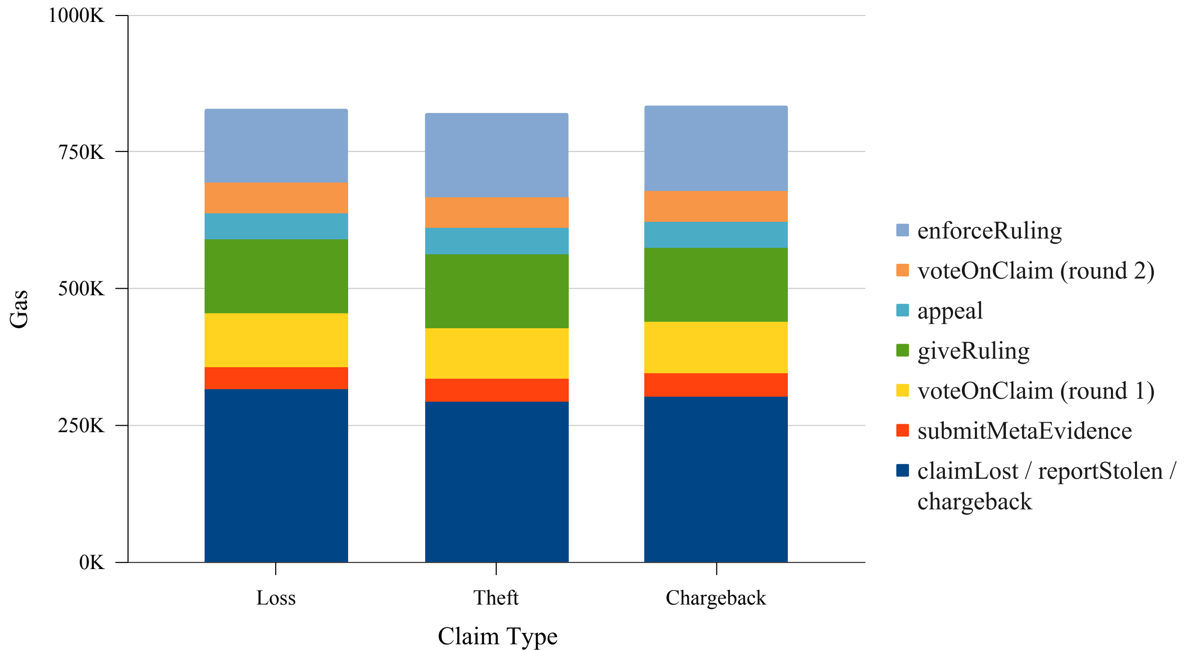


Figure 4.1: Average gas consumed by methods for each type of claim.

As far as gas consumption is concerned, we notice that in each scenario the most computationally expensive operation is the one that is responsible for submitting the claim as shown in Figure 4.1. This corresponds to *claimLost* for lost claims, *reportStolen* for stolen claims and *chargeback* for chargeback claims.

Note that some operations do not have a constant gas cost, i.e. a non-zero standard deviation, which could be attributed to the initialization and iteration of the data structures that store information about the different types of claims. This is supported by the fact that, for the *claimLost*, *reportStolen* and the *submitMetaEvidence* methods, its value only deviates once. However this is not true for *chargeback* calls since, apart from having to create the necessary data structures, it also has extra logic to manage pending transfers. All the other methods have a constant gas cost and this is not surprising since, in each scenario, they were called with the same arguments and so the same instructions were executed (they are deterministic functions).

Table 4.2: Gas breakdown for the Loss, Theft and Chargeback dispute resolution scenarios.

