Bachelor Informatica

University of Amsterdam

Using blockchain to validate audit trail data in private business applications

Rosco Kalis

June 8, 2018

Supervisor(s): Adam Belloum (UvA), Dan Haywood (Apache)

Signed: A.S.Z Belloum

Abstract

It is important to be able to trust the data within applications, and to ensure that it has not been tampered with. To achieve this, companies employ audit trails with a log of all changes made inside their applications. However, if this audit trail is saved in the same way as the application data, it is still vulnerable to tampering. To solve this, we research the use of blockchain with this audit trail. We discuss the fundamentals of blockchain technology, and how it achieves data integrity. Next, we discuss the different challenges of different blockchain implementations, of which the most important are data privacy on a public blockchain, and transaction limits/costs. We describe two different methods to overcome the data privacy concern: one based on data hashing and one based on data encryption. However, only the approach based on data hashing is able to scale in terms of transaction costs associated with the method. We create an Ethereum smart contract and integrate it with an Apache Isis audit trail. This audit trail can then be validated against the smart contract running on the Ethereum blockchain. We evaluate the method using several scenarios, which show that this implementation is able to detect data tampering, but not prevent it. Combined with a strong backup protocol and regular validations, however, it is able to provide more certainty regarding data integrity than a regular audit trail implementation.

Acknowledgements

I would like to thank my supervisors Adam and Dan for their input and guidance during the past months, Jeroen and Marc for enabling me to conduct this research and write this paper at Eurocommercial Properties, my coworker Sander for his LaTeX-foo, and my girlfriend Kevser for her patience and understanding during the past months of tight schedules and busy weekends.

Contents

CHAPTER 1

Introduction

US Director of National Intelligence James Clapper stated that the next big cyber threat is data manipulation [1] and technology magazine Wired has listed it as one of the biggest security threats in 2016 [35]. To combat this threat, it is paramount to ensure that data within applications can be trusted. As a way to achieve this, companies employ audit trails containing a log of application interactions. An interaction in this context is defined by a user interacting with the application in such a way that it changes the state of one or more entities within the persistent datastore, including creating or deleting existing entities.

The audit trail offers a way to verify the current state of the application against the state changes that have led up to it. However, if this audit trail is saved in the same way as the application data, it is still vulnerable to tampering. What this means is that the audit trail can only provide an indication of data integrity, but no guarantees.

A potential way to provide these guarantees is blockchain technology. Blockchain is a data structure that distributes all data over a network of nodes, so that there is no single point of failure, and no central control that might be compromised [32]. In order to further guarantee data integrity, blockchain uses a consensus algorithm which allows the independent nodes to approve correct transactions and reject malicious ones [7].

Since the strength of blockchain technology comes from its decentralised nature [32], there lies a challenge in effectively conveying this data integrity guarantee to private business applications, without giving unauthorised parties access to said application data.

Applications using the Apache Isis framework already have a way to easily integrate a regular audit trail through the AuditerService [1] and the Incode Audit Module [2]. This research aims to use these services and integrate them with blockchain technology in order to test the validity of this approach.

1.1 Research questions

In order to convey blockchain's data integrity guarantee to a private business application by building an audit trail on top of it, we formulated the following research questions:

Can we implement an automatic audit trail for Apache Isis applications, using blockchain technology? (1)

We further specify the requirements of this implementation in the second research question:

Is this blockchain audit trail able to ensure that it has not been tampered with, without sharing application data with unauthorised parties? (2)

[1] https://isis.apache.org/guides/rgsvc/rgsvc.html (Accessed 2018-06-06)
[2] http://platform.incode.org/modules/spi/audit/spi-audit.html (Accessed 2018-06-06)

These questions are answered by studying the nature of blockchain, and creating an implementation for an automatic audit trail for Apache Isis, that integrates with blockchain. This implementation is tested by simulating scenarios in which audit trail data gets tampered with.

CHAPTER 2

Theoretical background

2.1 Apache Isis

Apache Isis [1] is a Java software development framework based on Domain Driven Development and the Naked Objects pattern [11]. It can be used to rapidly develop complex business applications by leveraging its power to dynamically generate a UI and REST API from the domain model of the application at runtime. This allows for rapid prototyping of both applications and new features within these applications, since no time needs to be spent on the presentation layer of the application. If an application warrants a more complex, custom user interface, it is always possible to use the generated REST API.

2.1.1 Domain Driven Design

Domain Driven Design is a paradigm for software development in which software is strictly modelled after real-world systems and processes [26]. For this to work, it is important to work together with a *domain expert*, who precisely understands these real-world systems and processes. Between the domain expert and the developer, a Ubiquitous Language (UL) is defined, which contains definitions of terms used in describing the behaviour of the systems, so that both the domain expert and developer clearly understand the descriptions [26].

The real-world system and processes can then be specified with this UL, and the software can be modelled after the specification using different building blocks. These building blocks include [26]:

- *Value Objects*, which can contain different attributes, but no identity of its own. A date is an example of a Value Object.
- *Entities*, which can contain different attributes, but also have a distinct identity in the domain.
- *Domain Events*, which are events within the system that are deemed important by the domain expert.
- *Domain Services*, which are encapsulations of business logic that isn't necessarily related to any value object or entity.

Whenever the real-world systems change, this is captured in the domain model, and the software is adapted to reflect these changes in the domain model. This means that the software always reflects the actual real-world systems and processes.

[1] https://isis.apache.org/ (Accessed 2018-04-10)

2.1.2 Naked Objects

An oft-used pattern in software development is the Model-View-Controller (MVC) pattern. In this pattern the View displays information to the user, the Controller processes the interactions between the user and the View, and the Model contains the business logic [10]. However, in practice, the presentation logic and business logic are often not separated well enough in web applications, resulting in Controllers filled with business logic that belongs to the Model [10].

The Naked Objects pattern looks to fix these practical shortcomings by making the View and Controller roles completely generic [25]. By doing so, the business application is written solely in terms of domain entity objects. The entire presentation layer, otherwise coded with a View and Controller role, is then automatically generated from the domain model. This translates into the following three principles [25]:

- All business logic should be encapsulated in domain objects.
- The user interface is a direct representation of the domain objects.
- The user interface is derived automatically from the domain objects through reflection.

This leads to a faster development cycle and more agility, since the focus can remain on the domain objects, and no extra time needs to be spent on the View and Controller roles of the application.

2.1.3 Framework Domain Services

The Apache Isis framework offers different domain services in the form of APIs, which have been implemented by the framework and can be called by the application, and SPIs, which can be implemented by the developer inside the application, and will be called by the framework [12]. These services can be categorised by their uses and their place in the application.

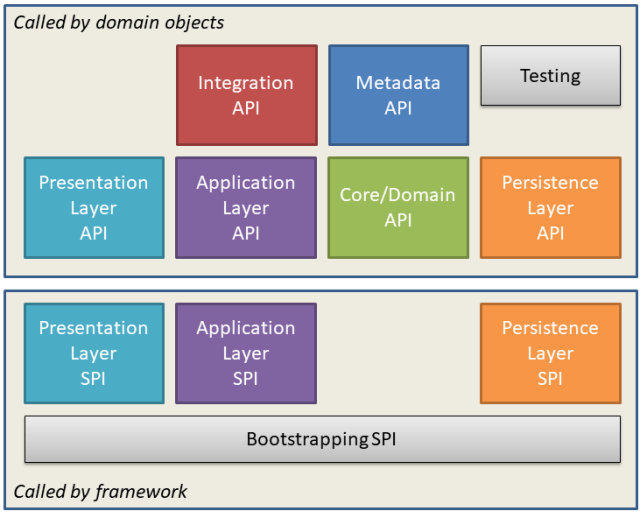

Figure 2.1: Domain service categories. Downloaded from [12]

2.2 Audit Trail

An audit trail is a log of all interactions that happen inside an application, and more importantly, their effects on the underlying data. This can be used as a way to verify the current state of the application against the interactions and changes that led up to this state. In order to do this, every piece of data in the application has to be traceable all the way back to the interaction that created it, including all changes that have been made along the way.

This makes it possible to see exactly when, by whom, and using which method the data was changed. If some data within the application is not traceable in such a way, it means that either the data in the application, or the data in the audit trail has been changed outside of the regular application functionality. However, if the audit trail and the application have been tampered with at the same time, in a way that the changed application corresponds correctly with the changed audit trail, it can be difficult to detect these changes without taking extra measures.

2.2.1 Apache Isis AuditerService SPI & related SPIs

Apache Isis has several domain services available as an SPI, where the implementation of the service is left to the application, and this implementation will automatically be called by the framework. Apache Isis offers several such SPIs for auditing (or similar) purposes, as illustrated in figure 2.2 [12].

