Off-chaining Smart Contract Data to RDBMS

Simon Fallnich

Patrick Friedrich

Tarek Higazi

Vincent Jonany

Dukagjin Ramosaj

Kevin Marcel Styp-Rekowski

Thanh Tuan Tenh Cong

March 15, 2018

Abstract

Simon Fallnich

Smart contracts are increasingly used in various domains as they enable credible computations without the need for a trusted third party. In order to ensure trustless properties, computations and storage have to be distributed on a large network of nodes, which makes it expensive to use more than a few hundred bytes of data in smart contract computations. In this report, we explore possibilities to *off-chain* smart contract data, i.e., storing it outside the blockchain without compromising the data integrity. We present a system which leverages cryptographic proofs to store the state variables of a smart contract in an untrusted relational database, while maintaining the ability to use those variables in trustless computations. The performance of this system is evaluated for two use cases to give a differentiated assessment, in which cases the benefit of storing data off-chain outperforms the overhead added by the applied integrity check mechanism. It is found that, although performance increases rely on the specific use case, off-chaining delivers the best results for smart contracts which use large state variables and functions that change many state variables at once. Finally, we present a concept of a translator which automatically transforms common smart contracts to use the presented off-chaining approach.

Contents

List of Figures

1 Introduction

Tarek Higazi, Dukagjin Ramosaj

One of the most significant technological achievements of this decade is the invention of blockchain technology. In the span of 10 years this technology has sprouted a new concept which the world has come to know as decentralized cryptocurrencies, and which today hold a combined value of hundreds of billions of US dollars [1].

The blockchain architecture gives us a unique combination of benefits which include data integrity, transparency, security and peer-reviewed actions. "Blockchain-based applications, however, may also suffer from high computational and storage expenses, negatively impacting overall performance and scalability." [2]

In recent years, blockchain technology has not only been utilized for cryptocurrencies, but rather to create a development platform as well. The concept of smart contracts was developed by Ethereum, a platform which, alongside its role in cryptocurrencies, is also known for the Ethereum Blockchain Development platform which utilizes blockchain. Through this concept, many great ideas and implementations have been produced. There is however a major issue which puts a limitation on the realization of these ideas. Just like any other program, there are local variables used to store different forms of data, yet the local variables in smart contracts are persistent on the blockchain network. This ultimately means that this data is replicated throughout all of the machines which participate in the blockchain network. Thus, storing data in smart contracts is expensive, and not practical.

The problem stated above calls for an approach which would enable users to store large amounts of data in a blockchain environment, while still benefiting from the features that come with it such as transparency, incorruptibility and data-integrity. The solution which we discuss and showcase in our paper is the approach of storing as much of the data as we can "off-chain" while preserving the integrity and the properties of the blockchain.

"Off-chaining" is basically the act of moving data or computation flows outside of the blockchain so that they can be stored or computed elsewhere [2]. However storing data in different databases, servers or any other location comes at a cost, and the blockchain's core properties may not be possible to maintain. Ultimately, the framework should remain "trust-less", meaning that no external trust into a single actor in the system is required.

In this paper we will describe how we developed a prototype which implements such an off-chaining approach with minimal impact to the blockchain's properties. We came up with different use cases as examples of where it would be beneficial to use blockchain technology, and more importantly, where our off-chaining approach would make the most sense. We will also describe the implementation details together with the design decisions which were incorporated into the prototype, and explain how it works. In addition, we will lay out the results of our benchmarking and compare the costs and execution times of storing the data on-chain vs off-chain, and try to reach a conclusion as to where and when this solution would be applicable and useful. We will also provide insights on how one could develop this off-chaining approach even further.

2 Background

Tarek Higazi, Dukagjin Ramosaj

2.1 Blockchain

A new concept for trustless, decentralized and transparent information processing and storage has been developed, introduced and already integrated into several mainstream applications in the last years. Bitcoin, developed by an unknown person or group under the name of Satoshi Nakamoto [3], led this initiative back in 2008 with its open-source, peer-to-peer network which cryptographically stores records in a chain and serves as a distributed ledger, this was named the blockchain. The currency in which these operations were valued and paid for with is called Bitcoin, and it was the first decentralized digital currency.

Blockchains are essentially continuously growing lists of information which are linked together and secured using cryptography. They consist of specific properties such as immutability, transparency, and the facilitation of cryptographically-secured peer-recorded transactions between users, and which are ratified by consensus on the network. The main benefit of blockchains is that they are designed to be incorruptible in the sense that the data they hold cannot be modified without mutual consent. The entire history of the blockchain is always stored and recorded on it and is verified by peers. It is possible for every user to parse the blockchain's history until its very beginning.

This technology has been so far mostly used in the financial sector in applications such as peer-to-peer systems, cryptography, consensus protocols, decentralized storage, decentralized processing and smart contracts. The consolidation of these concepts is what makes blockchains exceptionally innovative as a programmable platform and system at the same time. Being able to build up trustless interactions, along with business disintermediation, continues to be one of the most important objectives of using blockchain technology.

2.2 Ethereum and Smart Contracts

Smart contracts are a feature which the Ethereum blockchain is based on. A smart contract is basically a computer program which can check for the fulfillment of certain preconfigured conditions and control the transfer of funds or other assets between several parties [4]. While a standard contract enforces its terms through its legality, a smart contract actualizes this enforcement automatically through network consensus and a cryptographically secured system. The contracts are stored on the blockchain and thereby serve as a decentralized middle man which stores records, enforces conditions and is completely neutral and transparent. The most popular platform which utilizes this technology is the Ethereum blockchain. It empowers developers to create their own smart contracts and their own decentralized blockchain applications [4].

The runtime environment in which smart contracts are processed is called the Ethereum Virtual Machine (EVM). It continuously runs on every node on the chain, which means that every task which is executed inside of the EVM is executed by each full node [4]. This is a crucial part of the Ethereum consensus model and has the advantage that any contract on the EVM can call any other contract at zero cost, however, the disadvantage is that the computational steps on the EVM are exceptionally costly [4]. The EVM is also isolated in the sense that no outside framework or file system is accessible, this to ensure determinism. The smart contracts themselves are written using the Solidity programming language, which was designed for this purpose. Moreover, it allows users to create contracts which can be used for voting, crowdfunding, blind auctions, multi-signature wallets and more.

In order to ensure that the Ethereum network would not be abused or deliberately attacked, the Ethereum protocol charges a fee for every computational step. The way this cost is paid is through an attribute called 'gas'. The fee or the price of the gas is determined simply via supply and demand; the users' willingness to pay vs. the price for which the miners are willing to mine the next block in the blockchain. Basically, the way these fees work is that every transaction contains a 'gasPrice' attribute, which is the price per computational step, and the 'startGas' attribute, being the maximum amount of gas which the sender is willing to pay for the transaction. Therefore, at every execution of a transaction a prior evaluation of the transaction cost is performed [4].

$$gasCost(Tx) = gasPrice * startGas$$

The usage of the EVM for the most part makes sense when it comes to running business

logic applications ("if this, then that") such as confirming signatures or other cryptographic objects [5]. On the other hand, any utilization which incorporates using the EVM as a file storage platform or anything GUI related would not be practical due to the tremendous costs.

2.3 Integrity Checks

Simon Fallnich

This section describes basic principles to verify the integrity of data from an untrusted source.

2.3.1 Hashing

A *hash function* maps an input of arbitrary length to an ouput of fixed length, referred to as *hash* [6]. Although hash functions are used for a wide range of applications in the domain of computer science, the class of *cryptographic hash functions* is primarily relevant for our application. A cryptographic hash function, or *one-way function*, is a hash function which is infeasible to invert. This essential property lays the foundation for efficient integrity check mechanisms as the resulting hashes can be used to securely proof equality of the underlying data.

2.3.2 Merkle Tree

A *Merkle* tree, or *hash tree*, is a tree in which every leaf node stores the hash of a data block and every non-leaf node stores the hash of the concatenated hashes of its children [7]. This structure enables efficient, partial integrity checks on large data sets.

Figure 2.1 shows a binary Merkle tree with five leaves. As the node with index 4 is a leaf, the stored hash H_4 can be computed as

$$H_4 = h(D_0), \tag{2.1}$$

where $h(\cdot)$ is the employed hash function and D_0 is the corresponding data block for this particular leaf node. In contrast, the node with index 3 is a non-leaf node and its hash H_3 can hence be calculated as

$$H_3 = h(H_7 + H_8), \tag{2.2}$$

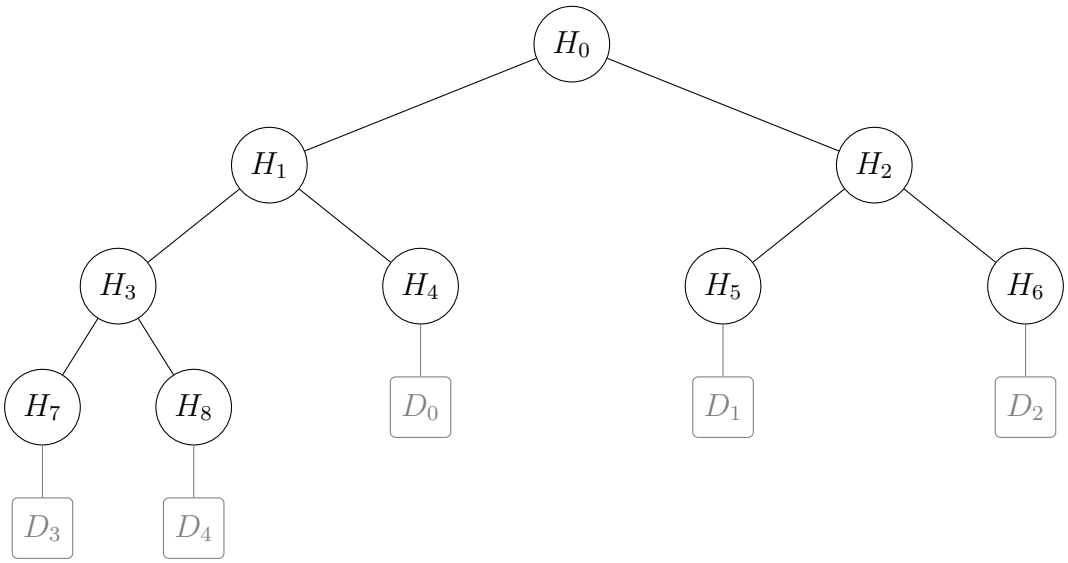


Figure 2.1: A binary Merkle tree with five leaf nodes. While H_i denotes the hash which is stored for the node with index i, D_j denotes the data block with index j.

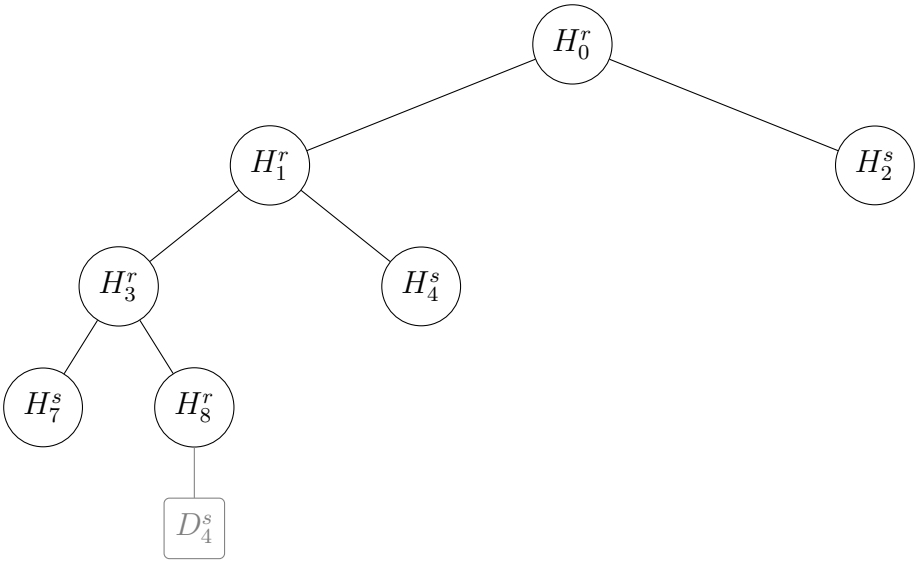


Figure 2.2: An exemplary Merkle tree integrity check. A superscript s denotes that the respective value is given by the untrusted sender, whereas a superscript r denotes a value which is computed by the receiver.

where $H_7 + H_8$ denotes the concatenation of the hashes of nodes 7 and 8. Similarly, the hash of the root node H_0, also referred to as *root hash* or *Merkle root*, results as

$$\begin{aligned} H_0 &= h(H_1 + H_2) && (2.3)\\ &= h(h(H_3 + H_4) + h(H_5 + H_6)) && (2.4)\\ &= h(h(h(H_7 + H_8) + h(D_0)) + h(h(D_1) + h(D_2))) && (2.5)\\ &= h(h(h(h(D_3) + h(D_4)) + h(D_0)) + h(h(D_1) + h(D_2))). && (2.6) \end{aligned}$$

Equation 2.6 indicates that, due to the recursive nature of the hash computation, there is no possibility of altering any data block without changing the root hash as well. Therefore, it is sufficient to know the root hash in order to be able to verify the integrity of any data block in the tree.

For instance, a sender wants to send data block D_4 to a receiver, which only knows the root hash H_0. Let D_4^s denote the sent data block, where the superscript s indicates that this data comes from the untrusted sender and could be different from the original D_4. In addition to the possibly altered data block D_4^s, the sender sends the hashes H_2^s, H_4^s and H_7^s. With the given data block and the given hashes, the sender then resconstructs the corresponding Merkle tree to compute the root hash H_0^r (see Figure 2.2), where the superscript r indicates that a hash is computed by the receiver:

$$\begin{aligned} H_8^r &= h(D_4^s) && (2.7)\\ H_3^r &= h(H_7^s + H_8^r) && (2.8)\\ H_1^r &= h(H_3^r + H_4^s) && (2.9)\\ H_0^r &= h(H_1^r + H_2^s). && (2.10) \end{aligned}$$

If the computed root hash H_0^r equals the known root hash H_0, the sender has successfully verified the integrity of the given data block ($D_4^s = D_4$) without knowing any other data block or non-root hash beforehand. This holds since, with the use of a cryptographic hash function (see subsection 2.3.1), the sender cannot determine a hash H_7' which, combined with an altered data block D_4', would yield the correct parent hash H_3.[1]

[1] For this, the sender would have to solve $H_3 = h(H_7' + D_4')$ for H_7', which requires the infeasible inversion of the cryptographic hash function $h(\cdot)$. One might argue that the hash H_3^r does not need to equal H_3 as long as the resulting root hash H_0^r equals H_0 — but this just propagates the problem of solving for an unknown input to $h(\cdot)$ to another level in the tree.

3 Related Work

Patrick Friedrich, Tarek Higazi

The initial concept of "off-chaining" was introduced to us by our project supervisor Jacob Eberhardt and the paper he published along with Professor Stefan Tai.

In their paper, they discuss the concept of "off-chaining" data from the blockchain and they propose several ways to achieve this while maintaining blockchain properties and data integrity. The most relevant approach we drew from was the "Content-Addressable Storage Pattern" [2], which proposes verifying the integrity of off-chained data via storing its hash on the blockchain. This allows us to trust that the off-chained data which is in the database hasn't been edited by any third party without our knowledge since any change to the data would later result in a different hash when compared [2].