| Action | Gas | Gas (%) | Cost (USD) | $\sigma$ |
|---|---|---|---|---|
| **Loss claim dispute resolution** | | | | |
| claimLost | 319035 | 38.52% | $15.01 | 9487 |
| submitMetaEvidence | 41546 | 5.02% | $1.95 | 531 |
| voteOnClaim (round 1) | 93406 | 11.28% | $4.39 | 0 |
| giveRuling | 134627 | 16.26% | $6.33 | 0 |
| appeal | 48323 | 5.83% | $2.27 | 0 |
| voteOnClaim (round 2) | 56768 | 6.85% | $2.67 | 0 |
| enforceRuling | 134467 | 16.24% | $6.33 | 0 |
| Total | 825172 | 100.00% | $38.95 | - |
| **Theft claim dispute resolution** | | | | |
| reportStolen | 293482 | 35.76% | $13.81 | 4743 |
| submitMetaEvidence | 42205 | 5.14% | $1.99 | 0 |
| voteOnClaim (round 1) | 92557 | 11.28% | $4.35 | 0 |
| giveRuling | 133774 | 16.30% | $6.29 | 0 |
| appeal | 48323 | 5.89% | $2.27 | 0 |
| voteOnClaim (round 2) | 55915 | 6.81% | $2.63 | 0 |
| enforceRuling | 154335 | 18.82% | $7.26 | 0 |
| Total | 820591 | 100.00% | %38.60 | - |
| **Chargeback claim dispute resolution** | | | | |
| chargeback | 303209 | 36.37% | $14.26 | 6415 |
| submitMetaEvidence | 43054 | 5.16% | $2.03 | 0 |
| voteOnClaim (round 1) | 93406 | 11.20% | $4.39 | 0 |
| giveRuling | 134623 | 16.15% | $6.33 | 0 |
| appeal | 48323 | 5.80% | $2.27 | 0 |
| voteOnClaim (round 2) | 56764 | 6.81% | $2.67 | 0 |
| enforceRuling | 154278 | 18.51% | $7.28 | 0 |
| Total | 833656 | 100.00% | $39.23 | - |

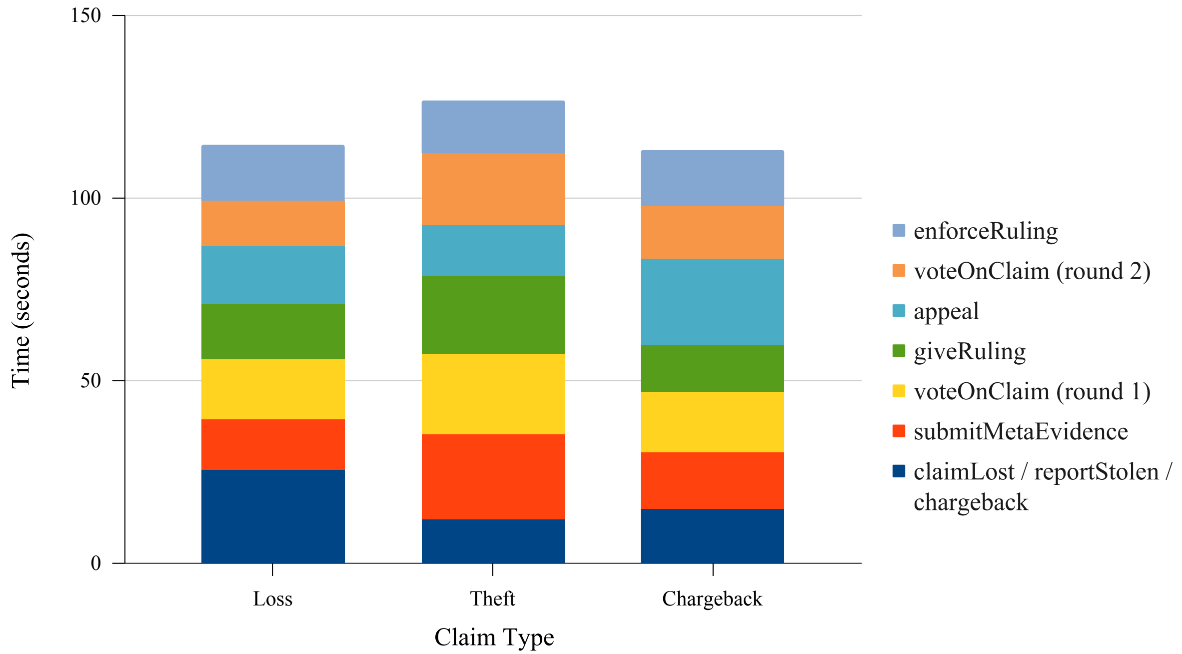


Figure 4.2: Average time to execute methods for each type of claim.

## 4.4 Time consumption

In this Section we present and analyse the results obtained in regards to time related metrics. Figure 4.2 is a chart of the *time* data shown in Table 4.3. As anticipated, there is no clear pattern that allows us to determine how long the execution of a method should take. The *appeal* method is a good example to demonstrate this point. Looking at the chart, we notice that it is not on average consistently faster or slower than the others. In the *theft* scenario it is the second fastest on average (13.66 seconds) and in the *chargeback* scenario it is the slowest (23.64 seconds). Although one of the reasons for this is due to an outlier – in one of the runs it took over 60 seconds for the transaction to execute – even if it is ignored the point still stands since, by looking at the gas usage in Table 4.2, the computational effort of executing it is exactly 48323 gas in all three scenarios.