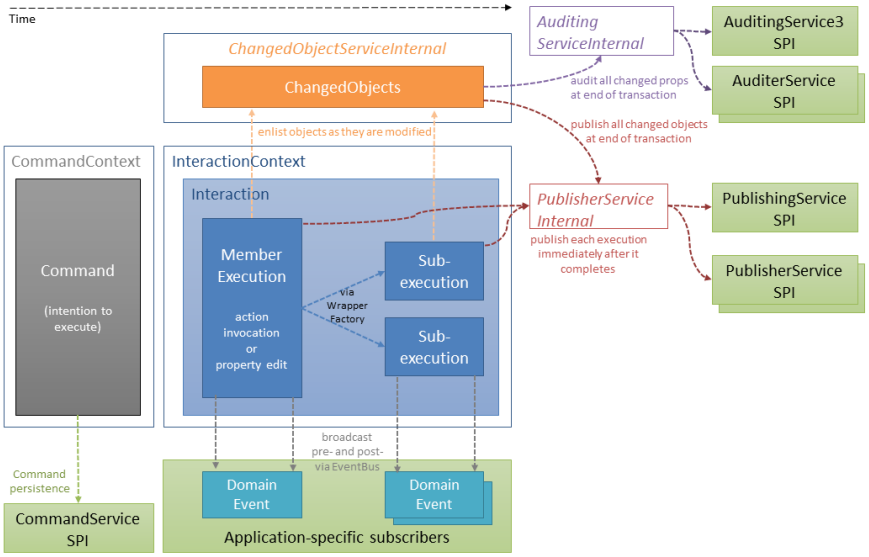

Figure 2.2: Domain services used for auditing. Downloaded from [12]

In this figure we can see the different phases of interactions within the application, and the different hooks that the services provide into this process. The CommandService SPI [2] captures a Command object, which includes the invoking user, the targeted object(s), and the corresponding interaction [12]. The AuditerService SPI captures the actual changes caused by the interaction by saving the invoking user, the changed property, and the pre- and post-values of the property [12]. The PublisherService [3] captures the actual interaction, as well as a summary of the changed properties [12]. These SPIs can be implemented to assist in auditing, depending on the requirements. It is not unusual to use these services together.

2.3 Blockchain

Blockchain is a data structure that is distributed over a number of different nodes. It uses a chain of so-called *blocks* to store data [7]. Every block header includes the root of a *Merkle tree*, which contains the actual data inside of the block [7]. Besides this, every block also includes a timestamp and a hash of the previous block in order to make it resistant to manipulation [7].

[2]https://isis.apache.org/guides/rgsvc/rgsvc.html (Accessed 2018-06-06)
[3]https://isis.apache.org/guides/rgsvc/rgsvc.html (Accessed 2018-06-06)

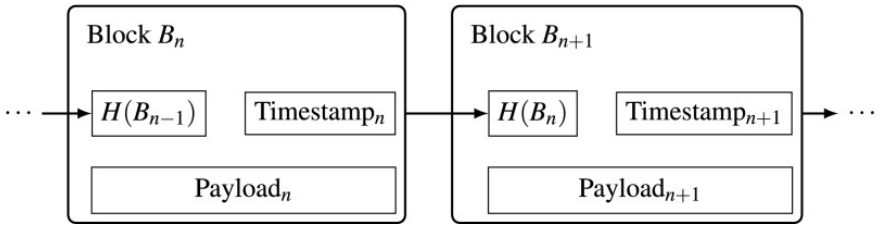

Figure 2.3: Block contents and linking. Downloaded from [22]

If an attacker would change a past block's data, its hash would change with it, and since the changed hash is never referenced by another block, it would not get accepted [7]. This means that in order to successfully change a past block they would have to fork the blockchain. The rule with forks in the blockchain is that the longest chain is always the leading one, so in order to have the modified block accepted by the network, the attacker would need to grow its chain faster than the rest of the network combined to eventually catch up and pass the longest chain, which is called a 51% attack [7]. Because the resources of the entire network are extensive in major blockchains, this sufficiently guarantees the integrity of the data in the blockchain [7].

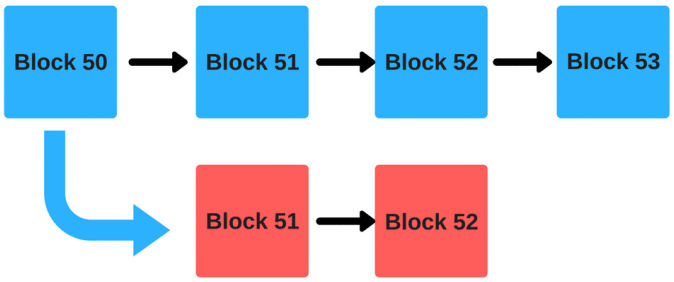

Figure 2.4: 51% attack in progress. Downloaded from [28]

2.3.1 Merkle Tree

Blockchain uses a Merkle tree to contain the data inside the block. Ethereum's white paper explains how this data structure functions [7]:

> A Merkle tree is a type of binary tree, composed of a set of nodes with a large number of leaf nodes at the bottom of the tree containing the underlying data, a set of intermediate nodes where each node is the hash of its two children, and finally a single root node, also formed from the hash of its two children, representing the "top" of the tree.

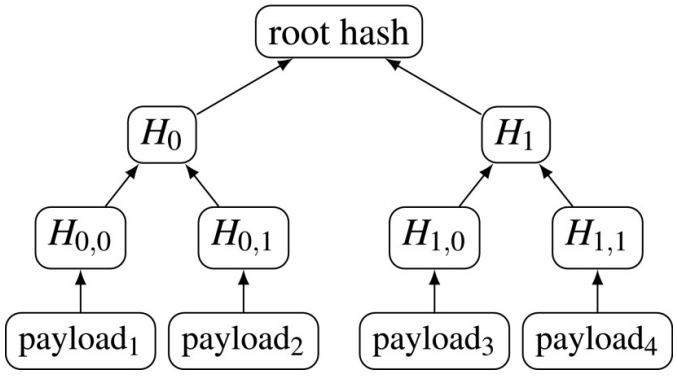

Figure 2.5: Structure of a Merkle tree. Downloaded from [22]

By using a Merkle tree, it is possible to receive the header of a block from one source, but parts of the tree from other sources, while still being assured all data is correct [7]. This works because all hashes propagate upwards into the tree, meaning that a change in one node modifies all hashes of the nodes above the one that was originally changed. This would finally lead to a different root hash, which would modify the block header, and would be perceived as a different block [7].

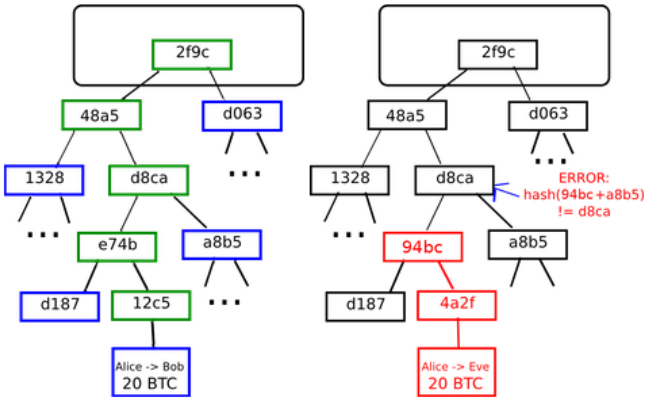

Figure 2.6: Merkle tree conflict. Downloaded from [7]

2.3.2 Decentralised consensus

One of the important parts of blockchain is the ability to reach a decentralised consensus between mutually distrusting parties. This problem dates back to the Byzantine Generals Problem, which was formalised in 1982 [17]. This Byzantine Generals Problem is an abstract description of the problems that arise when dealing when trying to reach consensus between these different mutually distrusting parties.

In this problem several divisions of the Byzantine army encircle an enemy city, where every division is commanded by a single general. After observing the enemy they must decide upon a plan of action by sending messengers to each other. However, some of the generals might be traitors who can try to prevent the loyal generals from reaching an agreement by relaying false information [17].

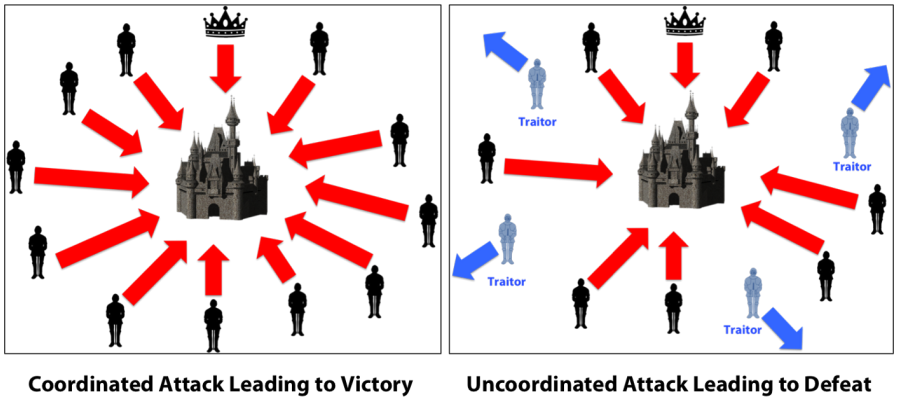

Figure 2.7: Only with a coordinated attack can the generals win the battle, an uncoordinated attack will result in defeat. Downloaded from [9]

In order to solve this problem, the Byzantine generals need to define an algorithm that allows the loyal generals to agree upon the same plan of action, while a small number of traitors cannot cause the loyal generals to adopt a bad plan [17]. An application that is able to satisfy these requirements is said to be Byzantine Fault Tolerant.

Blockchain addresses this problem with so-called *consensus algorithms*. These consensus algorithms allow the nodes of the blockchain to reach consensus about the blocks that are added to the blockchain. One of the most widespread algorithms for this is the *Proof-of-Work* algorithm, employed by the Bitcoin blockchain, as well as several other high-profile blockchains. Part of this algorithm entails giving out a financial reward for correctly validating transactions [24]. By adding a financial reward to correctly validating transactions, mutually distrusting parties are able to work together to keep the blockchain working.

