Institute of Parallel and Distributed Systems

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Bachelorarbeit

Design and Implementation of Secure Smart Contracts for Mobile Target Tracking Applications

Ali Salaheddine

Course of Study:	Informatik
Examiner:	Prof. Dr. Kurt Rothermel
Supervisor:	Dr. Frank Dürr
Commenced:	2019-12-17
Completed:	2020-08-12

Abstract

In recent years, cryptocurrencies implemented on top of Blockchains became very popular, with Bitcoin as the most prominent example. However, novel Blockchain-based platforms such as Ethereum also support distributed applications beyond cryptocurrencies through so-called smart contracts. Technically, smart contracts are programs, whose code and execution state is stored in the Blockchain, inherently featuring the ability to transfer (electronic) money during their execution.

In this Bachelor thesis, we investigate how smart contracts can be used to implement a distributed crowdsensing application for tracking mobile objects by a crowd of privately owned mobile devices. Such a system could be used, for instance, to find lost or stolen objects, such as keys, vehicles (cars, bicycles, . . .), or pets tagged with short-range radio transmitters implemented using readily available Bluetooth or RFID technology. These objects can then be detected by smartphones of private users in the vicinity of the object, effectively implementing a huge sensor network covering many parts of the world without any upfront investments by a central entity.

Although highly attractive, implementing a crowdsensing application on top of a Blockchain platform such as Ethereum comes with several challenges. First of all, users need incentives to participate in searching for mobile objects. A natural incentive is a monetary reward that participants automatically receive through the smart contract when reporting sightings (timestamped positions) of wanted objects. However, this directly brings up the problem of malicious participants (attackers) who try to get the reward without actually executing the work of searching for the object by simply reporting fake positions. Therefore, one major goal of this Bachelor thesis is to counter such attacks by proposing effective counter-measures, and implementing and evaluating them for the Ethereum platform. In detail, we propose a basic reputation-based approach for detecting fake positions which judges each sighting made by a mobile devices according to the reputation of that device, implemented by a smart contract. Furthermore, advanced attacks are identified compromising the basic reputation-based approach and effective counter-measures to these advanced attacks are proposed. Identified advanced attacks include reputation farming, where the attacker tries to aggregate reputation first before launching the attack, and the so-called copy cat attack, where the attacker simply copies already submitted valid sightings form honest participants, making his fake positions indistinguishable from valid positions.

Our evaluations analyses the monetary cost of executing smart contracts with and without our security mechanisms. The results show that the overhead included by our reputation-based approach is at maximum 45% of the cost of a smart contract without implemented security mechanisms.

Kurzfassung

In den letzten Jahren wurden Kryptowährungen, die auf Blockchains basieren, sehr populär, mit Bitcoin als prominentestem Beispiel. Neuartige Blockchain-basierte Plattformen, wie Ethereum, unterstützen jedoch auch verteilte Anwendungen jenseits von Kryptowährungen durch so genannte Smart Contracts. Technisch gesehen handelt es sich bei Smart Contracts um Programme, deren Code und Ausführungszustand in der Blockchain gespeichert wird und die inhärent die Fähigkeit besitzen, während ihrer Ausführung (elektronisches) Geld zu transferieren.

In dieser Bachelorarbeit wird untersucht, wie Smart Contracts dazu verwendet werden können, eine verteilte Crowdsensing-Anwendung zur Verfolgung mobiler Objekte durch eine Menge privater mobiler Geräte zu implementieren. Ein solches System könnte z.B. dazu verwendet werden, verlorene oder gestohlene Gegenstände wie Schlüssel, Fahrzeuge (Autos, Fahrräder, ...) oder Haustiere zu finden, die mithilfe von leicht verfügbaren Bluetooth- oder RFID-Technologie implementierten Funksendern ausgestattet sind. Diese Gegenstände können dann von Smartphones privater Nutzer in der Nähe des Objekts erkannt werden, wodurch ein riesiges Sensornetzwerk entsteht, das viele Teile der Welt abdeckt und ohne Vorabinvestitionen durch eine zentrale Entität aufgesetzt werden kann.

Obwohl die Implementierung einer Crowdsensing-Anwendung auf einer Blockchain-Plattform wie Ethereum attraktiv ist, bringt sie auch einige Herausforderungen mit sich. Zunächst benötigen die Benutzer Anreize, sich an der Suche nach mobilen Objekten zu beteiligen. Ein natürlicher Anreiz ist eine monetäre Belohnung, die die Teilnehmer durch den Smart Contract automatisch erhalten, wenn sie Sichtungen (Positionen mit einem Zeitstempel) gesuchter Objekte melden. Dies wirft jedoch direkt das Problem böswilliger Teilnehmer (Angreifer) auf, die versuchen, die Belohnung zu erhalten, ohne den Aufwand der Suche nach dem Objekt tatsächlich nachzugehen, indem sie einfach falsche Positionen melden. Ein Ziel dieser Bachelorarbeit ist es daher, solchen Angriffen durch den Entwurf wirksamer Gegenmaßnahmen zu begegnen und diese für die Ethereum-Plattform zu implementieren und auszuwerten. Im Einzelnen schlagen wir einen grundlegenden, durch einen Smart Contract implementierten reputationsbasierten Ansatz für die Erkennung von gefälschten Positionen vor, der jede Sichtung durch ein mobiles Gerät nach der Reputation dieses Geräts beurteilt. Darüber hinaus werden fortgeschrittene Angriffe identifiziert, die den grundlegenden reputationsbasierten Ansatz gefährden, sowie wirksame Gegenmaßnahmen gegen diese fortgeschrittenen Angriffe vorgeschlagen. Zu den identifizierten fortgeschrittenen Angriffen gehören so genannte Reputation-Farming-Angriffe, bei denen der Angreifer zuerst versucht, Reputation zu aggregieren, bevor er den Angriff startet. Des weiteren wird der so genannte Copy-Cat-Angriff identifiziert und behandelt, bei dem der Angreifer bereits eingereichte gültige Sichtungen von ehrlichen Teilnehmern kopiert, so dass seine gefälschten Positionen nicht von gültigen Positionen unterschieden werden können.

Unsere Bewertung analysiert die monetären Kosten der Ausführung von Smart Contracts mit und ohne unsere Sicherheitsmechanismen. Die Ergebnisse zeigen, dass die von unserem reputationsbasierten Ansatz verursachten Mehrkosten maximal 45% betragen.

Contents

Chapter 1

Introduction

In the last several years, cryptocurrencies have gained in popularity not only in the IT domain but also all other parts of society: with market capitalization reaching hundreds of billions of dollars [Mar19], big corporations starting to invest in their own cryptocurrencies [Scr19], governments starting to find regulations for these currencies [Sta18], and the debate whether cryptocurrencies will replace traditional currencies in the future, cryptocurrencies have become a promising technology [Kra18].

Over many years there have been many attempts in achieving a digital currency but many of them had the problem of relying on a centralized intermediary or on trusted computing, and if an approach overcame those boundaries, it had a lack of details on how to implement them [But20]. These problems where solved in 2009, when the cryptocurrency Bitcoin was released, utilizing cryptography principles and combining them with consensus algorithms. This made Bitcoin the first successful cryptocurrency [But20]. It uses a novel concept called "Blockchain" combined with a "Proof of Work" consensus algorithm.

What makes cryptocurrencies in the form we have them today so appealing? Cryptocurrencies generally have the properties of being decentralized and self-governing while offering a high level of security. They offer direct end-to-end transactions without any intermediary involved, resulting in an anonymous transaction system, which in a time where privacy is becoming more and more important is very appealing [But20]. Furthermore, the underlying technology of cryptocurrencies finds more and more applications in fields that exceed financial use cases such as supply-chain management, Internet of Things, and e-Voting [RH18].

Out of these promising advantages, the rise in popularity, and the innovative underlying concept, many new cryptocurrencies evolved [But20], each trying to evolve further from Bitcoin in its unique direction [YY18]. One of these directions is to enable the programmability inside a cryptocurrency. This extended cryptocurrencies from just being currencies to platforms, for which code can be written.

This allowed for the ability to create decentralized applications which run on top of a cryptocurrency. This is achieved through a concept referred to as Smart Contracts, integrated into such programmable cryptocurrencies. To this end, the code to run the application is stored in so-called Smart Contracts, which then are used to implement the logic of the application through a contract system. The advantage of such applications created through Smart Contracts is that they inherit the properties and advantages of cryptocurrencies. This also includes the immutability of the code written, transparency of the values calculated, and the decentralised nature of the application, which is beneficial for applications were privacy is important. Furthermore, Smart Contracts have a very simple and easy to implement specification allowing people with little experience in programming to develop decentralized applications. This means that cryptocurrencies are no longer limited to trading. Rather they can be used to create different types of decentralized applications, which can range from managing financial derivatives to running games [Kra18], [Gha20].

With these promising properties, and the hype surrounding them, it is important to investigate what applications are possible, what are considerations to make when creating such applications, and which limitations exist. Therefore, in this work, we consider a specific application on top of a smart contract system.