To gain further inspiration on how the data integrity in our smart contracts could be verified, we conducted research on other systems that potentially make use of similar mechanisms. As our system provides external data to a smart contract, that is on-chain, we considered Oracles to be a concept worth looking at. "An Oracle, in the context of blockchains and smart contracts, is an agent which finds and verifies real-world occurrences and submits this information to a blockchain to be used by smart contracts." [8] In addition, we conducted research on potential integrity mechanisms which distributed cloud systems (like AWS or Azure) implement in order to provide as features for their clients. This class of mechanisms is called Remote Data Integrity Checking and our findings on it are presented in the second part of this chapter.

3.1 Oracles

Initial Research Questions

- Could Oracles provide further ideas for integrity checks?
- How does a smart contract which requested data through an Oracle check the data's integrity?
- Does this check happen somewhere else?

Considered Oracle: Oraclize Oraclize is a leading Oracle provider (for Ethereum and other blockchains) [9]. Their simple code samples [10] show an event-based approach. In fact, their smart contract functionality seems similar to our smart contracts, minus the integrity check (at least none is applied in these examples).

The Oraclize documentation offers some entry points into advanced integrity checks [11]. These Authenticity Proofs (provided by Oraclize with requested data or put on IPFS) [12] include TLSNotaryProofs, Android Proofs and Ledger Proofs. They can be stored on IPFS as the proof may be huge and thus expensive for the smart contract to run it. [13]

In this way, Oraclize saves its users costs. The proof (that the pushed data is the requested one) is provided to the smart contract only on request so that the smart contract can run the verification of the proof. As this is costly (and thus seemingly rarely done in practice) the proof is stored on IPFS and can be run later to verify the correctness of the pushed data.

There are examples online that show how to integrate the verification process of proofs [14]. They use several steps for verification: signatures, hashing, prefix match and APPKEY1 provenance. Moreover, Oraclize provides tools to verify Authenticity Proofs [15, 16].

Proofs Used by Oraclize The following question thus arose: Could the proofs used by Oraclize be applicable in our project?

The TLSNotaryProof [17, 18, 19] does not seem appropriate for our project; from our understanding, there is no check that the integrity of the data is intact but it is a proof that one party went to a certain webpage and got a valid response from the server. The returned data from the server is not the part in question here.

The Android Proof [20] is an own development by Oraclize based on a Google technique. It is meant to ensure an application is running in a safe environment, i.e. the Android device is secure. There do not seem to be any security guarantees concerning data integrity.

The Ledger Proof [21, 22] provides a proof that the application is running in the environment of a true Ledger device (secure hardware wallet).

Thus, all three proofs do not account for data integrity but are focused on providing a proof that the intermediary (here Oraclize) has not tampered with the retrieved data. These techniques cannot be used for data integrity checks as envisioned in this project.

3.2 Remote Data Integrity Checking

RDIC Techniques RDIC is applied in Cloud Computing settings where users want to check the integrity of their data stored on the servers of the provider. There are several different RDIC techniques [23] and much research is focused on this domain producing new approaches on a regular basis.

This paper introduces several protocols [24]; an identity-based integrity verification protocol [25], a protocol where only parts of the data are encrypted [26] and one which uses a Third Party Auditor (this approach also achieves privacy) [27].

Common techniques for RDIC include Mirroring, RAID Parity and Checksum. A probabilistic method for data integrity assessment comprises the following steps: insert the testing tuples, encrypt all data, keep the testing tuples and perform the data integrity proof in the cloud storage system with random bits that are appended as metadata (encrypted as well). This might also be possible on the blockchain with the user doing the encryption of the tuples as this step must not be publicly visible, because otherwise the tuples do not have a security effect. There could also be augmented provenance for the data integrity by collecting and using the history of data items. This could be implemented in the middleware (keep provenance data there). [28]

The proposed RDIC techniques seem to be more appropriate if we performed integrity checks in the middleware. They are often based on randomness, which could pose a challenge on the blockchain.

Therefore, we needed to find answers to the following questions: Is randomness easily achievable on the blockchain? Could we include the respective approach in our smart contracts?

Randomness in Solidity In general, there are three options for achieving randomness on the blockchain. Each with different criteria for choosing one approach over another [29]. The three options are Block hash PRNG (Pseudo-random number generator), Oracle RNG and Collab PRNG.

Several approaches and exemplary implementations can be found online. The coin flipper [30] is based on the block number (thus falls into the category of Block hash PRNG). The Randao [31] is a decentralized autonomous organization for generating random numbers (thus, a Collab PRNG approach). Here, several participants act together to perform the protocol with three phases to provide a random number. It can be used as a service.

A step-by-step explanation for choosing the way in which to generate a random number in a smart contract is offered and insights on general challenges are given [32]. This comprises approaches that use the blockhash (Block hash PRNG). In one example [33], the author generates a random number from 0 to 100 and a random number from 0 to 2 to the power of n. This code could be included in our own smart contract. In terms of Oracle RNG approaches, Oraclize [14] is a popular provider (also see above). It acts as an external randomness source. In contrast, an internal solution like eth-random [34] uses the block timestamp and seed to generate pseudo-randomness. It should also be easy to use in an own smart contract.

3.3 Summary

The most promising findings in alternative integrity checks are coming from Oraclize (a blockchain oracle provider) and the field of Remote Data Integrity Checking, where several techniques have been developed and a lot of research is happening. Oraclize provides the proofs to the smart contracts which call their service (and they may run it) or stores the proof for later retrieval on the decentralized database IPFS, depending on the choice made by the smart contract. Oraclize uses three different Authenticity Proofs (TLSNotaryProof, Android Proof, Ledger Proof) to provide the smart contract or the owner or user with a certainty that the pushed data from Oraclize to the smart contract is the correct data. Further research would be required to fully understand these techniques. As for now, the proofs seem to only concern the intermediary (Oraclize) and its trustworthiness and not the integrity of the data per se (i.e. only that Oraclize has not changed the data but no checks on the data e.g. in contrast to a prior state, when stored or similar).

Remote Data Integrity Checking is applied in the domain of Cloud Computing. Users check the integrity of outsourced data stored on the providers' servers. Most of the analyzed techniques rely on either a Third-Party Auditor, encryption, randomness (bit samples) or a combination. Whereas encryption was proposed by Jacob Eberhardt in his paper as a potential integrity check, a third-party solution seems to be out of scope for this project. In general, there could be a verifier that takes the role of a mediator between the database and the smart contract and its user and performs the data integrity check. An immediate solution could include a random sample of the data to check and in this way reduce the amount of data to be verified (also see section "Randomized Sampling" in chapter Future Work).

A form of randomness in Solidity and on the Ethereum blockchain can be achieved in three different ways; Block hash Pseudo-Random Number Generators, Oracle RNG and Collab PRNG. As the name indicates, Block hash PRNG rely on the hash of the current block number and can be included rather easily into the smart contract. Even though the randomness of this method seems to be quite high, it relies on the miners and thus could be manipulated by those. It represents the least level of randomness of these methods but it might still be enough for our system. Further research could elaborate on this. Initiatives for Collaboration Pseudo-Random Number Generators have been launched. They promise a higher level of randomness (compared to block hash based) plus no dependency on a (more or less) trusted third party (as with an Oracle). Pivotal for their success is the participation of a plethora of users, making the outcome increasingly random with a higher number of participants. These services could be used in our smart contract but appear rather unnecessary for our purpose. Oracle-based approaches (as with Oraclize) provide real randomness to the system while practically trusting the Oracle provider (in theory the proof could be run on-chain, but this is unreasonably expensive). Again, we consider this to be an overkill for our project. In case the randomness achievable with Block hash PRNG does not suffice, the other two options might become relevant.

4 Off-chaining Approach

4.1 Architecture

Vincent Jonany

This section aims to give a broad view of the components and how all of them work together to satisfy the needs of our prototype.

As seen in Figure 4.1, we have divided our architecture into three sides:

- Client side
- Database
- Blockchain environment

Client Side The client side consists of the client side application, and the Ethereum node. The client side application is the biggest component as it bridges the database and the smart contract in the Ethereum node. Currently, we put trust into the client side to a certain extent. The extent of trust varies depending on the use case, but nevertheless, a certain level of trust has to exist. For example, upon insertion of new data, before any data goes into the smart contract to be processed, we trust that the client side application will not alter the values. This assumption also acts as a temporary solution to the problem which we have encountered in performing heavy computations inside the smart contract. This assumption has allowed us to take the computational burden from smart contract to the client side application. This problem and solution will be explained in more details in section 4.2.1.

Blockchain Environment The blockchain environment is the decentralized network where the smart contract and its local data are going to ultimately live in. The blockchain decentralized network is trustless. But what it truly means is that the trust is distributed amongst all nodes in the network. It also means that we do not enforce an external institution to make sure that the smart contract and its local data are accurate and consistent (integrity). The environment itself ensures the integrity of the data.

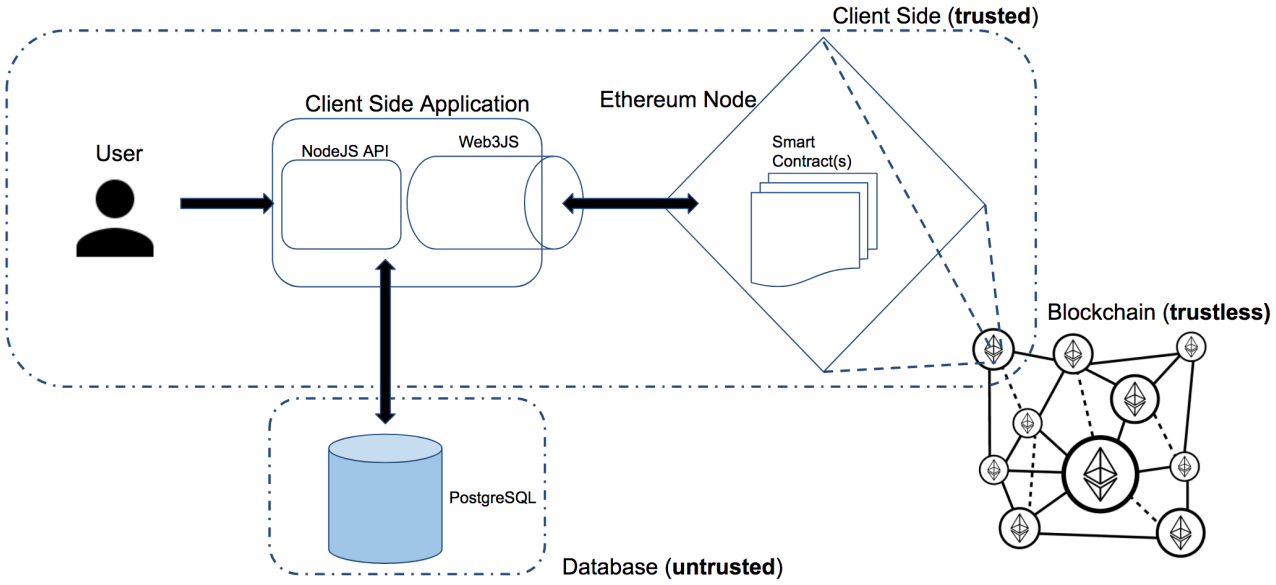


Figure 4.1: The architecture of the off-chaining approach.

Database The database however, is not trusted. Though in our approach we use the database to store data that is going to be used again in the smart contract, we cannot trust the database. The database is prone to internal attacks that can affect the integrity of the data. But at the same time, a database allows us to store large amount of data, and it can be easily integrated with other applications or systems, highly suitable for an off-chaining approach. Hence our approach includes leveraging data integrity check mechanisms when using off-chained data from an untrusted source, such as the database, in the smart contract.

Transportation Layer We have not only made the assumption that our client side is trusted, but also that the transportation layer is secured. Prior to the assumption, we have thought of attacks such as the man-in-the-middle attack, and how detrimental this attack is when we want to save raw data to the database, or when communicating with the smart contract. For example, it could happen that the hashes created in the client side application are altered upon sending them to the smart contract during the initial step. The first step of the approach includes the client side application hashing the raw data and sending the hashes to the smart contract to be stored, mapping the off-chained data to their hashes. Hence, if an attack changes a hash to a different data's hash, then someone may be able to pass the integrity check by reusing that altered hash's raw data value.

Application Flow There are two most basic and general approaches in which the user interacts with our client side application.

- Inserting a new data
- Performing a specific action with that data

In most general cases, the flow of our application when a user wants to insert a new data goes in the following way:

1. User posts a request to client side application with all the data that are required in the specified data model.
2. Client side application creates a Merkle tree using the data sent.
3. Client side application sends the Merkle root hash to the the smart contract.
4. Smart contract stores the root hash as a local variable.
5. Smart contract fires an Event to the client side application to let it know if it has successfully stored it.
6. Client side application performs a database query to store the data and the root hash into the database.
7. Database stores it.
8. Return success message to the user.

In most general cases, the flow of our application when a user wants to perform a specific task through the smart contract while using the off-chained data goes in the following way:

1. User creates a request to the client side application to perform a specific smart contract function.
2. Client side application triggers relevant smart contract function.
3. Smart contract triggers a smart contract event to the client side application to retrieve specific data from database. Specific data can be specified by using the root hash stored in smart contract previously.
4. Client side application listens to that event and queries required data from the database.
5. Client side application creates a Merkle tree from the queried data, and creates a Merkle proof.

6. Client side application sends required data and proof to smart contract via function call.
7. Smart contract performs integrity check using the proof from client side application.
8. Smart contract continues with the original intended task requested by the user when the proof has passed the integrity check.
9. Smart contract computes a new root hash from the new changed value, and stores it.
10. Smart contract triggers an event back to the client side application containing either the results, or an error when the proof does not satisfy the condition of the integrity check.
11. Client side application listens to event, and then stores the new root hash with the new changed value into the database.
12. Client side application finally returns either a success message, or a failure in case of a failed data integrity check.

4.2 Implementation

In this section we aim to showcase all of the design decisions that we, as a team, have made in the implementation. The goal of this section is to also provide the implementation details of the steps in our application flow. We will also include the technologies that have also helped us achieve our goals in the current implementation. We hope that the readers will have a good understanding of the mechanisms, techniques and technologies we used to implement our off-chaining approach.

4.2.1 Client-side Application

Vincent Jonany, Dukagjin Ramosa, Tarek Higazij

Technologies

We implemented our client side application using Node.js for the backend, and Express as our framework. We chose Node.js in our application because Node.js utilizes an event driven, non-blocking I/O model which helps with our integration with the smart contracts. Interactions with Blockchain technology, in this case smart contracts, result in long waiting times as transactions have to be mined. With the asynchronous property of Node.js, we can use this time to asynchronously perform other functions in an effective manner. Moreover, Node.js allows us to easily use and integrate libraries to interact with smart contracts, such as Web3.js.

Web3.js contains specific functionality for the ethereum environment and enables us to interact with a local or remote Ethereum node, utilizing an HTTP or IPC connection. These technologies have been proven to be the standard combination and have been used together in many developer communities.

In addition to Web3.js, we also use and integrate Sequelize, a promised-based Object-Relational Mapping library that allows us to easily create transactions with our database.

Hashing

In the course of our implementation of the Merkle Tree, we have encountered many blockers, and one of the bigger blockers is the implementation of the hash function in the client side application. Since the beginning, we have decided that we are going to have the same (or similar) function in our client side application as the one which the smart contract uses: SHA-3 hash function. We agreed on using SHA-3 (keccak), because it is the cheapest hash function available in a Solidity smart contract.