2.3.3 Smart Contracts

A smart contract is a way to digitally enforce a contract or an agreement between parties through code. This concept predates blockchain more than a decade, as the term was initially coined by Nick Szabo in 1996 [31].

However only when blockchain was implemented did it finally become possible to implement these smart contracts without the need for a trusted third party. Smart contracts on the blockchain were first implemented in the Ethereum blockchain, as proposed in the Ethereum white paper [7].

Ethereum offers the ability to publish smart contracts on its blockchain, which can be executed by the *Ethereum Virtual Machine (EVM)* [7]. By publishing these contracts to the Ethereum blockchain, all involved parties can easily inspect the contract and they will be assured that the contract will execute exactly as specified.

This offers possibilities to create decentralised applications, or *Dapps*, on the Ethereum blockchain. The Ethereum white paper lists several use cases, such as sub-currencies (tokens), decentralised file storage, and even entire *Decentralised Autonomous Organisations (DAOs)*, which run entirely on business logic encoded into smart contracts [7].

At the same time, smart contracts are still code, and can contain bugs like any other code. In the case of smart contracts, these bugs can have enormous consequences. One of these instances is the attack on *The DAO* in 2016, where it was unintentionally possible to drain funds out of this DAO into an attackers wallet, which is exactly what happened [18]. Millions of dollars worth of Ether was stolen because of this one bug in the code.

In this instance, the nodes in the Ethereum network agreed on a *hard fork*, intentionally changing the state of the blockchain, so that investors could retrieve the money they had invested in The DAO, while the attacker lost his share [18].

This does raise philosophical questions and even legal issues, since these smart contracts are supposed to be used to uphold a certain agreement. If the network decides not to honour this agreement, it diminishes the value these smart contracts hold. Even though in this case it was a clearly unintentional bug in the smart contract, it can be difficult to determine the difference between the intention behind a smart contract and its actual behaviour.

In order to solve this issue, ABN Amro stated that they deploy smart contracts accompanied with an attachment of the legal prose that the smart contract represents. If bugs should arise at any moment, this legal prose will be leading in determining the intentions behind a smart contract [6].

2.3.4 Public and private blockchains

There are different ways a blockchain can be deployed. There are blockchains that are completely open and public, so that anyone can see all activity on the chain, and conduct their own transactions [16]. This is called a *public* or *permissionless* blockchain. Opposed to this public or permissionless blockchain is the *private* or *permissioned* blockchain. In a permissioned blockchain, the owner or owners of the blockchain are in control of who can view the blockchain, and who can add new transactions [16].

These permissioned blockchains can be used by multiple corporations to work together in the same industry, or by multiple departments of the same corporations. Since there is more trust between parties in a private blockchain, there is less need for the more complex consensus algorithms such as Proof-of-Work for the blockchain to function [16]. This is clearly reflected in the consensus algorithm that blockchain framework Hyperledger Sawtooth uses for its consensus,

Proof-of-Elapsed-Time, which uses a lottery function deciding which node will be able to add the next block. From the Hyperledger Sawtooth documentation [14]:

> PoET essentially works as follows... Every validator requests a wait time from an enclave(a trusted function). The validator with the shortest wait time for a particular transaction block is elected the leader. One function, say CreateTimer creates a timer for a transaction block that is guaranteed to have been created by the enclave. Another function, say CheckTimer verifies that the timer was created by the enclave and, if it has expired, creates an attestation that can be used to verify that validator did, in fact, wait the allotted time before claiming the leadership role.

It is also possible to use a blockchain within a single, smaller, organisation, with only one node being able to write to the blockchain. However, this does not benefit from the decentralisation and distribution of regular blockchains, which means they would present no benefits over a regular relational database system in that regard.

2.3.5 Blockchains in practice

Blockchain was initially designed as the ledger of the Bitcoin cryptocurrency [24]. Today there are many more cryptocurrencies using a public blockchain as their ledger. But as the blockchain technology progresses, so do the use cases, piquing the interest of many corporations and even governments across the world. The Republic of Georgia was the first country to run pilots with land title registration on blockchain technology [30], which has since been successfully implemented [4].

Ethereum offers the ability to create decentralised applications with smart contracts. Among these Dapps are Golem [4], which is a project that allows people to rent out their idle computing power, and Augur [5], which allows people to bet on the outcome of any future event by creating peer-to-peer prediction markets.

However, Brian Behlendorf, the executive director of Hyperledger, stated that he is most excited about the use cases that can make a significant social impact [19]. In particular, the way blockchain is used for supply chain management by eliminating the need for a trusted third party to certify products as they travel through the supply chain. This allows a consumer to easily verify the product's authenticity and provenance, and by doing so they can ensure that the quality of the product is what they expect from it, or that it has been produced in accordance with their personal values [19].

[4] https://golem.network/ (Accessed 2018-05-26)
[5] http://www.augur.net/ (Accessed 2018-05-26)

CHAPTER 3

Method

3.1 Decision for blockchain implementation

There are several different ways to implement or use a blockchain in practice, many corporations are using different public blockchains, or private blockchain implementations. The next step is identifying the best way to use blockchain technology for our specific use case. We encapsulated the requirements to our use of blockchain technology in the research questions. To reiterate this: we want to ensure that the data in our audit trail has not been tampered with, without sharing application data with unauthorised parties. The challenges that arise with different blockchain implementations are discussed below.

3.1.1 Challenges in private blockchains

To see the challenges that arise in the use of a private blockchain, we have to look at the features of the blockchain. A blockchain is able to ensure data integrity by linking a block with all its ancestors using data hashes, and then making sure that the network as a whole has sufficient resources to repel any 51% attacks [7]. From this it is already clear that a completely private blockchain is not able to offer these same guarantees, as the network consists of a single node. If this node gets compromised, it can change any data at will, offering no more guarantees than a regular database.

To overcome this challenge it is possible to share a private network with other organisations. However, in the use case we are describing, there is no reason other parties would be interested in the shared data, which means that it would practically be a smaller, and therefore less-secure version of a public blockchain.

3.1.2 Challenges in public blockchains

From highlighting the limitations of a private blockchain it has become clear that we need to integrate with a public blockchain to really leverage the data integrity guarantee that it offers. However, this does come with its own drawbacks in the context of a private business application, because all data that is published to a public blockchain is publicly accessible. Since the data may contain sensitive elements, it can't just be published on a public blockchain without any privacy measures.

To overcome these complications, we discuss two different approaches to add data privacy to our implementation, while still leveraging the data integrity guarantees from a public blockchain. The first method stores data hashes on the public blockchain, while storing the actual data locally. The second method encrypts the data before publishing it to the public blockchain. These methods are further elaborated below.

3.1.3 Data Hashing & Blockchain Anchoring

US Patent application 20180025181 [2] presents a method to reliably store data and verify its integrity with blockchain technology. The method describes storing (potentially many) data files on a data storage module, and transmitting hashes of these data files to a public – or sufficiently decentralised private – blockchain [2]. With this hash they store metadata, such as a timestamp, file size, and other file metadata. The method then monitors these files, and whenever a change occurs, this process is re-initiated, storing the new hash on the blockchain, with a link to the previous one [2].

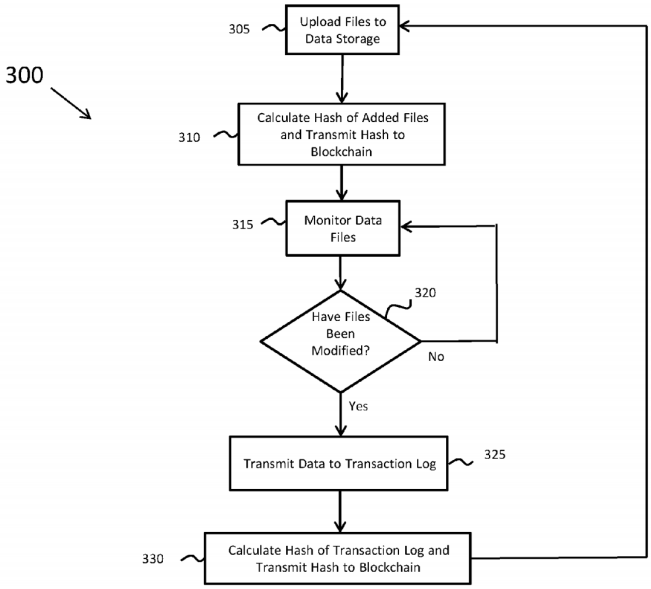

Figure 3.1: Visual representation of the method described in [2]. Extracted from [2]

While this method is specifically created for the validation of data files, this can be transferred to application data as well. The methods of hashing stay the same, and the monitoring process can be replaced by the Apache Isis framework, which automatically calls the SPI implementations whenever the application state changes. This does mean that the audit trail data needs to be stored separately from the public blockchain, since the data hashes that are stored on the blockchain can only be used to verify existing data. What this means is that this method doesn't prevent tampering, it is only able to detect it when the data has been tampered with. With a secure backup protocol, however, it is possible to retrieve the correct data after tampering has been detected. Because only data hashes are stored on the blockchain, this method is very scalable, since any size data can be hashed into a signature of 32 bytes [33].

