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 Oktober 2018

This thesis was created as part of the course

Secure Mobile Systems

during

Fall Semester 2018

Advisor:

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, October 23, 2018

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

In this work we 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.

We solve this problem by adding an additional contract to the app, which we will call "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 [2]. 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 [7].

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 [7, 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 [2, 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 ca 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* [1]. 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 [4].

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* [2, 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 [3], 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* [6, 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 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 [5]. At fault was the *reentrancy handling* bug. The DAO was a decentralized

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 [6].

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 we will briefly explore in this section. One example is *MedRec* a system for storing and managing medical data on the blockchain [1].

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 we have already 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 we need to focus 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. We will explore one of these tools, *Oyente*, 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"[6]. 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 [8].

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. We can download the image 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/`. We 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 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 our Project

To ensure the security of the smart contracts powering our cryptocurrency sportsbetting application, we make use of the aforementioned scanning tool. 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 Dependency* or *Re-Entrancy Vulnerability*. However the scanning tool produced 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. However, some of the positive results 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 our 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, we need 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. 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.

References

Literature

[1] 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).

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

[3] Usman Chohan. “The Leisures of Blockchains: Exploratory Analysis” (2017) (cit. on p. 4).

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

[5] Samuel Falkon. “The Story of the DAO - Its History and Consequences” (2017). URL: https://medium.com/swlh/the-story-of-the-dao-its-history-and-consequences-71e6a8a551ee (cit. on p. 4).

[6] Loi Luu et al. “Making smart contracts smarter”. 2016, pp. 254–269 (cit. on pp. 4, 5, 7).

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

Online sources

[8] Luu et al. *Oyente Github*. 2018. URL: https://github.com/melonproject/oyente (cit. on p. 7).