Each year the number of mobile devices and the number of sensors on each mobile devices is increasing. By utilising the huge crowd of mobile devices one can setup a huge sensor network, where normally setting up a dedicated sensor network would be much more expensive. With many people walking around with a smartphone equipped with sensors, we aim for a system where the crowd of smartphones can be used for mobile target tracking. This decentralized mobile target tracking system, which utilises the huge number of available smartphones as a giant sensor network, can be used to find lost or stolen objects, track pets wearing tags, or any other kind of mobile devices that are tagged using, for instance, Bluetooth or RFID tags. The collected data is processed by a Smart Contract which calculates a trace of the mobile device from the data collected. In return the tracking mobile devices get rewards in term of a cryptocurrency as an incentive.

Combining mobile crowd sensing into a decentralized application creates different types of security challenges and risks. One of them being sybil attacks where the attacker creates multiple malicious accounts in order to overrule or influence the contributions made by honest accounts. One major focus of this theses is to tackle this problem. In our work, we will contribute possible algorithms and methods to solve these. To this end, we propose an approach, where every account has something at stake and therefore discourage malicious behaviour in a fear of loosing that stake. This is achieved though a reputation-based system in which every account puts his reputation at stake to penalise malicious behaviour in order to minimize the effects of malicious behaviour. We will discuss possible exploitations of this approach and how to deal with them. Moreover, we will evaluate the overhead included by our approach, "the price of security", which directly translates into monetary cost for executing smart contracts.

As proof of concept, we implemented our concepts for the Ethereum Smart Contract system for two main reasons. On the one hand, it is currently one of the largest and most commonly known cryptocurrencies [Com19]. On the other hand it has one of the most active cryptocurrency communities [Com19], which determines the long-term success of a platform.

This thesis is structured as follows:

- In **Chapter 2**, the important background information are explained in order to understand the Ethereum blockchain and the Smart Contract concept. Furthermore, the chapter presents state-of-the-art mobile crowdsensing architectures.

- In **Chapter 3**, the current state of our mobile target tracking application is covered. After that, the goal of adding security for our application will be formalized.

- In **Approach 4**, the approach taken to realize the target tracking application and security through a reputation-based approach is presented.

- In **Chapter 5**, the specification for the realization of the target tracking application with and without security is presented. After that, advanced security threats to the presented target tracking application with security are highlighted, followed by discussing possible countmeasures to the advanced threats.

- In **Chapter 6**, challenging parts of the implementation of the previously presented approaches are highlighted.
- In **Chapter 7**, the results for measuring the monetary cost of the target tracking approach with and without security are presented.
- In **Chapter 8**, the summary of the thesis in addition to its results and suggestions for future directions are concluded.

Chapter 2

Background and Related Work

In this chapter, we will explain the background information needed. To this end, we will first explain the blockchain fundamentals followed by an explanation of Smart Contracts. Afterwards, we discuss related work from the fields of mobile crowdsensing.

2.1 Blockchain Fundamentals

In the following section, we will introduce the general concept of the Blockchain used by cryptocurrencies, since it builds the foundation for the decentralized nature of the approach created in this work.

2.1.1 Currency Decentralization vs. Centralization

In order to understand the basic design ideas for an independent digital currency, we need to understand how traditional transactions work and what is necessary for a shift to a decentralized currency approach.

In a traditional centralized banking system, in order to send money from one person to another, a trusted institution is needed to maintain all the necessary information including a log of all accounts with the corresponding owners and balances. If a specific owner wants to transfer money from its account to another one, the bank needs to check the validity of the transaction. The bank maintains the global state of all accounts, and through transactions it updates the balances, which leads to a new state, as shown in Figure 2.1. This can be done by the bank since every owner of an account trusts the bank to behave legitimately. Why do financial institutions track what the consistent states at a given moment are? This needs to be done for several reasons. They need to know what money an account has spent and what is still in his possession. This is important in order to prevent the same money from being spent twice ("double spending") or to check whether an account spends money that it does not have. This is implemented in banking systems by the following concept. The balances at one given moment and the ownership of these balances make up the current state. If then a transaction takes place and money is being transferred from one account to another the state changes. So financial institutions maintain state transition systems [But20, Kas17].

Now in order to decentralize a currency we do not have a financial institution in between, so we need to use a direct transaction system. The system gets constituted of connected entities referred to as nodes. Each node in the network has its own copy of the global state stored locally. If new

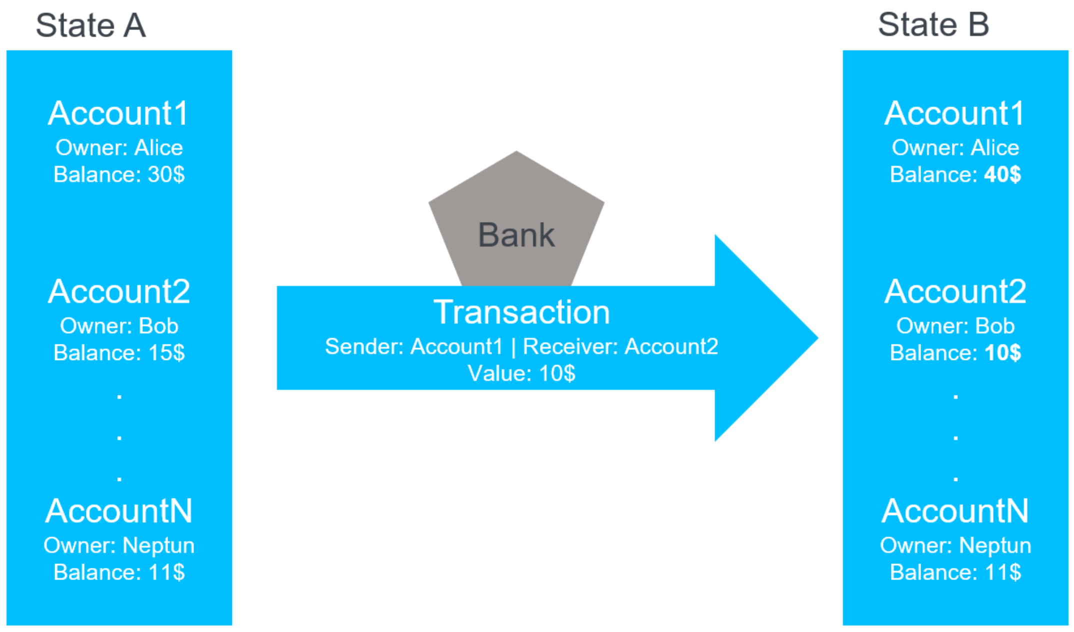


Figure 2.1: States and Transactions

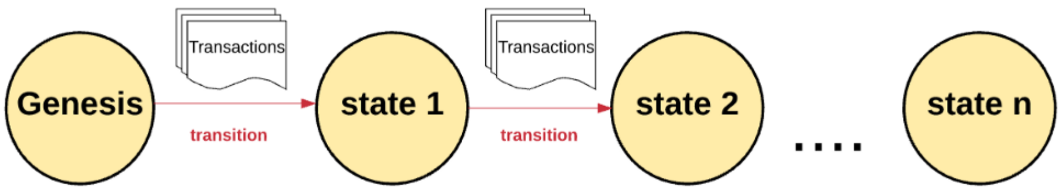


Figure 2.2: State Transition System
[Kas17]

transactions happen and the state changes, the changes get broadcasted to every node in the network to update their local state after verifying it [Woo20], [But20].

2.1.2 Blockchain Technology

The decentralization concept is implemented in the Blockchain technology, which is used by cryptocurrencies. So cryptocurrencies are basically state transition systems. Therefore, we need to understand how this state transition system is implemented in the Blockchain concept.

We start by explaining how the Blockchain concept handles transactions since there is no central intermediary doing this processing. After issuing a transaction it needs to be validated. This means checking whether the sender really owns the money (in the Blockchain context often referred to as coins) he tries to send. So after checking whether the money has not been spent yet, the other nodes need to get notified that this transaction took place. This all is handled by so-called miners. The miners are nodes in the network whose only task is to process transactions that took place. In return they get rewarded for every transaction they successfully processed.

Therefore, let us have a look at how transactions get handled in the Blockchain technology. It starts with one node deciding to make a transaction. A transaction can have different goals like a transfer of money or an exchange of information. A node is defined by its unique address. If a node wants to make a transaction to another node, it needs to address the other node. The network is built as a network of peers where each node knows the path to its neighbours referred to as peers. This means no single node knows the path to every node of the network. When sending a transaction, it gets broadcasted to its peers, and the peers broadcast the transaction again to their respective peers until everyone in the network received the transaction. Upon receiving, only special nodes in the network, the miners, put the transaction into a local pool of unverified transactions. Then these miners take several transactions from their mining pool and begin to verify these transactions. The verification consists of verifying, that the party issuing the transaction is not trying to spend money which has already been spent. After these transactions have been verified, they get compounded into a block [But20, Kas17, RH18].

