Recoverable Token: Recovering from Intrusions against Digital Assets in Ethereum

Filipe Miguel Fernandes Martins
INESC-ID, Instituto Superior Técnico, Universidade de Lisboa, Portugal
filipe.f.martins@tecnico.ulisboa.pt

***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.**

***Index Terms*—Intrusion Recovery, Blockchain, Digital Assets, Tokens, Ethereum**

I. 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]. However the learning curve for using the technology is still relatively high for non tech-savvy users. A minimum requirement is to have a set of asymmetric key pairs which are used to sign transactions that are then submitted into the blockchain. 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.

A. Ethereum and Tokens

Ethereum [2] is a public blockchain that was announced in 2014. 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 – the *Ethereum Virtual Machine* (EVM) – that is Turing complete. The 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 their 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]–[8] that are handled at the smart contract level. Tokens are handled by smart contracts that are owned by one account, so they encompass a form of centralization.

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

When a wallet is used to create an account, it generates an asymmetric key pair and derives the account address from the generated public key. This 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 corresponding *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*

[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/

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.

B. Blockchain Recovery

There is a considerable corpus of research on *intrusion recovery*, i.e., on undoing the effects of intrusions from the perspective of the user [10]–[16]. These works focus on removing the effects of an intrusion from the state of a system: mail server, web application, file system, etc.

In the blockchain domain, instead, *recovery mechanisms* are still scarce, something that may be an obstacle for the wider adoption of this technology. Recovery mechanisms in this context fall in two categories: *data redaction* [17], [18] and *transaction reversion* [19], [20]. *Data redaction* refers to removing data stored on the blockchain. This is useful in scenarios where data needs to be hidden or removed from the blockchain, e.g., references to illegal pornography or sensitive and private information. Nevertheless, this is not the approach we are interested in this work.

Transaction reversion is a form of *blockchain rollback*, where the goal is to either undo specific actions or moving the state of the blockchain to a previous point in time. Transaction reversion mechanisms allow for example the rightful owner to recover his funds in case an address has been compromised, e.g., because an attacker obtains access to the private key and proceeds to transfer all the funds to an address he owns.

When addressing issues such as losing access to a private key of an account, none of the previous works is fit to solve them without breaking fundamental properties of blockchains, specifically their *immutability*. This property means that it is possible to append blocks to the blockchain, but not to modify the blocks that are already part of the chain.

C. Our Approach

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, Pantheon). Our intrusion recovery approach is even more generic and applies to many other blockchains.

Our approach 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; 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 contains the balance of tokens of each account.

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

- *account loss* – user lost the access to the private key of an account and/or corresponding seed phrases and can no longer recover them;
- *account theft* – there is reasonable proof to believe that an account has been compromised;
- *chargebacks* – payment is made for any good or service and the payer believes that it did not receive what was agreed upon.

We implemented our approach in Solidity, the most used high-level EVM language, and thoroughly evaluated its performance on the Ropsten testnet. 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 related blockchains.

The remainder of the paper is organized as follows. In Section II we give an overview of the architecture of the solution and describe its components. The process of recovering an account is explained in Section III. Section IV details the implementation and Section V its evaluation. Finally in Section VI we discuss related work and in Section VII we state our conclusions.

II. Recoverable Token

This section presents our approach in detail.

A. Recoverable Token Architecture

The architecture of the system is shown in Figure 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) [21]. Users need access to an IPFS node so that they are able to store and access data using the IPFS network. However, other

