Using Blockchain Oracles as a Reliable Source of Information for a Cryptocurrency Sportsbetting App

David Mitterlehner

BACHELORARBEIT

Nr. 1610237018-A

eingereicht am
Fachhochschul-Bachelorstudiengang

Mobile Computing

in Hagenberg

im Juni 2019

This thesis was created as part of the course

Secure Mobile Systems

during

Fall Semester 2018

Advisor:

FH-Prof. DI Dr. Erik Sonnleitner

Declaration

I hereby declare and confirm that this thesis is entirely the result of my own original work. Where other sources of information have been used, they have been indicated as such and properly acknowledged. I further declare that this or similar work has not been submitted for credit elsewhere.

Hagenberg, June 17, 2019

David Mitterlehner

Contents

Abstract

Applications on the blockchain are becoming increasingly popular, since the introduction of the Ethereum Network. One of the main issues with those applications, also called *Dapps*, is the gathering of data from the outside world. Methods governing the functionality of the contracts on the network still have to be called by humans who supply the necessary input data. This is obviously a great weakness in a decentralized system, since the owner of a contract can exploit this by providing manipulated data that suits his or her interest.

For instance, contracts that carry out a certain action based on the occurrence of a certain event, need a reliable way to determine if that event did in fact happen or not. Relying on an administrator or system supervisor to decide whether or not it happened completely defeats the purpose of a decentralized system. One might as well fall back on traditional structures of computer networks with a central server in that case, because in terms of efficiency, scalability and speed those are superior to decentralized solutions.

For those reasons, a reliable way of gathering data without a central actor is needed. This can be accomplished with so-called "Blockchain Oracles", which connect applications on the blockchain to various data feeds available on the internet. There are different providers for oracles. Usually they work by providing an API which can be integrated into the smart contracts. The contracts can then be programmed to poll different kinds of web services such as weather information providers, news outlets, etc. by communicating through this API. The goal is to have a method for getting this data on demand, by using one or more oracle services that are available. Different methods for obtaining this goal are explored in this work.

Chapter 1

A Cryptocurrency Sportsbetting App

1.1 Introduction

The focus of this work is to show how an existing smart contract app, that relies on the input of truthful data about events that occurred, can be extended to include multiple independent sources to solve the problem of trust. The web app to explore is a sports betting app that works with cryptocurrencies. In the current state the app relies on the owners of the contract to supply necessary data. Only the contract creators have the permission to execute certain methods, which determine the outcome of a game. In this way, they basically have full control over who gets paid and who loses, as there are no checks against fraud by the administrators in place.

This problem can be solved by adding an additional contract to the app, which will be called "Oracle", that implements an already existing framework for obtaining data from the internet through an API. The other contracts then get their data from this "Oracle" contract, instead of the administrators.

The motivation for this work is to provide an example of how decentralized apps (Dapps) can be built in such a way as to fully incorporate the principle of decentralization. Although there was a lot of hype surrounding "Dapps", concrete use cases are still limited because of the drawbacks that come with decentralized networks, such as lower speed, throughput and very limited resources. In order to incentivize developers to choose to build apps on decentralized networks despite those drawbacks, a clear advantage needs to be demonstrated. This advantage is that trust in central actors is no longer necessary, which also means that there is no single point of failure for the system. If one of the sources of information becomes corrupted or is no longer available, there are other sources left to compare to. Checks and balances can be included in a contract to cross reference and check multiple sources, before accepting a certain piece of information to be true.

Some questions that will be answered in this work are "How can a smart contract have access to information that is not on the blockchain?". "What methods can be used to secure a system against fraudulent, manipulated data from one source?" , "How can funds be stored in a contract securely?" and "What measures can be taken to minimize the possibility of hackers exploiting weaknesses in the code?"

1.2 The Theory and Concept of Smart Contracts

Smart Contracts are a novel, useful technology based on the concept of executing code across a distributed network of nodes. There are many existing implementations of such networks that enable the deployment of so-called "DApps (Decentralized Apps)". The most popular of those platforms is the Ethereum Network.

Smart Contract Platforms		
Platform name	*Engine*	*Contract language*
Bitcoin	Bitcoin script	Ivy-lang, Balzac
Ethereum	EVM	Solidity
EOS	EVM / eWASM	C/C++
Neo	NeoVM	C, C++, GO, Py, JS

Table 1.1: Overview of the different available Smart Contract platforms.

The EVM (Ethereum Virtual Machine) was developed by Vitalik Buterin in 2013 [5]. It marked the beginning of the development of Turing complete Smart Contract platforms and programming languages. By offering an immutable, secure, transaction based state machine, it enables a way for participants to commit to a verifiable value transfer. This transfer of value is secured by the underlying computational power of the network, and can be publicly viewed at any time.

Proof of Work

The mechanism to make transactions secure is called *proof-of-work*. This proof is a way to show that computationally intensive work has been done, in order to obtain a certain piece of information, namely some input data that produces an output hash with a predefined format, as outlined by Nakamoto in the Bitcoin Whitepaper [11].

More specifically, the work has to be done by *miners* in order to include a transaction into a block, that is then appended to the public ledger (the blockchain). After a block has been added to the ledger, this cannot be undone, except by redoing the computational work. Because of this, users of the system can be confident that a transaction is very unlikely to be altered, if it has been included in a block, and more blocks have been appended after it, since it would be unprofitable and a waste of resources to try to redo all the computationally intensive work.

Potential Attacks on the Network

In the Bitcoin whitepaper, it has been demonstrated that the probability that an attacker will catch up with the honest chain of blocks drops exponentially as more blocks are added to the ledger [11, p. 7]. The trust in this system is therefore backed by the entire network of honest miners.

As long as the majority of the computational power of the network is controlled by honest participants, end users can be sure that the contracts they create, and the method calls on these contracts are recorded by the entire blockchain and the results are stored there irreversibly.

Use Cases of Smart Contracts

An example of a use case of contracts that are algorithmically enforced is law, as discussed by Buterin in the Ethereum yellowpaper [5, p. 2]. Instead of having participants in an agreement rely on a trusted authority that makes sure that what is stated in the agreement is executed accordingly, a contract can be written for one of the platforms mentioned in table 1.1. The proper execution of the agreement is then guaranteed by the computational power of the network behind this platform. One huge benefit of this approach is that the agreement can not simply be altered once it has been published on the blockchain, unless an upgrade mechanism is specifically built into the contract, for example by delegating to a contract address that can be changed.

An example for how smart contracts can offer benefits in the medical field has been explored by Azaria et al., with the development of the system *MedRec* [3]. In this work, an application is built which manages patient's medical records through SQL queries which are stored on the blockchain.

Another use case are web applications that utilize cryptocurrencies as a reward or penalty system for users, or a decentralized online exchange. An instance of the former will be discussed in the next section, a cryptocurrency sportsbetting app.

Cost of Code Execution

Since the network behind Ethereum consists of miners that want to make a profit by receiving *ether* for their work, code execution on the network costs actual money. The unit by which this is measured in the Ethereum space is *gas*. The cost of different instructions varies, reading data from a contract is usually free, but writing variables to the blockchain can be quite costly. This needs to be considered when writing a contract, to ensure maximum efficiency, and no unnecessary gas leaks.

1.3 Practical Implementation of Smart Contracts in an App

As part of a semester project, a web app has been developed to serve as a platform for placing bets on the results of various sports games. The framework that has been used for this is *Angular* in combination with TypeScript and the web3 Ethereum JavaScript API. The front-end was built using a template called "Material" by Creative Tim. The back-end was realized using the Solidity programming language for the definition of contract parameters and methods, and the Truffle Framework to manage it all. *Ganache*, which is part of the Truffle Suite, provides a local blockchain, which served in this project for testing, deploying and debugging the system.

Structure of the Contracts

The contracts for this app are split into three parts, a *Betting Factory*, a *Game Manager* and a *Bet Manager*. All contracts inherit from the *Ownable* contract, which ensures that certain methods can only be called by the creator of the contract, such as creating a new game for placing bets, or changing the state of a game. Data structures for a game have been implemented in the *Game Manager*, data structures for bets in the *Betting Factory*.

Design Choices

Since programming in Solidity involves thinking about the cost of code execution, some key concepts needed to be used in order to make sure no unnecessary *gas* is used, thus wasting money in the form of *ether*. *Gas* is a unit which measures the cost of specific instructions of the EVM. The contracts have to be created in such a way, that a minimum amount of information is stored on the blockchain, as those instructions are the most expensive. One also needs to make sure that the code is not bloated and contains redundant instructions, since this would increase the cost of deploying the contract to the network. Furthermore, when coding smart contracts, it is especially important to make sure that no leaks of *ether* can occur, due to sloppy implementations and lack of testing, which is a common problem for developers who come from traditional programming and are not used to dealing with money as an integer directly in a program [7].