Since we have a distributed network, a miner has to share with the other nodes that this transaction has been verified. But the goal was to create a trustless distributed system. Therefore, a consensus on who owns the money is necessary since someone could send the same money in different transactions. So miners must come to an agreement which transactions are valid. Therefore, the miners have to run a consensus algorithm, and the result of this consensus algorithm is a block. Which consensus algorithm the miners run depends on the Blockchain implementation [Sai18a]. Every node in the network can become a miner by running the consensus algorithm to create blocks. The miner receives a reward for mining a block that got added to the Blockchain.

The most prominent Blockchains run a consensus algorithm called "Proof of Work" where miners try to solve hard mathematical puzzles in order to proof that the miner had to put effort in creating the block. But there are several other different consensus algorithms being used by different currencies [RH18, Sai18a].

The block contains the transactions associated with it and depending on the implementation of the Blockchain, it can contain additional information. So basically blocks can be thought of as a record of verified transactions. In Blockchains, every new Block has to reference the block that has been created before this block. By creating a chain of block references where each new block references the previous block, nodes can reconstruct the momentary state, because this chain of blocks gives the order in which all transactions have taken place from the beginning. This reconstruction can be done implicitly or explicitly depending on the Blockchain implementation. Due to this chaining of blocks this technology got named Blockchain [But20, Kas17].

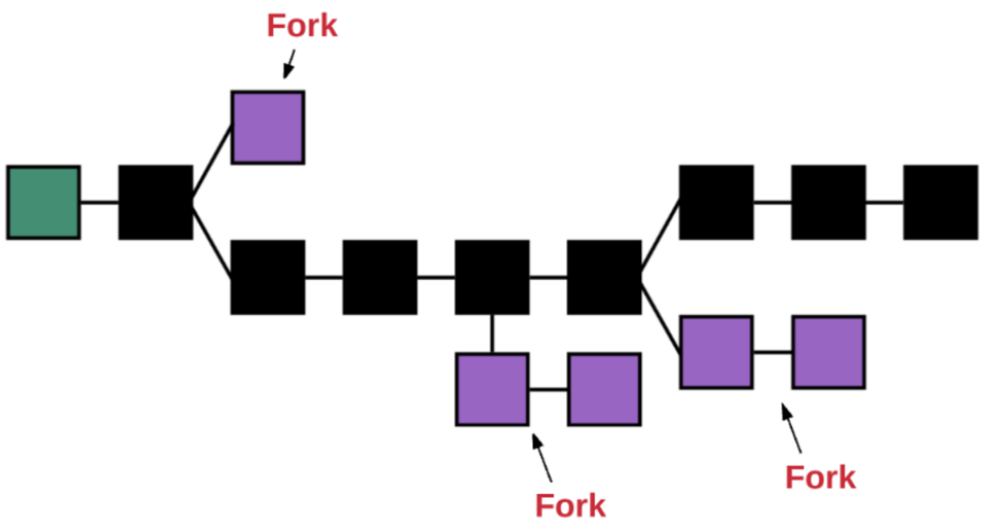


Figure 2.3: Blockchain with Forks
[Kas17]

The block added to the chain including all verified transactions, gets broadcasted, and upon receiving nodes run a simple verification process to check the legitimacy of this block. It checks whether the reference was correct and if the reference is wrong the block is ignored. After a successful verification, the block gets appended to the nodes local version of the Blockchain. So each node can constantly update the current global state without needing a central intermediary. Only the transactions that are part of blocks on the long chain count as verified [Kas17, Woo20].

If several miners reference the same block, the Blockchain will get "forked" as seen in Figure 2.3. Many Blockchains follow the rule that new blocks get appended to the longest chain with the most blocks. And transactions which are part of forked blocks that are not part of the main chain become broadcasted again in order to become mined again. That is why a transaction has to be part of the chain for several blocks before being sure that it will stay in the chain. Because the probability of a block that has several blocks after it getting changed is relatively low [But20].

2.2 Ethereum

As we have seen, Blockchains are decentralized, distributed, and public digital records, which are often used to build cryptocurrencies on top of them.

The first prominent cryptocurrency, Bitcoin, was mainly used to buy and sell items and services where anonymity was very important. Just in recent years cryptocurrencies started to find their way to different applications where privacy is not the primary focus. Furthermore, the key Blockchain architecture starts to be used in different use cases. Nowadays there are thousands of cryptocurrencies based on different goals. Figure 2.4 divides the goals of cryptocurrencies in six dimensions. Each currency can be used for a specific use case [YY18].

One dimension of cryptocurrencies, including Ethereum as the most popular example, offer a platform to create decentralized applications (dApps) which can autonomously run on the Blockchain.

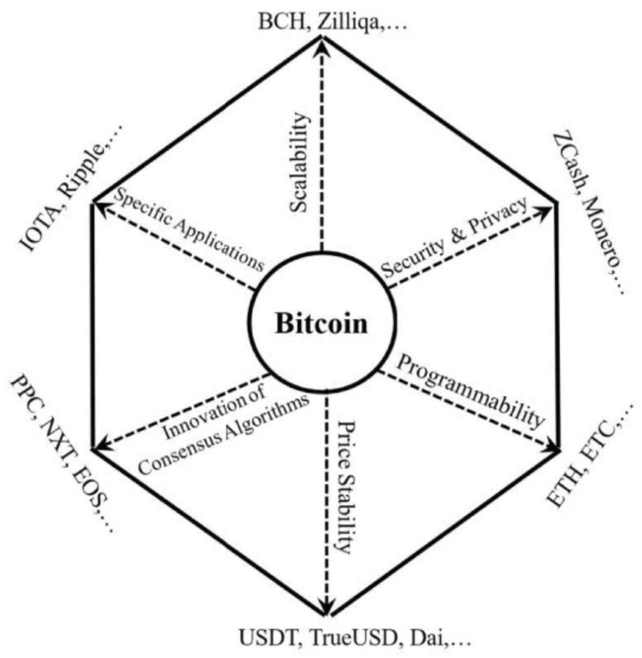


Figure 2.4: Six Dimensions of Cryptocurrencies
[YY18]

This means you can implement code, referred to as Smart Contract that gets stored in the Blockchain, which then the dApp uses to connect to the Blockchain. A specific property of Ethereum is to offer a platform to implement code, which then runs on top of the Ethereum Blockchain. Today it is one of the most important cryptocurrencies and has found widespread use for implementing the code which dApps use.

Ethereum was released in 2015 in a time where cryptocurrencies, especially Bitcoin, started to gain popularity. The founders of Ethereum noticed that many implementations of Blockchains did not support the implementation of applications on top of them [But20, Vit17]. In the following we will analyse what makes Ethereum the most prominent programmable Blockchain.

2.2.1 Ether

In order to have an independent digital currency, these cryptocurrencies offer their own currency often referred to as coins. Ether is the native currency offered by the Ethereum cryptocurrency. It can be used just like other currencies to transfer money and has to be used to pay for transaction fees.

2.2.2 State

As stated before, Blockchain-based cryptocurrencies need to maintain the global state (see Figure 2.1 & 2.2) in order to track who owns how much coins. In Ethereum the global state is made up of many small objects named accounts, and values are stored by these accounts. Accounts can be of two types: external accounts, which are controlled by human users through a private key, and contract accounts. Contract accounts contain and are controlled through code. Accounts possess a unique 20 bytes address to be globally identified and addressed. Each account contains four different fields:

- **nonce**: a positive integer value which increases by one with every transaction being sent from this account; it is needed to ensure that every transaction gets sent only once.
- **balance**: a positive integer value representing the Ether balance of the corresponding account stored in Wei (10^{18} Wei = 1 Ether).
- **codeHash**: for contract accounts the code hash contains a hash of the contract code; for external accounts it contains a hash of an empty string.
- **storageRoot**: the hash of the root node of the storage tree which includes the stored content of the account; empty by default.

An external account can use the address of other accounts to make transactions with other accounts. Possible transactions with external accounts are simple value transfers but transactions with contract accounts trigger the contract code, which can lead to the contract executing special operations like creating new contracts sending Ether to other accounts or store some Ether. It should be noted that contract accounts cannot initiate transactions but can send transactions if they have received one before. So basically a contract account is used to address the corresponding contract [Woo20, Kas17].

2.2.3 Transactions

Transactions have a central role in every Blockchain based cryptocurrency. As seen earlier, Ethereum is a transaction-based state-transition system. Like in other Cryptocurrencies, transactions cause the global state to be changed from one state to a new one. Transactions in Ethereum are data packages that contain information for the receiver, which gets deployed on the Blockchain by a miner. Every transaction gets signed and serialized. As explained in [Woo20], there are two types of transactions: "Message calls" and "Contract creations". Both consist of the following fields that need to be declared by the sender:

- **to**: the 20 byte address of the receiver of the transaction.
- **value**: if transaction is a message call, the value field defines the amount of Ether (in Wei) to transfer to the receiver; if transaction is a contract creation, the value field defines the amount of Ether (in Wei) the contract starts with.
- **nonce**: number of transactions the sender has sent; increases by one for every transaction sent; in order to track, which transaction was sent last
- **gasPrice**: Ether (in Wei) to be paid for a unit of gas; gas is the unit for the cost per computation step in a smart contract
- **gasLimit**: maximum amount of gas to be used by the transaction and all sub-transaction initiated by this transactions in order to limit the runtime of a smart contract.
- **v, r, s**: values Ethereum uses for determining who sent the transaction.