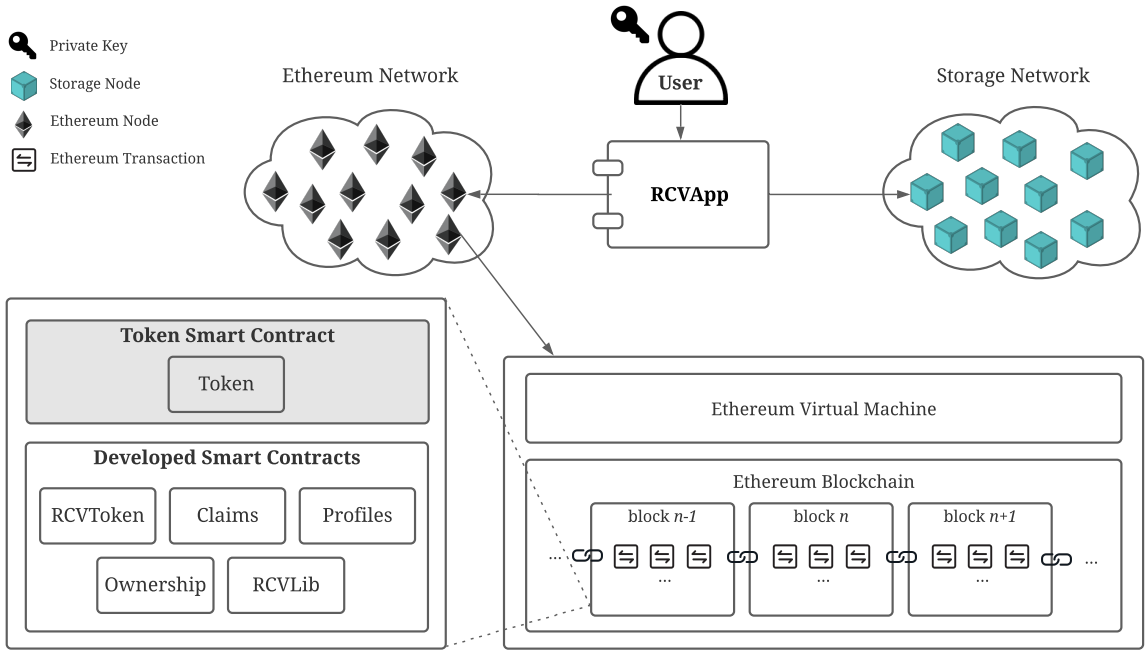

Fig. 1: Recoverable Token system architecture. A user which owns a private key is able to use the RCVApp to connect to a node of the Ethereum network and interact with the storage network. Each Ethereum node runs a local copy of an EVM and maintains a copy of the chain. The smart contracts that comprise the token recovery mechanism and the token which is to be recovered (shown in a grey background) are deployed on the blockchain.

decentralized storage systems [22], [23] may be used, each with their own trade-offs.

B. 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 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 [24]).

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).

C. Building Blocks

New features and standards for the Ethereum blockchain can be proposed through the submission of Ethereum Improvement Proposals (EIPs). Ethereum Request for Comments (ERCs) are subset of EIPs where application-level standards and conventions are proposed. These documents provide a technical specification and a rationale of the proposal. Our approach leverages some of these documents, although they are far from providing a full solution for our problem.

ERC-1080 is a proposal for a standard for an interface used to implement a recoverable token [25]. At the time of this writing, the proposal is still up for discussion and no implementations exist.

ERC-792 [26] and ERC-1497 [27] propose interfaces for arbitration and evidence submission, where a group of chosen arbitrators make rulings on disputes. The claimant is also able to submit evidence to a decentralized storage such as IPFS [21] to support his claim and then link the evidence to the dispute.

III. 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 I.

TABLE I: Public methods of the three contracts

Method Name	Description
RCVToken contract	
claimLost(lostAccount)	Reports the *lostAccount* address as being lost.
cancelLostClaim()	Reports the current address as not being lost.
reportStolen()	Reports the current address as being stolen. If successful, the sender's tokens are frozen.
chargeback(pendingTransferNumber)	Requests a reversal of the transfer number *pendingTransferNumber* on behalf of the sender.
get/setPendingTransferTime(account)	Get/Set the time an account has to chargeback a transfer.
get/setLostAccountRecoveryTime(account)	Get/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.

A. 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 false claims from being escalated to disputes. 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* [28] 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. 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.