As an extension to this method, a method called blockchain anchoring could be used. This consists of saving the actual data in a private blockchain, where the latest block of this private blockchain is periodically committed to a public blockchain [5]. This allows the implementation to be configured to store fewer hashes on a public blockchain, depending on the need of the implementation. Schneier & Kelsey state that a method for securing data integrity should be configured based on the estimated frequency that the data is at risk of being tampered with [29]. For instance, if it is estimated that the data is at risk of being tampered with about once a month, it would be sufficient to only store the data hash on a public blockchain once every two weeks.

3.1.4 Data Encryption

The second method works by encrypting the data before storing it on a public blockchain. In its most simple form this means that an encryption key needs to be saved to encrypt and decrypt the data that gets stored on the public public blockchain. Then, all data is encrypted before being stored on the blockchain, and it needs to be decrypted before being able to use the data again. This means that there is no need to store a copy of the data locally, as is the case with the first method. The application can rely fully on the data that is stored on the blockchain, which also means that, unlike the first method, this method is able to *prevent* instead of only detect tampering with the existing data.

As an extension to this method, more complex encryption schemes could be used in order to further protect the data, and to grant multiple parties access to (parts of) the data using their own authentication keys. One such method is described as part of the method for data integrity described by Schneier & Kelly [29]. This method describes using an initial authentication key, from which a new key is derived for every data entry. Then an encryption key is generated from this authentication key. Every subsequent key can be derived from the initial one, meaning the creator has access to all entries, but the keys for individual entries can be shared without granting access to all other entries [29].

3.1.5 Transaction Limits

The drawback of using the encryption method, or of saving the data without a privacy implementation is that it takes up a lot more space on the blockchain than the method using data hashes. Practically this means that bigger data entries need to be split up over different transactions, since there are limits on the amount of data that a block can hold, and therefore limits on the amount of data that a transaction can practically include.

With Ethereum, the limits per block are contained in the gas limit of the block, which is currently at around 8 million gas for the Ethereum Main Net, but this limit can scale depending on demand [34]. This gas is a unit of computation, and every operation that is executed on the blockchain has a different gas cost associated with it [34]. An overview of these gas costs is shown in figure 3.2.

The Yellow Paper specifies that every byte of non-zero data in a transaction costs 68 units of gas, and every transaction costs 21 000 gas to start with [34]. This means that theoretically, the maximum size of one transaction would be (8 000 000 - 21 000) / 68 000 $\approx$ 117 kB. However, this only includes transaction data that is not stored. The yellow paper also states that the fee to store a 32-byte word is 20 000 gas, translating to 640 000 gas per kilobyte of stored data [34]. This puts the limit to stored data per transaction at (8 000 000 - 21 000) / (640 000 + 68 000) $\approx$ 11 kB. This is not enough to store large values on the blockchain.

It is important to consider whether this limitation is relevant for the specific use case. When only storing single integer values, this limit will not be reached, but when storing arbitrary-size blobs, it is very likely that this data will need to be split up, thus adding extra complexity.

Besides these complications, every unit of gas on the Ethereum blockchain needs to be paid for, so the costs of this method could increase rapidly. At the time of writing the price of one unit of gas is around 10 GWei, or 0.00000001 Ether, and the price of one Ether is around €500. This means the price to store one kB of data is around €4. These increased costs make storing larger amounts of data impractical for realistic usage.

Appendix G. Fee Schedule

The fee schedule G is a tuple of 31 scalar values corresponding to the relative costs, in gas, of a number of abstract operations that a transaction may effect.

Name	Value	Description*
G_{zero}	0	Nothing paid for operations of the set W_{zero}.
G_{base}	2	Amount of gas to pay for operations of the set W_{base}.
$G_{verylow}$	3	Amount of gas to pay for operations of the set $W_{verylow}$.
G_{low}	5	Amount of gas to pay for operations of the set W_{low}.
G_{mid}	8	Amount of gas to pay for operations of the set W_{mid}.
G_{high}	10	Amount of gas to pay for operations of the set W_{high}.
$G_{extcode}$	700	Amount of gas to pay for operations of the set $W_{extcode}$.
$G_{balance}$	400	Amount of gas to pay for a BALANCE operation.
G_{sload}	200	Paid for a SLOAD operation.
$G_{jumpdest}$	1	Paid for a JUMPDEST operation.
G_{sset}	20000	Paid for an SSTORE operation when the storage value is set to non-zero from zero.
G_{sreset}	5000	Paid for an SSTORE operation when the storage value's zeroness remains unchanged or is set to zero.
R_{sclear}	15000	Refund given (added into refund counter) when the storage value is set to zero from non-zero.
$R_{suicide}$	24000	Refund given (added into refund counter) for suiciding an account.
$G_{suicide}$	5000	Amount of gas to pay for a SUICIDE operation.
G_{create}	32000	Paid for a CREATE operation.
$G_{codedeposit}$	200	Paid per byte for a CREATE operation to succeed in placing code into state.
G_{call}	700	Paid for a CALL operation.
$G_{callvalue}$	9000	Paid for a non-zero value transfer as part of the CALL operation.
$G_{callstipend}$	2300	A stipend for the called contract subtracted from $G_{callvalue}$ for a non-zero value transfer.
$G_{newaccount}$	25000	Paid for a CALL or SUICIDE operation which creates an account.
G_{exp}	10	Partial payment for an EXP operation.
$G_{expbyte}$	10	Partial payment when multiplied by $\lceil \log_{256}(exponent) \rceil$ for the EXP operation.
G_{memory}	3	Paid for every additional word when expanding memory.
$G_{txcreate}$	32000	Paid by all contract-creating transactions after the *Homestead transition*.
$G_{txdatazero}$	4	Paid for every zero byte of data or code for a transaction.
$G_{txdatanonzero}$	68	Paid for every non-zero byte of data or code for a transaction.
$G_{transaction}$	21000	Paid for every transaction.
G_{log}	375	Partial payment for a LOG operation.
$G_{logdata}$	8	Paid for each byte in a LOG operation's data.
$G_{logtopic}$	375	Paid for each topic of a LOG operation.
G_{sha3}	30	Paid for each SHA3 operation.
$G_{sha3word}$	6	Paid for each word (rounded up) for input data to a SHA3 operation.
G_{copy}	3	Partial payment for *COPY operations, multiplied by words copied, rounded up.
$G_{blockhash}$	20	Payment for BLOCKHASH operation.

Figure 3.2: Fees in gas for operations on the Ethereum blockchain. Extracted from [34]

3.2 Comparison of blockchain frameworks

If we use the data hashing approach, and extend it with blockchain anchoring, the data needs to be stored on a private blockchain, which can be anchored to a public blockchain. To assist in building private blockchains, multiple corporations have created frameworks for blockchain construction, each with their own distinct features. Three of the biggest of these frameworks are Hyperledger Sawtooth, Hyperledger Fabric, and Quorum. These frameworks are discussed in the sections below, finally deciding on a best fit for this project.

3.2.1 Hyperledger Sawtooth

Hyperledger Sawtooth is a framework in the Hyperledger family, which focuses on modularity and flexibility [20]. As such, it allows the creator of the blockchain to use their own consensus algorithm by implementing a distinct interface for the publishing of a block, the verification of this block, and the resolution of forks [20]. While Sawtooth provides these possibilities, it also includes its own consensus algorithm, Proof-of-Elapsed-Time, which works using a random lottery function [20].

Another distinguishing feature of Hyperledger Sawtooth is the ability to batch transactions. When a user submits transactions, this is always done in a batch, although a batch can contain as little as one transaction [20]. These batches are then submitted to a validator node, which checks the (pre-defined) validity of all transactions inside the batch. If any of the transactions inside the batch are invalidated, the entire batch is rejected. If all transactions are deemed valid, they are applied by the validator node and added to the blockchain [20].

Next, Sawtooth defines *transaction families* as the way to change the state of the blockchain. These transaction families are defined groups of operations that are allowed on the blockchain. By limiting the amount of operations that are allowed within a transaction family, Sawtooth is able to reduce the risk at mistakes with these transactions [23]. Sawtooth has several pre-defined transaction families that can be readily used within blockchains built on top of it, from the Integer Key family with just the *increment*, *decrement* and *set* operations, to a full EVM

transaction family [23]. Besides these transaction families, network operators can define their own transaction families by implementing a so-called *transaction processor*, which can be programmed in a variety of programming languages [23]. By choosing the allowed transaction families within the network, it is possible to select the level of versatility/risk thats right for the specific network [23].

Finally, applications are able to interface with a Sawtooth blockchain through a well-defined and clearly documented REST API, and Sawtooth offers SDKs in multiple languages to assist in development [14].

3.2.2 Hyperledger Fabric

Hyperledger Fabric is similar to Hyperledger Sawtooth in the way that it allows for custom algorithms for consensus and also for identity verification. Consensus within Hyperledger Fabric is divided into three steps: endorsement, ordering, and validation and commitment [21].

When a transaction is proposed by a client, it is sent to *endorsing peers*. These endorsing peers simulate the transaction, and return so-called *RW sets*, which capture all the data that the transaction reads from and would write to the world state [21]. If the transaction passes the pre-configured endorsement policy (e.g. a majority of peers must endorse a transaction), the endorsed transaction is sent to an ordering service [21].