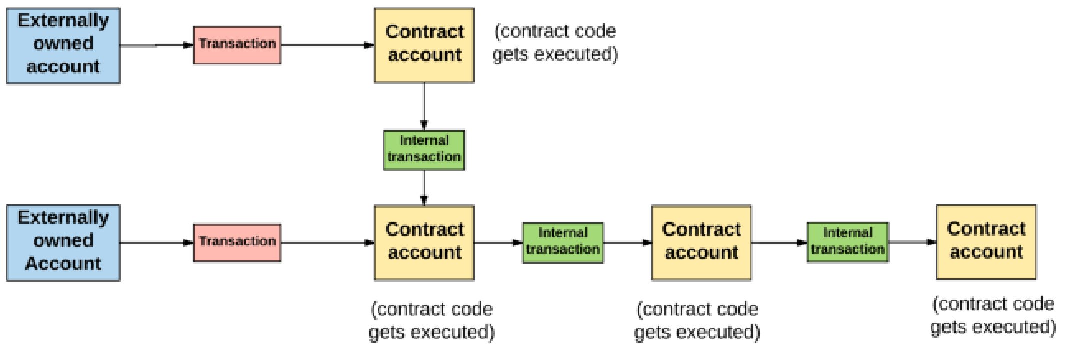


Figure 2.5: Ethereum Accounts
[Kas17]

Contract creation transactions need additionally an **init** field containing the code of the contract that gets executed if the contract receives message calls. Message calls need additionally a **data** field containing data which contracts can use to update their status (see Section 2.3).
As discussed earlier in the account section, transactions interact differently with external and contract accounts. Therefore, let us have a look at which accounts can send which type of transactions.

As seen in Figure 2.5, external accounts can use their private key to send transactions

- either to other external accounts, which is basically just a transfer of Ether,
- or they can use message calls to send data or Ether to contract accounts, which triggers the implemented logic.

Contract accounts can create and send contract creation transactions to other contracts but they can do this only re-actively. This means they have to be created before by an external account.
Contracts can also send messages between each other. Messages are often referred to as internal transactions since they happen only inside the Blockchain. But since messages only function between contracts they do not have a gas limit. The gas limit is set by the transaction triggered by the external account. The external account has to set the gas limit high enough so all messages have enough gas.

2.2.4 Gas

Another important concept used in Ethereum is the concept of fees. Every computation that takes place in a transaction costs a certain fee. The unit of the fee is called gas. Every particular computation costs a certain amount of gas. Additionally, if one transaction gets created the initiator must set a specific gas price. The gas price is the amount of Ether he is willing to pay for every gas used, measured in wei. For every transaction, a sender has to set a gas limit and a gas price. The gas limit defines the maximum amount of gas the transaction is allowed to use. By multiplying gas price with gas limit one can derive the maximum transaction fee. This can look like the following. The sender sets the gas limit to 10000 gas and the gas price to 200 Gwei (10^9 Gwei = 1 Ether). Then multiplying them the sender knows she has to pay 0.002 Ether for that transaction. The sender must own this money when sending the transaction otherwise the transaction will not be executed. Consequently all the gas that does not get used is refunded to the sender [Woo20, But20, Kas17].

The gas goes to the miner as a reward for mining the transaction. This is the only way for a miner to earn Ether in Ethereum apart from receiving Ether through transactions. By setting the gas price higher, the miners receive more Ether as a reward for mining this transaction, which leads to the transaction getting verified faster. Otherwise, if the gas price is too low, transactions can starve (never get mined) [Doe17]. Miners generally have a minimum gas price for a transaction that can be looked up. But not just computation has to be paid, storage usage too. The fee is proportional to to the smallest multiple of 32 bytes. This was introduced in order to keep the size of the Ethereum Blockchain small [Kas17]. This concept of gas shows the necessity of writing time and space efficient code since inefficiency leads to the price of running the dApp to increase drastically, which then can lead to the failure of the application.

Ethereum uses this concept of fees, for two major reasons:

- There are nodes in the Ethereum network that have to process every transaction in the entire Blockchain from the first to the newest block. These nodes are referred to as full nodes. Miners for instance are these nodes. They have to do this because the mining process requires them to do so. Since they have to put such a huge effort in running the mining procedure, Ethereum discourages computation- and storage-heavy transactions by using the fee concept. This keeps the Ethereum Blockchain smaller and makes mining faster and more efficient [But20, Kas17].

- The other important reason why fees are necessary is to handle Turing-complete programs that run on the Blockchain. As said before, Ethereum offers the ability to implement programs. To this end, Ethereum offers a Turing-complete programming language. Since it is Turing-complete it supports infinite loops. When a program with an infinite loop is executed on the Blockchain, it could disrupt the entire Network. But since every transaction has to set a gas limit, it limits how long a program can run in the Blockchain. When it has used up all its gas, the transaction gets reverted and the program stops running [Woo20, Kas17].

2.3 Smart Contracts

Ethereum was designed to be able to implement decentralized applications on top of Ethereums Blockchain easily and this is implemented by the "Smart Contract" concept. Decentralized applications can range from managing financial derivatives to running games [But20, Gha20].

In order to understand the Smart Contract concept, we will have a look at the following example of a decentralized application based on Smart Contracts. In this example we want to implement a decentralized crowdfunding application. To this end, the Smart Contract would need to be defined as follows. The initiator of the crowdfunding would define the amount of money he would like to receive and a time limit for the funding inside the contract. The contract could then be addressed to receive money. It collects the money and if the contract received the aspired amount of money defined by the crowdfunding initiator in the specified amount of time, the fundraiser receives the money from the contract. Otherwise, the money is returned to the people who sent the money to the contract. Since the contract runs on the decentralized Blockchain, we are able to implement the application without needing a centralized intermediary managing the collection and distribution tasks.

In the following, we will show how Smart Contracts function on the Blockchain. Smart contracts are most commonly written in a higher level languages such as Solidity and Serpent [Mor20]. The application in this thesis is written in Solidity. We will not go into further details for Solidity because it is not necessary for understanding this thesis. Ethereum has its own virtual machine called the Ethereum Virtual Machine (EVM) using EVM bytecode. After the code is written, a contract creation transaction is sent to a contract account. Now the transaction gets mined and inserted into a block. This includes storing the contract in the storage of the associated contract account. Afterwards, the block gets appended to the Blockchain. Then, every node which downloads and validates the updated Blockchain containing the new blocks runs the contract on the local Blockchain, because every node is required to do so by the block validation algorithm. So physically the code runs on every node. Every node is also able to see the code written since it is now stored visibly in bytecode inside the Blockchain on their machine. This is important to remember because it will lead to problems in our approach with respect to security and privacy, see Section 7. Contracts are deterministic: this means for everyone who executes the code, the resulting outcome will be the same. Now the contract can be addressed to receive Ether or messages in order to fulfil their logic [Mor20, Kas17].

2.4 Related Work

The designed application in this work is a mobile crowdsensing application, and therefore, in the following we will show the spectrum of challenges and ideas, and additionally show the findings made by related research handling these.

2.4.1 Mobile Crowdsensing

In order to build a common foundation on mobile crowdsensing we start off by an explanation of why and how mobile crowdsensing concept is designed.

The number of actively used mobile devices is constantly increasing and additionally the number of sensors on mobile devices are increasing and performing better [GYL11], [LSZ16]. Used for improving user experience on mobile devices, the field of mobile crowdsensing emerged for researching how this sensing power could be used as a collective for different types of applications [LYL^{+}18]. In [LSZ16], three different types of application for mobile crowdsensing are suggested: environmental, infrastructural, and social applications. All these types generally attempt to improve the quality of experience. In environmental applications the sensing mobile devices are used to track natural phenomena like noise or air pollution in an area. Whereas infrastructure applications tend to be looking for the proximity between mobile devices, which can be used for tracking of mobile devices for lost and found scenarios or traffic control like rerouteing traffic and temporary speed limits. Social applications of mobile crowdsensing could be catering recommendations or social networks [XJX^{+}18], [WWDR11], [LSZ16].

With all these promising approaches and the approach presented in this work, it is important to understand the concept of mobile crowdsensing.

Mobile crowdsensing includes four different general roles:

1. A typically cloud-based mobile crowdsensing platform implemented by servers
2. Access Points typically Wifi or Cellular Networks, to connect mobile devices to the Internet and crowdsensing platform
3. Mobile device users the so-called participants
4. The creator of the mobile crowdsensing job, the so-called task initiator

[XJX^{+}18], [LYL^{+}18], [WYJ^{+}19], [LSZ16] describe the process of mobile crowdsensing similarly. The task initiator submits the sensing job to the mobile crowdsensing platform, and pays a certain price for the completion of the job. Then, the platform recruits participants for the sensing job. The platform manages the data exchange between the sensing devices. As participants in the sensing job generally get paid too, the platform additionally makes sure all incentives for the participants are distributed as defined by the task initiator. After the submission of the job to the platform, the participating device receive the sensing job, which includes the information needed to be sensed for. Then the participants can decide whether to participate in sensing. Upon acceptance the dedicated sensor or sensors get enabled, and the participant begins the sensing. The sensed information gets stored on the sensing device and afterwards uploaded to the platform through an access point. The platform then extracts the information needed and removes redundant data. At the end, each device which submitted meaningful data (see Section 2.4.2 for an explanation of meaningful data) receives its reward from the platform for submitting the data, and the task initiator receives the data collected by the participants from the platform.