B. 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.

1) 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* of 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 *cancelLostClaim* method using the account that is claimed to be lost). This is necessary

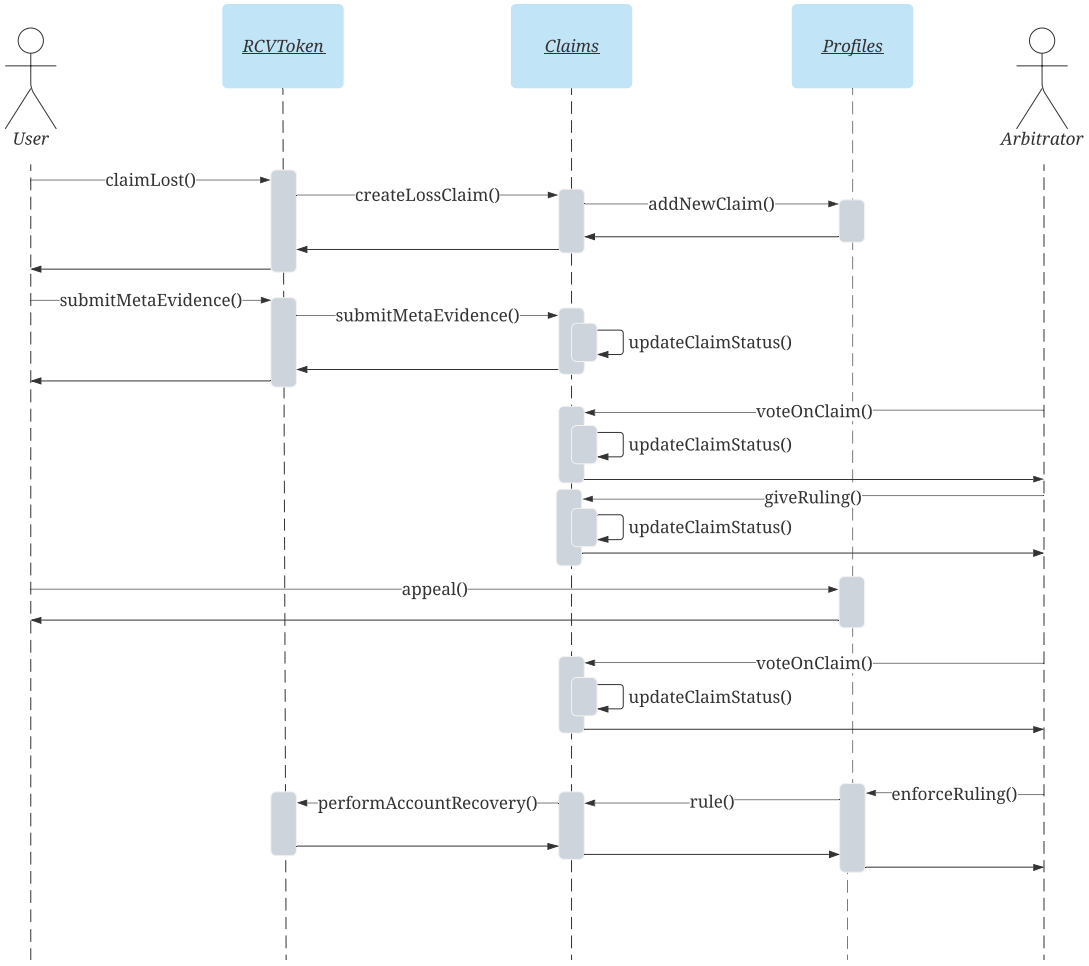

Fig. 2: Example of a successful loss claim.

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.

2) 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*.

3) 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.