Initially, we used the native default SHA-3 (keccak) hash function provided by the Web3.js library. However, we quickly encountered a major blocker. We first used simple strings to create a proof-of-concept. While the results of the hash function on both platforms, Javascript and Solidity, are equal, the representations are different: Solidity returns the hash result in a hex string representation ‘0x’, and follows 32 bytes of the hahs results, whereas Web3.js returns the result without ‘0x’. This gets more complicated as Merkle Tree requires hashing a hash.

Solidity's SHA-3 function automatically removes the first two characters '0x' before hashing it, whereas in Web3.js we had to manually slice those characters out, and had to specify that the input has a hex encoding.

The other huge blocker which we encountered includes the inconsistency of the data variables in Javascript and Solidity. Our next proof-of-concept includes incrementing a counter value in the smart contract, and that counter value is kept off-chained. Immediately, we received two different hash results from hashing the counter value, an integer, from Web3.js and Solidity. The problem is that when we pass in a number literal in Solidity's SHA-3 function, it will convert the integer to the smallest possible integer type, in this case, a literal value of 1 will be considered as uint8. Hence, the solution requires us to keep the bytes representation of the input consistent. For example, we had to left-pad the byte representation of the integer by 2 hex characters, which calculates to 8 bytes, to match the representation of uint8 in Solidity.

The solutions mentioned are not the best, as our implementation becomes very complex and dirty; we have to pad 64 hex characters to the left of the input in order to match the representation of the uint256 variable in Solidity before hashing it using Web3.js. Fortunately, we found a third party library which provides a SHA-3 (keccak) function that matches the characteristics of the function in Solidity. <https://web3js.readthedocs.io/en/1.0/web3-utils.html> It allows us to state the exact Solidity data type that we want to hash it as, and it also takes care of the appending and the removal of the extra '0x' hex characters of the hash results.

Revert

The client side application is also responsible for reverting the previous smart contract transaction in case of any error when updating the results into the database, such as a new computed root hash from the smart contract. In case of error, the client side application will create a new transaction for the smart contract and revert it to the previous stored root hash.

Gas Cost Problem

As mentioned in the Architecture section, we have taken the computational burden of creating the initial Merkle Tree in the client side application. This initial Merkle tree creation process

requires all of the data - the leaf nodes - to be present. Hence, we were blocked by the gas cost limit error when we were doing all of this inside the smart contact. The gas cost overhead does not happen during the creation of the tree, rather it happens when we send all of the leaves as parameters in the smart contract transaction.

We do try to keep as many of the responsibilities inside of the smart contracts as possible to avoid any damage to the external trust and to reduce dependencies on the client side application. However, in the end, with our limited resources and the urgent need to continue on with our implementation and testing, we decided to move the initial creation of the Merkle tree to the client side application. Possible solutions to this problem are discussed in the chapter 7.

4.2.2 Smart Contracts

Vincent Jonany, Dukagjin Ramosaj

Technologies

Our implementation of smart contracts in the Blockchain was done through the Solidity programming language. Solidity being one of the most used contract-oriented high-level programming languages for executing smart contracts. Solidity compiles smart contracts to bytecode which is executable on the Ethereum Virtual Machine (EVM).

As our development framework in Blockchain, we decided on using Truffle, since its the most popular development framework for Ethereum. Through Truffle we were able to handle the process of compilation, linking, deployment and binary management of smart contracts. In the scope of this project, we decided to start our implementation with the Truffle framework. One of the main reasons was due to the advantage Truffle has in terms of wider and more active community support and the availability of online resources, more than any other Ethereum development framework. Moreover, the possibility of Truffle integration with Node.js became an important and crucial advantage for our implementation due to our team's advanced skills on JavaScript programming [35].

In order to test our smart contracts on a local blockchain network, we used Ganache. We used Ganache since it provides us with a possibility of simulating the blockchain network, allowing us to make Remote Procedure Calls(RPC) calls in it without the need of actually mining the blocks. Moreover, Ganache provides a nice user interface in which developers can see the details

of all the transactions, mining status, costs, and other things to aid them during development and debugging.

Responsibilities

In addition to performing data integrity checks, and performing the tasks they are programmed to do, it is also the responsibility of the smart contracts to create the new root hash when any data was changed. For example, when the smart contract has finished incrementing a counter value, it is the responsibility of the smart contract to create the new root hash. It can create the new root hash using the proof given for the integrity check (also see section 4.2.3) and using the new counter value's hash value.

We are able to keep this responsibility in the smart contracts because sending Merkle proofs is not expensive, as it is only sending a fraction of nodes from the whole tree. Recreating the Merkle tree in the integrity check, and creating a new root hash is also not computationally expensive.

4.2.3 Integrity Check

Simon Fallnich

To verify the integrity of the data, which is sent from the client-side application to the smart contract, we utilize Merkle trees (see subsection 2.3.2), where each data block of a Merkle tree corresponds to a state variable of the smart contract. Although the computations for constructing a Merkle tree are straightforward, the order, in which the given data blocks and hashes are concatenated, is crucial. Therefore, information about the position of the given data blocks and hashes in the resulting Merkle tree needs to be sent to the smart contract as well. In the following, we will summarize all data other than the root hash, which is required to verify the integrity of one or more given data blocks, under the term *proof*.

For the case of verifying one data block, a simple proof could consist of the data block, an array which contains the required hashes in the same order in which they are used in the computations, and an array of booleans of the same length, in which each boolean defines whether to concatenate a specific hash from the left or from the right side. Such a proof can be generated and verified without great effort and implementations are available. Nevertheless, this principle cannot be extended to support multiple data blocks in a single proof — with this approach, each

data block requires its own proof, which dramatically increases the overhead added by the integrity check mechanism. To minimize the overhead, we implemented an integrity check which enables the verification of multiple data blocks within a single proof, referred to as *multiple item proof*.

Proof Generation

To generate a multiple item proof, the client-side application first constructs a Merkle tree of *all* existing data blocks. For this, an array is created, which represents the nodes of a full binary tree whose number of leaf nodes equal the number of all data blocks, stored in level order.[1] The hashes of the data blocks are inserted into the array and the remaining hashes are computed recursively to construct a Merkle tree. Figure 4.2 visualizes the relation between the resulting array and the respresented Merkle tree for an exemplary proof with a total of four data blocks, D_0, D_1, D_2 and D_3.

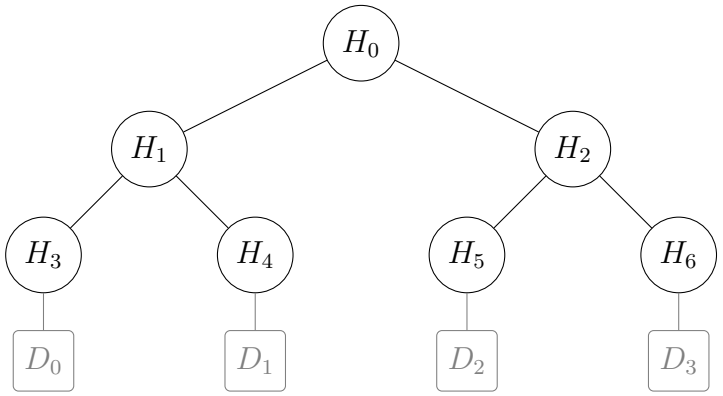
In the next step, a new array of booleans with the same length is created, in which each boolean represents whether the hash of the corresponding node has to be computed during the proof verification or not. The hashes which need to be computed are, on the one hand, the hashes of the data blocks (leaf nodes) which should be verified. On the other hand, the hashes of all (direct and indirect) parent nodes of these nodes need to be computed as well, as they depend on the previously unknown hashes of the respective data blocks. Figure 4.3 shows the resulting array and the correlation to the represented Merkle tree for the exemplary proof with four data blocks, in which D_1 and D_2 should be verified.

Subsequently, the hashes which are required for the proof are determined. For this, another array with the same structure is created an initialized with null values. The hashes are inserted for every node in the array of booleans which has a "false" value and a parent with a "true" value — to compute the hash of the parent node, the hash of such a node has to be concatenated with the hash of its sibling and therefore is required, but does not need to be computed itself. Figure 4.4 shows the resulting array of hashes for the previously mentioned example proof.

Finally, to complete a multiple item proof, the actual data blocks which should be verified and the index of the first leaf node are included, which is used to determine whether an index refers to

[1] This data structure is sufficient to represent a binary tree as the parent's index of a node with index i can be computed as $\lfloor \frac{(i-1)}{2} \rfloor$ and the indices of the left and right childs can be computed with $2i+1$ and $2i+2$, respectively.

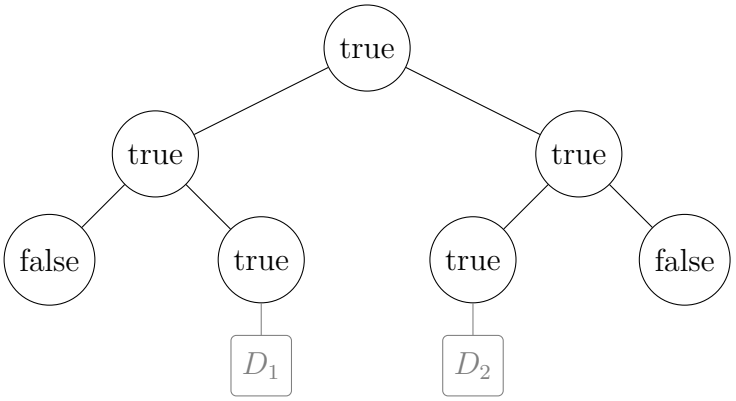
$[H_0, H_1, H_2, H_3, H_4, H_5, H_6]$



(a) Data structure. **(b)** Representation.

Figure 4.2: The array data structure and the corresponding representation of a Merkle tree with four data blocks.

[true, true, true, false, true, true, false]



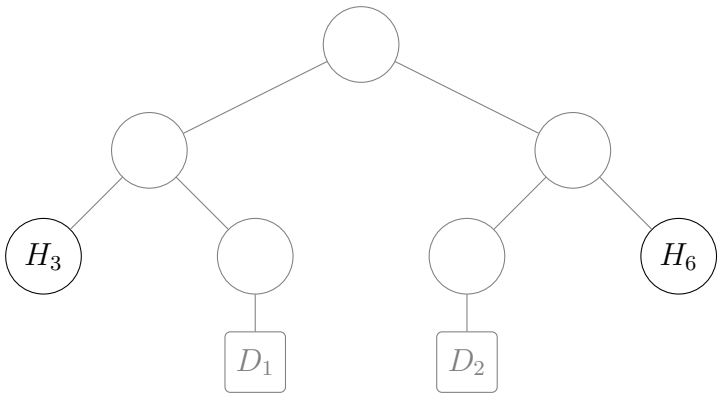
(a) Data structure. **(b)** Representation.

Figure 4.3: The array of booleans and the corresponding representation for a multiple item proof with four data blocks, in which D_1 and D_2 should be verified.

a leaf node in the proof verification process. The complete example proof with four data blocks, in which D_1 and D_2 should be verified, consist of the following:

- Array of booleans: [true, true, true, false, true, true, false]
- Array of hashes: [null, null, null, H_3, null, null, H_6]
- Data blocks: $\{D_1, D_2\}$
- Index of first leaf: 3.

$[\text{null}, \text{null}, \text{null}, H_3, \text{null}, \text{null}, H_6]$



(a) Data structure. **(b)** Representation.

Figure 4.4: The array of hashes and the corresponding representation for a multiple item proof with four data blocks, in which D_1 and D_2 should be verified.

Proof Verification

To verify a proof, the root hash for the respective Merkle tree has to be computed from the given proof data and compared to the stored root hash (see subsection 2.3.2). Algorithm 1 shows the basic principle of the recursive root hash calculation in an off-chained smart contract. To get the hash for a node with index i, it is first determined using the array of booleans, whether the respective hash needs to be computed or is given in the array of hashes. If the hash is given, this is a base case of the recursion and the hash is simply returned (see line 8). Otherwise, if i is greater or equal to the index of the first leaf node, we know that the node with index i has to be a leaf due to the level order in the underlying data structure (see Figure 4.2). This is also a base case of the recursion and the hash of the corresponding data block is returned (see line 11). In all other cases, the hash needs to be recursively computed from the hashes of its children (see line 15). This principle is used to compute the root hash (see line 2), which is compared with the stored root hash. If the hashes are equal, the integrity of the off-chained data was successfully verified and the respective data blocks can be used in further computations.

Algorithm 1 Proof verification

```
1:  procedure VerifyProof(booleans, hashes, dataBlocks, indexOfFirstLeaf)
2:      rootHash ← ComputeHash(0, booleans, hashes, dataBlocks, indexOfFirstLeaf)
3:      if rootHash = storedRootHash then return true                ▷ Proof is valid
4:      else return false                                            ▷ Proof is invalid
5:
6:  procedure ComputeHash(i, booleans, hashes, dataBlocks, indexOfFirstLeaf)
7:      if booleans[i] = false then                  ▷ Hash does not need to be computed
8:          return hashes[i]
9:      else if i ≥ indexOfFirstLeaf then  ▷ Hash needs to be computed from a data block
10:         dataBlockIndex ← i − indexOfFirstLeaf                    ▷ Index of the data block
11:         return hash(dataBlock with index dataBlockIndex)
12:     else                              ▷ Hash needs to be computed from children's hashes
13:         left ← 2i + 1                                            ▷ Index of the left child
14:         right ← 2i + 2                                           ▷ Index of the right child
15:         return hash(ComputeHash(left, ...) + ComputeHash(right, ...))
```

4.2.4 Database

Vincent Jonany, Dukagjin Ramosaj

In the scope of this project, a relational database is used for storing and retrieving the off-chained data from our client side application. We decided to use PostgreSQL as an open source and highly scalable object-relational database system. [36]. And as mentioned previously, we used a third party library to connect the database with the client side application. The client side application interacts with the database when it needs to store data, such as root hashes, as seen in our application flow. In addition to that, the database also allows the client side application to insert filters and specify the order when querying data.

4.3 Translator

Simon Fallnich

Manually integrating our solution into an existing smart contract which uses state variables (on-chained contract) requires advanced knowledge of the implementation of both the given contract and our integrity check mechanism. This inevitably introduces a barrier to potential users as they have to familiarize themselves with implementational details of our solution. Moreover, it prevents applications where knowledge about the implementation of a contract cannot be obtained with reasonable effort, e.g., in the case of automatically generated contracts. Since translating an on-chained contract into an off-chained one is a static procedure, we decided to develop a proof-of-concept of a program which automates this process, referred to as *Translator*, to make off-chaining accessible and easy to use.

The functionality of the Translator can be divided into the following steps:

1. Check given contract
2. Parse contract, variables and functions
3. Compute off-chained values
4. Render off-chained values to contract template.

After the validity of a given contract was checked, the contract is parsed to determine the individual state variables and functions. Each state variable is split up into its name, type, size (in the case of an array), and the original descriptor string, which describes the variable in the smart contract. Subsequently, each function is parsed to determine which state variables are used and which are modified. The original arguments and modifiers of the functions are parsed

```
contract Onchain {

  // State variables
  uint exampleInt;
  string exampleString;
  bool exampleBool;

  // Constructor
  function Onchain(...) {...}

  // Contract functions
  function exampleFunction(...) {...}

}
```

(a) On-chain contract.

```
contract Offchained {

  // State variables
  bytes32 rootHash;

  // Events
  event RequestExampleFunctionEvent(...);
  event ExampleFunctionReturnEvent(...);
  event IntegrityCheckFailedEvent(...);

  // Constructor
  function Offchained(...) {...}

  // Off-chained contract functions
  function requestExampleFunction(...) {...}
  function executeExampleFunction(...) {...}

  // Integrity check
  function _verifyIntegrity(...) {...}
  function _computeHash(...) {...}
  function _leftChildIndex(...) {...}
  function _rightChildIndex(...) {...}

}
```

(b) Off-chained contract.