One major issue is that the application would not be user friendly anymore if the contracts are not designed with those points in mind because in the end the user of the system has to pay the fees that result from method calls of the contract. The way the Ethereum Network is set up, a user can decide independently how much they want to spend for the *gas* on a transaction such as a method call of a contract. This is called the *gas price* [5, p. 7]. However, since transactions have to compete to get included in the next block, miners will only chose transactions with a certain minimum *gas price*. This price depends on the load of the network.

Critical Issues and Security Bugs

The proper execution of a contract hinges on the integrity of the Ethereum blockchain. While it is a very robust system, as many existing applications using smart contracts have shown [6], there are still some major risks and security concerns to keep in mind when developing a decentralized application. Those security issues come, for instance, from uncertainty of transaction ordering, false timestamps (due to malicious miners) or from mishandling exceptions. In the following table the most common security bugs are listed, with a short description of the cause, and an example of a countermeasure that can be taken in order to prevent the bug from being exploited.
The most common of these bugs is, according to an analysis conducted with the the security scanning tool *Oyente* [9, p. 11], the *mishandled exception*. In this analysis, out of a sample of 19.366 contracts, 27.9% were flagged by the tool. This bug is followed

Common Security Bugs in Smart Contracts		
Bug name	*Caused By*	*Countermeasures*
Transaction-Ordering Dependence	Two transactions in the same block invoking the same contract	Guarded transactions, Locking
Timestamp Dependence	Malicious miners sending manipulated timestamps	Deterministic timestamps
Mishandled Exceptions	Not validating return values	Better exception handling
Reentrancy Vulnerability	Multiple calls exploiting an intermediate state	Reentrancy Detection

Table 1.2: Typical security issues in decentralized apps [9].

by *transaction-ordering dependence* (15.7%), *reentrancy handling* and the least common bug, *timestamp dependence*. One of these bugs, despite not being very prevalent in the sample, has caused a great amount of financial damage in the past, in the infamous "The DAO" hack [8]. At fault was the *reentrancy handling* bug. The DAO was a decentralized autonomous organization for crowdfunding, managing around 12.7 million ether in user funds. As a result of the exploit, 3.6 million ether were stolen, valued at the time at around $70 million. The lost funds were later returned by forking the entire Ethereum blockchain, which was the cause for a very heated debate about the fundamentals of blockchain (immutability, irreversibility). However, the majority of users agreed on this change. As a result the current, most widely used version of Ethereum is the one where the funds have been returned.

Occurrences like this demonstrate just how important it is to thoroughly test for security vulnerabilities before deploying smart contracts to a public blockchain. If funds are lost, usually they can never be returned to the users. In the case of the DAO it was simply the huge amount of lost funds and the vast media attention that caused the Ethereum developers to take action and reverse the theft.

1.4 Security and Trust in the Current State of the System

One major weakness that stands out in the web app, is the question of trust. The idea behind decentralized apps is to distribute trust across nodes and build a consensus this way. However, in the system at hand, there is still a single point of control, namely the power to create games that users can bet on, and to enter the final results, which ultimately determine the outcome of bets. This is done by specific methods that can only be called by the creators of the contract. While the creators of this contract, the administrators, don't have direct access to user's funds they can still manipulate and exploit this system for personal gain by providing untruthful game results.

This key issue needs to be addressed by delegating the authority over deciding what the results are to a diverse set of independent sources, so called *blockchain oracles*.

Chapter 2

Related Work

2.1 Managing Data on the Blockchain

There are numerous existing implementations of decentralized apps, some of which will briefly be explored in this section. One example is *MedRec* a system for storing and managing medical data on the blockchain [3].

Managing a web application through an administrator backend access is often associated with a lot of time and effort, since the administrator or a team of managers constantly have to monitor the system, and push changes manually as needed. An example of how management can be made easier and more efficient is demonstrated in the system *MedRec*.

Specifically, in this system permission management is done completely automatic by contracts designed for this purpose. A patient can decide what parts of their medical data can be accessed by which parties. It is not necessary to rely on a central server, managed by a trusted authority, to control who can view the data and to which extent. Instead, this information is stored on the distributed ledger, and thus enforced by all the participating nodes. No additional administration is needed, the two parties consisting of patient and care provider can interact directly with each other. Furthermore, the parties don't have to interact with the blockchain directly, as this would be not very user friendly, but an interface is provided which translates the user input into the appropriate API calls that form the connection to the contracts. Querying of data is also done through the blockchain, i.e. the SQL strings for retrieving the data are stored there, and sent to an application for further processing if triggered by an authorized party.

This reduction of management overhead and increase in efficiency can be extended to the application accompanying this work, the cryptocurrency sportsbetting app. The administrators only have to provide information about the available games users can bet on. Other tasks, like managing user funds, access control to the total funds in the contract, and who can withdraw under which circumstances are implemented in the contracts.

2.2 Security Tools for Smart Contracts

Programming in the realm of smart contracts on decentralized networks always involves dealing with money in the code, as has already been demonstrated in chapter 1.3. Because of this, security has a very high priority, to ensure that users can trust in the system and funds are not at risk of being stolen by hackers. Another reason for the need of improved security measures is the fact that a lot of smart contract platforms, including Ethereum, are completely open to the public. Anyone with internet access can call contracts if they have the address, so the worst-case scenario always needs to be considered when developing apps for those platforms. Especially if a *Dapp* gains popularity, it has to be assumed that hackers are interested in exploiting the system for personal profit.

Another issue that needs to be focused on, is the fact that network participants (miners), are the ones who decide which transactions are accepted, how they should be ordered and they also set the block timestamp, which can be a source of manipulation.

There are a variety of security scanning tools available, mostly for code written for the Ethereum Network. One of these tools, *Oyente*, will be described in more detail in this section.

The Security Scanning Tool Oyente

Oyente is a symbolic execution tool developed by Luu et al. and described in their work "Making Smart Contracts Smarter"[9]. It is capable of analyzing EVM bytecode directly, without needing access to the high level representation in Solidity or Serpent. This is important, because often access to the high level code is not available, as the Ethereum blockchain only stores the EVM bytecode and in many cases developers don't provide access to public repositories to review the source code. However, the tool is also capable of analyzing source code directly, so one can specify a Solidity source code file as input and the tool will scan this contract for critical bugs. The code for *Oyente* is open source and can be found on GitHub [15].

The easiest way to start using Oyente is to make use of the pre-fabricated docker image that is available under `luongnguyen/oyente`. This image contains all the necessary dependencies for the application, which makes the installation process very easy. The image can be downloaded by issuing the command `docker pull luongnguyen/oyente` on the terminal, and then creating a new container from this image by executing `docker run -i -t luongnguyen/oyente`, or better starting the container in privileged mode with `docker run --privileged -i -t luongnguyen/oyente`, to have access to all devices on the host. It should be noted that root access is required to interact with the docker daemon.

Inside this container, the Python program `oyente.py` is available in the directory `/oyente/oyente/`. One can evaluate a contract by changing into this directory and then running `python oyente.py -s Contract.sol`, for Solidity source code. To evaluate a contract that is only available in EVM bytecode, the flag "-b" needs to be appended to the command. For example, `python oyente.py -s BytecodeContract -b` evaluates a file which contains EVM bytecode.

Since Oyente runs in a docker container that is created from the image from scratch

every time, in order to make changes in this container permanent, the modified container needs to be saved explicitly. The starting container has example Solidity contracts for analyzing, but to run the tool on user defined contracts, those need to be copied into the container first. This is done by obtaining the container ID via `sudo docker ps` and then executing `sudo docker cp [source_path] [container_id]:[destination_path]`.

The version of the Solidity compiler `solc` which is found in the container created from the image is outdated. Because of this, it is recommended to update all software packages in the container. The operating system present in the container is Ubuntu 17.10, which is based on Debian, so it uses the Advanced Package Tool (APT) for managing the software on the system. Using APT, updating packages can be done by running first `apt-get update`, and then `apt-get upgrade`. After making all these changes, the modified container can be saved by executing `sudo docker commit [container_id] luongnguyen/oyente:version2`.

Analyzing the Contracts in the Project

To ensure the security of the smart contracts powering the cryptocurrency sportsbetting application, the aforementioned scanning tool will be used. Running *Oyente* on the *BetManager* contract, which inherits from all other contracts, produces a satisfactory, albeit not perfect output. There were no extremely critical bugs, such as *Parity Multisig Bug, Callstack Depth Attack Vulnerability, Transaction-Ordering Dependence, Timestamp Dependence* or *Re-Entrancy Vulnerability.* The Parity Multisig Bug, which wasn't described before in this paper, would enable an attacker to acquire ownership of the contract, by re-initializing it with her address, and to drain the funds stored in the contract. In the contracts that are used in this project, this is not possible, because the only way to withdraw funds is through a dedicated *withdrawFundsForBet* function, and there is no initializing function to change the owner of the contract. However the scanning tool did produce a positive result concerning *Integer Underflow* and *Integer Overflow* for certain contracts.