This ordering service determines the order in which the transactions are added to a block [21]. The implementation of this ordering service can be customised by the creator of the network; the default implementation of the ordering service is one based on Apache Kafka [21]. While this default ordering service is crash tolerant, it is not yet Byzantine Fault Tolerant, and this is one of the next big steps that Hyperledger Fabric is working towards [21].

After ordering, the transactions are passed back to the endorsing peers, as well as to *committing peers*. These peers verify once more that the RW sets of the transactions still match the current world state, based on this they validate or invalidate the transaction and commit them to the blockchain. Even when they are invalidated they will still get logged to the blockchain, but marked as invalid and they will not be executed [21].

One of the most important distinguishing features of Hyperledger Fabric is the ability to create *channels*, which are full-fledged separate blockchains within the same network [21]. By limiting who can join certain channels in the network, it is possible to keep certain transactions private between the involved parties inside the channel.

Finally, the transaction logic and smart contracts within Hyperledger Fabric are called *chaincode*, and are written in Go [21]. Chaincode invocations execute transactions against the current state data. In order to make this more efficient, the most recent key/value pairs for all assets are stored in a LevelDB database [21]. This database is an indexed view on the blockchain's committed transactions, and can always be regenerated by replaying all transactions.

Applications can interface with a Fabric blockchain through a Node or a Java SDK, and several other SDKs are actively being developed [13].

3.2.3 Quorum

Quorum is a blockchain framework created by J.P. Morgan as a way to create permissioned versions of the Ethereum blockchain. In their implementation they focused on staying very close to the original Ethereum implementation so Ethereum developers would have no trouble switching over to Quorum. The smart contracts engine is very similar to that of Ethereum, and it is possible to run regular Ethereum smart contracts on a Quorum network without any modifications [15].

Since Quorum is intended to be used for private permissioned blockchains, they offer different kinds of consensus algorithms depending on the needs of the network. The first one is based on the Raft protocol [1], which works by electing a leader, who is the sole block creator within the network [15]. This allows for high throughput within the network, but is not Byzantine Fault Tolerant, so it is not viable in every situation.

[1] https://raft.github.io/ (Accessed 2018-04-24)

As an alternative, Quorum implements the *Istanbul BFT* consensus protocol, which works by picking a new validator node to be a *proposer* every block [15]. This selection is done in a round robin fashion, meaning they are selected one after another, in a circular way. This proposer then broadcasts a new block proposal to the other validator nodes, which all validate the block proposal [15]. Once done, they broadcast a *COMMIT* message to all other nodes. Then, when a node has received a *COMMIT* message from at least two thirds of all validators, the block is inserted into the blockchain [15].

Finally, one of the most important features of Quorum is the addition of private transactions to their blockchain. Hyperledger Fabric also offers private transactions, but the methods Quorom and Fabric use to achieve this are different altogether. While Fabric uses *channels*, Quorum has a separate Transaction Manager, which keeps track of the people who have access to certain transactions [15]. Prior to broadcasting such a transaction, a node first replaces the original transaction payload with a hash of the encrypted version of the payload that it receives from the Transaction Manager. Parties that are included in the recipients of the transaction will then receive the actual payload, while others will only see the hash [15].

3.2.4 Decision for blockchain framework

Because for this project a private blockchain will run within a single node, it will offer no Byzantine Fault Tolerance, regardless of blockchain implementation. This also eliminates the need for private transactions, because the data in this private blockchain does not need to be shared. What is important for this project is the way application logic is written in the framework, how an integration with Apache Isis applications can be made, and how an integration with a public blockchain can be made.

In Hyperledger Sawtooth, the application logic is encapsulated within transaction families, which can be programmed using several programming languages [23]. Integrations can be written in several languages as well, but the Java SDK is still being worked on and can not be reliably used [14]. This means that an integration with Apache Isis applications would be difficult.

Hyperledger Fabric does offer a production ready Java SDK to create integrations with its blockchain [13]. However, all actual application logic is encapsulated within *chaincode*, which can only be written using Go [21]. This would mean a lot of context switching. Since the project already encapsulates a lot of context switching between blockchain and business applications, using this could be detrimental to the project's productivity.

Quorum is a bit different, since it is essentially built as a private version of the Ethereum blockchain. This means that application logic is encapsulated in almost exactly the same way as on a regular Ethereum blockchain, by creating smart contracts with a language such as Solidity [15]. Most existing Ethereum tooling is also functional for a Quorum-based blockchain, which offers possibilities. There is an extensive Java library [2] to connect with an Ethereum blockchain, which also has functionality to include Quorum based blockchains. Because it is so similar in use to Ethereum, the same code code be reused between the private and public instances if necessary. Quorum seems to be the best choice for this project.

3.3 Implementation design

We presented two different solutions to the privacy problems of using a public blockchain. The first one is saving the audit trail data on a private blockchain, and saving hashes of this data on a public blockchain. The second solution is encrypting the audit trail data and adding the encrypted data itself to a public blockchain.

While using data hashes is not able to prevent tampering from happening, it is the only method that is able to scale to larger data entries, as storing encrypted data would cause the costs of data storage to skyrocket. Therefore, the implementation will store the actual audit data entries in the application's own database, while a hash of this audit data entry will be stored on the Ethereum blockchain for validation.

[2] https://web3j.io/ (Accessed 2018-04-26)

We will not add it in this implementation, but this method can be extended by storing the audit data on a local instance of Quorum, which can be anchored to the Ethereum blockchain. By using this extension it would be possible to further limit the amount of transactions to the public Ethereum blockchain, by only committing certain block hashes.

CHAPTER 4

Implementation

4.1 The AuditTrail Smart Contract

The cornerstone of our implementation is the AuditTrail smart contract that we use to store hashes of the data we are saving locally. This smart contract is written for the Ethereum blockchain, using the Solidity programming language. For the demo application, this contract is deployed on the public Rinkeby Ethereum test network.

4.1.1 Solidity

Ethereum smart contracts are executed by the EVM. There are several languages that compile to EVM bytecode, of which Solidity is the most important and most widely used. Solidity is a language based on JavaScript that was specifically created to write smart contracts. As such, it has been made to work with the limitations of the EVM, such as the gas limits.

This is visible in the data types of Solidity, which are quite rudimentary. These data types are divided in value types, reference types, and mappings [8]. The value types consist of integers (signed and unsigned), addresses, and fixed-length byte arrays. These value types can be further specified in size, in order to fully utilise the available storage space. Signed and unsigned integers can be specified in steps of eight bits from `(u)int8` to `(u)int256`, and fixed-length byte arrays can be specified in steps of one byte from `bytes1` to `bytes32` [8].

The reference types consist of arrays and structs. An array can hold any other data type (including reference types), and can be dynamic [8]. Structs can be used to create new data types, by defining named members, quite similar to structs in C [8]. Next, mappings are very basic hashtable, that is just able to map a key to a value, without any other features, like being able to iterate over all keys or values [8]. Finally, Solidity has the ability to emit events, which are added to the transaction log on the blockchain, and can be accessed from outside of the contract [8].

4.1.2 Truffle & Ganache

To assist in smart contract development, there is a suite of development tools for smart contract development called Truffle [1]. Truffle makes it easy to compile, link, and deploy contracts to the Ethereum blockchain. It also offers features for contract migrations, and it offers an extensive framework for unit testing smart contracts, with tests written either in Javascript or Solidity.

The same company also develops Ganache [2], a local development version of the Ethereum blockchain. This local development blockchain only runs a single node, so it is very fast in transaction times. It also provides a set of ten unlocked test accounts with 100 Ether each. This makes Ganache perfect for fast development and testing of smart contracts.

[1] http://truffleframework.com/ (Accessed 2018-05-26)
[2] http://truffleframework.com/ganache/ (Accessed 2018-05-26)

4.1.3 Smart contract functionality

To assist in explaining the functionality of the smart contract excerpts of the most important parts of the contract are added below.

The AuditTrail smart contract's main functionality comes from its `dataHashes` mapping. This mapping contains the corresponding data hash for every transaction identifier that has been audited by the contract. Besides this, it also saves an array of all transaction identifiers so it can be iterated over.

The `audit` function of this contract saves the given transaction identifier and data hash in this mapping, saves it to a list of audited transaction identifiers, and emits an Audit event. The contract also contains a function to validate transactions, by comparing the passed transaction identifier and data hash with the values inside the `dataHashes` mapping.

```
contract AuditTrail {
    ...

    bytes28[] public auditedTransactions;
    mapping(bytes28 => bytes32) public dataHashes;
    event Audit(bytes28 transactionIdentifier, bytes32 dataHash);

    ...

    function audit(bytes28 transactionIdentifier, bytes32 dataHash)
    external ownerOnly {
        require(dataHashes[transactionIdentifier] == 0,
                "A transaction can only be audited once");
        dataHashes[transactionIdentifier] = dataHash;
        auditedTransactions.push(transactionIdentifier);
        emit Audit(transactionIdentifier, dataHash);
    }

    function validate(bytes28 transactionIdentifier, bytes32 dataHash)
    external view returns(uint8) {
        return dataHashes[transactionIdentifier] == dataHash ? 0 : 1;
    }
}
```

4.2 Integrating with Apache Isis applications

With a smart contract ready that can audit transactions, the next step is to implement an audit trail using Apache Isis, that submits hashes of its audit data to this smart contract.