For sensing the data in mobile crowdsensing there are generally two different approaches [GYL11]:

1. participatory sensing
2. opportunistic sensing

Participatory sensing is a rather manual sensing approach, where the participant needs to actively participate in the sensing procedure by taking a picture or giving a feedback for a meal. In contrast opportunistic sensing is a very autonomous form of sensing where only a little user involvement is necessary like sending positional or temperature information.

2.4.2 Challenges and Solutions in Mobile Crowdsensing

When creating mobile crowdsensing applications, there are several design challenges, which have to be taken into consideration. We will highlight a few in this section and show how they are dealt with in the literature.

The first type of challenges in mobile crowdsensing is due to the fact that the participants devices are controlled through private entities. Therefore, we have to assume selfish behaviour of these devices. This means not all the data received by all sensing devices is legit. One typical example threat in this are sybil attacks. In sybil attacks on mobile crowdsensing, multiple malicious sensing devices report back fake sensing information in order to overrule the contributions made by good sensing devices. This leads to malicious devices receiving the payment for the crowdsensing job [XJX+18], [LYL+18].

[WCMA14] offers a solution to this problem by evaluating the reliability of the data send through a trust-based system. In such systems the devices which send the sensing information have a reputation value associated to them. This value increases by submitting real data and decreases by sending falsified data. When evaluating the data received by a participant, the mobile crowdsensing platform takes the reputation of this device into consideration and gives a higher importance to data received by an account with a higher reputation value.

Another problem in crowdsensing is the trustworthiness of the mobile crowdsensing platform. Because it controls the data exchange and payment, one has to make sure that every participant receives the money deserved and the task initiator receives the submitted data requested for [WYJ+19].

The paper [WYJ+19] highlights how using a blockchain based mobile crowdsensing platform can deal with the problem of untrustworthy mobile crowdsensing platforms. They argue, since blockchains have the properties of being fully transparent meaning that every decision made by the blockchain is visible for everyone outside the blockchain, malicious behaviour implemented into the platform would be visible.

The last challenge is the overall user acceptance or acceptability of such systems. On the one hand, the process of sensing and transmitting the information can consume time and battery. On the other hand, the information is connected to the specific location of the participant, and, therefore, the privacy of the participant is threatened. Both of these problems could lead to users being reluctant to participate in crowdsensing showing the importance of implementing privacy into a mobile crowdsensing system and giving the user an incentive for participating [LYL+18], [XJX+18].

The general solution to this problem presented by most of the research regarding mobile crowdsensing for example by [LYL+18], [SLD15], [WYJ+19] is achieved through incentive mechanisms. This means giving participants an incentive for legitimately participating in the sensing process issued by the task initiator. There are different types of incentives like receiving a certain service in return for participating, having a game-like experience while sensing, or plain monetary reward for the participation. Most currently used incentive mechanisms are monetary rewards for participation in the crowdsensing.

In the above presented approaches each concept gets analysed on its own not in combination. Therefore, we have investigated how well the approaches of blockchain-based applications and mobile crowdsensing can be combined. To this end, we designed and implemented a blockchain based mobile target tracking application utilizing a reputation based trust system to handle fake data contributions.

Chapter 3

System Model and Problem Statement

In this chapter, we present the system model and problem to be solved by this work.

3.1 System Model

In the following, we introduce the system model in order to give the reader an understanding of the main components of our system and the main assumptions.

The goal of this system is to design and implement a crowdsensing-based decentralized mobile target tracking application for lost and found scenarios. This allows to find lost mobile objects equipped with an unique tag using the help of other mobile devices.

Our mobile target tracking system consists of the following components:

- **Tagged mobile objects:** These are the objects that shall be located and tracked by the mobile target tracking system, such as lost or stolen objects, vehicles, pets, etc. We assume that mobile objects are tagged, for instance, using Bluetooth beacons or active RFID tags. Using these tags, the mobile objects can be sensed and identified by mobile sensors in their vicinity (tens to maximal hundreds of meters distance).
- **Mobile sensor devices:** These mobile devices search for mobile objects. To this end, they are equipped with sensors that can sense the tags of mobile objects. These mobile devices are equipped with a positioning system like GPS. Typical examples of mobile sensor mobile devices are smartphones equipped with Bluetooth to sense Bluetooth beacons in their vicinity. We assume that these mobile devices are owned and controlled by private users.
- **Smart contracts:** They define search tasks (which objects shall be tracked), the reward for the users to participate in sensing by providing their sensor mobile devices for the search task, manage the reward distribution, manage sightings from the mobile sensor devices. We assume that the Smart Contracts are immutable upon creation.
- **Ethereum blockchain:** It is the platform on which the smart contracts get stored.
- **Mobile application:** It distributes the search task to the mobile sensor devices. Then the owner of the sensor mobile devices needs to accept the search request and the mobile device

starts to sense and detect the location of the requested mobile object. When the mobile application detects the requested mobile object, it sends a sighting (location and timestamp) to the smart contract, which is subsequently stored in the blockchain as part of its state.

With respect to the users involved in our system, we can distinguish the following roles:

- **Mislayer:** representing one person who wants to find a tagged mobile object. To this end, the Mislayer sets up a smart contract.
- **Searcher:** the users searching for the Mislayer's object using their mobile sensor devices.

The process of tracking the Mislayer's mobile object is divided into five phases:

1. **Search request creation**: Here, the Mislayer creates a search task where it defines all necessary parameters for the Searchers to know. This includes how much the Mislayer is willing to pay for the mobile object to be found, a deadline for the search task until which sightings can be reported by Searchers, and the unique id of the tag of the lost mobile object. The Mislayer then creates a Smart Contract from this information.
2. **Request distribution**: The mobile application notifies the set of all mobile sensor devices of the search request. This includes the tag id of the Mislayer's mobile object and the reward. The owner of the sensor device needs to accept the search request. Then, the device starts scanning for the Mislayer's mobile object, turning the owner into a Searcher.
3. **Search and data collection**: Each Searcher device starts looking for the wanted mobile object object. This means the Searcher Mislayer's mobile objects start to look for the unique tag id of the Mislayer's mobile object by using a wireless short range radio technology like Bluetooth. If the Mislayer's mobile object is seen by a Searcher device, it sends a message to the Smart Contract including the current location and time. This phase continues until the deadline of the contract.
4. **Data evaluation**: If the deadline of the search task has been reached, the Smart Contract automatically starts calculating the longest path on which the Mislayer's mobile object could have been moving according to the sightings reported to the contract and stored as part of the contract's state. The idea of calculating the longest path, as the valid path representing the movement of the Mislayer's mobile object accurately, borrows the idea of the blockchain that if the majority (more than 50%) of Searchers are honest, then it is likely that they will contribute the most sightings and the longest path. The Mislayer can then retrieve this path from the state of the contract.
5. **Reward distribution**: Each Searcher who contributed a sighting to the longest path receives a reward. A Searcher is paid by each sighting she contributes to the longest path. The reward for each sighting contributed to the longest path is calculated according to:

   $\text{reward per sighting} = \frac{\text{Reward defined by Mislayer}}{\text{\#sightings on longest path}}$

Let us summarise the key definitions from the above description used also below in the problem statement:

Definition 3.1.1 ***Sighting*** *A sighting includes the position (consisting of an x,y coordinate) and timestamp of sensing the mobile object created by Searchers.*

Definition 3.1.2 ***Honest and Malicious Searchers*** *A honest Searcher is a Searcher, who only submits valid sightings to the longest path calculation. In contrast, a malicious Searcher is a Searcher, who submits sightings which are not valid to the longest path calculation.*

Definition 3.1.3 ***Valid and Fake Sightings*** *A valid sighting represents the actual position of the mobile object at a certain time and is reported by an honest Searcher. In contrast, a fake sighting is reported by a malicious Searcher to get the reward, without actually sensing the mobile object at this position.*

Definition 3.1.4 ***Valid Longest Path*** *A valid longest path only consists of valid sightings.*

3.2 Problem Statement

In this section, we define the problem to be solved by our approach in detail.

As stated above, we assume that the longest path reported by Searchers is the valid path following the intuitive idea: if a majority of Searchers are honest, then should contribute a majority of correct sightings.

This first of all brings up the question, what is the longest path given a sequence of sightings reported by Searchers? If we calculate a path from a sequence of sightings, this path must be plausible, i.e., fulfilling some **consistency** criteria such that the path could actually exist in reality. Obviously, if an object was reported to be in Stuttgart at 1:00 pm and then in New York at 1:05 pm, these two sightings do not form a plausible path assuming that mobile objects are restricted in their speed of movement. So in order to define the **longest consistent path**, we assume a given maximum speed v_{max} constraint for mobile objects.