On average, the time it took to complete each operation is between 10 to 30 seconds. These values were to be expected since this is the rate at which blocks in the Ropsten network are mined. Moreover, the relatively high standard deviation values for most operations are an indicator of how volatile the Ropsten network activity can be.

Table 4.3: Time breakdown for the Loss, Theft and Chargeback dispute resolution scenarios.

| Action | Time (s) | Time (%) | $\sigma$ |
|---|---|---|---|
| **Loss claim dispute resolution** | | | |
| claimLost | 25.48 | 22.27% | 13.44 |
| submitMetaEvidence | 13.82 | 12.08% | 12.24 |
| voteOnClaim (round 1) | 16.40 | 14.34% | 12.32 |
| giveRuling | 15.11 | 13.20% | 16.89 |
| appeal | 16.08 | 14.05% | 10.18 |
| voteOnClaim (round 2) | 12.38 | 10.82% | 6.57 |
| enforceRuling | 15.17 | 13.24% | 11.82 |
| Total | 114.44 | 100.00% | - |
| **Theft claim dispute resolution** | | | |
| reportStolen | 11.83 | 9.35% | 8.92 |
| submitMetaEvidence | 23.62 | 18.67% | 16.26 |
| voteOnClaim (round 1) | 21.71 | 17.16% | 18.75 |
| giveRuling | 21.60 | 17.07% | 15.24 |
| appeal | 13.66 | 10.80% | 5.76 |
| voteOnClaim (round 2) | 19.65 | 15.53% | 13.29 |
| enforceRuling | 14.45 | 11.42% | 9.79 |
| Total | 126.52 | 100.00% | - |
| **Chargeback claim dispute resolution** | | | |
| chargeback | 14.84 | 13.14% | 8.05 |
| submitMetaEvidence | 15.60 | 13.81% | 9.68 |
| voteOnClaim (round 1) | 16.44 | 14.55% | 10.90 |
| giveRuling | 12.80 | 11.33% | 6.82 |
| appeal | 23.64 | 20.92% | 19.73 |
| voteOnClaim (round 2) | 14.30 | 12.66% | 7.75 |
| enforceRuling | 15.35 | 13.59% | 12.85 |
| Total | 112.97 | 100.00% | - |

## 4.5 Summary

This chapter presented the results of our experimental evaluation. It started by describing the scenarios in which the experiments were ran and the metrics that were measured. The Chapter concluded with an analysis of the results obtained from the performed experimental evaluation.

# Chapter 5

# Conclusions

In this chapter, we conclude this document by summing up our accomplishments and proposing features and research to improve upon this work.

## 5.1 Achievements

This document presented Recoverable Token, a system that combines several standards in order to provide an opportunity for recovering digital assets stored on the Ethereum blockchain without modifying its fundamental properties. We believe this to be useful to increase adoption of blockchain systems as a means for storing value and performing transactions.

To evaluate our system we applied it to an application that implements an ERC-721 token and demonstrated that it is possible to recover the tokens in a variety of scenarios, i.e. *account loss*, *account theft* and *chargeback*.

Our evaluation has shown that the mechanism allows doing recovery in a reasonable amount of time and at a reasonable cost, given the benefits of being able to do such an operation that is currently not supported in Ethereum or any other EVM-based blockchain.

## 5.2 Future Work

In spite of all the work that was developed, it contains several aspects that could be improved upon.

### 5.2.1 Privacy

If we consider our *evidence submission* mechanism, its current design allows anyone to have access to the evidence submitted for each and every claim. One possible improvement would be to create a permissioned system to only give access to the arbitrators assigned to each claim.

We studied the possibility of leveraging the *Enigma* [61] project in our system and believe that, when integration with the Ethereum blockchain is implemented it can be an interesting approach. With Enigma it would be possible to perform computation on encrypted data and use it in Ethereum contracts.

Another privacy concern is when related to the current *voting* mechanism. Again, just like all evidence is publicly available so are the votes submitted by arbitrators. Although there is no direct incentive for arbitrators to vote according to the majority, we argue that implementing a commit-reveal mechanism would still be an improvement. This would prevent arbitrators from being influenced by the already submitted votes.