Discovered Vulnerabilities		
Affected contract	*Integer Underflow*	*Integer Overflow*
Bet Manager	True	True
Betting Factory	False	True
Game Manager	True	True
Ownable	False	False

Table 2.1: Overview of discovered vulnerabilities.

Upon closer inspection, some of these overflows could be fixed by using the `SafeMath` library for mathematical operations. This library provides mathematical operations with safety checks, the revert the operation on an error. For instance, with a multiplication a * b = c, it implements a *require* statement c / a = b. In case of an integer overflow, this statement wouldn't hold true, and the transaction would get reverted. Some of the positive results flagged by the security scanner were simply false positives that could not be reproduced.

Chapter 3

The Problem of Trust and Centralization

3.1 Why Centralized Control is Undesirable in this Project

The reason for building a decentralized application is mainly that users don't have to trust a central authority with their money and their data. Another reason is that users can rely on the integrity of the data provided by the distributed system. In the fully decentralized peer-to-peer model, there is no central point that can be attacked or tampered with. So the end user can be sure that the data they received from one of the nodes of the network is authentic, or at least they can verify this with digital signatures.

However, if there is a central instance that provides the data, as is the case in the current state of the project, users can never be sure that this central source has not been compromised. As of now, the power lies in the hands of the administrators who control what games users can bet on, and also the results of these games. While security precautions have been taken to ensure that not *anyone* can alter game results or manipulate bets, since only the creators of the `BetManager` contract on the blockchain can call the functions responsible for this, there is still the possibility of the administrators being bribed, or their private keys being stolen. Furthermore, the administrators could take advantage of their power and use their control for personal financial gain, by providing game results they have bet on, etc.

For all these reasons mentioned above, in its current state, the project ultimately defeats the purpose of being provided on a decentralized platform because it still sources crucial data from a central hand. The whole point of blockchain systems is to not have this central point. Systems built based on the traditional model with a central instance have many advantages like better speed, latency and throughput compared to those novel platforms. The factor of having no single controlling instance is essential and must be preserved in order to not defeat the purpose of this project. The target user, which is someone who rejects centralization in favor of decentralized, distributed systems, can not be satisfied with the system as it is. Countermeasures need to be taken and solutions provided to mitigate these issues. This is where so called *Blockchain Oracles* come in.

3.2 The Concept of Blockchain Oracles

In order to receive data from the world outside of the Ethereum network, there needs to be a way for smart contracts to be able to interact with regular web servers, for example through APIs that can be queried over regular HTTP or HTTPS. Blockchain applications themselves cannot directly fetch the data they require, like price feeds, weather data, sports game results, election results, etc. A blockchain oracle is basically just a connector that makes this possible. Usually, the oracle itself is also a contract deployed on the blockchain, that can be accessed at its contract address, and queried through specific methods [18]. There are a number of different implementations of such oracles, for various blockchain platforms.

Blockchain Oracle Implementations			
Blockchain Protocol	*Orisi*	*Town Crier*	*Oraclize*
Bitcoin	yes	no	yes
Ethereum	no	yes	yes
Rootstock	no	no	yes
EOS	no	no	yes
R3 Corda	no	no	yes
Hyperledger	no	no	yes
Fabric	no	no	yes

Table 3.1: Blockchain platforms supported by various oracles [13][18][21]

In the table above, it becomes apparent that the most versatile oracle clearly is the *Oraclize* implementation. This service, *Oraclize*, will be used in the cryptocurrency sportsbetting app, because it provides an extensive documentation and a rich interface.

3.3 Existing Solutions

3.3.1 Oraclize

Oraclize is the leading oracle service in the blockchain space, supporting a variety of platforms, and even non-blockchain applications [18]. It provides a solution which can demonstrate that the data fetched from the third party source, for example over an API request, has not been manipulated. So called *Authenticity Proofs* are used to accomplish this.

Authenticity Proofs

In order to guarantee data integrity, a document can be requested from a smart contract in addition to the desired third party data, which is a strong cryptographic guarantee providing that the data has not been tampered with. For this, the trust is shifted away from Oraclize, to technology providers or device manufacturers with good reputation and a lot at stake. An example is the *Ledger proof*, which uses the hardware attestation feature provided by BOLOS, to enable code execution on the Ledger platform. BOLOS

is a crypto-embedded operating system built for Secure Elements and Secure Enclaves. Ledger, which is a hardware manufacturer for cryptocurrency cold storage, is a device manufacturer with good reputation and a lot at stake. This is one example of how an authenticity proof might be generated. The Oraclize service currently implements three different types of authenticity proofs:

- TLSNotary Proof
- Android Proof
- Ledger Proof

TLSNotary Proof

TLSNotary is an open-source protocol and a mechanism for independently audited HTTPS sessions. The technology is developed and used by the *PageSigner* project[14].

It works by splitting the secret data (premaster secret) used to setup the https session to the desired data source, i.e. the server, between the auditee (in this case Oraclize) and the auditor (in this case a locked-down AWS instance of an open-source Amazon Machine Image). In this way, it is possible for the auditee, Oraclize, to prove to the auditor that certain web traffic has occurred between them and a server, and that the traffic has come, in fact, from the server (data origin authentication). The server would be the data source requested by the issuer of the "oraclize_query" call, such as a machine that serves data over a REST API URL. This authenticity proof is indisputable, provided the auditor trusts the public key of the server (the data source)[1][18].

The basic steps of how the TLS secret data is split into two parts, and the control flow of how the auditor makes sure that the auditee doesn't produce manipulated data, can be outlined as follows:

1. Auditee and auditor generate 24 bytes of the premaster secret each, the full premaster secret is not known to either of them.
2. Both, auditee and auditor, apply a hash function to those 24 bytes, generating 48 bytes (auditee $\rightarrow H_1$ and auditor $\rightarrow H_2$).
3. Auditor gives the first half (H_{21}) of their hashed secret to auditee.
4. Auditee gives the second half (H_{12}) of their hashed secret to auditor.
5. Using these halves, by applying the XOR logical operation, ($H_{11} \oplus H_{21} \rightarrow$ auditee's master secret, and $H_{22} \oplus H_{12} \rightarrow$ auditor's master secret), each party constructs their half of the master secret, which is then further processed to obtain the initialization vectors (IVs), encryption keys, and message authentication codes (MACs) used in the TLS protocol.
6. Both parties compute a hash (140 bytes) of their master secret halves.
7. The auditor gives approximately 120 bytes (exact number depends on the cipher suite) of this hash to the auditee, enabling them to compute IVs, client MAC key, and encryption keys (by XORing the bytes of the hashes). At this point, the client can write the https request and send it to the server. This can be seen as the start of the audit.
8. The server (the desired data source of truth) then responds, the response is received by the client, but not decrypted. Instead, the network traffic is logged, a hash is computed and sent to the auditor (commitment of the auditee). This is

the key point in the whole auditing process, as this commitment (the hash), can later be used to prove the authenticity of the data, since there was no way for the client to fabricate fake traffic data using the server mac that they don't have at that point.

9. After the auditor receives this commitment, they reveal the remaining approximately 20 bytes, and send them to the auditee. This allows them to compute the server mac key. The TLS process continues from there, the auditee decrypts the network traffic, and checks authenticity with the server MAC.

The process is illustrated in detail in the following figures:

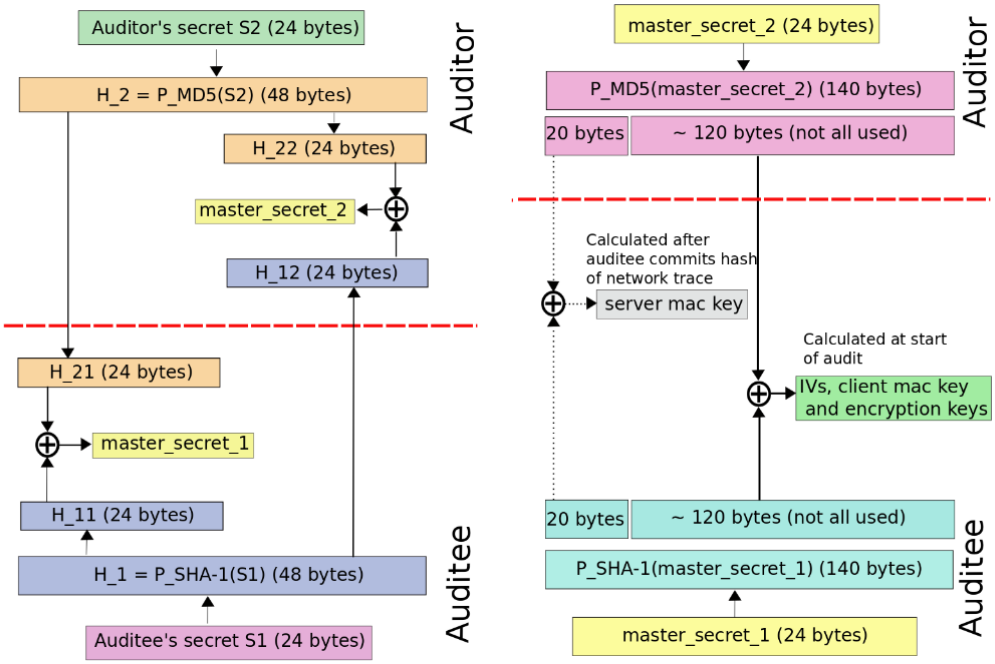


Figure 3.1: Control flow of the TLS secret splitting and auditing process. [1]

This series of complex steps is done so that the auditee (Oraclize) cannot create a fake version of the HTTP traffic from the server, since they do not have the server mac key. Only once the auditor releases the remaining 20 bytes of the expanded key block (generated from the master secret key halves), the auditee can complete the TLS decryption and authentication steps, view the data, and forward it to the issuer of the oraclize request.