4) 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

```
/* MetaEvidence.json */
{
    fileURI:
        "ipfs://QmWRUgLu9iRk...",
    fileHash:
        "QmWRUgLu9iRk...",
    fileTypeExtension: ".txt",
    category: "Lost 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 1: Example of a meta evidence file

supported by an evidence submission standard proposal [27] 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 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 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.

C. 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.

1) Voting: Our dispute resolution mechanism starts with the creation of a new dispute which is triggered after the steps explained in Section III-B 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 III-B4 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 [24]; 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.

2) 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.

D. 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 transaction but swapping the source and destination addresses).

E. Recoverable Token Application

As user interface (UI), we created an application tailored to both regular users and arbitrators (RCV App in Figure 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.

IV. Implementation

In this section we delve deeper into details regarding the implementation.

To implement the bulk of the Recoverable Token functionalities we developed a set of smart contracts written in the Solidity programming language. Figure 3 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.

It implements the IArbitrator interface of the ERC-792 [26]. Additionally, it references the *Ownership* library which has the necessary functions to validate the *proof of ownership* of an account.

The role of the *Claims* contract is to hold the information related to any sort of dispute, i.e., lost, stolen or chargeback

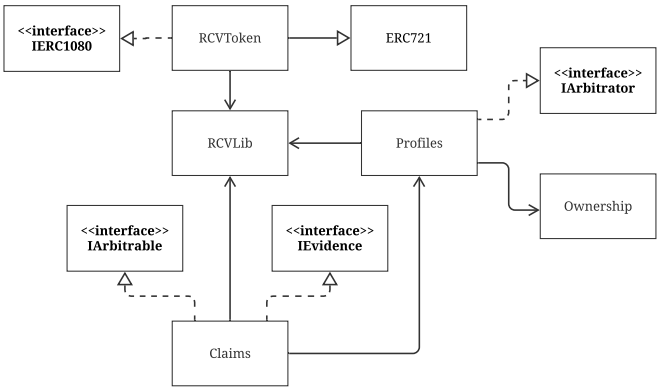

Fig. 3: Smart contracts used with inheritance relations

```
/* 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 2: Changes to transferFrom function

claims. Through it arbitrators can vote on claims and enforce rulings by calling the methods shown in Table I.

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 methods described in Table I. 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 2.

All the mentioned contracts share a library – *RCVLib* – that stores definitions of structures used across them. As for the UI (Section III-E), it is essentially a command line tool that allows interacting with the developed smart contracts, implemented using NodeJS.

Figure 4 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.

V. Experimental Evaluation

For our evaluation we applied the *Recoverable Token* system to an application [4] that makes use of an ERC-721 token. We deployed the smart contracts on *Ropsten*, a public *test network* for Ethereum that to most extent mimics the 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). In the evaluation we assess source code metrics (Section V-A) for our smart contracts as well as gas (Section V-B) and time (Section V-C) for the methods commonly used by all three main use cases of the system: trying to recover from the scenarios *S1*, *S2* and *S3* (Section I).

For *gas* and *time*, the metrics represented are the result of calculating the average of 10 claims. 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* [5] 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.

A. Source code

TABLE II: 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 [6].

Looking at Table II, 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

[4] https://github.com/CodinMaster/Crypto-Car-Battle
[5] https://infura.io/docs
[6] All historical data was gathered from: https://www.etherscan.io/

Fig. 4: Using the RCV App to call the signUp method of the RCVToken smart contract.

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.

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 [29] 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 [30], [31] 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.

B. Gas consumption

Any transaction requires *gas* to be executed. 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.[7] 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 [32]. 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. 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 (σ) of the obtained values.

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 5. 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).

C. Time consumption

In this Section we present and analyse the results obtained in regards to time related metrics. Figure 6 is a chart of the

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

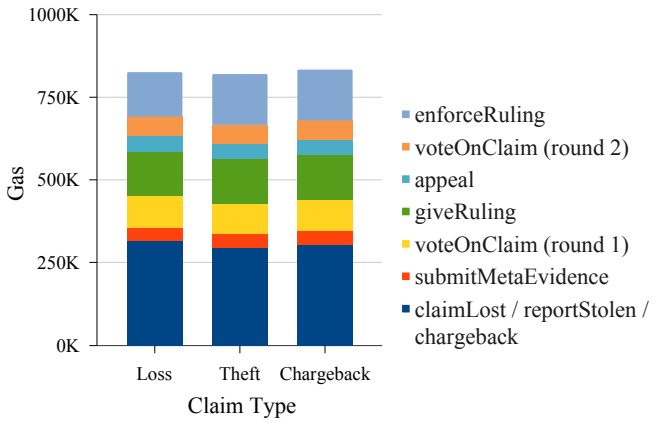

Fig. 5: Average gas consumed by methods for each type of claim.