4.2.1 Web3j

To integrate Java applications with Ethereum-based blockchains there is a library called Web3j [3]. It uses the Ethereum JSON-RPC [4] to interface with the blockchain, and it provides straightforward bindings to these RPC calls. Besides, it offers the ability to generate Java smart contract wrappers from compiled Solidity code or Truffle specifications. This allows for seamless integrations between the blockchain and the application.

To effectively use Web3j in Apache Isis applications, we created a specific domain service *Web3Service*, which contains a web3j instance, the credentials for an Ethereum wallet, and a wrapper around a deployed AuditTrail smart contract. Through configuration properties in Apache Isis configuration files, it is possible to specify the private key for this Ethereum account and the URL used to connect with a running blockchain instance. If no account credentials or

[3] https://web3j.io/ (Accessed 2018-05-17)
[4] https://github.com/ethereum/wiki/wiki/JSON-RPC (Accessed 2018-05-17)

Ethereum URL are provided, the implementation uses a local Ganache instance with one of its test accounts.

4.2.2 AuditerServiceUsingBlockchain implementation

To create an automatic audit trail, we implemented the Apache Isis AuditerService SPI. This AuditerService SPI is defined as the following interface:

```
1  public interface AuditerService {
2      boolean isEnabled();
3      public void audit(
4          final UUID transactionId,
5          final int sequence,
6          String targetClass,
7          final Bookmark target,
8          String memberIdentifier,
9          final String propertyId,
10         final String preValue,
11         final String postValue,
12         final String user,
13         final java.sql.Timestamp timestamp
14     );
15 }
```

In the signature of the `audit` method of this interface, the `transactionId` signifies the application interaction that caused the changes, while this `transactionId` in combination with the `sequence` signifies the corresponding database transaction. The `targetClass` and `target` signify the object that was changed, and the `memberIdentifier` and `propertyId` the specific changed property.

The audit method is called by the framework for every state change that occurs as a result of an interaction. This will most likely occur more than once per interaction, since a single interaction can lead to multiple changes in state. If every audit call leads to a transaction on the blockchain, the costs of this method would increase, keeping the base costs of a transaction in mind.

To keep the number of blockchain transactions down, we choose to aggregate all changes within a single database transaction and save this as a single ThreadLocal *Audit Entry*. At the end of the transaction, a hash of the Audit Entry is taken, and is sent to the AuditTrail smart contract, together with a transaction identifier, which is a representation of the `transactionId`, `sequence` and `timestamp` of the Audit Entry. This was achieved by also implementing the PublisherService SPI. This SPI gets called at the end of each transaction, allowing us to commit the aggregated Audit Entry.

The blockchain transaction is sent asynchronously, so that the rest of the application can continue running while the transaction is executed on the blockchain. When the transaction completes, a callback is called, adding the Ethereum transaction hash to the Audit Entry, so the specific Ethereum transaction can be looked up.

4.2.3 Audit trail validation

Now that the hashes of all Audit Entries are getting added to the smart contract on the blockchain, the final step is being able to validate that the Audit Entries that are stored in the application's database are still correct. To achieve this, individual Audit Entries can be validated with a validate action, which calls the `validate` method on the AuditTrail smart contract. This validate action stores the validation result on the Audit Entry, and updates its last validation date.

The entire audit trail can also be validated, by validating every single Audit Entry in the audit trail. This ensures that all Audit Entries in the audit trail are valid, but in order to fully validate the audit trail, it is also checked for missing entries. This is done by iterating over

all audited transaction identifiers that are saved in the smart contract, and verifying that the corresponding Audit Entry can be found within the application. The results of this validation are presented as three different lists – of validated audit entries, of invalidated audit entries, and of missing audit entries.

When Audit Entries are missing or invalidated, steps need to be taken in order to restore the correct data. The timestamp and transaction id of Audit Entries are known and stored on the blockchain, which makes it easier to identify which data needs to be restored.

CHAPTER 5

Evaluation

To evaluate the implementation, three different scenarios have been created, in which the data inside the audit trail would be invalidated. After executing these scenarios we validate the audit trail using the implementation described in the previous chapter.

For these scenarios, we have added the blockchain audit trail implementation to two demo applications, which are based on actual applications that are being used. The first is based on Incode's Contact App [1], and is used for internal contact management within companies. The second is a larger scale application based on Estatio [2], which is a full-fledged estate management system.

5.1 Scenario 1 - Shift the blame

Sven, an employee of Acme Corporation, maliciously changes the email address of a contact in the application. After doing so, he attempts to shift the blame to his coworker Peter by editing the audit trail data outside of the application straight in the database. Sven hopes to gain a professional advantage over Peter, who is a great source of jealousy for Sven, by reporting Peter's alleged incompetency to their superior.

Unknown to Sven, however, Acme had just upgraded their audit trail to the new implementation that is validated against the Ethereum blockchain. After Sven's report, his superior decides to validate the audit trail and sees that the corresponding audit entry is invalidated and has been tampered with. The corresponding audit entry had not yet been included in the company's nightly backups, so while they could see that there had been tampering, they could not see who the actual user was that changed the contact's email address. In this case, there was enough evidence to suspect Sven's intentions, and as a result he is placed under closer supervision.

The steps to simulate this scenario are:

- Find a contact in the application.
- Change this contacts email address.
- Open the database that is connected with the application, and find the corresponding audit entry.
- Change the user field of this audit entry from Sven to Peter.

After following these steps, running the audit trail validation inside the application results in the corresponding audit entry being invalidated. Figure 5.1 shows that the edited audit entry with user Peter has been invalidated.

[1] https://github.com/incodehq/contactapp (Accessed 2018-05-29)
[2] https://github.com/estatio/estatio (Accessed 2018-05-30

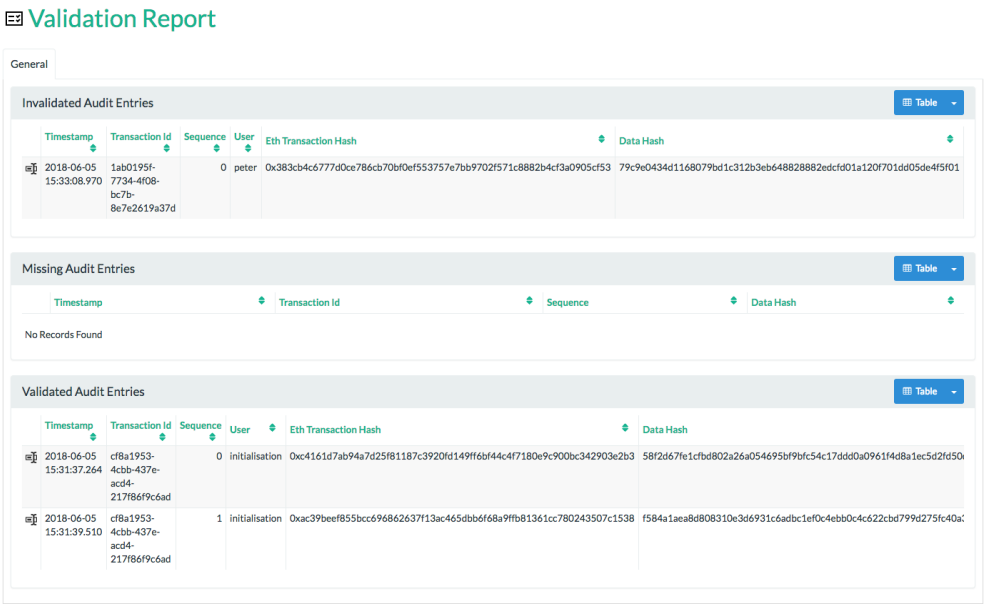

Figure 5.1: The edited audit entry with user Peter has been invalidated.

5.2 Scenario 2 - Cover your tracks

Mary has worked at Acme Corporation for fifteen years, but lately she has been feeling close to a burnout, and has decided she wants to quit her job and retire to the Colombian countryside. To fill the gap in her finances, she changed the recipient on one of the company's larger invoices to her husband's IBAN, which she edited back to the original after it had been paid. A few days after this, she quits her job and is preparing to leave her old life behind. But as she is getting ready, she finds out that her old company employs an audit trail, which contains all actions taken inside the application, and now she wishes to cover her tracks. She still has her old company credentials, which she uses to gain access to the application database. In the database she removes all audit entries that log her changes.

Unknown to Mary still, Acme has the policy to routinely validate their audit trail against the Ethereum blockchain at the end of every working day, so they notice the missing audit entries. Since these entries had already been included in the company's backups, the correct data can easily be restored, and Mary's malicious actions come to light. She is reported to the authorities, and apprehended at the airport, just as she was leaving the country. Because the data in the audit trail can be validated against the blockchain, this is reliably used as evidence in court, and Mary and her husband are convicted of fraud.

The steps to simulate this scenario are:

- Change the IBAN of an invoice that still needs to be paid.
- Go through the payment process to.
- Open the database that is connected with the application, and change the IBAN back.
- In the database find the corresponding audit entries.
- Delete these audit entries and their ChangedProperties.

After following these steps, running the audit trail validation inside the application results in several audit entries being reported as missing. Figure 5.2 shows that two audit entries with timestamps around 18:25 are missing, which contained the interaction that changed the IBAN and the payment to this IBAN. The final missing audit entry with timestamp around 18:39 contained the action that changed the IBAN back.