In this way, the creator of a smart contract using Oraclize can be sure that the data forwarded by this service has not been tampered with and is authentic.

Android Proof

Android Proof is another type of Oraclize's authenticity proofs. It uses Google's *SafetyNet* Software Attestation and Android Hardware Attestation in order to provide an environment that is secure and auditable, which can deliver reliable data.

The basic concept is to use a service application, which is running on a trusted physical Android device, to fetch and deliver this data. The integrity of this device is guaranteed by *SafetyNet*, which can prove that the list of root certificate authorities stored on the device has not been modified. It does this by verifying the full chain of certificates against publicly available certificate revocation lists owned by Google. *SafetyNet* can also detect whether an Android device is in a tampered state or not.

Another important part of this type of proof is the Hardware Attestation Object, which is required to prove that the key has been generated inside the KeyStore of the trusted physical Android device.

The process of retrieving data from an API while using the Android Proof attestation can be outlined in the following way[17]:

- The URL provided by the issuer of the oraclize request (the data source URL) is sent to the trusted Android device. The service application running on this device then establishes a HTTPS connection with this URL, and the entire response is retrieved from the server. Then the SHA256 hash of this response is calculated and signed by the application, using the hardware attested key pair from the Android KeyStore on this device.
- A call to Google's SafetyNet API is made using the data from before as parameter. The API returns an *AttestationResponse* in the JSON Web Signature format (JWS).
- The service application running on the trusted device then sends the entire JWS response, the HTTPS response with its SHA256 signature and the requestID to Oraclize. There the proof is validated and the data from the HTTPS response is forwarded to the issuer of the oraclize request. The SafetyNet AttestationResponse and the Hardware Attestation Object are also sent to the issuer.

The following picture demonstrates this process in a simplified way:

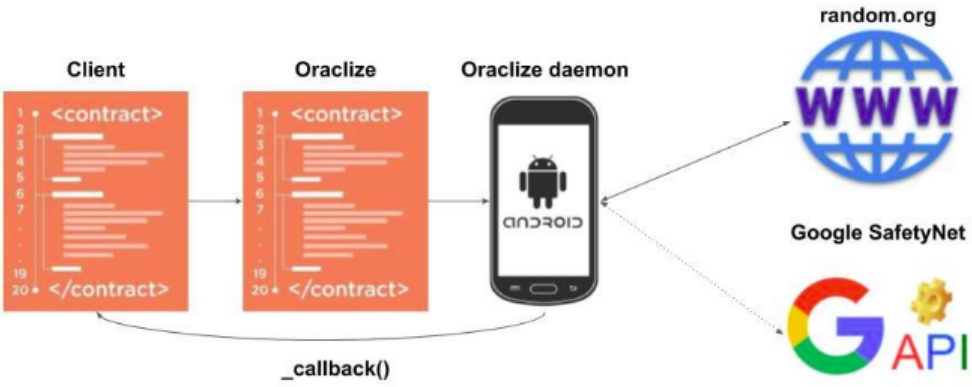


Figure 3.2: Oraclize API request using Android Proof[17]

This type of authenticity proof can be verified by any third party by checking the following elements:

- JWS Verification, this can be done by validating the certificate chain found in the JWS Header against a certificate revocation list by a known root certificate authority.
- SafetyNet Authenticity. For this, the Google Device Verification API can be consulted to check if the JWS has indeed been generated by Google.
- SatetyNet Response Verification
- Hardware Attestation Verification

In order for the Android Proof to be a suitable method for guaranteeing data authenticity, a number of features of the Android platform are utilized. Worth mentioning here are the Android Hardware Keystore and the Hardware Attestation, first implemented in Android Nougat. Both features are implemented in a TEE (Trusted Execution Environment). Furthermore, the concept relies on Google's SafetyNet Software Attestation APIs and the Android App Sandbox model. It is assumed that this model is secure and prevents apps from manipulating memory or data that do not belong to them. This sandbox model is one of the key features of the Android OS. Google's SafetyNet is a feature which can be used by developers to check whether a device is in a tampered (rooted, monitored, infected with malware) state or not.

The steps of generating this Android Proof are outlined in more detail in the following graph, taken from the Android Proof whitepaper by Oraclize.

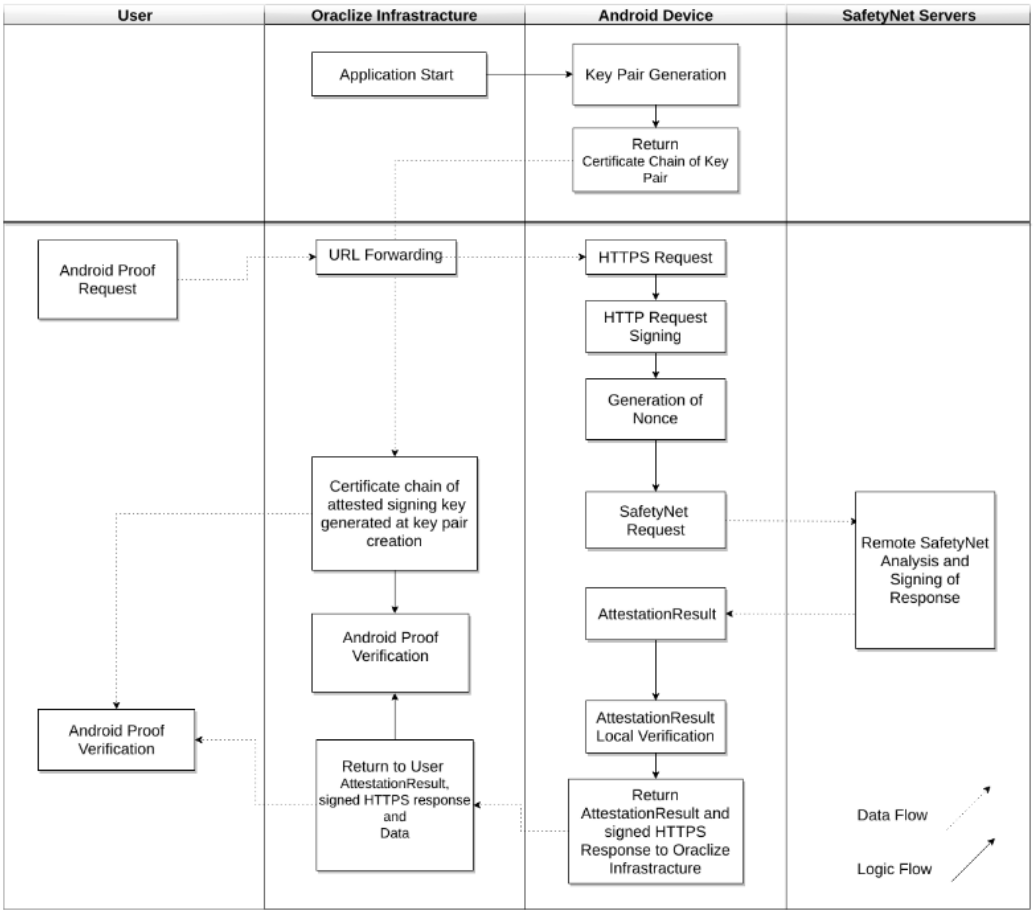


Figure 3.3: A detailed graph showing how the Android Proof is generated and verified[17]

Ledger Proof

The third kind of authenticity proof provided by the Oraclize service is called *Ledger Proof*. *Ledger* is a French company specializing in the field of producing hardware-enforced cryptocurrency wallets. Their flagship product is the Ledger Nano S. Their devices are equipped with a STMicroelectronics secure element, a controller and the operating system *BOLOS*. Developers can build applications for this OS using the BOLOS-SDK.