Figure 4.5: Comparison of the structure of an on-chain contract and the corresponding off-chained contract.

as well since they also need to be included in the off-chained version of the respective function. After the contract was decomposed into the individual data structures (contract, variables, and functions), the required values for the off-chained version of the given contract are computed. These values comprise the names and arguments of the respective functions and events of the off-chained contract in several required formats.[2] The resulting values are then rendered into a template to produce the off-chained contract.

Figure 4.5 shows an overview of the structures of an on-chain contract and the corresponding off-chained contract which is produced by the Translator. It is noticeable that the off-chained contract is longer due to the functions added for the integrity check mechanism. Furthermore, the example function is split up into two functions — one to request the required data and the other one to verify the integrity and execute the actual functionality.

[2]These formats include lists of variables *with* the type of each variable, for the use in function/event declarations, and *without* the respective types, for the use in function/event calls.

4.4 Deployment

Simon Fallnich, Vincent Jonany

This section describes the steps required to run and use our application.

Prerequisites

- Docker 17.05 or higher[3]
- npm 5.5.0 or higher[4]
- Git 2.1.0 or higher[5] (optional)
- Postman 5.0.0 or higher[6] (optional)

Docker is a container runtime, which enables platform-independent development and deployment by packing an application's resources, configurations and dependencies to a bundle, referred to as *container*. We chose to use a container engine in general to consistently configure, build and run the different components (client-side application, blockchain, database) of our solution, and Docker in particular as it has a large ecosystem, which provides readily configured and maintained base images for common applications. The package manager npm is used to easily execute deployment commands. Moreover, Git is utilized for version control, which is optional for deployment as it is solely used to obtain the source code. Finally, we employ Postman, a tool for API testing, to provide predefined API calls for optional end-to-end testing of our solution.

Source

To obtain a copy of the source code, simply clone our repository by running

```
$ git clone https://github.com/simonfall/offchainer.git && cd offchainer
```

or, without Git, downloaded the source as a ZIP archive from `https://github.com/simonfall/offchainer/archive/develop.zip` and unpack it.

[3] See `https://docs.docker.com/install/`.
[4] See `https://docs.npmjs.com/getting-started/installing-node`.
[5] See `https://git-scm.com/downloads`.
[6] See `https://www.getpostman.com/apps`.

Core System

Build and run our system with

```
$ npm run staging
```

This command first builds the images for the client-side application, database and local blockchain if they do not exist already and creates containers from the individual images. Furthermore, the output of the client-side application is piped to the current console and the used ports are bound to the host machine so that they can be accessed locally. The client-side application listens over HTTP on port 8000 of the local machine (`http://127.0.0.1:8000`). To exit, simply press `Ctrl` + `C`, which stops the containers and unbinds the used ports.

Predefined API Requests

We provide Postman collections, which contain predefined requests for our use cases.[7] To test a use case, import (File → Import) the desired JSON file from the `postman` directory to the Postman application and send the intended requests.

Benchmarking

To run the benchmarking, execute

```
$ npm run benchmarking
```

Similar to running the core system, this command first builds the required images, creates the respective containers and pipes the output to the current console. Furthermore, it binds parts of the current filesystem to the container's filesystem in order to output benchmarking results. The benchmarking is performed and the resulting statistics are written to CSV files in the `benchmarking-results` directory.[8]

[7] For more details on the use cases see chapter 5.
[8] For more details on benchmarking see chapter 6.

Unit Tests

We provide unit tests, which target the implementation of our integrity check mechanism. To run the unit tests, execute

```
$ npm run testing
```

This builds the required images, creates the containers and binds the output to the current console. The unit tests are performed and a summary of the passed and failed test cases is shown.

Translator

The Translator can be used by running

```
$ npm run translate <path to contract file>
```

An example contract can be translated with

```
$ npm run translate translator/examples/counters.sol
```

These commands build the Translator image, create a container and bind parts of the filesystem to the container's filesystem to output the resulting contract. The translation is performed and the off-chained contract is written to the `translator-output` directory.[9]

[9] For more details on the Translator see section 4.3.

5 Use Cases

Thanh Tuan Tenh Cong, Patrick Friedrich

In companies large amounts of data records are retrieved and modified each day. But what if data records are changed or manipulated in a way that the results of your calculations with the data become wrong and decisions are made based on these results? Guaranteeing the integrity of data records in a distributed environment is a big challenge.

This challenge can be tackled by using blockchain technologies. Storing data in the blockchain can guarantee the integrity of your data at all time. Although it gives you many other advantages, storing data, especially large amounts of data, on the blockchain can be very expensive.

The goal of this project is to find ways to guarantee the integrity of data stored in a RDBMS with the help of smart contracts. The general idea is to perform integrity checks on data in the smart contract, whenever data is required to perform any transaction. To show the practical relevance of this approach, realistic uses cases have to be found and developed. The use cases shall show the potential benefit of storing large amounts of data off-chain while still being able to guarantee the integrity of the data. This is always the case when it comes to financial or personal/contract data.

Throughout the implementation process in this project, we have been continuously coming up with use cases, and going over them again in order to produce more relateable and complex use cases. The idea is to generate simple uses cases first, apply these to the current prototype implementation, test them in terms of performance and efficiency and show ways to further develop the use cases as well as to apply different integrity check mechanisms based on the results of the tests.

In the end, we have come up with two different use cases, produced through our feedback sessions and discussions as a group. The use cases should represent a real case scenario and highlight the need for off-chaining data.

The following points should summarize the properties of our use cases. As mentioned before, the developed uses cases should still be feasible and integratable with the current prototype implementation.

- Multiple Columns: In order to make use of the hash tree algorithm a data table with multiple columns is required.
- One table per Use Case: In the current project stage, the Use Cases should use only one table each. Multiple entities and relations would make the Use Case too complex. In the future, a more complex Use Case with multiple entities and relations could be developed.
- Big Data: In real world examples large numbers of data records are stored and evaluated every day. Our use cases should consider real world examples, where a large amount of data is required. With the current prototype implementation and the posed constraints by the Ethereum network, a large number of data records could hit the gas limit very quickly. Therefore, the number of data records should be limited to a defined number.
- Different Scenarios: For presentation purposes, the two use cases should not only use different data types but also do different things with the acquired data.

5.1 Employee

Thanh Tuan Tenh Cong

5.1.1 Concept

Description In a real world scenario, a company could decide to use the blockchain and the off-chaining approach to verify employee data, which is stored in a private database (RDBMS). Additionally the company wants to make the changes in salaries for each employee as transparent as possible. Together with a union the company can agree on details (e.g. percentage of pay raise, affected departments, etc.) for a pay raise increase and store as well as execute the pay raise on the blockchain.

The general idea behind implementing this use case is to create two separate Smart Contracts: The employee contract and the pay-raise contract. In the employee contract, we need to store the root hashes, which can be used to verify the integrity of employee data records. The details of a pay raise agreed together with a company and a union are stored in a separate contract called the pay raise contract. When performing the pay raise, the employee contract should be addressed which in turn should address the pay raise contract to extract the details for the pay raise.

Example: A Employees Table

RecordID	FirstName	LastName	StartDate	Salary
1	Max	Mustermann	20171230	45500
...	...	...	...	...

As mentioned before, the Employee smart contract stores the root hashes of each employee record. The table above shows an example Employee table with one record. The merkle tree is constructed with the values in 'FirstName', 'LastName', 'EntryDate' and 'Salary' in the given record. The root of the constructed merkle tree is stored in a map with the 'RecordID' as an identifier for the root hash.

5.1.2 Implementation

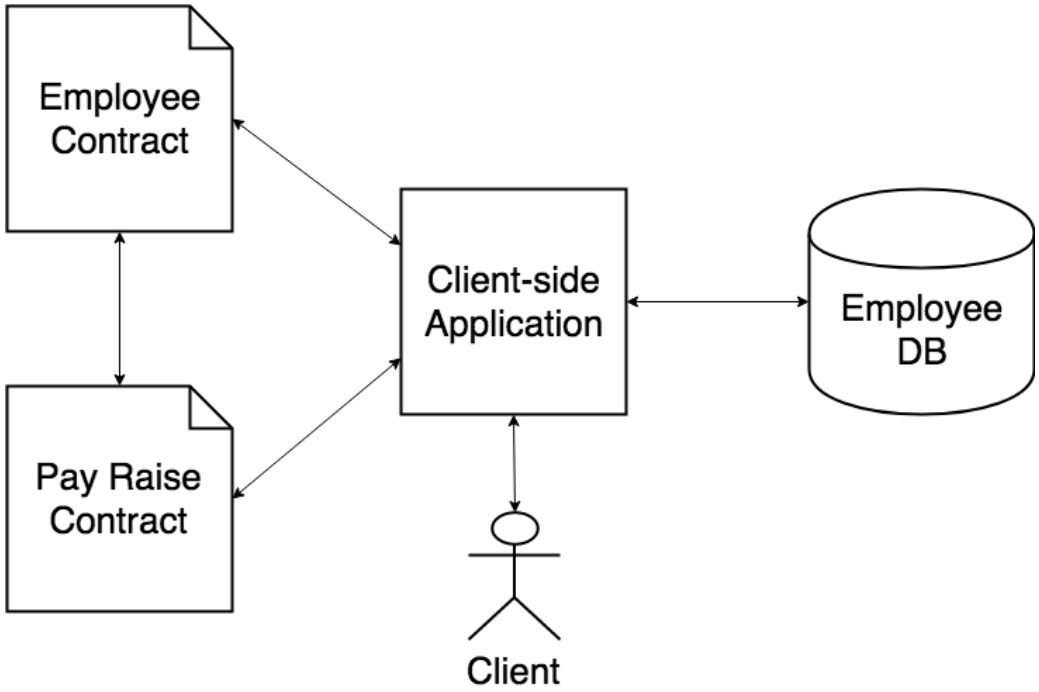


Figure 5.1: Overview of all components of the Employee Use Case

Overview As mentioned in the previous chapter, this use case requires two smart contracts, which should interact with each other. Figure 5.1 shows all the components of this use case and how they interact with each other. In this use case, we have the following components: Client, Client-side Application, Employee DB, Pay Raise Contract and Employee Contract.

The client is basically the user who is interacting with the whole system. They trigger functions to the client-side application in order to perform transactions. Applied to the use case, the client can be either the company or the union.

The client-side application is the main part of this use case. It is connected with the employee database, where the employee data is stored in an employee table. By using the Web3 API it is able to create and interact with the employee contract as well as the pay raise contract.

Both contracts, the employee contract as well as the pay raise contract, can exist on their own. But only one employee contract should be created for this use case. Everytime data is inserted or updated in the employee database, the employee contract is updated accordingly. E.g. when a record is inserted in the database, an additional mapping item with the identifier and the Merkle root of the Merkle tree constructed from this record is inserted into the contract. In case a record is updated, a new Merkle tree has to be constructed in the client-side application and the new Merkle root has to be used to replace the old Merkle root in the equivalent mapping item.

Necessary Steps This subchapter briefly describes the necessary steps for a client to interact with the components in the use case. Also it gives some insights on how the components interact with each other.

1. Create Employee Contract

In order to perform further transactions, an employee contract has to be created. Via REST interface, the client shall call a function to create a contract. The client-side application takes the request and creates the contract in the network as well as a contract instance inside the application with the contract address, to perform further transactions with the employee contract.

2. Create/Import Employee Data

The client can now create data records or use the import function, to create multiple employee data records. The client-side application uses the input of the client to create data records in the employee table in the database. Moreover, it creates a Merkle tree for each new employee record and adds a mapping item with the Merkle root and the "record id" of the employee in the database into the smart contract.

3. Create Pay Raise Contract

Before a pay raise can be given to employees, specific conditions have to be formulated, for example the percentage, the affected department, etc. Those conditions are formulated in the Pay Raise Contract. This contract contains only the formulated and agreed upon conditions between the company and the union, as well as getter functions to retrieve these conditions later from the Employee Contract, where the pay raise is executed.

4. Increase Salary

In the Increase Salary step, the client triggers a function on the client-side application to increase the salary of all affected employees according to the created pay raise contract in step 3. Therefore the client sends a request with the contract address of the pay raise contract. The client-side application calls a function in the employee contract by passing the pay raise contract as input parameter. After that the employee contract retrieves the conditions (e.g. percentage, department, etc.) from the pay raise contract and returns it to the client-side application as an event. The client-side application listens to this event and uses the conditions as query parameters when querying the database. For example, if the department is 'IT', the SQL Query would look like this:

```
SELECT *
FROM Employee
WHERE Department = 'IT'
```

For each single employee in the output of the query, a single smart contract transaction is required to increase the salary of the employee. If for example, the number of employees in the IT department is three, then the whole transaction needs three additional smart contract transactions. These three smart contract transactions are chained, meaning they need to be executed sequentially. The reason for that is because the employee contract will send data back to the client-side application via an event. The design decision was to shut down the event listener in the client-side application after it listened to one event. Triggering three smart contract transactions would result in three different events as a result of each transaction. Triggering all three smart contract transactions at once would mean that the client-side application would only listen to one event and the whole transaction cannot be completed.

As a first step in one smart contract transaction for increasing the salary of one single employee, the client-side application calls a smart contract function and sends the whole data record of the employee in the database table as well as the Merkle proofs of the salary of the single employee

as input parameters to the smart contract. In the smart contract the integrity of the employee data record is checked. The Merkle root can be retrieved from the mapping item with the record identifier and the Merkle root stored in the smart contract. With the Merkle proof and the hash of the salary of the employee, the Merkle root can be constructed. If the constructed Merkle root is the same as the saved Merkle root for the employee record, the integrity check was successful. The salary is then increased by the percentage stored in the pay raise contract. A new Merkle root is constructed out of the new salary and the proofs, which is then stored into the mapping item and sent back as an event to the client-side application together with the new value of the salary. The client-side application takes the new value of the salary of the employee and stores it into to the database. Subsequently, it will continue to create a new smart contract transaction for the next employee pay raise. The whole transaction is finished when all employees in the query result are processed.

In case the integrity check fails, the smart contract will send an event to notify the client-side application about the failure. Even if one integrity check of an employee fails, the other employees are still processed. The end result is that the client will receive as a response a list of all the employees whose salary have been increased.

Discussions In this use case the client-side application needs to trust the database to return the correct results. For example if there are actually five people working in the IT department, the database should return the records of those five employees. In this case we are putting some trust in the database to return all of those five employees. However,the database could be manipulated by an administrator, so that one or more employees from the IT department are missing in the query result. As a result a pay raise would not affect those missing employees. A possible solution would be to store the number of employees for each department in the smart contract and check whether the number of query results matches the number of employees for a specific department stored in the smart contract.

Table 5.1: An Example of a Sales Table

ID	Product	Date	Amount	Price
1	Wallet	20171230	1	20
..	...	....	..	...