Definition 3.2.1 ***Maximum Speed Constraint*** *The Mislayer's mobile object can only travel with the maximum speed v_{max}.*

Definition 3.2.2 ***Consistent Sightings*** *Two sightings consisting of two positions p_1 and p_2 at times t_1 and t_2, respectively, with $t_2 \geq t_1$, are consistent if: $v_{max} \cdot (t_2 - t_1) \geq dist(p_2, p_1)$.*

With this definition of two consistent positions, we can now define the **longest consistent path** as follows:

Definition 3.2.3 ***Longest Consistent Path***

- *Given: a sequence of sightings $s = (p_1, ..., p_n)$ ordered by the timestamps of the sightings such that $t(p_1) \leq t(p_2) \leq ... \leq t(p_{n-1}) \leq t(p_n)$*
- *Then the longest consistent path is the maximum subset of sightings $s^{'} \leq s$, such that for all pairs of positions (p_i, p_j) from $s^{'} \times s^{'}$, $dist(p_i, p_j) \leq v_{max} \cdot (t(p_j) - t(p_i))$. That is all pairs must be consistent with respect to the maximum speed constraint v_{max}*

We assume that this consistency criteria is known to all Searchers, also malicious Searchers, called **attackers**. This brings up the problem, that attackers will try to report fake sightings fulfilling the consistency criteria in order to build the longest consistent path and get the reward of the search request defined in the smart contract. Thus, the fundamental problem to be solved by this work is to prevent such attacks.

3.3 Attacker Model

In the following, we will further evaluate an attacker model to get a better understanding about the concrete attacks this work needs to consider to solve the problem.

As mentioned in the problem statement, we need to prevent attacks that threaten the security of the system model where malicious Searchers report fake sightings to build the longest consistent path to get the reward.

To this end, the attacker (malicious Searcher), sends a large amount of consistent sightings in order to form the longest consistent path. By doing this the attacker receives the reward from the Mislayer without contributing valid sightings. Honest Searchers would be discouraged from participating in future search tasks since they did not receive the reward for participation, and the Mislayer would be discouraged because he did not receive any helpful information about the tracked mobile object.

A simple counter-measure could be to let each Searcher report a limited number of sightings. Then, a single attacker could not define the longest consistent path himself, and also collusion between attackers is more difficult since a critical mass of individual attackers would be required to overrule honest Searchers. However, this counter measure is only effective if we can prevent so-called sybil attacks.

In sybil attacks, an attacker creates and controls multiple identities in the system under attack which is designed in a way that each individual should control one identity. After that he utilizes his mass of identities to behave maliciously in order to gain a personal profit [JCK15].

For this attack on our system, the attacker creates multiple search accounts. Afterwards, if these newly created accounts receive the search requests, all accounts start to consistent sightings. Then, if the deadline of search task is reached and the longest consistent path is formed, the longest path will contain or be solely a path of nodes from the attacker, even though these malicious Searcher accounts have sent false information.

A sybil attack is possible because the presented system does not limit the number of Searcher accounts created by one individual user. Additionally, the creation of Searcher accounts has little to no cost associated with it. This allows anyone to create multiple accounts. Now the question arises why this attack works. First the attacker can send sightings at no cost, meaning that if the attacker sends invalid sightings there is no risk of losing something. This is referred to as the nothing-at-stake-problem. Therefore, a solution to this attack needs to make the attacker put something at stake which she looses when sending invalid sightings. Since the system cannot differentiate between attackers and honest participants, all senders of data need to put something at stake. So, we need to ensure that what a sender puts at stake is for a honest Searcher acceptable, while for an attacker, who creates multiple accounts, is not feasible or at least very unattractive w.r.t. the reward gained by a successful attack.

Chapter 4

Approach

In the previous chapter, we have presented the system model and problem to be solved. In this chapter, we present the approach for preventing the attacks identified previously as follows:

- smart contracts as the mobile crowdsensing platform and the final system interaction.
- how the Searchers track the Mislayers Device.
- how sybil attacks are prevented in blockchains.
- how our application realizes the prevention of sybil attacks.

4.1 Smart Contracts as Mobile Crowdsensing Platform

As one of the goals of this work was to create a mobile crowdsensing application, we need to deploy the previous discussed logic onto the mobile crowdsensing platform. As we wanted to create a decentralized blockchain based mobile crowdsensing application, we used the (Ethereum) blockchain as the mobile crowdsensing platform. This has two advantages:

- In the Section 2.4.2, we have seen that a crowdsensing platform has multiple risk factors like holding back information or being able to not give the full incentives, due to a lack in transparency in the reasoning made by the platform. This can be countered by utilizing a blockchain as crowdsensing platform because if the logic gets encoded into a blockchain it is fully disclosed. Therefore, every participant in the process can see the logic encoded into the platform and detect malicious behaviour in it.
- The logic deployed onto the blockchain gets encoded through a smart contract. Once deployed a smart contract is immutable, meaning that no party involved in the crowdsensing process can tamper with the deployed contract.

The final general system design can be seen in Figure 4.1.
The following sections present the approaches taken and later implemented into a smart contract.