The feature in the *BOLOS* operating system which is important to Oraclize is the kernel-level API which can conduct cryptographic operations and attestations. The attestation feature enables any application, by querying the appropriate API, to retrieve a signed hash of the kernel binary. This hash is signed using a key which is controlled by the BOLOS kernel and can not be accessed by the application developer. Furthermore, this key has a full chain of trust with the root in a master key that resides on a hardware security module controlled by Ledger.

These features of the BOLOS operating system are utilized by Oraclize to attest to the user of the Oraclize query system that the applications built by Oraclize are indeed

running in a trusted execution environment (TEE) of a true Ledger device[18].

There are tools available to verify the proofs generated by Oraclize [19].

Oraclize Engine

The basis for the functionality of the Oraclize service is the *Oraclize Engine*. Internally, it replicates a logical, conditional model. Thus, certain conditions can be verified repeatedly and data will only be returned if those conditions are met. In order to form a valid request, the engine needs to be provided with at least two, or optionally three arguments:

- A data source type
- A query
- (Optional) An authenticity proof type

Those arguments are passed as string arguments to the `oraclize_query` function call. An example call could look like this:

```
oraclize_query("URL", "xml(https://www.example.at/api).element");
```

Data Source Types

In the example above, the data source type was "URL", however Oraclize supports various other types, such as[18]

- WolframAlpha: native access to WolframAlpha engine
- IPFS: access to any file stored on IPFS (InterPlanetary File System)
- Random: provides random data utilizing the Ledger Authenticity Proof type
- Computation: the result of a computation specified by the issuer of the query

Furthermore there are meta data source types, including:

- Nested: used for nesting multiple data source types of different kind, or multiple requests to one source, returns one unified result
- Identity: returns the query
- Decrypt: decrypts data which was encrypted with the Oraclize private key

Queries

The query is the central element to be used in the Oraclize engine. It specifies the details of the request containing information where the data is to be retrieved from, for example a specific URL.

On a technical level, the query is an array of parameters, with the first one being mandatory. An example is the URL where a certain resource is located. However, the result of this query may not be suitable in the original format, it may need to be parsed. For this, Oraclize provides parsing helpers, which can be specified within the query.

Query Parsing Helpers

There are JSON, XML, XHTML and binary parsing helpers offered by the Oraclize service. They are used by prefixing the name of the data format (e.g. "json") to the

query and appending object, tag names or operators, depending on the format. An example using a parsing helper for binary data source types could look like this:

`binary(http://www.siemens.com/pki/ZZZZZZVS.crl).slice(0,300)`

The parameters of the slice operator are `(offset, length)`. This example will return the first 300 bytes in this binary file.

3.3.2 Verity

Verity is a platform for decentralized data feeds. It has a conceptually different approach on fetching data, when compared to Oraclize. Instead of connecting to Web-APIs, it sources the data by using wisdom of the crowd and blockchain-as-a-court-system[10].

Basically, the core concept of this service is to create an incentive for people to provide truthful data about real-world events. This is done by giving participants monetary rewards, in the form of cryptocurrencies, if they provide correct information about events that have happened in the world.

The correctness of information in this system is determined by a rather complex consensus mechanism. Data providers (mostly people observing real world events) report on events via a desktop or mobile app. They can enter what the values of pre-defined parameters of a data request are, for instance, which team won a football game. The data request is made by developers of smart contracts that rely on data from the outside world, such as the outcome of a sports game, how many yellow cards were raised in a game, how many people attended an event, etc. The basic flow of information from the data providers to the developers is illustrated in the following figure:

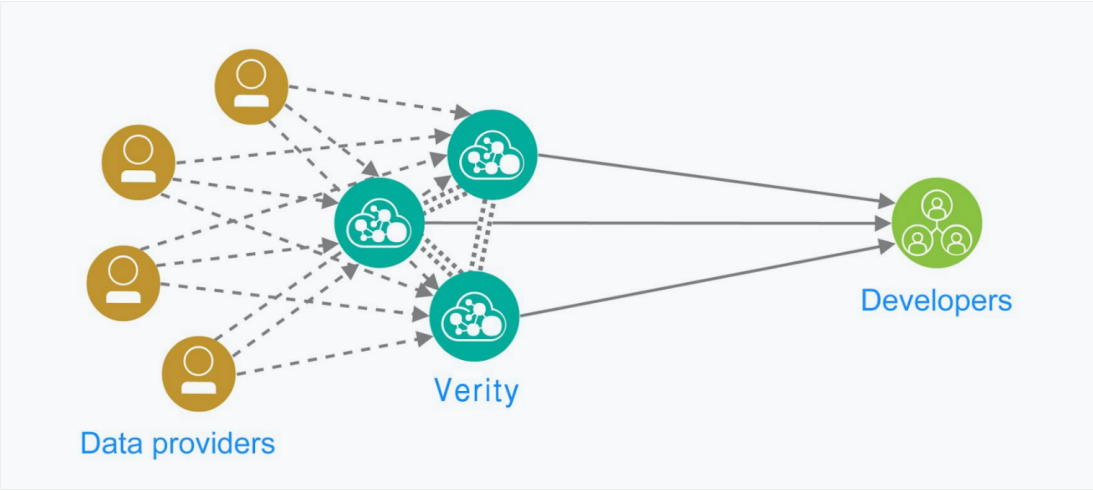


Figure 3.4: The flow of information in the verity network[10]

On the left hand side are the people that observe the events, of which the developer (of a smart contract) wants to know the outcomes. In the middle are the Verity network nodes, which validate the data by forming a consensus (e.g. a specified percentage of data providers report the same information). The parameters for this consensus, such as percentage of same value reports, minimum number of participants, etc., can be defined by the developer.

Verity System Architecture

The architecture of the system is split into three parts, the core layer, the services layer, and the application layer. The core layer implements the consensus mechanism and node communication (discovery, routing, etc.) and the services layer implements the Verity API, a data feed engine and a marketplace system. The application layer consists of validation nodes, data providing nodes, and provides a Verity SDK and analytics.

The Ethereum blockchain is used as the base layer for the system, plus *Swarm*, which is a decentralized storage platform and content distribution service[20].

The following graphic provides a good overview over the architecture of the entire system:

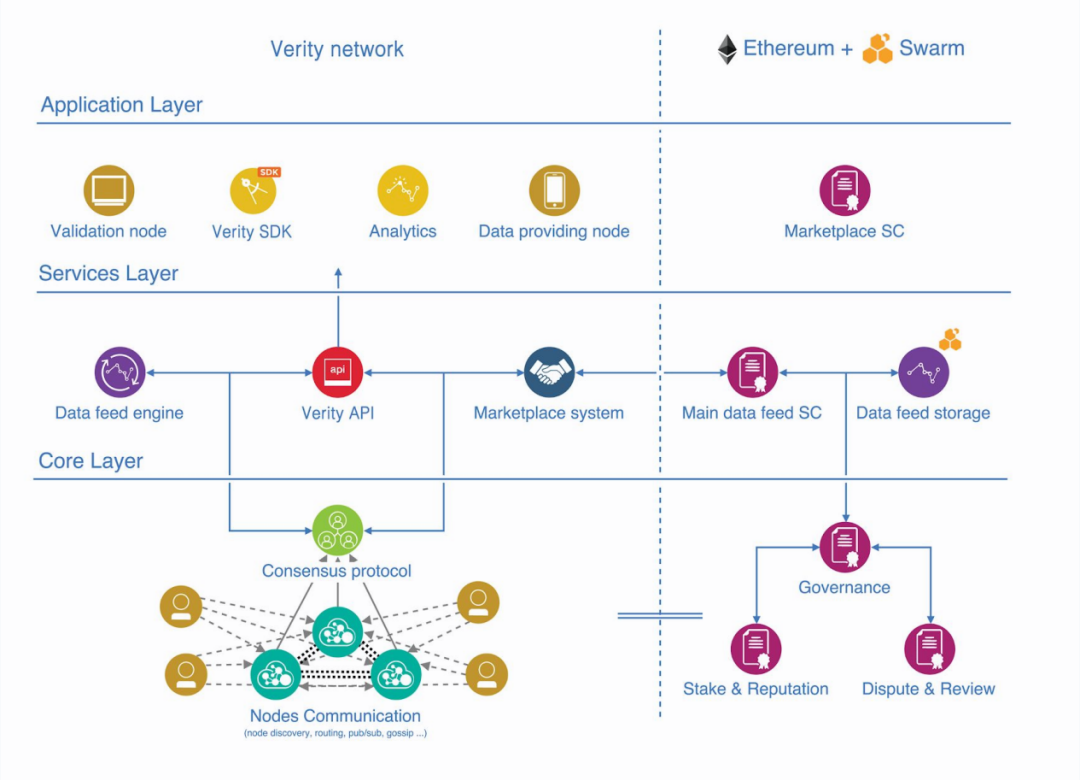


Figure 3.5: The architecture of the Verity Network[10]

Data Markets

In order to receive data from the Verity network, a developer first needs to specify what kind of data she wants, in what format the data should be provided, how the consensus for this data should be established, and how much she is willing to pay for this data[10]. This is necessary to incentivize data providers to provide high quality, truthful data.