5.2 Financials

5.2.1 Concept

Description An external auditor wants to check the financial situation of a company. One task could be to check the stated weekly or monthly sales amount against the sum of all sales records in the specific year. One of the first steps would be to verify (check the integrity) of all sales records in the specific year.

This use case consists of two parties, a financial auditor and the company that holds the financial data. An assumption that we make is that the middleware is open-sourced, produced by an auditing company, and it is being consumed by the company to store their financial data in the blockchain environment. The middleware allows the company to off-chain their financial records from the smart contract into a database, while maintaining the integrity of the data using the blockchain environment. The middleware also allows the auditor to easily audit companies' financial records without having to worry about the integrity of data once they are appended. It also allows the auditor to query the financial records with a filter, such as the date of the records, performing query completeness. The middleware however does not allow users to edit the records through the middleware, though it is possible to do it on the database directly, which will then result in an integrity check error when the records are used or verified by the auditor.

The different table rows represent moments in time (e.g. every Friday night after 00:00 or every first of the month) and thus new records are appended to the table. In this way, the records can be tracked back in time and an auditor could double-check the records for e.g. the last 6 months or the last 104 weeks. The smart contract entry then consists of the root hash over every table row (with each column being a leaf in the Merkle tree).

As the database is under the control of the company storing their financial figures, the financial auditor cannot be sure that those numbers were inserted correctly (same as without blockchain). By having the hashes of the records on the blockchain, the auditor can be sure that the company was not able to change the figures later on and thus has the certainty that there

are no accounting tricks and malpractices in place e.g. at the end of the year or quarter. The figures that were once recorded by the company can be double-checked afterwards (or it can be noticed that the company recorded wrong numbers or tried to change them in hindsight). It is worth mentioning that the smart contract never stores the financial figures and they are thus not available on the blockchain either. While upon insertion, there isn't a way to check that the numbers are true, but we can assert that these numbers will not be prone to illegitimate changes.

Furthermore, this use case could be valuable for rewarding benefits to employees. For example, the CEO of a company could receive a benefit by the shareholders (indirectly through the company itself but signed by the shareholders) if certain financial data are met. Or a sales person in the company could receive a benefit for sales data records for a certain month that went especially well.

To conclude, this use case asserts that large quantities of data (like financial figures of a company) cannot be modified after reporting them while only storing a relatively cheap representation of that data (hash) on the blockchain. Third parties and internal controllers are thus able to rely on the integrity of the recorded data.

5.2.2 Implementation

Initial Concept: Whole Table Verification In this initial approach and concept, we created an assumption that the financial rows are prone to changes, and that the smart contract is going to keep track of the total or the aggregation of the rows. Hence, a verification of the whole table is needed to ultimately maintain the integrity of the "total" row.

The smart contract stores the current root hash of the whole table. It could also store the root hashes of the prior states of the table (this information could otherwise be found in the blockchain history) to allow for additional functionalities.

The smart contract creates a Merkle tree over the hashes of all rows. The hashes of the rows are provided by the middleware. Afterwards, the smart contract verifies that the root hash of the created Merkle tree matches the current stored root hash. If that is the case, it calculates the new total value(s), and adds the hash of the new row (new entry) to the Tree to create the new root hash after these processes. This new root hash is then stored as the current one.

Merkle tree is used here when a column, or multiple column values are changed in a row, and we need to recalculate the new values for the "total" row. Instead of verifying the whole data in a row, we only need to verify the nodes that are affected by the change.

For the Merkle tree implementation, there is no need to implement a new one or change the existing one, as we can use the Multiple Item Proof here (note: an item here will correspond to a row, not a column).

These are the implemented steps for appending a row into the database:

1. First, the client-side application gets all the data from the Database.
2. It then creates the proof and sends it to the smart contract.
3. The smart contract does the integrity check.
4. Then the new total values are calculated.
5. The smart contract creates a root hash for that total row.
6. It then recreates the tree with that new roothash from the total row, and the hash of the newly added item row.
7. Store the new root hash of the whole table.
8. Send back the total row with its root hash, and the root hash of the added row.
9. Client-side application receives it, and stores it into the database.

Realization We quickly realized the "red flags" this approach is producing in regards to the gas cost. In the process of adding a new row, we are adding a new leaf, or a new hash into the current Merkle tree (step 6). This cannot work the way we imagined it, as we are planning on having only the Merkle proof, the nodes that are not affected by the change. Hence, to approach this we always have to send all of the leaves, the hashes of the rows, so that we can create this new tree in addition to a new leaf, the new hash of the inserted row. This quickly creates an overhead in the gas cost when adding a new row into the table. Hence, we took a turn in our approach, and we have to come up with something more usable, and worth it for the users. While the same gas cost problem will still exist in the next approach, it will however provide a much larger incentive for using the function.

Query Completeness The possibility to query RDBMS is a pivotal functionality of those systems and thus represents a great entry point for efforts to link the technologies of blockchain and RDBMS as targeted in this project. The trust that today's users of RDBMS put into the correctness of the returned results for their queries could be nullified by the trustlessness offered by the blockchain - trustless query results in a sense. Currently, users may not be sure if the results were not tampered with or only part of the truth were returned to them.

The general idea is to use the blockchain and its properties to counteract the four ways a database system could falsify the query results. In detail, the following measurements have to be prevented:

- Firstly, the database system could try to not consider all database records while performing the query. This means, a mechanism has to be implemented that verifies that all records were looked at.
- Secondly, the database system could try to add records to the query results that were not part of the database before. To counteract, a trustless system needs to show that all considered records were part of the database already and that the returned results are actual database entries.
- Thirdly, the database system could try to leave out actual database records that fulfill the query in the returned set of records. Accordingly, we need to proof that that all records that fulfill a user's query find their way into the results that the user obtains.
- Finally, the database system could try to include actual database records that do not fulfill the query in the returned set of records. A trustless system thus has to check that only records that fulfill the query are returned to the user.

Any system that successfully counteracts these ways to counterfeit query results achieves query completeness as defined in this paper. An example on how this mechanism is applied to our use case includes a user who wants to query all sales data in the period from Nov 2017 to Jan 2018, and expects that the results have not been tampered with.

Initial Implementation Ideas Our initial approach to implementing query completeness includes, verifying that all entries were considered and that none were left out in the returned results. We can have the database responds to original query and to the negated query (e.g. original: all sales data in the period from Nov 2017 to Jan 2018, negated: all sales data

Table 5.2: A Representation of How the Records Are Stored in the Database

ID	company_name	roothash	total_sales	date	cogs	sc_id	...
1	CompanyAB	0x123	1000	20160523	123	0	..
2	CompanyAB	0x134	2000	20161215	421	1	..
3	CompanyAB	0x321	3000	20170601	222	2	..
4	CompanyAB	0x789	9000	20180130	980	3	..

NOT in the period from Nov 2017 to Jan 2018). And in the end, check if the combined size of both returned lists equals total number of entries (mapping counter in SC). Extending this idea, we can also send the two results, queried results, and the negated results to the smart contract to check if both lists will result into the stored root hash of the whole table.

Alternatively, we can verify that all results from the database match the query through smart contract, by storing all of the required data in the smart contract.

The above statements show that there is a clear trade-off between achieving query completeness and this project's goal of saving gas costs by off-chaining data. Either practically all data has to be stored on-chain to enable our smart contract to guarantee query completeness, or a plethora of computations (Merkle proofs for each row in the data table) have to be on-chained which would hit the gas limit relatively fast.

As we are facing the challenge described previously, we decided to implement our proof-of-concept for query completeness by adding assumptions to make it feasible and easier for testing for large amount of data: building on our initial assumption that we trust our client-side application, we calculate the Merkle proofs on the client side and thus save a considerable amount of computation on-chain. Nevertheless, the smart contract will still have to check the integrity of the data and perform a simplified query to guarantee query completeness. We also only allow the "date" column to be used in the query in this proof-of-concept.

Proof of Concept This section describes the step-by-step actions between our client-side application, smart contract and the database. The proof of concept includes the implementation of appending a new financial row, and query completeness.

Appending A Financial Row Steps Below shows the implemented step by step actions when a user requests to append a new row into the records. Figure 5.2 shows the activity diagram of the steps described below:

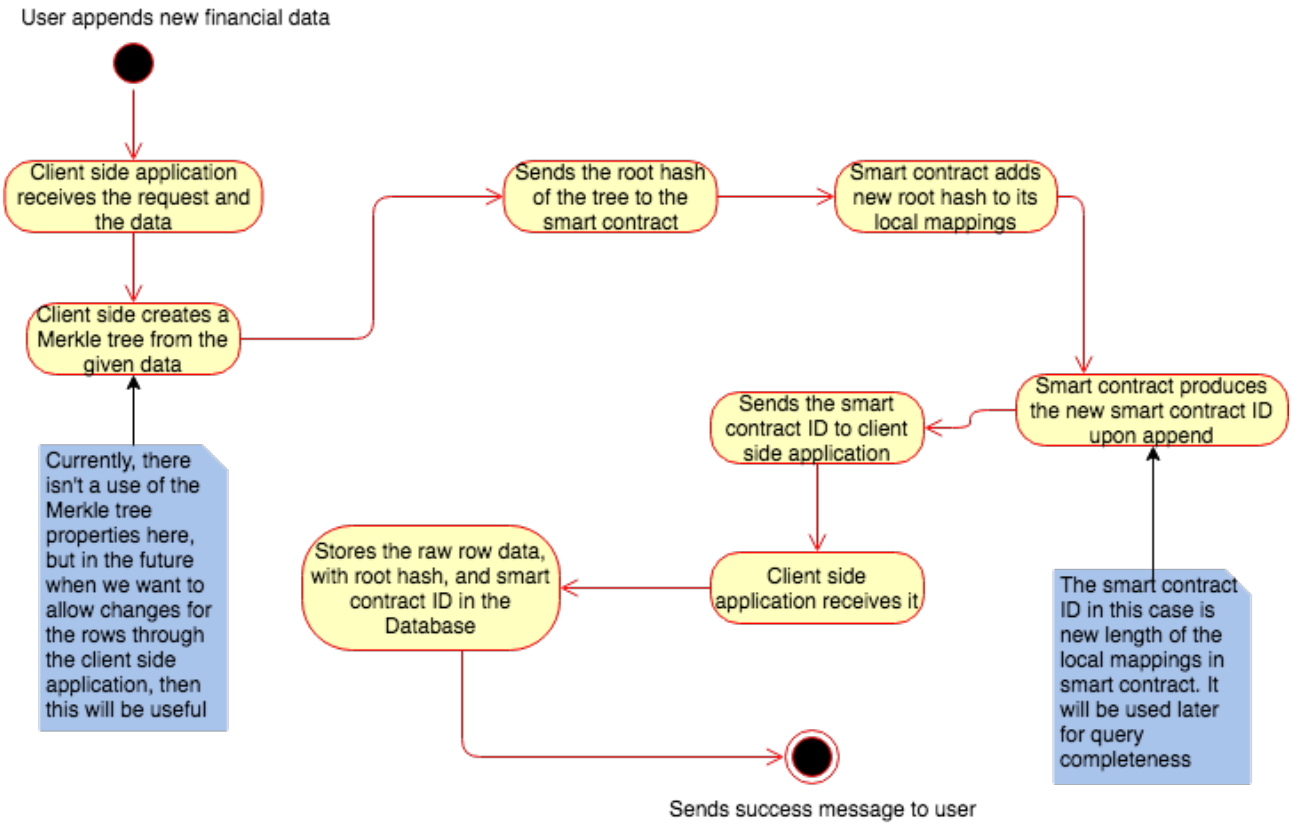


Figure 5.2: Activity diagram of appending a row to the financials record

1. User appends a new financial data, providing all the required information to be in the database
2. Client-side application then creates a Merkle tree from the given raw data
3. Sends the root hash of the created Merkle tree to the smart contract
4. Smart contract appends that new root hash into its local mappings.
5. Smart contract increments the new length of the mapping, which is the smart contract ID that is going to be returned. This acts as an identifier to which hash belongs to which financial row. It is also used to preserve the ordering of the hashes which will later be used for query completeness.
6. Smart contract sends back the smart contract ID to the client-side application
7. Client-side application saves the raw financial data, with the root hash, and the smart contract ID in the database. (See also Table 5.2 to see how records are saved in the database)

Query Completeness Steps Below shows the implemented step by step actions when a user requests to perform query completeness. Figure 5.3 shows the activity diagram of the steps described below:

1. User requests to get all of financial data that are within the date range the user has requested.
2. The client-side application gets all data from the database in the ascending order of the smart contract ID. This ID is generated by the smart contract to keep the ordering of the hashes that are stored in the smart contract. It is generated upon appending a financial record through the smart contract.
3. The client-side application hashes each of the rows in the table.
4. It will then map the hashes of the row and the raw date value to the smart contract ID.
5. The client-side application sends the mapping with the query to the smart contract.
6. The smart contract checks if it got the right data, the right amount of rows and the correct table overall by iterating over the provided root hashes and comparing them to the stored hashes in the local mapping.
    a) If smart contract cannot verify the rows, then it will throw an integrity check error to the client-side application as an event.
    b) If the smart contract can confirm this, it checks the query condition for every row and returns an array of booleans indicating all the indexes of rows that fulfill the query. Dates are intentionally stored as "uint" to ease the query function in the smart contract. For example 1, March, 2018 -> 20180301
7. Smart contract triggers an event to return back all the smart contract IDs that have satisfied the query.
8. The client-side application listens to this event and returns the specified rows to the user subsequently.

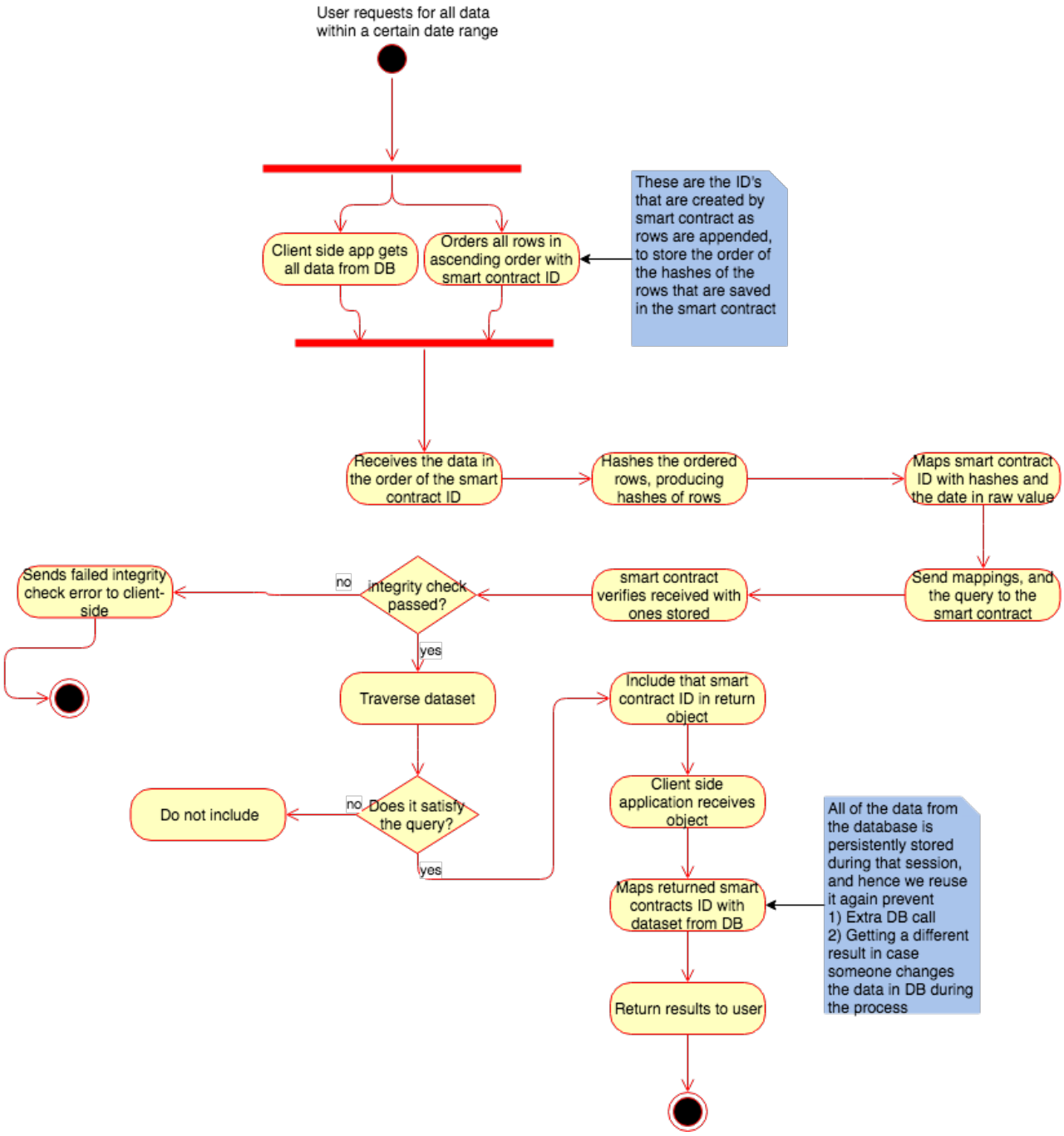


Figure 5.3: Activity diagram for query completeness

6 Evaluation

Kevin Marcel Styp-Rekowski

In this section, we are going to discuss how our evaluation was conducted. We automatized the benchmarking, therefore, our benchmarks are repeatable and easy to execute. The findings we made will also be discussed.