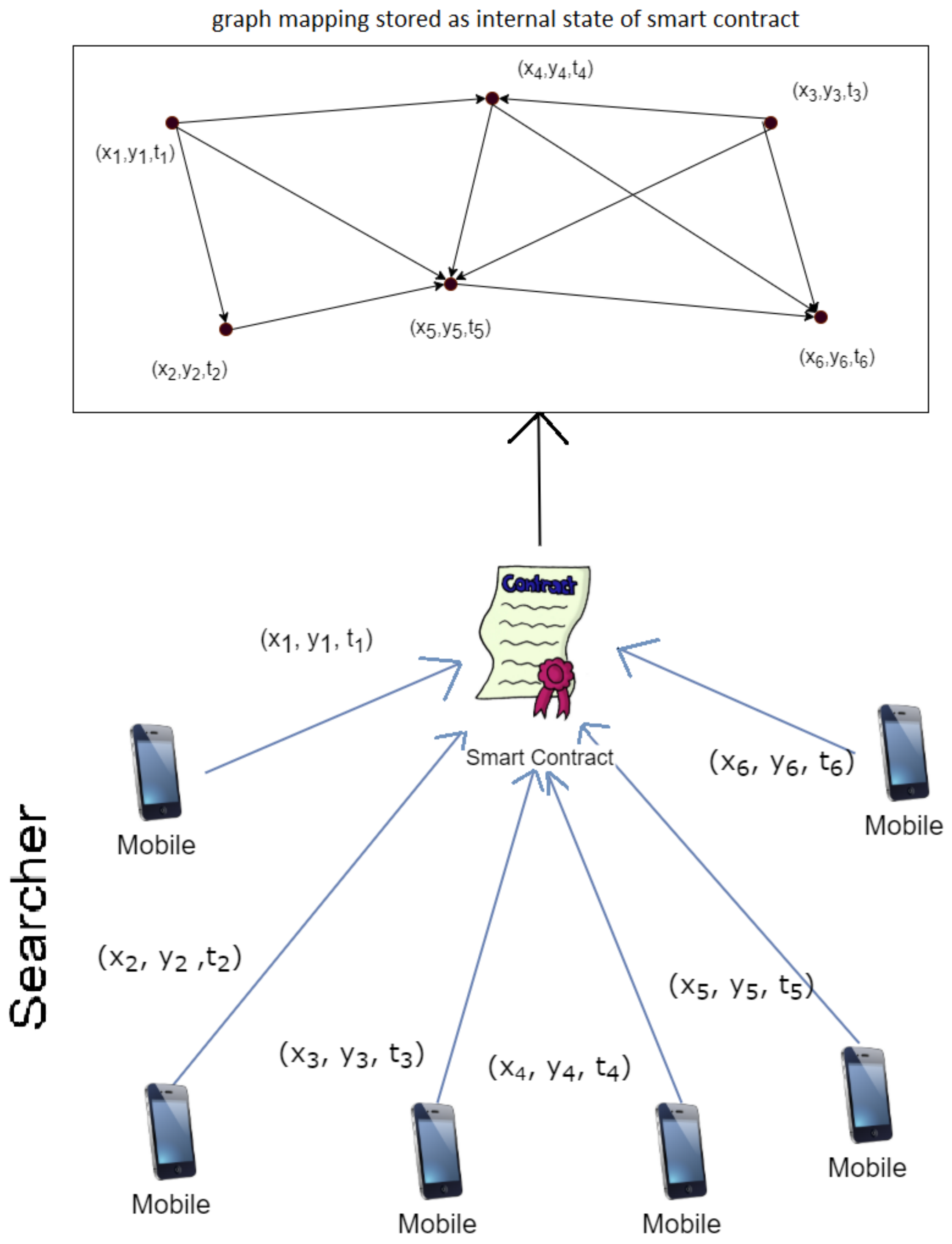


Figure 4.1: Reporting of Sightings and Graph Mapping

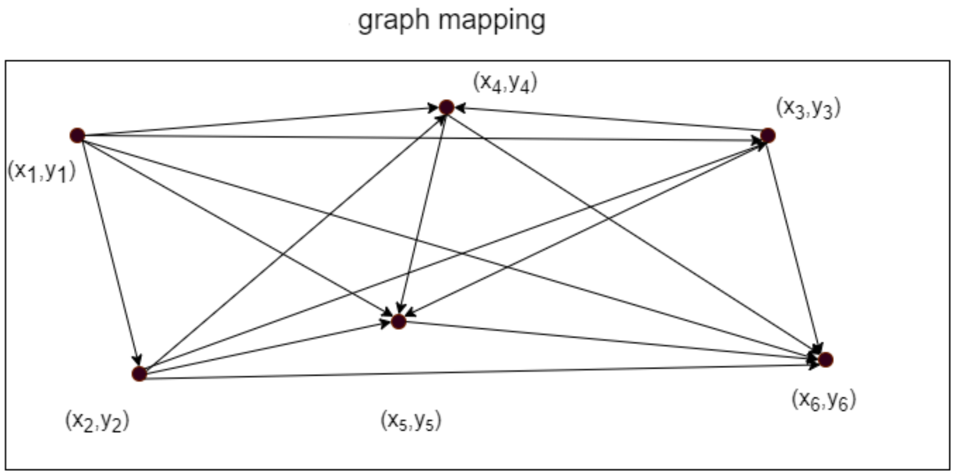


Figure 4.2: Example Directed Graph without Consistency Criteria

4.2 Tracking Mislayer Objects

Before we present our counter-measures to prevent attacks, we first describe in more detail the tracking process for finding the Mislayer's mobile object by Searchers.

As described in the system model, each Searcher device transmits sightings about the Mislayer's mobile object. In this section we present the approach taken on how this information can be used in order to construct the longest consistent path of the mobile object? .

The main goal of our application is to track the location of a Mislayer's mobile object. Since we assume in our application that the mobile object will be at different locations and constantly moving, the mobile object will come across multiple other mobile objects owned by the Searchers. As mentioned in the problem statement, our approach uses these sightings to form a longest consistent path.

The next paragraph explains how to create the graph and which properties it has.

As seen in Figure 4.1, the Searchers sends a sighting consisting of an x, y coordinate and a timestamp to the smart contract. This server then maps these into graph representation, where each position transmitted represents one node. This raises the question how to create the edges in this graph.

The simplest idea would be to connect each position to all other positions. This causes several problems. The most striking is that a longest consistent path calculation does not deliver meaningful results. The reason being, that if all nodes are connected, we would have cycles and therefore infinite many paths. Additionally, we would not utilize the aspect in our data that the positions we receive comes in a timely ordered manner. Since the Searcher devices transmit a location, when they come across the Mislayer's mobile object, the positions transmitted to the crowdsensing platform are ordered by time. This means the edges in the graph can be unidirectional by letting only edges be created from older positions to newer positions. The graph created from the positional information is then a so-called directed acyclic graph, as shown in Figure 4.2. In Section 6, we will see how this type of graph reduces the time and space complexity of our longest path algorithm.

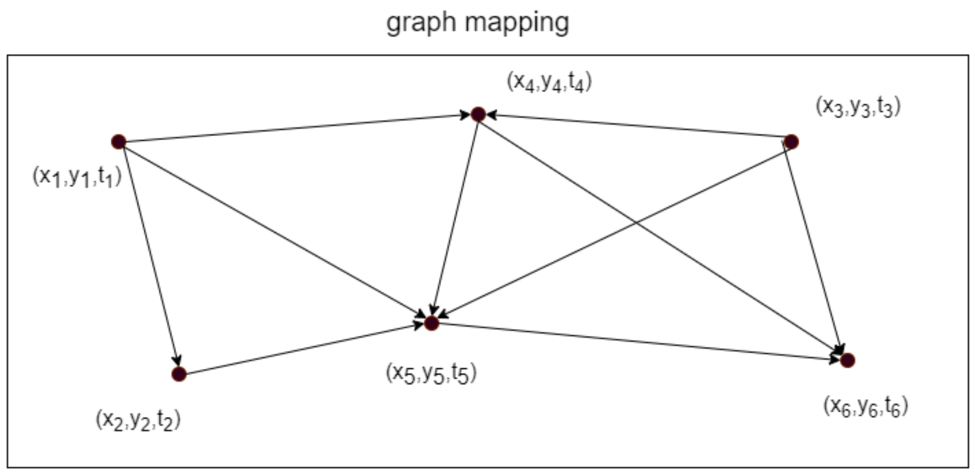


Figure 4.3: Example Directed Acyclic Graph using Consistency Criteria

The presented graph still has one problem. By only requiring that an edge has to point from an old position to a newer position, we allow edges in the graph to be created, which violate the maximum speed constraint introduced in Section 3.2. Depending on the use case this application is used for, the maximum speed parameter needs to changed. For example, if the Mislayer's mobile object is a smartphone, which has been lost in a subway, it does not make sense to assume that the smartphone can travel with a speed of 200 km/h rather it would make sense to take a speed of 60 km/h. In order to see the advantage, we illustrated an example graph utilizing the maximum speed constraint in Figure 4.3, created from the graph shown in Figure 4.2. Compared to the graph seen in Figure 4.2, this graph shows how this reasonable addition reduces the amount edges which would need to be created. So, in order to create the longest consistent path utilizing the maximum speed constraint not only the positions need to be transmitted by a Searcher but also the timestamp for that position is required. This is the reason why we previously defined that our sightings consist of both the position and a timestamp.

Definition 4.2.1 ***Weight of an Edge in Approach without Reputation*** *An edge between two sightings in the graph representation must satisfy the consistency criteria. The weight of that edge is the Euclidean distance between the two sightings.*

4.3 Prevention Sybil Attacks

Previously we only discussed the logic for tracking the location of the Mislayer's mobile object. Now we show the approach taken to ensure that sybil attacks performed in a manner as presented in Section 3.3 have a low effectiveness. We decided when investigating approaches for dealing with sybil attacks, the approaches should come from concepts from the fields of blockchain technology, since our application is a blockchain-based application and as blockchains need to deal with sybil attacks too.

Blockchains prevent sybil attacks through the usage of consensus algorithms. These algorithms rely on some resource like compute power which cannot easily be tackled. Our previously presented system uses a voting based approach where the majority of consistent sightings form the longest consistent path. The sybil attack abused the aspect that a single attacker could send multiple votes.

Therefore, we investigated several blockchain consensus protocols and tried to map them to our system for prevention of sybil attacks.

1. **Proof of Work**: Proof of work achieves consensus on the data being transmitted to the blockchain by making it computationally hard to submit information. Hereby, the transmission of falsified information becomes more difficult and requires an upfront investment. It requires from malicious miners in order successfully transmit falsified information to the blockchain to own more than half of the resource power in the entire network, see Section 2.

   Applying proof of work to the problem of invalid sightings transmitted by multiple Searchers in a sybil attack manner seems promising. This approach could be implemented by letting Searcher devices solve a computationally hard puzzle which consumes large amounts of the mobile objects battery and time. This is effective to prevent sybil attacks by reducing the ease of submitting a location to the mobile crowdsensing platform by adding a time and resource barrier. Just like in proof of work for blockchains, the transmission process could be designed in a way that a malicious Searcher would need more than half of the processing power of all the Searchers processing power to successfully transmit sightings which would violate our security of the system model definition.

   The disadvantage with this approach is that, it discourages potential honest Searchers to participate in the searching process because it would cost them a lot of energy to participate. This could potentially lead to the application conflicting with the everyday usage of the Searchers mobile object. If a transmission of sightings via a smartphone would use up more than half of its battery life to transmit one sighting, honest Searchers would be reluctant to participate in the searching procedure. As discussed in Section 2.4.2, for successful crowdsensing, it needs to be integrated seamlessly into the normal usage of the mobile object. Due to these problems, we decided in this thesis that we would not use a proof of work based approach to prevent sybil attacks.

2. **Proof of Elapsed Time**: In proof of elapsed time, each miner needs to wait a random time before being allowed to append a new block to the blockchain [Sai18b]. This approach prevents sybil attacks by limiting the amount of contributions submitted to the blockchains. This could be adapted to our problem with sybil attacks by letting each miner wait a random time before being allowed to transmit a sighting. This could solve the problem of sybil attacks performed by one attacker where one malicious Searcher accounts sends multiple invalid sightings in order to create the longest consistent path and violate the security definition. This approach has the following disadvantages, which were the reason we did not implement it into our application.