This process is called *market generation*. In Verity, a market can consist of one or more *data feeds*, and each data feed contains at least one or more *data fields*, which represent events, such as whether a penalty kick was successful (boolean), what the end score in a tennis game is (two integers), or even more complex things, like what the

name of the current president of a certain country is (string).

The following figure illustrates what a data market in Verity consists of:

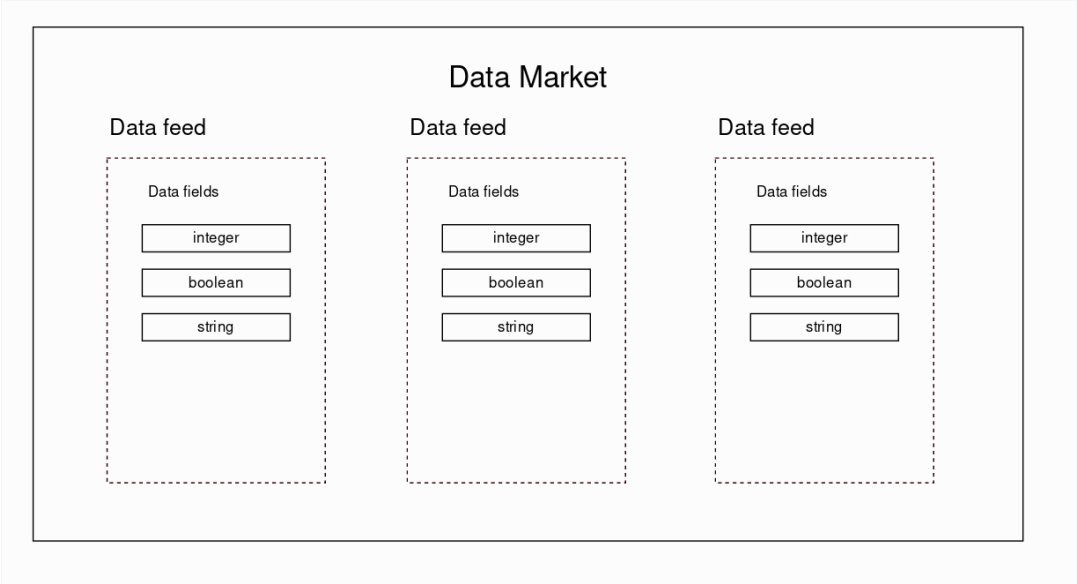


Figure 3.6: The structure of a data market in Verity

Verity Network Nodes

In the Verity network, there are two different kinds of nodes: *Validation* nodes, and *Data Providing* nodes.

Validation Nodes

Validation nodes capture data, form the consensus according to the rules set by the developer, and send the data to the developer. The also earn fees for correctly processing data (according to the system's rules). Validation nodes are part of a reputation system, and can vote in the governance of the entire system, where each vote is weighted by the node's reputation.

Before a market is created, validation nodes for validating the data in this market are selected randomly. Validation nodes always process *encrypted* data[10].

Data Providing Nodes

Data Providing Nodes do not partake in any consensus forming or voting, they are only there to send data to the validation nodes.

3.3.3 Astraea

Astraea is a decentralized blockchain oracle, described in a paper by John Adler et al.[2], however it has never been implemented as a real world service. Other than Oraclize,

it is not an intermediary for fetching data from third party sources, but attempts to deliver data based on a voting game that decides the truth or falsity of propositions.

In the proposed system, there are players (submitters, voters and certifiers) that behave in the following way:

- Submitters allocate money to fund propositions that get voted on
- Voters submit deposits, and are given the chance to vote on one of the available propositions, chosen at random. The maximum amount a voter can deposit is a parameter of the system.
- Certifiers choose an available proposition, and place a large deposit in order to certify its truth or falsity. The minimum deposit for a certification is a system parameter and should be large enough to carry a certain risk.

In the paper it is shown that such a system can be set up in a way that manipulation by an adversary is almost impossible, even if the adversary contains up to 25% of all votes[2].

3.3.4 ChainLink

ChainLink is another provider of oracle services in the blockchain space. Currently, the service is available only on the Ethereum test networks, that is Ropsten, Rinkeby and Kovan.

One unique characteristic of *ChainLink* is that the service utilizes the dedicated *LINK* token to carry out transactions.

Requirements for requesting data

In order to request data using the ChainLink oracle, some requirements need to be met:

- The contract which carries out the request needs to be funded with LINK tokens. On the test networks, one LINK token amounts to one request.
- Depending on the type of data to be requested, a "Job ID" needs to be specified. This is different on every network.

For testing purposes, LINK tokens can be obtained from a faucet, which is a contract that gives out free tokens. An example for this is a faucet that runs on the Ropsten testnet, accessible via HTTP at `https://ropsten.chain.link/`. The address of the contract which calls the request function of the ChainLink contract is entered into the input field on this website, in order to receive LINK tokens.

The "Job ID" for each type of data is found in the ChainLink documentation: `https://docs.chain.link/docs/addresses-and-job-specs`. For instance, if the contract owner wants to request boolean data, the corresponding "Job ID" would be

`7ac0b3beac2c448cb2f6b2840d61d31f`. It is advisable to store those jobs ids in a constant in the contract, like this:

```
1 bytes32 constant UINT256_MUL_JOB = bytes32("493610cff14346f786f88ed791ab7704");
```

To specify HTTP POST or GET parameters, or to navigate through a JSON object, methods within `Chainlink.Request` can be used. For example, to fetch the current Ethereum USD price over an API, the function can look as follows:

```
1 function requestPrice(string _currency) public returns (bytes32 requestId) {
2   Chainlink.Request memory req =
3   newRequest(UINT256_MUL_JOB, this, this.fulfillEthereumPrice.selector);
4   req.add('get', 'https://some-crypto-api.com/data/eth');
5   req.add('path', _currency);
6   requestId = chainlinkRequest(req, 1 * LINK);
7 }
```

Receiving data

In the `requestPrice` function implemented previously, a callback function that has been assigned to `requestId` is returned. This function is triggered, when the Chainlink network responds to the query with a result.

External Adapters

Chainlink provides a feature called "External Adapters". Basically, these adapters represent the bridge between the Chainlink node, which handles, amongst other things, signing transactions for the blockchain and writing to the blockchain, and complex external APIs that may require some form of authentication.

Publicly open APIs can be read by the Chainlink node directly, without any adapters. However, some APIs require authentication with sensitive credentials. These should not be stored on the Chainlink node directly, as that would pose a security risk. Credentials can be stored in whatever way it is specified in the external adapter.

Furthermore, these adapters can include additional functionality. They can be written in any language[16].

Architecture Overview

The Chainlink system architecture is outlined in a simplified way in the following figure:

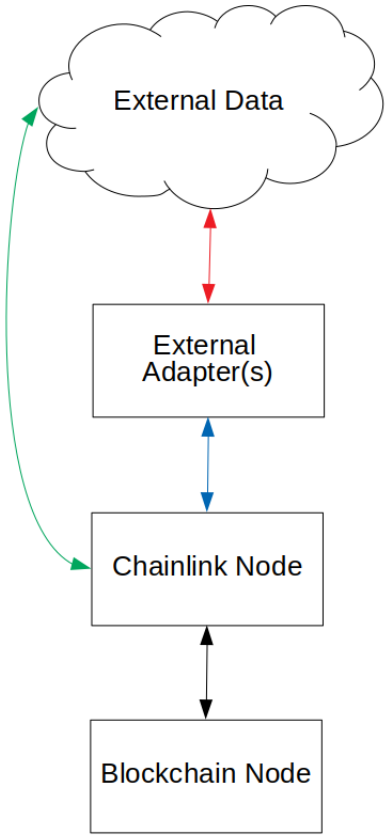


Figure 3.7: The ChainLink system architecture, as described in the official documentation[16]

3.4 Implementation of an Oracle in an App

In the following section, the implementation of an Oracle in an App is demonstrated using the *Oraclize* service. The first steps of extending an app to work with Oraclize, is to fetch the connector contract from Github. It can be found under this URL: `https://github.com/oraclize/ethereum-api/blob/master /oraclizeAPI_0.5.sol`. This source code needs to be downloaded and saved into the `/contracts` directory. The name should be `usingOraclize.sol`. This contract contains the main logic and the adapters for communicating with the external APIs.