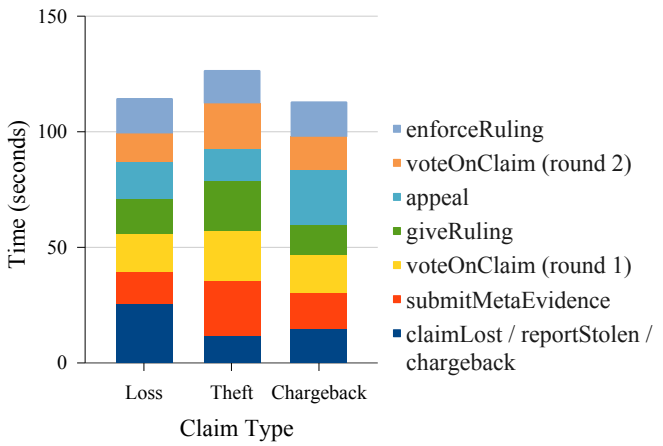

Fig. 6: Average time to execute methods for each type of claim.

TABLE III: Gas breakdown for the Loss, Theft and Chargeback dispute resolution scenarios.

Action	Gas	Gas (%)	Cost (USD)	σ
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	-

time data shown in Table IV. 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 III, 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 IV: Time breakdown for the Loss, Theft and Chargeback dispute resolution scenarios.

Action	Time (s)	Time (%)	σ
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%	-

VI. Related Work

A. Intrusion Recovery

The problem of intrusion recovery has been studied for different kinds of systems. In [10] the authors present a generic algorithm to recover from intrusions that works in three steps: a rewind step that rolls back the system to a point in time prior to the attack, a repair step in which the faulty operations are erased from the log, and a replay step to re-execute every operation in log. By the end of the third step the system no longer has the effects of the attack but it keeps every legitimate operation that occurred. This three-step algorithm was adopted in other works [11]–[16]. Our work aims to solve a similar problem by reverting the effects of undesired operations. However, the approach we use is more in line with the execution of compensating transactions [33], [34]. These kind of transactions aim to revert the effects of the intrusion without requiring the system to be rolled back to a previous point in time allowing the system to be recovered without shutting it down. A compensating transaction can be thought of as an inverse operation of the intrusion. Some systems that use this method of recovery are [13], [35]–[40].

B. Blockchain Recovery

Forks which are an expected occurrence as a result of the design of the blockchain may also be leveraged to perform blockchain recovery. One instance where a fork was used as a recovery mechanism was the response to The DAO hack [41]. Essentially a smart contract that implemented it had a vulnerability which allowed the attacker to steal over $50M USD worth of ether at the time. After a number of proposals the community as a whole voted for forking the chain to a state before the hack ever happened. As a result some community members which did not agree with the decision as they argued that it put into question the ledger immutability attribute of blockchains decided to continue with the original Ethereum chain which is now Ethereum Classic.

Removing or editing data [17], [18] from the blockchain has also been a topic of research. The general idea is to be able to modify confirmed blocks without breaking the links between them. This may be useful when dealing with removing references to illegal or unwanted information that was submitted to the blockchain, or to work towards making the blockchain GDPR [42] compliant.

For cases where a private key may have been stolen and transactions performed without the consent of the rightful owner, transaction reversion mechanisms such as Reversecoin [19] and more recently Blockd [20] have been proposed. Usually these types of work rely on replacing or cancelling transactions while they have not yet been submitted to or confirmed in the blockchain.

C. Dispute resolution

The idea to bring dispute resolution to the blockchain is being explored. Kleros [43] has been working on a decentralized arbitration application in which crowdsourced arbitrators give rulings on disputes. The arbitrators are expected to rule correctly and fairly as a result of game theoretical incentives.

Aragon [44], 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 whereas Kleros does not have that restriction.

Mattereum [45] is working on creating 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 had to be developed. The approach taken was akin to the standard used in common arbitration courts. When a dispute is raised, either both parties had a predetermined agreement where the arbitrator had already been chosen, or alternatively and by default, an arbitrator is appointed from a panel of arbitrators.

VII. Conclusion

This paper 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.

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.

We concluded by discussing work related to intrusion recovery, blockchain recovery and dispute resolution on the blockchain.

Acknowledgements This research was supported by the European Commission under grant agreement number 822404 (QualiChain) and by national funds through Fundação para a Ciência e Tecnologia (FCT) with reference UIDB/50021/2020 (INESC-ID).