As already mentioned this project was carried out as a prototyping project. Therefore, we need to benchmark our results against the current best practice. Current state of the art for deploying contracts to the Ethereum blockchain includes saving all the state variables of the contract on the blockchain and thus, on all machines participating in the blockchain. As proposed we want to save those state variables off of the blockchain and thus, save on gas which explains why gas costs are our most important measurement.

6.1 Automated Benchmarking

We automated the process of benchmarking within our system. Thus, we achieved a repeatable, fast and reusable way to evaluate our application with various small changes to our contracts.

As our application gets accessed through API routes, this was the most natural point to call when automating our benchmarking. Also, in the JSON response of those API calls we receive information about the gas costs of the transaction and the time needed for the computation that happened within our application.

The automatization was realized using the npm packages “mocca” and “supertest”. These packages represent a test framework for JavaScript, where it is possible to define test sets which contain multiple test cases. Moreover, it enables us to call our express-application through the defined routes.

After extracting the gained insights about our application and the behaviour of our contracts, we save these results into a CSV-file which can later easily be used to generate statistics about our results.

6.2 Benchmarking of Use Cases

We chose to benchmark the employee use case in particular, as the employee use case is the more complex of our two use cases. As a preparation for benchmarking we first needed two different smart contracts. One smart contract that works completely on-chain without using any external database, hashing functions or Merkle trees and another smart contract that uses all of the introduced methods that are needed in order to off-chain our data to an external database. The two variants behave completely similarly, except for the saving of the state variables. Both variants are completely functional and could be used as they are if someone would like to deploy the employee use case. These facts were exceptionally important to us because only this way we could make sure that our benchmarks lead to meaningful results. Within the employee use case we measured different scenarios which will be described in the corresponding sections, mainly a time measurement as well as a simple and a complex measurement for the gas cost have been made.

Gas cost was the initial motivation for off-chaining, as we expected to save on gas costs when we off-chain the state variables. Gas cost translates to real currency as a participant of the blockchain either needs to buy Ether or receive it by mining blocks and as a reward from the network, which is needed to pay for the resulting gas cost of a transaction. Time measurement was done since we wanted to gain insights into how much overhead our application adds to a normal blockchain application. This was possible as our technology stack includes ganache, which is capable of automatically mining new transactions into new blocks instantly, so that we are able to measure only the overhead that our application introduces.

6.2.1 Time Measurement

First we will measure the additional computation time of our application for executing different functions of the smart contract. The results can be seen in Figure 6.1.

As we measured the computation time multiple times, we decided to unite the results into a box plot chart. Thus, we can see how the system behaves most of the times and single outliers can easily be identified. Throughout our measurement we always compare the on-chained approach against the off-chained approach. Every single function got called multiple times and the results have been summed up into the corresponding box plot.

For the *employee contract* creation we can see that the time differs by around 50ms when comparing the on-chained approach to the off-chained approach. The on-chained approach

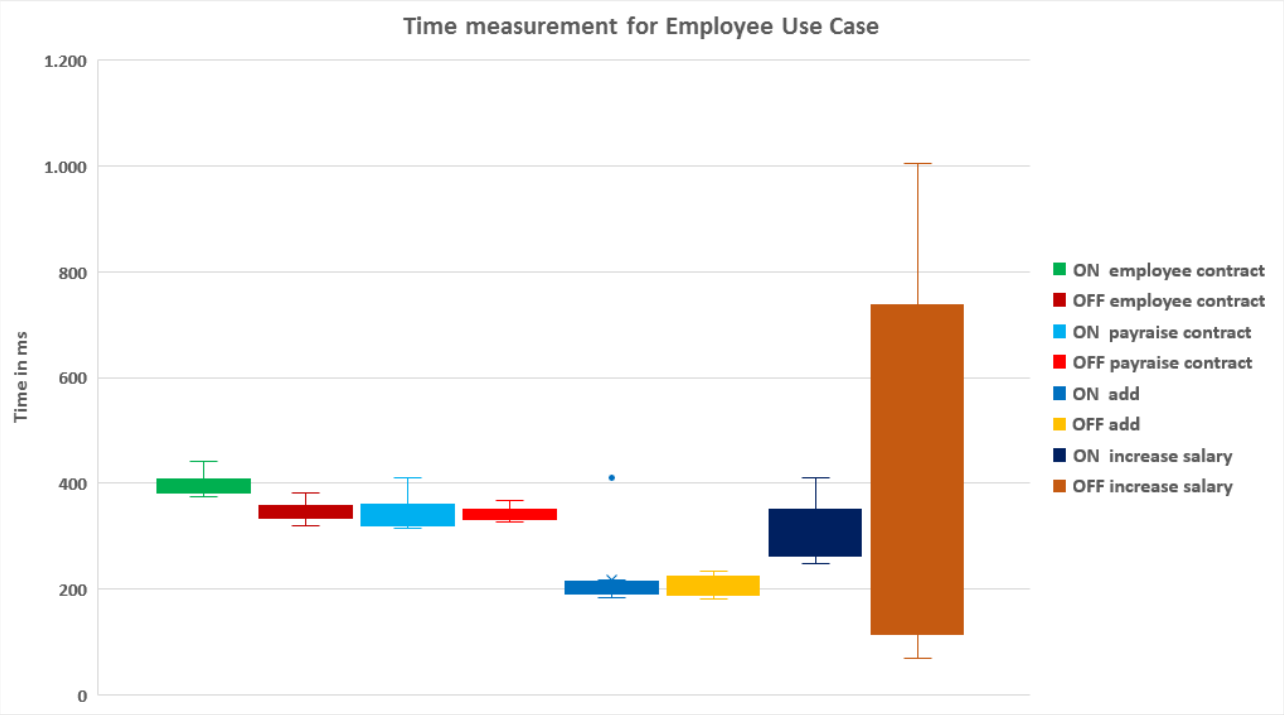


Figure 6.1: Time measurement for Employee Use Case.

needs about 400ms of computation time in the application while the off-chained approach needs 350ms. Both timings are negligible for us, as the average block mining time of the Ethereum blockchain is currently about 14 seconds [37]. Thus, our application has more than enough time to compute a new transaction when weighing nearly half a second against 14 seconds.

The *payraise contract* creation needs about the same time for the on-chained as for the off-chained approach. Both need about 350ms of computation time and are negligible for the same reason as with the *employee contract* creation. The same holds true for the *add* functionality, which takes even less time and thus, the timing is negligible here as well.

More interesting are the timing needs of the *increase salary* functionality. We can easily see that the off-chained variant has a much higher variance than the on-chained variant and we have peeks in the computation time of up to one second. Hereby it is very important to mention that the time needed for the off-chained variant increases with the size of the employee dataset. This happens because for every added employee the *increase salary* function needs to generate another transaction. As every transaction needs to be mined first, the increased timing needs of our application are also negligible, as what we see in our chart is the accumulated computation time for all of the transactions which happened during this function call. All of these transactions need to be mined within blocks of the Ethereum blockchain which takes

a lot longer, namely about 14 seconds currently, than the computation of our application. Plus, it could possibly happen that the single transactions will be divided on multiple blocks. Therefore the introduced timing needs of our application are also negligible for this function and additionally all measurements of the *increase salary* function were less than a second which is quite fast.

As every function call in the off-chained approach does not introduce a real overhead compared to the on-chained approach and our application always performed under one second, we concluded that we do not need to deeper analyze what causes the most computation time or how to optimize for time in our application, as this would not result in a great benefit for our use case. Therefore, no further measurements concerning the computation time have been made. We can conclude that the off-chained approach did not introduce any additional computation time and is as good as the on-chained approach timewise.

6.2.2 Gas Cost Measurement

Measuring the gas needed for executing the functionalities of our smart contract is of utmost importance to us. The initial motivation for our prototype was to gain insight into whether it is possible to save on gas costs, when off-chaining state variables from smart contracts. As we receive the used gas for each transaction within the JSON-response when we call the smart contract functions through our API endpoint, we were able to gain different insights into the use of gas when on- or off-chaining.

In Figure 6.2 we visualized the gas used for calling single functions of our smart contract for the employee use case. Similarly to our time measurement, we again measured the main functionalities of our smart contract and compared the on-chained variant with the off-chained variant. The results of our analysis will be discussed below.

For the *employee contract* creation we can see that the off-chained variant needs about 130 thousand more gas than the on-chained variant. Both smart contracts do not save any state variables apart from empty lists for either hash values or employees, depending on the approach. The difference in the gas costs originates mainly from the added functions for the Merkle tree proof and verification which were added to the smart contract for the off-chained approach. As a result, a new task for future work could be identified. Extracting the functions which are needed in order to iterate through the tree and verify the Merkle proof into a library in the sense of Solidity would benefit the off-chaining approach. Calling functions of libraries in

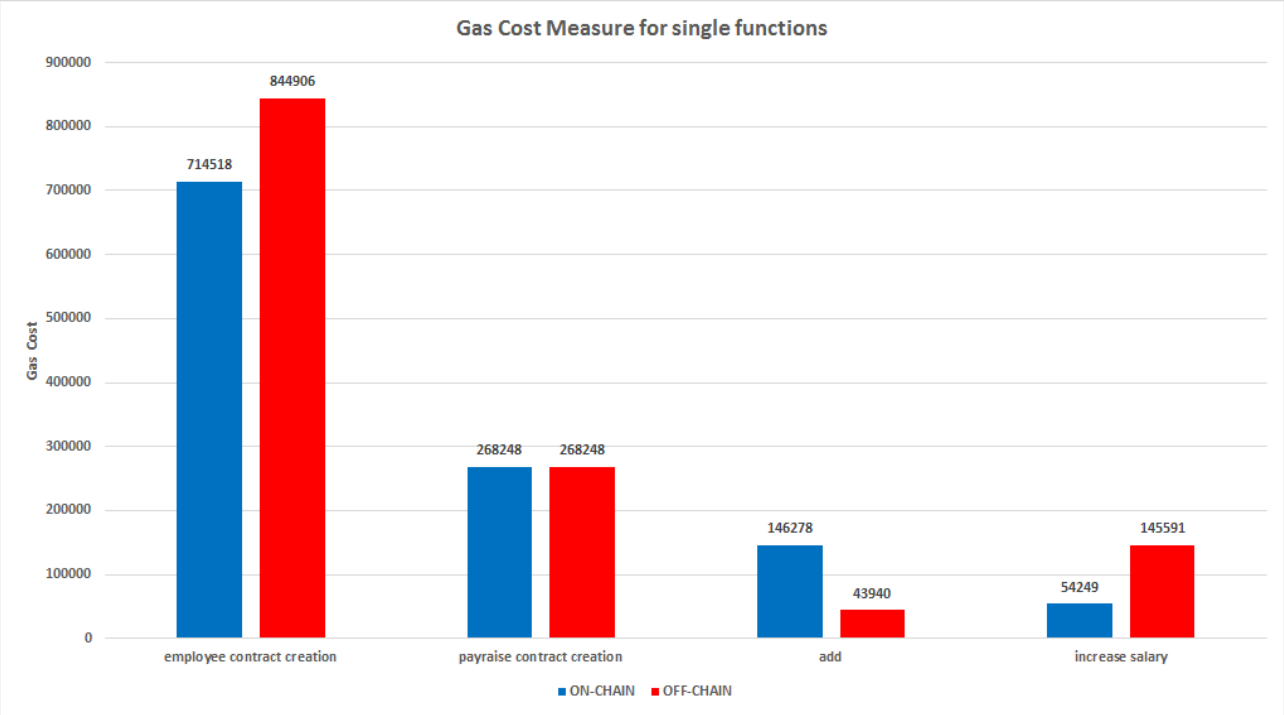


Figure 6.2: This is one employee.

Solidity is in general very beneficial in terms of gas cost. Anyways the gas costs of the off-chained approach exceed the gas costs of the on-chained approach only by approximately 18% which is rather minor. All in all, it can be said that the increased gas costs of the off-chained approach are negligible as the contract creation needs to be paid for only once throughout the whole life cycle of the smart contract. Analyzing the offered functions of the smart contract is much more important as these will be called multiple times.

The gas costs for the creation of the *payraise contract* do not differ from each other as both approaches use the same implementation for the payraise contract. The payraise contract holds very little information in its state variables which are basically information about the department for which the payraise is relevant and a percentage value for the payraise. Off-chaining this very small contract would not have made sense and thus, it was kept as an on-chain approach because this contract is not very data intense.

One of the most interesting functions of the smart contract is the *add* functionality. This is one of the main features of the employee contract, which we expected to yield the biggest opportunity to save on gas costs. Our measurements confirmed our expectations, as we save approximately 102 thousand gas on each call of the *add* functionality, which translates to a saving of about 70%. An important thing to mention at this point is that the gas costs of 43940 gas for the off-chained *add* functionality are constant because that is the gas cost for

saving one root hash in a smart contract. So, independently of how big the employee data item looks like, so independently whether it uses very long or short names, we will always only need to pay the constant 43940 gas for storing the hash of this record. This would also hold true if there were multiple new variable fields added to employee records in general. Comparing the savings of the *add* functionality to the introduced overhead when saving employee contracts, where we needed to pay about 130 thousand gas more makes the constant overhead look very small because by adding only one employee we nearly balanced this overhead out. Bigger savings could be achieved by adding multiple employees which represents a normal use case.