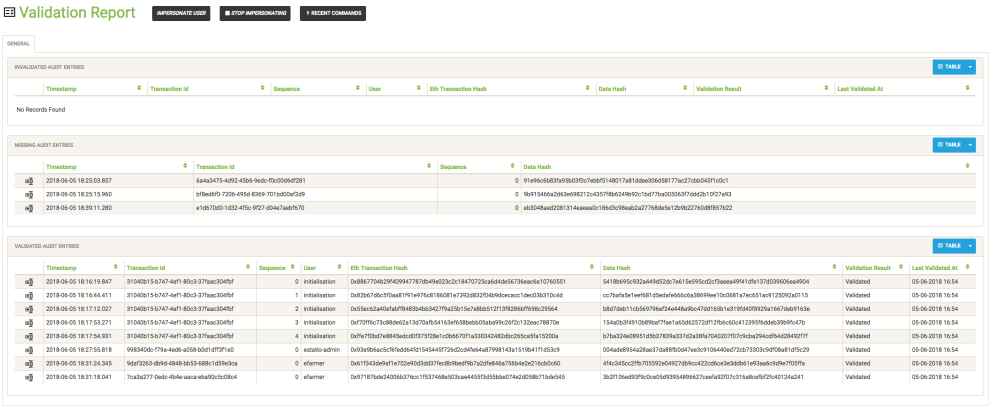

Figure 5.2: The audit entries with timestamps 18:25:03, 18:25:15 and 18:39:11 are reported as missing by the validation.

5.3 Scenario 3 - Inexperienced admin

Paul is one of the newer system administrators at Acme Corporation, and he is still getting trained to use the systems and processes that are employed by the company. At his first day working in the field he gets a request from his coworker Lucy to restore some files from a backup a few days back. Not being knowledgeable enough, he performs a full server backup, which incidentally includes the application database. Happy he could help, Paul informs Lucy that she can access the files she needed, and he goes on with the rest of his day.

At the end of the day, the audit trail is validated, and Paul's mistakes are discovered. Luckily, they can reconstruct most of the correct data with the correct backups, but the changes that had been made the same day as Paul's mistake could not be recovered. This highlights a part of the weakness of our implementation, but it also displays the way these kinds of mistakes can at least be detected quite early on.

The steps to simulate this scenario are:

- Take a backup of the database.
- Make some changes in the application.
- Restore the backup.

After following these steps, running the audit trail validation inside the application results in the last few audit entries being reported as missing. Figure 5.3 shows that the last two audit entries are reported as missing, as these are the ones that were deleted while restoring the database backup.

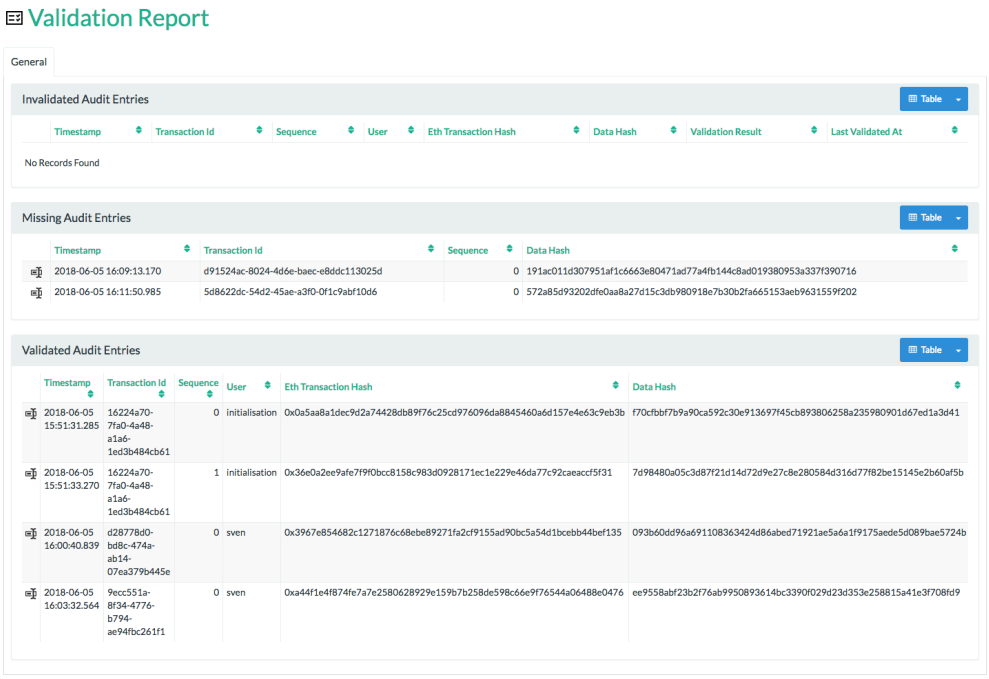

Figure 5.3: The final two audit entries are missing as they have been removed in the restoring of the backup.

CHAPTER 6

Discussion

6.1 Conclusions

The objective of this study was to implement an automatic audit trail with Apache Isis, and use blockchain to ensure that it has not been tampered with, without sharing application data with unauthorised parties. In order to achieve this we have compared two different methods to use blockchain.

The first method stores the audit trail in the regular application database, and stores hashes of this audit trail data on the Ethereum blockchain to validate its integrity. The second method stores the entire audit trail data in encrypted form on the Ethereum blockchain. The first method is not able to **prevent** tampering from happening, and is only able to **validate** the audit trail data. However, the second method is unable to scale because of the costs associated with storing data on the blockchain, and is therefore not realistically viable.

We implemented the first method by implementing the Apache Isis AuditerService and PublisherService SPIs and using Web3j to connect it with an Ethereum smart contract that maps transaction identifiers to their corresponding data hashes. We evaluated this implementation by defining three different scenarios in which a regular audit trail would be insufficient to detect data tampering. We described the steps needed to simulate these scenarios, after which we showed the way our implementation would detect the tampering.

Using this research we are able to answer the research questions.

> *Can we implement an automatic audit trail for Apache Isis applications, using blockchain technology?*

Section 4.2 describes the implementation that was used to integrate an Ethereum smart contract with an Apache Isis application. We manage to leverage the strength of Apache Isis by implementing the AuditerService and PublisherService SPIs, and we are able to link this audit trail to a deployed instance of the smart contract described in section 4.1.

We further specified the requirements of the implementation in research question 2:

> *Is this blockchain audit trail able to ensure that it has not been tampered with, without sharing application data with unauthorised parties?*

Section 3.3 describes the design of our implementation, as well as its limitations. Because only hashes are stored on the blockchain no actual data is shared with the public. Because the audit trail data is stored in the application's own database, it is not resistant to tampering. This means that our implementation can only be used to validate the integrity of the data, but not to prevent tampering from happening. However, when paired with a regular backup and validation protocol, it can be possible to retrieve and restore the correct data after tampering has been detected.

We conclude that our implementation is not able to ensure that it has not been tampered with, but it offers more certainty about the integrity of the data than a regular audit trail

implementation does. We also conclude that the implementation offers this increased certainty without sharing application data with unauthorised parties.

6.2 Limits of the implementation

We discussed in detail the limitation of only being able to detect tampering, rather than completely prevent it. This is not the only limitation to our implementation. Another weakness is application crashes and outages during the auditing process. It could be possible to have an application crash or other form of outage during the process of auditing a transaction. This could lead to an incomplete audit entry being saved to the database, but nothing being transacted to the blockchain, invalidating the audit entry.

Next, since some audit entries are impossible to retrieve or correct after being invalidated, they will always be displayed in the list of invalidated audit entries. When an application is in production for a long time and these incidents happen regularly, the list of invalidated audit entries can grow to the extent where it will become difficult to detect newer errors because the interface is cluttered with older invalidated entries.

Finally, when more than five blockchain transactions are pending simultaneously, new ones don't go through. This is usually not a problem, but it could be when a user is performing many different actions in quick succession. In order to overcome this problem, a solution could be a protocol to to automatically resend failed transactions after waiting for some time, or a protocol to automatically queue new transactions when there are already too many pending transactions.

6.3 Recommendations for further research

To overcome the limitations of our hashing-based approach, it would be valuable to look for a way to store the data itself in a way that could prevent tampering. As we discussed, encrypted data could be stored on the blockchain directly, but this would lead to incredibly high costs. It would be interesting to see if an implementation for this could be achieved, disregarding transaction costs.

There could also be potential in IPFS [1]. IPFS is a distributed files system that offers deduplication and version history for all stored data. IPFS would allow data to be stored in a way that is resistant to tampering because of its distributed nature and checksum verification. [3]

These kinds of guarantees look similar to the ones provided by blockchain, showing the potential of IPFS to store the full audit trail data in encrypted form to prevent tampering from happening. The IPFS website also shows the possibility to integrate IPFS with blockchain technology by linking to certain specific versions of data in IPFS [27]:

> IPFS and the Blockchain are a perfect match! You can address large amounts of data with IPFS, and place the immutable, permanent IPFS links into a blockchain transaction. This timestamps and secures your content, without having to put the data on the chain itself.

[1] https://ipfs.io/ (Accessed 2018-05-31)

References

[1] Spencer Ackerman. "Newest cyber threat will be data manipulation, US intelligence chief says". In: *The Guardian* (2015). URL: https://www.theguardian.com/technology/2015/sep/10/cyber-threat-data-manipulation-us-intelligence-chief (visited on 04/17/2018).