### 5.2.2 Decentralization

Decentralization is one of the most desired features of a public blockchain. As mentioned in Section 3.2.3, the method used to select arbitrators also has the drawback of introducing a degree of centralization since users can only become arbitrators if all current arbitrators approve it. There are more decentralized alternative methods to select arbitrators such as the one used by both Kleros [47] and Aragon [48] which essentially requires users to stake their tokens so that they are allowed to arbitrate.

## 5.3 Final remarks

The research performed in this work leads us to believe that the study of recovery mechanisms for blockchain systems is worth pursuing. As mentioned throughout this document, there are a variety of use cases for these systems, from trying to fix user mistakes to recovering from exploits that might compromise the trust in the blockchain as a whole [21]. Although it is common for works in this area to sometimes be disregarded as they commonly break the immutability inherent to the blockchain, opt-in approaches similar to the work we presented, are in our opinion, a good starting point. It gives users the option to rely on recovery systems if they feel the need to do so, which could lead to an increase in adoption of blockchain since it introduces the possibility of recovering from scenarios where it would not be possible otherwise.

# Bibliography


[1] S. Nakamoto. Bitcoin: A peer-to-peer electronic cash system. online, 2008.

[2] G. Wood et al. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper*, 151(2014):1–32, 2014.

[3] C. McFarlane, M. Beer, J. Brown, and N. Prendergast. Patientory: A healthcare peer-to-peer EMR storage network v1. *Entrust Inc.*, 2017.

[4] P. Snow, B. Deery, J. Lu, D. Johnston, and P. Kirby. Factom: Business processes secured by immutable audit trails on the blockchain. *Whitepaper*, Nov. 2014.

[5] D. Serranito, A. Vasconcelos, S. Guerreiro, and M. Correia. Blockchain ecosystem for verifiable qualifications. In *Proceedings of the 2nd IEEE Conference on Blockchain Research & Applications for Innovative Networks and Services*, Sept. 2020.

[6] F. Vogelsteller and V. Buterin. EIP 20: ERC-20 Token Standard. URL `https://eips.ethereum.org/EIPS/eip-20`. [Online]. Accessed: 2020-12-05.

[7] W. Entriken, D. Shirley, J. Evans, and N. Sachs. EIP 721: ERC-721 Non-Fungible Token Standard. URL `https://eips.ethereum.org/EIPS/eip-721`. [Online]. Accessed: 2020-12-05.

[8] A. M. Antonopoulos and G. Wood. *Mastering Ethereum: building smart contracts and dapps*. O'Reilly Media, 2018.

[9] V. Buterin. Deterministic wallets, their advantages and their understated flaws, 2013. URL `https://bitcoinmagazine.com/articles/deterministic-wallets-advantages-flaw-1385450276`. [Online]. Accessed: 2020-12-05.

[10] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr. Basic Concepts and Taxonomy of Dependable and Secure Computing. *IEEE Trans. Dependable Secur. Comput.*, 1(1):11–33, Jan. 2004. ISSN 1545-5971. doi: 10.1109/TDSC.2004.2.

[11] A. B. Brown and D. A. Patterson. Undo for operators: Building an undoable e-mail store. In *USENIX Annual Technical Conference*, pages 1–14, 2003.

[12] A. Goel, K. Po, K. Farhadi, Z. Li, and E. de Lara. The Taser Intrusion Recovery System. *SIGOPS Oper. Syst. Rev.*, 39(5):163–176, Oct. 2005. ISSN 0163-5980. doi: 10.1145/1095809.1095826.

[13] T. Kim, X. Wang, N. Zeldovich, and M. F. Kaashoek. Intrusion Recovery Using Selective Re-execution. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, pages 89–104, Berkeley, CA, USA, 2010. USENIX Association.

[14] T. Kim, R. Chandra, and N. Zeldovich. Recovering from Intrusions in Distributed Systems with DARE. In *Proceedings of the Third ACM SIGOPS Asia-Pacific Conference on Systems*, APSys '12, pages 10–10, Berkeley, CA, USA, 2012. USENIX Association.

[15] D. R. Matos, M. L. Pardal, and M. Correia. Rectify: Black-box Intrusion Recovery in PaaS Clouds. In *Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference*, Middleware '17, pages 209–221, New York, NY, USA, 2017. ACM. ISBN 978-1-4503-4720-4. doi: 10.1145/3135974.3135978.