For simulating a blockchain environment, the *Ganache* application is used, as mentioned in chapter 1. The local blockchain will be served from the host `http://192.168.0.108:7545`, although that URL can be changed in the application settings. When first launching the application, no transactions will be visible in the "Transactions" tab, as seen in the following figure:

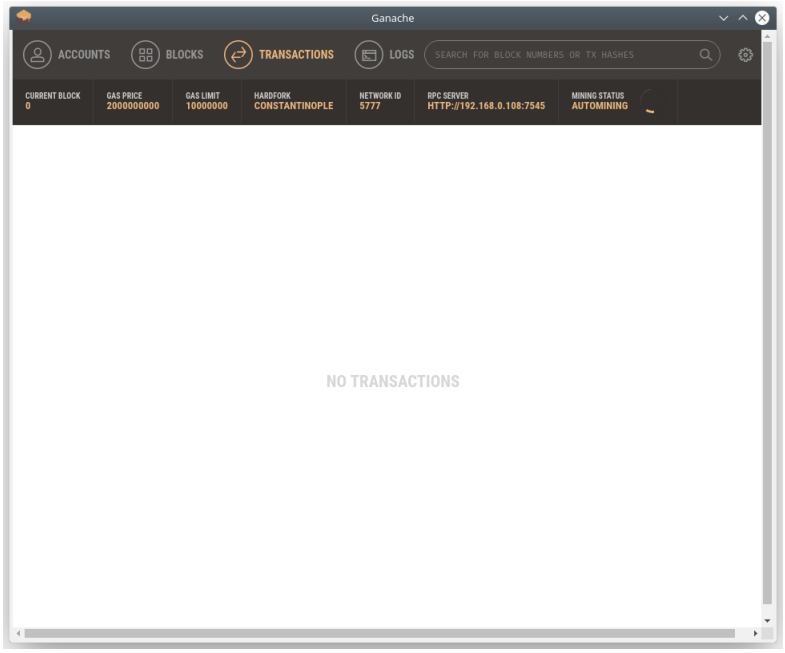


Figure 3.8: The Ganache application at first startup. The list of transactions is empty.

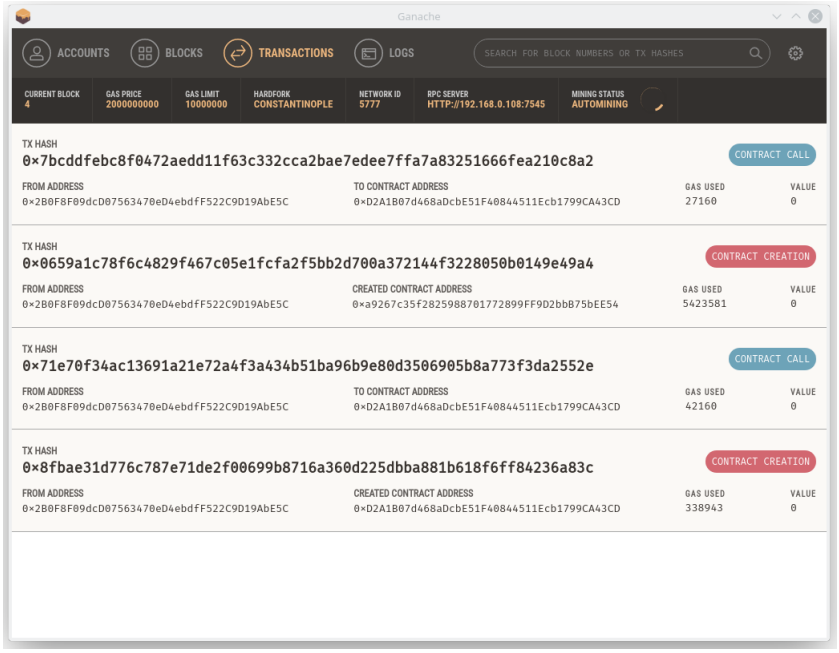
The next step is to compile all of the smart contracts, and deploy them to this local blockchain using the migration script `1_initial_migrations.js`. After that the second migration script `2_deploy_contracts` is run. These processes can be automated by simply running the command `migrate -reset` in the *truffle console*.

```
truffle(development)> migrate --reset
```

After successfully completing, a summary of the deployment costs is displayed on the console:

```
   > gas used:       5423581
   > gas price:      20 gwei
   > value sent:     0 ETH
   > total cost:     0.10847162 ETH



   > Saving migration to chain.
   > Saving artifacts
   -------------------------------------
   > Total cost:     0.10847162 ETH


Summary
=======
> Total deployments:  2
> Final cost:         0.11525048 ETH

truffle(development)>
```

This console is provided by the *Truffle Suite* software. It greatly speeds up the development, and especially the debugging process of Smart Contract applications. As one can see now, the transactions that have been sent to the local blockchain by the `migrate -reset` command appear in the "Transactions" tab of the Ganache application.



Now the application lives on this local blockchain. However, there is still no connection to the Oracle, because the *Oraclize* service requires an additional component, the *Ethereum Bridge*, in order to function correctly. The source for this bridge can be obtained from Github: `https://github.com/oraclize/ethereum-bridge`. The bridge is run by issuing the command `ethereum-bridge -H 192.168.0.108:7545 -a 9 -dev` in the terminal, in the root folder of the ethereum-bridge. The parameters in this command can be described as follows:

- -H specifies the host on which the local blockchain is served
- -a specifies which account to use (in this case the 9th account)
- –dev specifies that development mode should be used

The Ethereum bridge deploys some contracts onto the local blockchain:

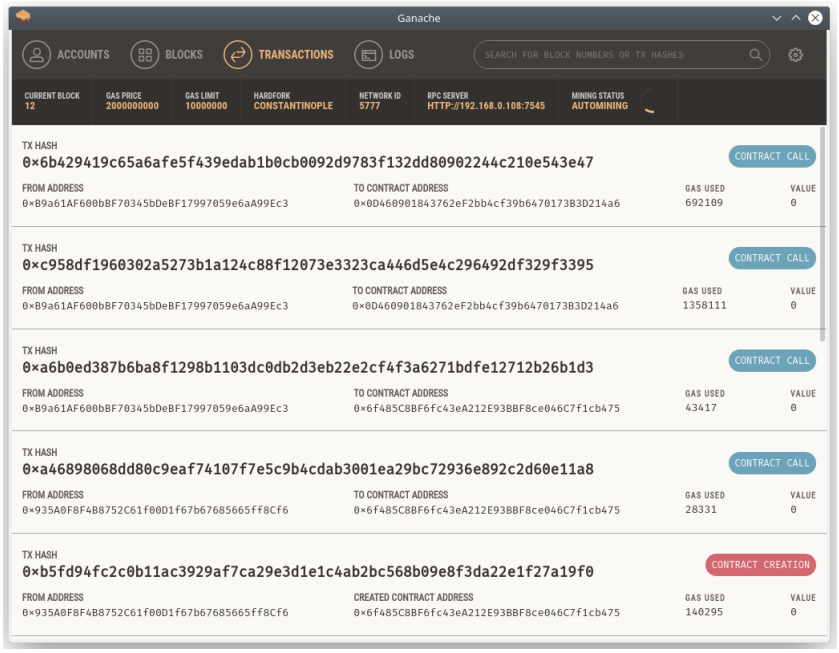


Figure 3.9: The transactions generated by the Ethereum list can be viewed in this list

Then the Oraclize address resolver is provided by the bridge and needs to be inserted into the constructor of the `OracleObserver` contract:

```
Please add this line to your contract constructor:

OAR = OraclizeAddrResolverI(0x6f485C8BF6fc43eA212E93BBF8ce046C7f1cb475);

[2019-03-21T07:01:22.662Z] WARN re-org block listen is disabled
[2019-03-21T07:01:22.662Z] INFO Listening @ 0x0d460901843762ef2bb4cf39b6470173b3d214a6 (Oraclize Connector)
```

Figure 3.10: The Oraclize address resolver from this terminal output needs to be added to the constructor

After that, the setup is completed so far - the Smart Contract can now communicate with the Oraclize service, if the methods are implemented correctly in a contract that inherits from the connector contract. In this project application, that contract is the `OracleObserver` contract. The crucial functions that are responsible for actually fetching the data, and storing it in a variable in the contract are these following two:

```
1  function __callback(bytes32 queryId, string memory result) public {
2    if (msg.sender != oraclize_cbAddress())
3      revert();
4    emit newDataResult(result);
5  }
6
7  function update() public payable {
8    emit newOraclizeQuery("Oraclize query was sent, standing by for the answer..");
9    oraclize_query("URL", "xml(https://www.datasource.com/ws/rest/gameresults).game.
       home");
```

```
10 }
```

Listing 3.1: The implementation of the crucial connector functions in the OracleObserver contract

After these steps, the data can be successfully fetched from the web API, via these connector functions, and stored in a variable residing in the smart contract. Lastly, there needs to be an interface between the frontend (the GUI presented to the user in the browser), and the backend (the application logic that resides in the contracts and the script functions on the server). This is enabled by the *MetaMask* plugin, which manages the Ethereum accounts, and injects them into the web application through the *web3* library.