References

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

[2] G. Wood *et al.*, "Ethereum: A secure decentralised generalised transaction ledger," *Ethereum project yellow paper*, vol. 151, no. 2014, pp. 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*, Sep. 2020.

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

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

[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, [Online]. Accessed: 2020-12-05. [Online]. Available: https://bitcoinmagazine.com/articles/deterministic-wallets-advantages-flaw-1385450276

[10] A. B. Brown and D. A. Patterson, "Rewind, repair, replay: three r's to dependability," in *Proceedings of the 10th ACM SIGOPS European Workshop*, 2002, pp. 70–77.

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

[12] D. Nascimento and M. Correia, "Shuttle: Intrusion recovery for PAAS," in *IEEE 35th International Conference on Distributed Computing Systems*, 2015, pp. 653–663.

[13] D. Matos and M. Correia, "Nosql undo: Recovering NoSQL databases by undoing operations," in *IEEE 15th International Symposium on Network Computing and Applications (NCA)*. IEEE, 2016, pp. 191–198.

[14] A. Goel, K. Po, K. Farhadi, Z. Li, and E. De Lara, "The Taser intrusion recovery system," in *Proceedings of the 20th ACM Symposium on Operating Systems Principles*, 2005, pp. 163–176.

[15] 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*, 2010, pp. 89–104.

[16] F. Hsu, H. Chen, T. Ristenpart, J. Li, and Z. Su, "Back to the future: A framework for automatic malware removal and system repair," in *IEEE 22nd Annual Computer Security Applications Conference*, 2006, pp. 257–268.

[17] 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*, 2017, pp. 111–126.

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

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

[20] R. M. Forster. (2019) Blockd.co In-Depth: Features and Future. https://medium.com/blockd/blockd-co-in-depth-under-the-hood-and-into-the-future-c142d6d54777. Last accessed: 2020-05-17. [Online]. Available: https://medium.com/blockd/blockd-co-in-depth-under-the-hood-and-into-the-future-c142d6d54777

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

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

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

[24] M. Correia, "From byzantine consensus to blockchain consensus," in *Essentials of Blockchain Technology*. CRC Press, 2019, ch. 3.

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

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

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

[28] M. J. Dworkin, "Sha-3 standard: Permutation-based hash and extendable-output functions," Tech. Rep., 2015.

[29] 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*. Springer, 2020, pp. 203–232.

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

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

[32] 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*, 2019, pp. 24–31.

[33] H. F. Korth, E. Levy, and A. Silberschatz, *A formal approach to recovery by compensating transactions*. University of Texas at Austin, Department of Computer Sciences, 1990.

[34] P. Liu, P. Ammann, and S. Jajodia, *Rewriting Histories: Recovering from Malicious Transactions*. Springer US, 2000, pp. 7–40.

[35] X. Xiong, X. Jia, and P. Liu, "Shelf: Preserving business continuity and availability in an intrusion recovery system," in *Proceedings of the Annual Computer Security Applications Conference*, 2009, pp. 484–493.

[36] İ. E. Akkuş and A. Goel, "Data recovery for web applications," in *Proceedings of the 40th IEEE/IFIP International Conference on Dependable Systems and Networks*, 2010, pp. 81–90.

[37] 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*. ACM, 2017, pp. 209–221.

[38] D. R. Matos, M. L. Pardal, and M. Correia, "RockFS: Cloud-backed file system resilience to client-side," in *Proceedings of the 2018 ACM/IFIP/USENIX International Middleware Conference*, 2018.

[39] P. Ammann, S. Jajodia, and P. Liu, "Recovery from malicious transactions," *IEEE Transactions on Knowledge and Data Engineering*, vol. 14, no. 5, pp. 1167–1185, 2002.

[40] R. Chandra, T. Kim, M. Shah, N. Narula, and N. Zeldovich, "Intrusion recovery for database-backed web applications," in *Proceedings of the 23rd ACM Symposium on Operating Systems Principles*, 2011, pp. 101–114.

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

[42] R. Viorescu *et al.*, "2018 reform of eu data protection rules," *European Journal of Law and Public Administration*, vol. 4, no. 2, pp. 27–39, 2017.

[43] C. Lesaege and F. Ast, "Kleroterion, a decentralized court for the internet," Jun. 2017.

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

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