[16] S. Haber and W. S. Stornetta. How to Time-stamp a Digital Document. *J. Cryptol.*, 3(2): 99–111, Jan. 1991. ISSN 0933-2790. doi: 10.1007/BF00196791.

[17] M. Castro and B. Liskov. Practical Byzantine fault tolerance and proactive recovery. *ACM Transactions on Computer Systems*, 20(4):398–461, 2002.

[18] A. Bessani, J. Sousa, and E. E. P. Alchieri. State machine replication for the masses with BFT-Smart. In *Proceedings of the IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 355–362, June 2014.

[19] P. Piasecki. What is a transaction's step-by-step life cycle?, 09 2017. URL `https://bitcoin.stackexchange.com/a/4731`. [Online]. Accessed: 2020-12-05.

[20] R. Viorescu et al. 2018 reform of eu data protection rules. *European Journal of Law and Public Administration*, 4(2):27–39, 2017.

[21] C. Jentzsch. Decentralized autonomous organization to automate governance. *Whitepaper*, Nov. 2016.

[22] G. Ateniese, B. Magri, D. Venturi, and E. Andrade. Redactable blockchain–or–rewriting history in bitcoin and friends. In *2017 IEEE European Symposium on Security and Privacy*, pages 111–126, 2017.

[23] D. Deuber, B. Magri, and S. A. K. Thyagarajan. Redactable blockchain in the permissionless setting. In *2019 IEEE Symposium on Security and Privacy*, pages 124–138, 2019.

[24] O. N. Challa. Reversecoin: Worlds First Cryptocurrency With Reversible Transactions. https://docs.google.com/document/d/1hMCkEQUYm9oFCQpxtIWFqVpt66pTQn1zCDW8WX0b7hw/, 2014. [Online]. Accessed: 2020-12-05.

[25] Opt-in Replace by Fee FAQ. URL https://bitcoincore.org/en/faq/optin_rbf/#i-heard-opt-in-rbf-was-added-with-little-or-no-discussion. [Online]. Accessed: 2020-12-05.

[26] Bitcoin Improvement Proposal 125. URL https://github.com/bitcoin/bips/blob/master/bip-0125.mediawiki. [Online]. Accessed: 2020-12-05.

[27] T. Lindholm, F. Yellin, G. Bracha, and A. Buckley. *The Java virtual machine specification*. Pearson Education, 2014.

[28] G. A. Pierro and H. Rocha. The influence factors on ethereum transaction fees. In *IEEE/ACM 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain*, pages 24–31, 2019.

[29] E. Foundation. Geth: Official Go implementation of the Ethereum protocol. https://github.com/ethereum/go-ethereum. [Online]. Accessed: 2020-12-05.

[30] OpenEthereum: The fast, light, and robust client for the Ethereum mainnet. https://github.com/openethereum/openethereum. [Online]. Accessed: 2020-12-05.

[31] Hyperledger Besu: An enterprise-grade Java-based, Apache 2.0 licensed Ethereum client. https://github.com/hyperledger/besu/. [Online]. Accessed: 2020-12-05.

[32] The Solidity Contract-Oriented Programming Language. https://github.com/ethereum/solidity. [Online]. Accessed: 2020-12-05.

[33] Vyper: Pythonic Smart Contract Language for the EVM. https://github.com/vyperlang/vyper. [Online]. Accessed: 2020-12-05.

[34] T. Krajewski and R. Lettiere. Efforts Integrating Blockchain with Intellectual Property. *les Nouvelles - Journal of the Licensing Executives Society*, Volume LIV No. 1, Mar. 2019.

[35] M. O'Dair and Z. Beaven. The networked record industry: How blockchain technology could transform the record industry. *Strategic Change*, 26(5):471–480, 2017.

[36] V. P. Ranganthan, R. Dantu, A. Paul, P. Mears, and K. Morozov. A decentralized marketplace application on the ethereum blockchain. In *2018 IEEE 4th International Conference on Collaboration and Internet Computing (CIC)*, pages 90–97, 2018. doi: 10.1109/CIC.2018.00023.

[37] G. Fenu, L. Marchesi, M. Marchesi, and R. Tonelli. The ico phenomenon and its relationships with ethereum smart contract environment. In *2018 International Workshop on Blockchain Oriented Software Engineering (IWBOSE)*, pages 26–32. IEEE, 2018.