Another very important function is the *increase salary* functionality of the employee contract. We can see that the gas cost for increasing the salary of a single employee increases by approximately 91 thousand gas for the off-chained approach. This happens because in the case of raising salary only one integer field of an employee gets changed which is a very small variable type. The on-chained variant only needs to iterate through its locally saved employees and increase the salary for a given employee, in our case the list would only contain one employee. Raising the salary translates into saving a new integer variable as a state variable of the smart contract which in comparison to the off-chain approach is rather cheap in terms of gas costs. The off-chained approach however sends the integer field for the salary of the employee together with a complete Merkle proof for verifying the integrity of this salary field. The smart contract would then need to start its computations by verifying the integrity of the received employee, respectively the salary of the employee. The overhead introduced by this behavior explains the risen gas costs for the off-chained approach.

Introducing more functionalities would make the evaluation even more complex. One could think of a new function that would modify multiple string fields of the employee, which would create the need to pay for the newly saved string fields. In such a similarly constructed function like the *increase salary* functionality the off-chained approach would not perform worse than the on-chained approach. The on-chained approach would have to pay high gas costs for the saving of multiple string variables into its state, while the off-chained approach would only have the same overhead of the sent Merkle proof and verification.

As we identified that there exist circumstances in which the off-chained approach performs better and circumstances in which the state of the art on-chained approach performs better we decided to benchmark a more complex scenario of the employee use case. As the gas costs for the creation of the contracts are constant, we decided to focus on the functionalities which can be called multiple times and can have varying results. We introduce a scenario where we add

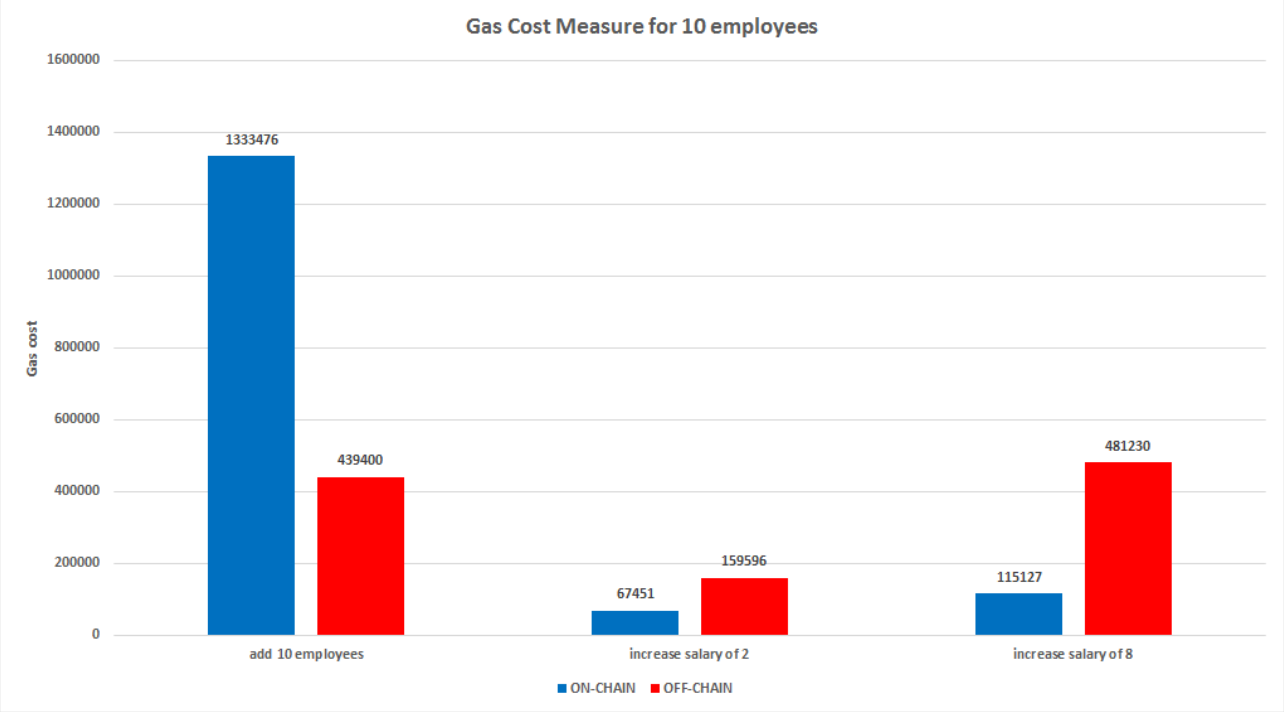


Figure 6.3: This is ten employees.

10 different employees with varying data which are divided up into two different departments. 8 of those employees work in the 'IT' department while the other 2 employees work in the 'Marketing and Management' department. This was introduced so that we can benchmark how the *increase salary* function behaves when it gets executed on the majority of the employees or respectively on the minority. The results for the cumulative gas costs of this benchmark are visualized in Figure 6.3.

When adding 10 employees to the employee contract, we can see that the overall saving on gas was approximately 894000 gas. Also, we see that the off-chained approach needed exactly 439400 gas which is exactly ten times the constant gas cost for adding a single root hash to the smart contract, which aligns with our thoughts on that. The cumulative gas costs for the on-chained approach are put together by slightly differing gas costs for each add call as we used different data for the added employees so the single add values would fluctuate.

Most interesting for us is the finding that the *increase salary* function behaves differently when calling it on the majority or respectively on the minority of the employees. If we look at the percentage-wise increase of the gas cost for the off-chained variant, a huge difference becomes apparent.

$$159596/67451 \approx 237\%$$

$$481230/115127 \approx 418\%$$

Here we can clearly see that when we update the majority of the employees, the overhead of the off-chained variant becomes greater. This has mainly two reasons. The first reason lays in the behavior of the on-chained approach, because when calling the *increase salary* function it iterates over all of the saved employees in order to find the employees of the requested department. This iteration is more costy when only few of the employees get matched, so the on-chained approach is weaker in terms of performance when updating the minority. The second reason lays in the behavior of the off-chained approach. The off-chained approach can easily query the departments in the RDBMS without using blockchain computation power and afterwards send only the needed employees to the smart contract and thus save when updating the minority. On the other hand, with every sent employee the overhead for the Merkle proof and verification gets added as well as the cost for creating a transaction to the blockchain, which leads to a bigger increase when updating the majority of the employees.

As mentioned before, there are functions in which the off-chained approach performs better and other functions where it is less efficient. The most important thing when deciding whether to off-chain an existing contract is to know the use case very well, especially the ratio of how often the functions are called in comparison to each other. In our use case we can assume that employees will get added more often than the salary will be increased, this is why off-chaining makes sense here. Also, it is important to mention that when off-chaining there is no possibility to decide whether to off-chain the add method but keep the *increase salary* function on-chain because each approach needs the data at a certain place. So, either all functions get off-chained or none. Thus, one has to carefully decide whether the savings on the one side will outweigh the overhead on the other side. There are plenty of use cases for which this is true.

In general, we can say that off-chaining makes sense especially when smart contracts use mainly large state variables and functions which change large or many variables in the state of the smart contract. For smaller contracts which do not save a lot of variables, or are rather computationally intense, off-chaining does not make sense as it would introduce an unnecessary overhead. We can see an example for that in the payraise contract. Since this contract was so small we did not off-chain it as this would have introduced a bigger overhead due to the Merkle proof and verification.

Comparing the measurements for the employee use case to the financial use case, we can say that the results would probably have been very similar as the financials use case has a similar structure when compared to the employee use case. In the financials use case we

save a lot of financial data so we would probably also save on the *add* functionality, but the verification of the financial records would be conducted offline and not on the blockchain. This is why an evaluation of the financials use case would look very similar to the evaluation of the employee use case with the main difference of not gaining the insights we gained when analyzing the *increase salary* function as the financials use case has no equivalent for this function.

7 Future Work

Patrick Friedrich, Tarek Higazi, Vincent Jonany, Dukagjin Ramosaj

In this section, we provide possible future steps that show how to continue and further increase the value and advantages of the off-chaining approach. During the course of the project, we have identified several relevant additional ideas to extend our implemented off-chaining system that we present one after the other subsequently. We have ordered the list according to what we believe will give the highest value.

Smart Translator The current implementation of the "Translator" is at its proof-of-concept stage. However we believe that extending the development of the translator can lead to an even more useful and an interesting feature to have. A smarter translator can make a decision on whether off-chaining makes sense or not for a given smart contract. In addition to translating the original smart contract to its off-chained variant, a future implementation of the translator may also include the creationg of the client-side application and database components that would be necessary for the overall system.

Implemented Smart Contracts Offered as a library The functionalities of our developed off-chaining solution could be included in Ethereum as a library living on the blockchain, allowing developers to use them in their smart contract. The library functions could for example include our implemented integrity check mechanisms (e.g. multi-row and multi-column Merkle Tree, compatible with Web3.js) and off-chaining concept (with e.g. events, also general setup for avoiding gas costs through storage). We hope that by paying the constant gas cost to publish these functions developers can save when they create their contracts on the blockchain by simply using the already existing functions of the library, thus effectively lowering their used gas.

In addition to (or possibly instead of) the library we could provide the smart Contract as an entity living on the Ethereum blockchain that developers could have their Smart Contracts inherit from and thus include the functions and structure of our implemented off-chaining solution.

Make application listen on events publicly and offer contract functions publicly (Multi-User) In the current implementation, only the creator of the smart contract can trigger its functions and make use of the off-chained data. As the smart Contract is living on a public blockchain, it could be beneficial to offer the functionalities of one off-chained Smart Contract to the network though and leverage its power in terms of potential applications. There could be contracts that react if called by any network participant or only by an entitled one. Depending on the access rights and the new ways to interact with our system the application scope of the implemented system is broadened widely and could e.g. be used by several team members, all departments in a company or across a collection of organizations within an industry with their respective Ethereum accounts.

Make Middleware Trustless By deploying the now trusted locally hosted middleware as a dapp we could make it trustless (as the blockchain). It could e.g. be hosted on a decentralized database like IPFS and listen to the events fired by an off-chained Smart Contract. This could enable further applications domains and allow certain user groups to build more trust in our solution. For example, a financial auditor that knows that the middleware used in the Financials use case is hosted in a trustless environment is assured that the company was not able to apply any accounting tricks based on the code in the middleware. Also, this approach could enable us to bring the implemented solution into the hands of more users and offer it as a service to them.

Derived Ideas from Research on Oraclize Could our system be seen as a personal data Oracle? On request, it retrieves the wanted data from outside the blockchain and pushes it on-chain. Not from websites though but from a (Relational) Database (that is either trusted or can be double-checked by the trusted middleware) and through the locally running middleware (not a Third party service).

Could our system function as an Outbound Oracle (or represent a part of it) [8]? An example could be a Smart Contract by a company with one of its suppliers that lives on Ethereum. The supplier claims he has delivered the goods to the company by calling the function "requestPaymentForDeliveredGoods" in the smart Contract. The smart Contract fires an event to get the invoice data from the company's database. The company's middleware gets the data and pushes it into the smart Contract. The smart Contract verifies this data (e.g. a contract about the delivered goods) and fires another event. This event could be used by the company's accounting software to trigger the money transfer to the supplier.

Oracle queries can be encrypted [38]. Could on-chain encryption of the data thus behave like an integrity check? The smart Contract stores a magic header, encrypts the data with this magic header (concatenated at the beginning) and the database stores it. When the smart Contract needs the data, it requests it and the database provides the needed data. The smart Contract then decrypts it and controls the magic header (correct decryption, i.e. useful decryption output). The question arises whether Smart Contracts can decrypt and encrypt, that is without anyone being able to decrypt the encrypted output. This is not yet possible, as the pubkey encryption does not really apply to Smart Contracts (only to user accounts). Much research is going on in this direction.

On conflict between data from the database and the stored hash on the blockchain we could issue "Data Discrepancy Forms" to be resolved by the user manually later on.

Randomized Sampling Another idea that we derived from our initial research on integrity check mechanisms is the Randomized Sampling approach. This could be used for query completeness as well as for integrity checks. To reduce the gas costs, a random sample from the query results that the database returns could be drawn and then verified by the blockchain. In contrast, in our proof-of-concept implementation for query completeness every record is looked at on-chain. In a similar fashion, the Randomized Sampling could be used as an integrity check mechanism that trades less security against decreased gas costs. When the smart Contract requests the off-chained data all of that data is currently checked afterwards on-chain (which is costly). Again, a random sample could be drawn from the off-chained data that is handed to the smart Contract and only that random sample is verified. The user of the system could be offered the choice whether she wants a full integrity check performed or a Randomized Sampling.

Encryption-Based Instead of hashing, the data could also be encrypted (i.e. signed by the user) and could thus not be modified in any meaningful way [39]. It could be relatively easy and efficient to perform integrity checks in this way. This could follow a similar implementation structure as with the hashing and Merkle Tree beforehand; first a whole row is encrypted and can thus be checked. Then, a more sophisticated mechanism (like a Merkle Tree for encryption) could be built on top of that. With more research in the field and more experience further ideas beyond the presented and implemented ones could arise and prove themselves as good candidates for integrity check mechanisms.

Mature Query Completeness The full benefit of the Merkle Tree Integrity check mechanism will reveal itself once our implementation supports an even more profound query functionality. Currently, the functions of the smart Contract can be executed and the integrity of the needed data for them is checked but the correct execution of queries to the database can only be verified for the implemented proof-of-concept (queries for date range). The query completeness could be broadened to assured any query by the blockchain as well. This includes the smart Contract capabilities to check that the returned query results (by the database) were not modified, that the returned results match the query predicate, that the records that were not returned do not match the query predicate and that the sum of the two sets (returned and not returned) equals to the right amount of stored records. When it comes to benchmarking, it would be interesting to find a limit for the rows that can be checked in this way. Every number bigger than that one needs to cope with compromises concerning query completeness (e.g. only checking that all rows were considered and are part of the table but no actual value checking) because of the high costs and the off-chaining of data constraint.

Make use of a more optimized database for Merkle Trees Tarek Higazi

One of the steps which were originally thought of was to integrate a specifically optimized database for the storing of our Merkle Tree structures, this in order to save on memory and processing within the flows which make use of the Merkle Tree.

In our current implementation it would not have had much of an effect since we used relatively small Merkle Trees for our use cases and demos, however it is not the most efficient since we re-create the Merkle Tree whenever necessary, rather than storing it somewhere and updating it. Possible optimizations to the way our information is stored in the Merkle Tree are also theoretically possible, for example, regarding how items are stored in the Merkle Tree, dividing our datasets into different size groups and mapping them so that we can access them faster.Another potential improvement could be to make use of a graph database such as the neo4j platform due to its speed and efficiency.

Oraclize Approach to Save Gas During our research about alternative approaches to check the integrity of the off-chained data, we analyzed the Oracle provider Oraclize (see part 3.2) and found an approach that Oraclize uses to save gas costs which could present a further opportunity to extend this project's system as well. By giving its users the choice whether to provide the proof that the data pushed by their Oracle is correct to the smart Contract that requested the data in the first place and that may want to run the integrity check before actually using it or to store the proof for later use on a trustless database (IPFS) Oraclize can

save its users a lot of gas in contrast to running the proof of correct data on-chain every time data is pushed to a Smart Contract. Accordingly, the presented system in this paper could provide its users with the same choice and further improve on its goal to save gas. At the same time, the integrity of the data can still be secured.

Merkle DAG Merkle DAG (Directed Acyclic Graph) is one of the features that we included into our potential roadmap after the mid-term presentation. However, due to the limited time and resources we have, we decided to prioritize other things in the roadmap.