Chapter 4

Comparing Different Types of Oracles

After examining the different implementations of Blockchain Oracles on the market, one can divide them into the categories of hardware based oracles, and software based oracles. Consensus bases oracles represent any combinations of the former.

4.1 Hardware Oracles

Hardware oracles are those that rely on data directly from the physical world. For instance, sensors that collect data on temperature, humidity, speed, etc. can be put in this category. Furthermore, RFID sensors that send data to a blockchain are a kind of hardware oracle.
Another type of oracle that can be assigned to this category, is one that mainly relies on hardware attestation features, such as Trusted Execution Environments (TEEs). Certain implementations of *Oraclize* can be mentioned here, namely those that use the hardware authentication proofs. Those are:

- Oraclize with Ledger Proof
- Oraclize with Android Proof

4.2 Software Oracles

Software oracles handle information online. Those are Oracle services that provide connectors to web APIs, and various external adapters that can extract information from any kind of data format that is used online, such as JSON, XML, XHTML, etc.
Known examples that can be mentioned here are:

- Oraclize in the basic implementation, as a simple connector to web APIs
- Oraclize using the TLSNotary Proof to guarantee data authenticity
- ChainLink, with its various external modules that can be used for obtaining information from data sources where authentication is needed, or more complex filtering.

4.3 Consensus Based Oracles

Consensus based Oracles are implementations that gather information from more than one Oracle and establish a consensus mechanism. For example, one could use two different Oracle services, fetching data from 5 different sources. Only if all results returned by the services are identical, the data is accepted as valid. But also less strict modes of this kind of model could be used, such as requiring a certain percentage to deliver the same result, etc.
Another concept that can be mentioned here is the Astraea Oracle, described in chapter 3. It is a decentralized version of an oracle that uses a game theoretic approach to solving the problem of establishing truth in a system of participants that do not trust each other.

4.4 Advantages and Disadvantages of these Types

Those different kinds of oracles have certain advantages and disadvantages. One feature that stands out in hardware oracles would be the fact that authenticity proof is provided by secure physical devices. So there is a high level of security involved, as long as the hardware modules on these devices are implemented properly and the devices themselves are always up to date, containing the latest security patches. By using devices from reputable manufacturers such as Ledger or Google, a fairly high level of data integrity can be established. So the data provided by one of these oracles is highly unlikely to have been tampered with.
However, one limitation of these hardware based approaches is, for instance, the limited throughput, as is also mentioned in the official Oraclize documentation [18].
Software oracles, on the other hand, do not have this limitation, they don't rely on special hardware, but on secure software protocols that guarantee data authenticity. TLSNotary is such a protocol, that provides authenticity proof from a software level. The drawback here is that the software protocols can be exploited under certain circumstances, as described by the TLSNotary group in the TLSNotary whitepaper [1].
For this project, a basic, software based oracle is the preferable choice, since it can handle high throughput, and it is more modular than relying on a specific hardware infrastructure. For example, the Oraclize service can be replaced by ChainLink, since on the software level, they both provide very similar interfaces and functionality.

Chapter 5

Conclusion and Discussion

The problem of connecting external data sources to the blockchain can be solved in many ways. There are various types of Oracles available, and they are all suited for different use cases. For web applications that need to query basic APIs that can be reached via HTTP requests, the Oraclize service is a suitable solution.
Other services that support more complex scenarios, such as ChainLink with its external modules may be used in more complex applications, and applications that require authentication to the data source.
For most use cases, a solution already exists on the market. In this way, applications on the blockchain, or so called "Dapps" can become a lot more useful. As these connectors grow in functionality, adoption and variety, one can foresee that those kinds of apps will also see an increase in popularity. There are already a great number of ideas that have been implemented in a basic way out there, but they are not well known or adopted yet. Examples are *DTube*, a decentralized video platform, *Steemit*[12], a decentralized social media platform, *IPFS*, the InterPlanetary File System[4], which is used by D.tube, and many more.
On the contrary side, the disadvantages and shortcomings of blockchain applications are lower throughput than centralized services, and in many cases the user friendliness is not at a satisfactory level yet. As developers become more familiar with these technologies, and users get accustomed to working with those kind of apps, these shortcomings can be overcome. The technology is innovative and disruptive, and, by revolutionizing the way trust works on the internet, clearly has a bright future.

Appendix A

CD-ROM/DVD Contents

```
CD-Content
├── thesis/
│   └── [latex-files]
├── Mitterlehner_BAC2.pdf
├── references/
└── sources/
```

Description of the CD-ROM contents:

- CD-Content: This is the root level. It contains subfolders and the main pdf file, called `Mitterlehner_BAC2.pdf`
- thesis: This is the folder in which all the LaTeXsource files are located.
- references: In this folder, all external sources that have been used in the thesis can be found.
- sources: This folder contains all the application source code for the project.

References

Literature

[1] Dan Smith et al. "TLSNotary - a mechanism for independently audited https sessions". TLS Notary Group, 2014. URL: https://tlsnotary.org/TLSNotary.pdf (cit. on pp. 11, 12, 28).

[2] John Adler et al. "Astraea: A Decentralized Blockchain Oracle". University of Toronto, 2018 (cit. on pp. 19, 20).

[3] Asaph Azaria et al. "Medrec: Using blockchain for medical data access and permission management". In: *Open and Big Data (OBD), International Conference on*. IEEE. 2016, pp. 25–30 (cit. on pp. 3, 6).

[4] Juan Benet. "IPFS - Content Addressed, Versioned, P2P File System". 2019. URL: https://ipfs.io/ipfs/QmR7GSQM93Cx5eAg6a6yRzNde1FQv7uL6X1o4k7zrJa3LX/ipfs.draft3.pdf (cit. on p. 29).

[5] Vitalik Buterin. "Ethereum Yellow Paper". Yellow Paper. 2013. URL: https://ethereum.github.io/yellowpaper/paper.pdf (cit. on pp. 2–4).

[6] Usman Chohan. "The Leisures of Blockchains: Exploratory Analysis". University of New South Wales, 2017. eprint: SSRN papers. URL: https://papers.ssrn.com/sol3/papers.cfm?abstract_id=3084411 (cit. on p. 4).

[7] Kevin Delmolino et al. "Step by step towards creating a safe smart contract: Lessons and insights from a cryptocurrency lab". Springer, 2016, pp. 79–94 (cit. on p. 4).

[8] Samuel Falkon. "The Story of the DAO - Its History and Consequences". *Medium* (2017). URL: https://medium.com/swlh/the-story-of-the-dao-its-history-and-consequences-71e6a8a551ee (cit. on p. 5).

[9] Loi Luu et al. "Making smart contracts smarter". National University of Singapore, 2016, pp. 254–269 (cit. on pp. 4, 5, 7).

[10] Luka Perović Martin Mikeln. "Verity: Platform for Decentralized Real-World Data Feeds". 2018. URL: http://verity.network/whitepaper.pdf (cit. on pp. 17–19).

[11] Satoshi Nakamoto. "Bitcoin: A Peer-to-Peer Electronic Cash System". White Paper. 2009. URL: https://bitcoin.org/bitcoin.pdf (cit. on p. 2).

[12] "Steem - An incentivized, blockchain-based, public content platform." 2018. URL: https://steem.com/steem-whitepaper.pdf (cit. on p. 29).

[13] Fan Zhang et al. “Town Crier: An Authenticated Data Feed for Smart Contracts”. Cornell University, 2016. eprint: IACR ePrint (cit. on p. 10).

Online sources

[14] Dan Smith et al. *PageSigner*. 2019. URL: https://tlsnotary.org/pagesigner.html (cit. on p. 11).

[15] Luu et al. *Oyente Github*. 2019. URL: https://github.com/melonproject/oyente (cit. on p. 7).

[16] ChainLink. *Chainlink Documentation*. 2019. URL: https://docs.chain.link/docs/ (cit. on pp. 21, 22).

[17] Oraclize. *Android Proof: Authenticated Data Gathering using Android Hardware Attestation and SafetyNet*. 2018. URL: https://provable.xyz/papers/android_proof-rev2.pdf (cit. on pp. 13, 15).

[18] Oraclize. *Oraclize Documentation*. 2019. URL: https://docs.oraclize.it/ (cit. on pp. 10, 11, 16, 28).

[19] Oraclize. *proof-verification-tool*. 2019. URL: https://github.com/oraclize/proof-verification-tool (cit. on p. 16).

[20] Swarm. *Swarm Homepage*. 2019. URL: https://ethersphere.github.io/swarm-home/ (cit. on p. 18).

[21] Orisi Team. *Orisi White Paper*. 2019. URL: https://github.com/orisi/wiki/wiki/Orisi-White-Paper (cit. on p. 10).