[38] F. Victor and B. K. Lüders. Measuring ethereum-based erc20 token networks. In I. Goldberg and T. Moore, editors, *Financial Cryptography and Data Security*, pages 113–129, Cham, 2019. Springer International Publishing. ISBN 978-3-030-32101-7.

[39] M. Fromberger and L. Haffke. Ico market report 2018/2019–performance analysis of 2018's initial coin offerings. *Available at SSRN*, 2019.

[40] P. Technologies. The Multi-sig Hack: A Postmortem. https://www.parity.io/the-multi-sig-hack-a-postmortem/, July 2017. [Online]. Accessed: 2020-12-05.

[41] P. Technologies. A Postmortem on the Parity Multi-Sig Library Self-Destruct. https://www.parity.io/a-postmortem-on-the-parity-multi-sig-library-self-destruct/, Nov. 2017. [Online]. Accessed: 2020-12-05.

[42] B. Leatherwood. EIP-1080: Recoverable Token [DRAFT], May 2018. URL https://eips.ethereum.org/EIPS/eip-1080. Ethereum Improvement Proposals, no. 1080 [Online]. Accessed: 2020-12-05.

[43] ERC-1080: RecoverableToken Standard discussion. URL https://ethereum-magicians.org/t/erc-1080-recoverabletoken-standard/364. [Online]. Accessed: 2020-12-05.

[44] F. Elkouri, E. A. Elkouri, and K. May. *How arbitration works.* Bureau of National Affairs Washington, DC, 1973.

[45] S. Mentschikoff. Commercial arbitration. *Columbia Law Review*, 61(5):846–869, 1961.

[46] D. Allen, A. Lane, and M. Poblet. The governance of blockchain dispute resolution - harvard negotiation law review, vol. 25. 25:75–101, 04 2020.

[47] C. Lesaege and F. Ast. Kleroterion, a decentralized court for the internet, June 2017.

[48] L. Cuende and J. Izquierdo. Aragon network: A decentralized infrastructure for value exchange. *Whitepaper*, 24:2018, 2017.

[49] The Mattereum Team. Mattereum protocol: Turning code into law, 2018. URL https://mattereum.com/wp-content/uploads/2020/02/mattereum-summary_white_paper.pdf. [Online]. Accessed: 2020-12-05.

[50] C. Lesaege. EIP 792: ERC-792 Arbitration Standard. URL https://github.com/ethereum/EIPs/issues/792. [Online]. Accessed: 2020-12-05.

[51] S. Vitello, C. Lesaege, and E. Piqueras. EIP 1497: ERC-1497 Evidence Standard. URL https://github.com/ethereum/EIPs/issues/1497. [Online]. Accessed: 2020-12-05.

[52] J. Benet. IPFS-content addressed, versioned, P2P file system. *arXiv preprint arXiv:1407.3561*, 2014.

[53] S. Wilkinson, T. Boshevski, J. Brandoff, and V. Buterin. Storj a peer-to-peer cloud storage network. 2014.

[54] J. Benet and N. Greco. Filecoin: A decentralized storage network. *Protoc. Labs*, pages 1–36, 2018.

[55] M. Correia. From byzantine consensus to blockchain consensus. In *Essentials of Blockchain Technology*, chapter 3. CRC Press, 2019.

[56] M. J. Dworkin. Sha-3 standard: Permutation-based hash and extendable-output functions. Technical report, 2015.

[57] S. K. Kim, Z. Ma, S. Murali, J. Mason, A. Miller, and M. Bailey. Measuring ethereum network peers. In *Proceedings of the Internet Measurement Conference 2018*, IMC '18, page 91–104, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450356190. doi: 10.1145/3278532.3278542. URL https://doi.org/10.1145/3278532.3278542.

[58] A. Singh, R. M. Parizi, M. Han, A. Dehghantanha, H. Karimipour, and K.-K. R. Choo. Public blockchains scalability: An examination of sharding and segregated witness. In *Blockchain Cybersecurity, Trust and Privacy*, pages 203–232. Springer, 2020.

[59] J. Poon and T. Dryja. The bitcoin lightning network: Scalable off-chain instant payments, 2016.

[60] J. Stark. Making sense of ethereum'slayer 2 scaling solutions: state channels, plasma, and truebit, 2018.

[61] G. Zyskind, O. Nathan, and A. Pentland. Enigma: Decentralized computation platform with guaranteed privacy. *arXiv preprint arXiv:1506.03471*, 2015.