Over the past few years, some papers have been published in regards to extending the Merkle hash technique from just trees but to other data structures [40], such as directed acyclic graphs. In addition to that, another paper has been published in regards to a revised hashing technique [41] for directed acyclic graphs. The paper suggests that the traditional hashing technique is not suitable for a more complex data structure, such as the directed acyclic graph. And lastly, there exists a proof-of-concept [42] and further disucussions for Merkle DAG.

The data structure compatible on the client side application will affect the complexity of the use cases that we can handle. However, at the same time, we also try to not tremendously increase the gas cost. The Merkle DAG increases the complexity of the data structure from a binary tree to a directed acyclic graph. It allows us to have a more complex relationship. For example, we could create a complex Role Based Access management application that leverages the smart Contract to maintain the integrity of the hierarchy status of each role.

We could of course implement another data structure that allows more complex behavior, however as mentioned previously, we want to keep the gas cost reasonable. Hence, the second requirement is that the data structure has to have an efficient integrity check mechanism. As the name suggests, Merkle DAG also implies the similar Merkle proofing mechanism that a Merkle tree has. Hence, an integrity check on the root hash, to make sure that the children have not been changed is efficient as well.

Such complexity and a level of efficiency in the data integrity check this mechanism provides can provide higher values to the users. Hence, incentivizing them to use the off-chaining approach in comparison to the traditional approach.

Potential Additional Features Depending on the data and its format a different database could be chosen (automatically by the middleware), e.g. Relational or NoSQL. Specifically, an optimized database for storing the Merkle Proof could increase performance.

Additional Database access control mechanisms could be included.

The user could specify if and which kind of blockchain guarantees she wishes to have for the data she wants to store (e.g. blockchain needed at all?, which integrity check?). A remote database (in place of the currently local one) can be integrated into the implementation. This could simulate the case that the user does not control the database herself and lead to further refinements of our solution.

The database could sign the records it stored and the user may keep these as proof that the records once existed in the database (proof of existence, in case the database provider denies that the database ever knew about a record). [39]

We could optimize the calculation of proofs by placing variables which are used together as neighbours in the merkle tree. Thus, when calculating the merkle root / merkle proof we save on iterations as we can chop more parts of the tree away. Of course, this optimization has its limitation when certain variables get used together with multiple others in different functions. (This could also be done automatically with an AI)

8 Project Organization

Vincent Jonany

This chapter aims to provide an overview of our internal project organization. Firstly, we are going to cover our software development cycle. We will then provide brief descriptions on the methods and technologies we used to help us maintain the best practices in our software development cycle.

We chose to follow an agile software development methodology and we used SCRUM as our framework. We mainly chose SCRUM due to the nature of our project - it is a highly complex and potentially huge project, and we needed regular feedback from our supervisor. SCRUM encourages short, iterative sprints, weekly meetings, and daily stand-ups. In addition to that, the number of members in the group seems very appropriate for SCRUM. Hence, we made SCRUM the backbone of our project organization.

As we have been given a fixed schedule for the project, we were able to create two iterative cycles, each cycle ending with a presentation of our current findings and results to everyone who takes part in the project. The cycle consists of brainstorming, research, implementation, and a demo.

In addition to our quarterly presentations, every two weeks we demonstrated the current prototype to our product owners, which were the project supervisors. Every week the team members, excluding the previous roles mentioned previously, met twice a week for a general discussion and sprint planning respectively. We chose to have shorter sprints (weekly), because we learned that it was very easy for us to lose track from the project goal. Hence, the weekly meeting helped us to stay on track. Moreover, we also had daily virtual stand-up meetings. This aimed to further prevent each other from going out of the scope, and for everyone to be more aware of what each member was doing at any given point in time, or if they were blocked with the current task, and required immediate reinforcements.

We used several technologies and mechanisms to aid us in following SCRUM with the best practices. One of the most important tools we used is ClickUp. ClickUp is a project management platform that helped us manage and track tasks, stories, epics, and sprints. During our sprint planning, our project manager would create all the necessary elements for the developers to start with their tasks. For example, creating a new sprint board, with all the members' tasks. The developers must then proceed with adding their own descriptions, user stories, and

acceptance criteria into their own tasks. At the same time, the project manager could move and add tasks into and out of the backlog, depending on the resource allocation of each member for that sprint.

As the sprint started, ClickUp still played a huge role during the development phase. Another tool that we strongly leveraged on is GitHub, a version control for collaboration platform. Since each task is isolated, ClickUp allows users to track each commit and/or branch to each task, making our pull-requests organized, understandable and also isolated.

As a team, we also created a guideline on how to better isolate and organize branches, commits and ultimately the tasks themselves. For every task created on ClickUp, there was a unique identifier which we used to link the name of the branches and commits. For example,

- For branches: _#{task id}_short_description_of_branch.
- For commits: #{task id}_commits_message.

Once developers were done with their features, they created a pull request and they could move that specific task to the "QA" category in ClickUp for another developer to review the pull request. We made sure that we had at least two approvals before the pull request could be merged, and only that assigned reviewer could move the task to "Done".

We heavily relied on Slack and Google Hangouts for our day-to-day communication between team members, including the product owners, and our daily stand-up calls or messages.

Lastly, we used Google Drive to store our presentations, research documents, schedules, and most importantly, the meeting protocols. These protocols are created whenever there were more than three people meeting for the project. The protocol should contain all of the members present during that meeting, the agenda of the meeting, discussions, and next steps. The protocol helped project members to look back on what had been discussed in regards to design decisions and current or future tasks.

9 Conclusion

Dukagjin Ramosaj, Kevin Marcel Styp-Rekowski

In this project, we emphasized the need for off-chaining approaches as a method to deal with the limitations of today's blockchain development environment, to broaden their usefulness and to reduce further usage costs. Taking into consideration our problem statement, by off-chaining data to a RDBMS while compromising blockchain properties as little as possible, we introduced an approach on how we can solve the problem at hand. Following the numerous aforementioned challenges which we encountered during this project, we introduced a client side application which is connected to a database and offers its functionality through an API. The client side application is capable of communicating with smart contracts which exist on the Ethereum blockchain and it enables smart contracts to use data from the RDBMS without compromising blockchain properties under certain preconditions. We exhibited the possibility of using and modifying the data from the RDBMS within the blockchain, as well as, utilizing multiple rows and multiple columns from the RDBMS simultaneously within the blockchain.

This concept was implemented on two use cases, namely the employee use case and the financial use case. We introduced concepts on how to prove query completeness of a potentially untrusted database with the help of the blockchain. Moreover, a first prototype version of a translator was developed which is capable of automatically generating contracts which implement off-chaining, while using the approach proposed in this paper, from state of the art contracts.

We anticipate that in the future, blockchain systems will continue their development and further improve in regards to scalability issues by combining and implementing new concepts. Our analysis of our new approach showed that it can indeed save on gas costs. Altough, it needs to be mentioned that this saving is not achievable for all use cases, but still for many. To summarize, we continue to consider off-chaining approaches to be crucial features in blockchain-based applications as they can potentially result in huge benefits in terms of costs.

References

[1] *MarketCap*. URL: https://coinmarketcap.com (visited on 03/10/2018).

[2] J. Eberhardt and S. Tai. "On or Off the Blockchain? Insights on Off-Chaining Computation and Data," in: *Service-Oriented and Cloud Computing: 6th IFIP WG 2.14 European Conference, ESOCC 2017, Oslo, Norway, September 27-29, 2017, Proceedings,* ed. by F. De Paoli, S. Schulte and E. Broch Johnsen. Cham: Springer International Publishing, pp. 3–15. ISBN: 978-3-319-67262-5. DOI: 10.1007/978-3-319-67262-5_1. URL: https://doi.org/10.1007/978-3-319-67262-5%5C1.

[3] S. Nakamoto. "Bitcoin: A peer-to-peer electronic cash system," in:

[4] V. Buterin. *Ethereum White Paper*. URL: https://github.com/ethereum/wiki/wiki/White-Paper (visited on 03/09/2018).

[5] *Ethereum Development*. URL: https://github.com/ethereum/wiki/wiki/Ethereum-Development-Tutorial (visited on 03/09/2018).

[6] A. J. Menezes, P. C. Van Oorschot and S. A. Vanstone. *Handbook of applied cryptography,* CRC press.

[7] R. C. Merkle. "A digital signature based on a conventional encryption function," in: *Conference on the Theory and Application of Cryptographic Techniques,* Springer, pp. 369–378.

[8] *Blockchain Oracles*. URL: https://blockchainhub.net/blockchain-oracles/ (visited on 03/08/2018).

[9] *Oraclize Dev Center*. URL: https://dev.oraclize.it/ (visited on 10/28/2017).

[10] *Solidity Examples showing how to use the Oraclize API*. URL: http://dapps.oraclize.it/browser-solidity/%5C#gist=9817193e5b05206847ed1fcd1d16bd1d&version=soljson-v0.4.19+commit.c4cbbb05.js (visited on 10/28/2017).

[11] *Oraclize Documentation*. URL: http://docs.oraclize.it/%5C#home (visited on 10/28/2017).

[12] *Oraclize Documentation - Security Deep Dive*. URL: http://docs.oraclize.it/%5C#security-deep-dive (visited on 10/28/2017).

[13] Y. Fu. "Off-Chain Computation Solutions for Ethereum Developers," URL: https://medium.com/@YondonFu/off-chain-computation-solutions-for-ethereum-developers-507b23355b17 (visited on 10/28/2017).

[14] *Oraclize random datasource Solidity example*. URL: https://github.com/oraclize/ethereum-examples/tree/master/solidity/random-datasource (visited on 10/28/2017).

[15] *Oraclize Documentation - Network Monitor*. URL: http://docs.oraclize.it/%5C#development-tools-network-monitor (visited on 10/28/2017).

[16] *Tool to verify the validity of proofs generated by Oraclize*. URL: https://github.com/oraclize/proof-verification-tool (visited on 10/28/2017).

[17] *TLSNotary*. URL: https://tlsnotary.org/ (visited on 10/28/2017).

[18] *How does Oraclize handle the TLSnotary secret?* URL: https://ethereum.stackexchange.com/questions/201/how-does-oraclize-handle-the-tlsnotary-secret (visited on 10/28/2017).

[19] *Introducing PageSigner*. URL: https://tlsnotary.org/wp/ (visited on 10/28/2017).

[20] *Oraclize Documentation - Android Proof*. URL: https://docs.oraclize.it/%5C#security-deepdive-authenticity-proofstypes-android-proof (visited on 10/28/2017).

[21] *Oraclize Documentation - Ledger Proof*. URL: https://docs.oraclize.it/%5C#security-deepdive-authenticity-proofstypes-ledger-proof (visited on 10/28/2017).

[22] *Welcoming our brand new "Ledger proof"*. URL: https://blog.oraclize.it/welcoming-our-brand-new-ledger-proof-649b9f098ccc (visited on 10/28/2017).

[23] C. S. JNTUA, Anantapuramu, P. C. S. Bindu and A. JNTUACE. "A Study on Remote Data Integrity Checking Techniques in Cloud," URL: http://pkiindia.in/pkia/Views/presentations/papers/A%20Study%20on%20Remote%20data%20integrity%20checking%20techniques%20in%20cloud.pdf (visited on 10/28/2017).

[24] L. Song, D. Zhao, X. Chen, C. Cao and X. Niu. "A Secure and Effective Anonymous Integrity Checking Protocol for Data Storage in Multicloud," URL: https://www.hindawi.com/journals/mpe/2015/614375/ (visited on 10/28/2017).

[25] Z. Hao, S. Zhong and N. Yu. "A Privacy-Preserving Remote Data Integrity Checking Protocol with Data Dynamics and Public Verifiability," URL: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.185.858&rep=rep1&type=pdf (visited on 10/28/2017).

[26] Y. Yu, M. H. Au, G. Ateniese, X. Huang, Y. Dai, W. Susilo and G. Min. "Identity-based Remote Data Integrity Checking with Perfect Data Privacy Preserving for Cloud Storage," URL: http://www.academia.edu/35496687/Identity-based%5C_Remote%5C_Data%5C_Integrity%5C_Checking%5C_with%5C_Perfect%5C_Data%5C_Privacy%5C_Preserving%5C_for%5C_Cloud%5C_Storage (visited on 10/28/2017).

[27] M. Imran, H. Hlavacs, I. U. Haq, B. Jan, F. A. Khan and A. Ahmad. "Provenance based data integrity checking and verification in cloud environments," URL: http://journals.plos.org/plosone/article?id=10.1371/journal.pone.0177576 (visited on 10/28/2017).

[28] H. Yu, Y. Cai, S. Kong, Z. Ning, F. Xue and H. Zhong. "Efficient and Secure Identity-Based Public Auditing for Dynamic Outsourced Data with Proxy," URL: https://www.google.de/url?sa=t&rct=j&q=&esrc=s&source=web&cd=57&ved=0ahUKEwjitejo-KzYAhWGzaQKHRDYBUs4MhAWCFIwBg&url=http%3A%2F%2Fwww.itiis.org%2Fdigital-library%2Fmanuscript%2Ffile%2F1828%2FTIIS%2BVol%2B11%2C%2BNo%2B10-19.pdf&usg=AOvVaw0o0bNPNb3i8-tL0VXqievl (visited on 10/28/2017).

[29] *On how to pick your random generator wiseley for Ethereum Contracts*. URL: https://github.com/rolandkofler/ether-entrophy (visited on 10/28/2017).

[30] *35 coin flipper sol*. URL: https://github.com/fivedogit/solidity-baby-steps/blob/master/contracts/35%5C_coin%5C_flipper.sol (visited on 10/28/2017).

[31] *RANDAO: A DAO working as RNG of Ethereum*. URL: https://github.com/randao/randao (visited on 10/28/2017).

[32] *How can I securely generate a random number in my smart contract?* URL: https://ethereum.stackexchange.com/questions/191/how-can-i-securely-generate-a-random-number-in-my-smart-contract (visited on 03/09/2018).

[33] *A very simple random generator*. URL: https://gist.github.com/alexvandesande/259b4ffb581493ec0a1c (visited on 03/09/2018).

[34] *Generate pseudo random numbers inside the Ethereum blockchain*. URL: https://github.com/axiomzen/eth-random (visited on 03/09/2018).

[35] *Truffle Suite - Your Ethereum Swiss Army Knife*. URL: http://truffleframework.com/ (visited on 03/09/2018).

[36] *PostgreSQL*. URL: https://www.postgresql.org/about/ (visited on 03/09/2018).

[37] *Ethereum Average BlockTime Chart*. URL: https://etherscan.io/chart/blocktime.

[38] *Oraclize Documentation - Encrypted Queries*. URL: http://docs.oraclize.it/#ethereum-advanced-topics-encrypted-queries (visited on 03/08/2018).

[39] J. Eberhardt. *Slide Deck Strategies,*

[40] C. Martel, G. Nuckolls, P. Devanbu, M. Gertz, A. Kwong and S. G. Stubblebine. "A General Model for Authenticated Data Structures," in: *Algorithmica* 39.1 May 2004, pp. 21–41. ISSN: 1432-0541. DOI: 10.1007/s00453-003-1076-8. URL: https://doi.org/10.1007/s00453-003-1076-8.

[41] A. Kundu and E. Bertino. "On Hashing Graphs." In: *IACR Cryptology ePrint Archive* 2012, p. 352.

[42] *JRFC 20 - Merkle DAG*. URL: https://github.com/jbenet/random-ideas/issues/20 (visited on 03/12/2018).