[2] Igor Barinov et al. "SYSTEM AND METHOD FOR VERIFYING DATA INTEGRITY USING A BLOCKCHAIN NETWORK". Pat. 20180025181. 2018. URL: http://www.freepatentsonline.com/y2018/0025181.html (visited on 05/06/2018).

[3] Juan Benet. "IPFS - Content Addressed, Versioned, P2P File System (DRAFT 3)". Unknown. URL: https://github.com/ipfs/papers/raw/master/ipfs-cap2pfs/ipfs-p2p-file-system.pdf (visited on 05/31/2018).

[4] Bitfury Group. *Blockchain Land Registry*. 2017. URL: https://exonum.com/napr (visited on 04/19/2018).

[5] Bitfury Group Limited. "On Blockchain Auditability". 2016. URL: https://bitfury.com/content/downloads/bitfury_white_paper_on_blockchain_auditability.pdf (visited on 05/31/2018).

[6] Tim Blankers. "Blockchains: taking ecosystems to the moon". AWESOME IT. 2018. URL: https://awesomeit.nl/ (visited on 04/21/2018).

[7] Vitalik Buterin. "Ethereum: A Next-Generation Smart Contract and Decentralized Application Platform". 2013. URL: https://github.com/ethereum/wiki/wiki/White-Paper (visited on 04/21/2018).

[8] Ethereum. *Solidity Documentation*. 2017. URL: https://solidity.readthedocs.io/en/v0.4.23/ (visited on 04/26/2018).

[9] Debraj Ghosh. "How the Byzantine General Sacked the Castle: A Look Into Blockchain". In: *Medium* (2016). URL: https://medium.com/@DebrajG/how-the-byzantine-general-sacked-the-castle-a-look-into-blockchain-370fe637502c (visited on 04/17/2018).

[10] Sander Ginn. "CLIsis: An Interface for Visually Impaired Users of Apache Isis Applications". University of Amsterdam, June 2016. URL: https://esc.fnwi.uva.nl/thesis/centraal/files/f270412620.pdf (visited on 05/31/2018).

[11] Dan Haywood. *Apache Isis*. Updated 2018. URL: https://isis.apache.org/ (visited on 04/10/2018).

[12] Dan Haywood. *Apache Isis Domain Services*. Updated 2018. URL: https://isis.apache.org/guides/rgsvc/rgsvc.html (visited on 04/12/2018).

[13] Hyperledger. *Hyperledger Fabric Documentation*. 2017. URL: https://hyperledger-fabric.readthedocs.io/en/release-1.1/ (visited on 04/26/2018).

[14] Intel Corporation. *Hyperledger Sawtooth Documentation*. 2017. URL: https://sawtooth.hyperledger.org/docs/core/releases/latest/ (visited on 04/19/2018).

[15] J. P. Morgan. *Quorum Wiki*. 2017. URL: https://github.com/jpmorganchase/quorum/wiki (visited on 04/24/2018).

[16] Praveen Jayachandran. "The difference between public and private blockchain". In: *Blockchain Unleashed: IBM Blockchain Blog* (2017). URL: https://www.ibm.com/blogs/blockchain/2017/05/the-difference-between-public-and-private-blockchain/ (visited on 05/31/2018).

[17] Leslie Lamport, Robert Shostak, and Marshall Pease. "The Byzantine Generals Problem". In: *ACM Transactions on Programming Languages and Systems* 4.3 (1982), pp. 382–401. URL: https://web.archive.org/web/20170205142845/http://lamport.azurewebsites.net/pubs/byz.pdf (visited on 04/17/2018).

[18] Matthew Leising. "The Ether Thief". In: *Bloomberg* (2017). URL: https://www.bloomberg.com/features/2017-the-ether-thief/ (visited on 04/21/2018).

[19] LinuxFoundationX. "Chapter 3: The Promise of Business Blockchain Technologies". In: Blockchain for Business - An Introduction to Hyperledger Technologies. 2017. URL: https://courses.edx.org/courses/course-v1:LinuxFoundationX+LFS171x+3T2017/courseware/5f4fa9501e284f4ebebbc00085699e27/79cebeccac2b43908eabd75e76ca8e24/3 (visited on 04/19/2018).

[20] LinuxFoundationX. "Chapter 6: Hyperledger Sawtooth – Key Components and Transaction Flow". In: Blockchain for Business - An Introduction to Hyperledger Technologies. 2017. URL: https://courses.edx.org/courses/course-v1:LinuxFoundationX+LFS171x+3T2017/courseware/9faeeb6df17c4d25a205c965473925e5/f26e2957264d421bac8af52e97bfb4de/1 (visited on 04/24/2018).

[21] LinuxFoundationX. "Chapter 7: Hyperledger Fabric – Key Components and Transaction Flow". In: Blockchain for Business - An Introduction to Hyperledger Technologies. 2017. URL: https://courses.edx.org/courses/course-v1:LinuxFoundationX+LFS171x+3T2017/courseware/f0db5224eb0e4bbb8cc1e93a6819012c/5ebddaca983d4d6d952e83f95ea9e281/1 (visited on 04/24/2018).

[22] Weizhi Meng et al. "When Intrusion Detection Meets Blockchain Technology: A Review". In: *IEEE Access* 6 (2018), pp. 10179–10188. URL: https://ieeexplore.ieee.org/document/8274922 (visited on 04/19/2018).

[23] Dan Middleton. "WTF?! (Whats a Transaction Family?!)" In: *Hyperledger Blog* (2017). URL: https://www.hyperledger.org/blog/2017/06/22/whats-a-transaction-family (visited on 04/24/2018).

[24] Satoshi Nakamoto. "Bitcoin: A Peer-to-Peer Electronic Cash System". 2008. URL: https://bitcoin.org/bitcoin.pdf (visited on 04/18/2018).

[25] Richard Pawson. "Naked Objects". PhD thesis. University of Dublin, Trinity College, June 2004. URL: https://isis.apache.org/versions/1.13.1/resources/thesis/Pawson-Naked-Objects-thesis.pdf (visited on 04/12/2018).

[26] Andrew Powell-Morse. "Domain-Driven Design – What is it and how do you use it?" In: *Airbrake Blog* (2017). URL: https://airbrake.io/blog/software-design/domain-driven-design (visited on 04/12/2018).

[27] Protocol Labs. *IPFS is the Distributed Web*. Unknown. URL: https://ipfs.io/ (visited on 05/31/2018).

[28] Ameer Rosic. "Hypothetical Attacks on Cryptocurrencies". In: *Blockgeeks* (2018). URL: https://blockgeeks.com/guides/hypothetical-attacks-on-cryptocurrencies/ (visited on 04/21/2018).

[29] Bruce Schneier and John Kelsey. "Secure Audit Logs to Support Computer Forensics". In: *ACM Transactions on Information and System Security* 2.2 (May 1999), pp. 159–176. URL: https://www.schneier.com/academic/paperfiles/paper-auditlogs.pdf (visited on 04/18/2018).

[30] Laura Shin. "The First Government To Secure Land Titles On The Bitcoin Blockchain Expands Project". In: *Forbes* (2017). URL: https://www.forbes.com/sites/laurashin/2017/02/07/the-first-government-to-secure-land-titles-on-the-bitcoin-blockchain-expands-project/#48f363dc4dcd (visited on 04/19/2018).

[31] Nick Szabo. *Smart Contracts: Building Blocks for Digital Markets*. 1996. URL: http://www.fon.hum.uva.nl/rob/Courses/InformationInSpeech/CDROM/Literature/LOTwinterschool2006/szabo.best.vwh.net/smart_contracts_2.html (visited on 04/21/2018).

[32] The Economist. "Blockchains: The great chain of being sure about things". In: *The Economist* (2015). URL: https://www.economist.com/news/briefing/21677228-technology-behind-bitcoin-lets-people-who-do-not-know-or-trust-each-other-build-dependable (visited on 04/12/2018).

[33] Unknown. *Descriptions of SHA-256, SHA-384, and SHA-512*. Unknown. URL: http://www.iwar.org.uk/comsec/resources/cipher/sha256-384-512.pdf (visited on 05/26/2018).

[34] Gavin Wood. "Ethereum: A Secure Decentralised Generalised Transaction Ledger — Byzantium Version f72032b – 2018-05-04". 2018. URL: https://ethereum.github.io/yellowpaper/paper.pdf (visited on 05/06/2018).

[35] Kim Zetter. "The biggest security threats we'll face in 2016". In: *Wired* (2016). URL: https://www.wired.com/2016/01/the-biggest-security-threats-well-face-in-2016/ (visited on 04/17/2018).

Acronyms

API Application Programmer Interface. 11, 12, 23

DAO Decentralised Autonomous Organisation. 16

EVM Ethereum Virtual Machine. 16, 23

IBAN International Bank Account Number. 32

IPFS InterPlanetary File System. 34

JSON JavaScript Object Notation. 28

MVC Model-View-Controller. 12

REST Representational State Transfer. 11, 23

RPC Remote Procedure Call. 28

SDK Software Development Kit. 22–24

SPI Service Provider Interface. 12–14, 20, 29, 33

UI User Interface. 11

UL Ubiquitous Language. 11

URL Uniform Resource Locator. 28

Appendices

APPENDIX A

Code repository

The full implementation code can be found and inspected at https://github.com/rkalis/blockchain-audit-trail. This repository features a comprehensive README with further instructions on where to find the specific code files, as well as detailed local installation instructions.