   - We do not want to limit the amount of data sent by one Searcher. After all, if honest Searchers frequently send information, the effectiveness of the application increases because crowdsensing applications thrive from more honest data contributed.
   - This approach alone does not prevent sybil attacks performed by attackers using multiple malicious Searcher accounts. As sending data from multiple Searcher accounts comes with little to no cost, the attacker could create a large number of accounts and still be able to transmit as many data at any given time. So one malicious Searcher of the set of all malicious Searchers can always send, if the attacker owns a large group of malicious Searchers. If you used this approach together with some expensive resource, such as an

Intel CPU with SGX extension for implementing waiting (without processing) [CV17], using it as a trusted compute platform, might work. Since then (expensive) equipment is identified by the manufacturer (such as Intel), you would need to buy many expensive resources (Intel CPUs) to get more sightings submitted. But this requires all the honest Searchers to also own such resources leading to less users participating.

3. **Proof of Stake**: In proof of stake each miner has the probability to append a block to the blockchain proportional to the number of coins the miner owns from the total stake of coins afloat. For example, if there are a total of 10 coins afloat and a miner owns two coins, the miner has a 20% chance to successfully append the next block to the blockchain. The assumption taken in this approach is: the higher the stake of one person in the cryptocurrency, the more this person has to loose when it behaves maliciously, i.e., the higher stake miners are more likely to behave honestly than the lower stake miners [Sai18b].

   When adapting this approach to prevent sybil attacks for the application of this thesis one problem occurs. Currently there is nothing at stake for the Searcher when behaving maliciously. One would need to relate the ”right” to report sightings to the amount of money (Ether) a Searcher owns. This would discourage potential honest Searchers with only small amount to participate in the system.

By the analysis of all consensus mechanisms presented, we decided that in our application we needed to have Searchers put something at stake in order to discourage malicious behaviour. From the analysis of proof of work, we additionally decided that we would not have the Searchers put money or time at stake since we assumed this could lead to discouraging potential Searchers to participate in the sensing process. Additionally, the investigation of proof of elapsed time showed us that we do not want to limit the amount of sightings transmitted by one Searcher due to reduction in effectiveness of the crowdsensing application. Therefore, we decided to use an alternative approach based on reputation.

4.4 Reputation-Based Approach to Prevent Sybil Attacks

This section presents the reputation-based approach used in this work to prevent sybil attacks. We present how the reputation is inserted into the application and how it fulfils the defined security requirement in Section 3.2.

Before explaining the reputation-based approach we make clear:

- We do not define the concrete algorithm or formula to calculate reputation values.
- We assume that reputation value can be calculated based on the past contribution to the longest consistent paths.
- The basic information to calculate reputation values is available through the Blockchain through past search request jobs and results.

In our system, each Searcher gets a reputation value associated with it. When an Searcher submits a sighting, a node gets added to the graph corresponding to the reported sighting. Now when calculating the distance through a node, the distance gets weighted by the reputation value of the Searcher adding the node. This creates a new graph, on which we calculate a weighted longest consistent path. The reputation of an account increases by submitting a location, which was in previous tracking jobs part of the weighted longest consistent path and decreases if not. The

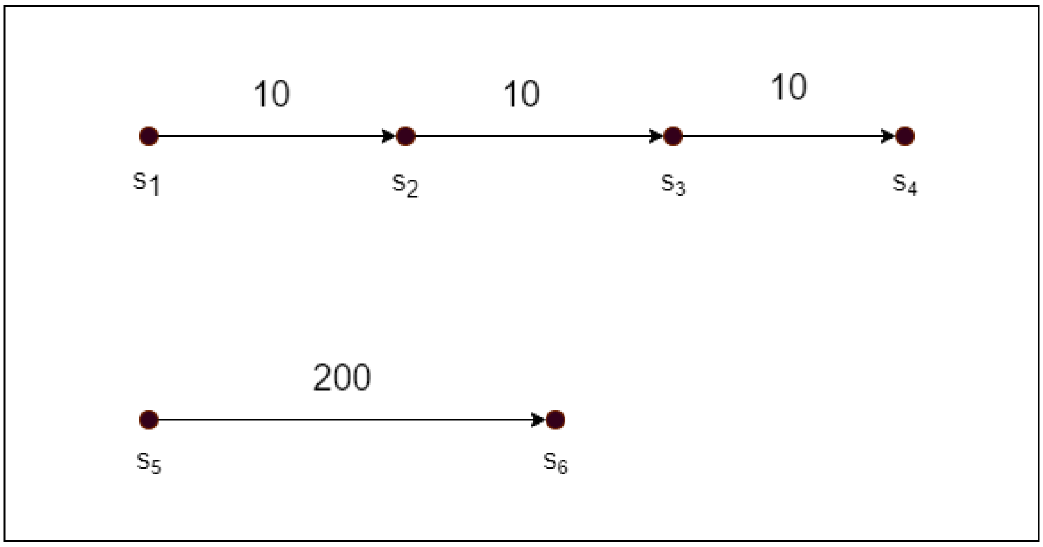


Figure 4.4: Scenario with two Unconnected Paths without RDP

reputation value of a user is implicitly stored in the blockchain by looking at all finished search requests. This is possible because everything that happens on the blockchain is publicly visible, so the blockchain offers full transparency.

In this reputation-based approach we calculate the weights of each edge as a Reputation-Distance-Product.

Definition 4.4.1 ***Reputation-Distance-Product (RDP)*** *An edge created between two sightings s_1 and s_2 in the graph representation must satisfy the consistency criteria. The weight of that edge is then calculated by:*
$r(s_1, s_2) = dist(s_1, s_2) * reputation(s_1) * reputation(s_2)$
This weight is referred to as Reputation-Distance-Product, short r

This definition does not change which edges would get created between sightings. It only affects which path would be selected as the longest consistent path.

To better understand the reputation-based approach and see why it reduces the effectiveness of sybil attacks, we will now demonstrate this through an example. Assume we have the scenario given as seen in Figure 4.4.

In this scenario, an implementation of the approach without reputation would have created two unconnected paths each one fulfilling the consistency criteria. According to the Definition 4.2.1, each edge has a weight according to the distance between the sightings. The first path has a total weight of 30 and the second path has a total weight of 200. Therefore, the Searcher s_5 and s_6 would be rewarded for finding the longest path. That is, the path with the longest distance. That is, the path with longest distance. Now we assume the sightings submitted by s_5 and s_6 are invalid sightings, so the simple scenario shows how the malicious Searchers win without reputation-based system.

Next, we assume, each Searcher account has a reputation value associated with it. Given a reputation distribution as follows: reputation(s_1) = 10, reputation(s_2) = 10,reputation(s_3) = 10, reputation(s_4) = 10, reputation(s_5) = 1, reputation(s_6) = 1. After applying RDP to the edges seen Figure 4.4 Then the two paths would be, as shown in Figure 4.5.

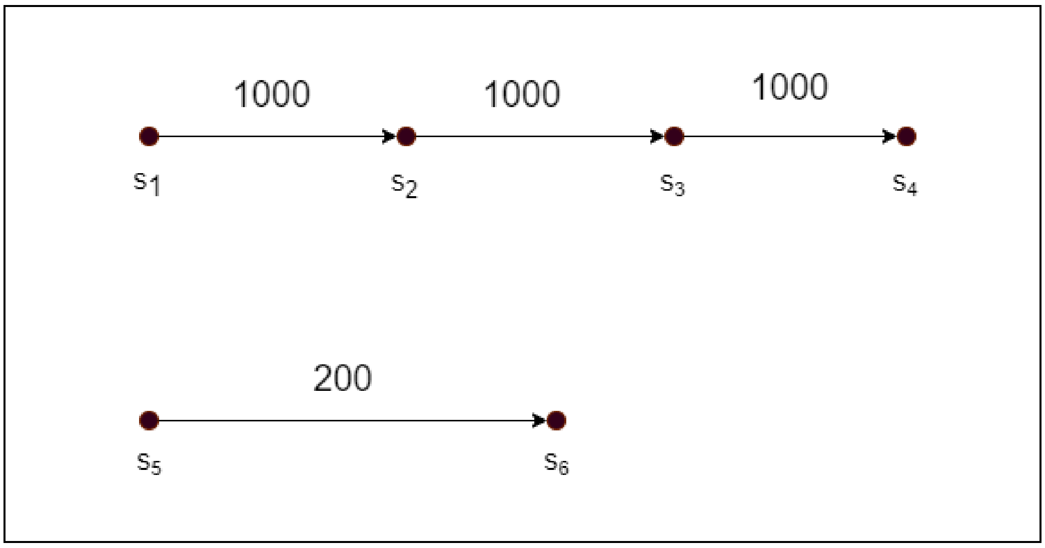


Figure 4.5: Scenario with two Unconnected Paths including RDP

As we can see the edges between the nodes are still the same only the weight changed. The longest path is now the path with the highest total RDP weight. That is, reputation only affects the selection of a longest consistent path, no the consistency criteria, which still needs to be fulfilled for a plausible path. The first path now has a weight of 3000 while the other has a weight of 150. So, this time, Searcher s_1, s_2, s_3, and s_4 would be selected as winners, and path one would be the longest path. Since s_1, s_2, s_3, and s_4 submitted valid sightings, this approach ensures that as long as the reputation values are calculated reasonably, the probability of a fake path selected gets minimized.

In this section, we have seen the approach taken to find the Mislayer's mobile object, how to prevent sybil attacks from being effective, and where these will be encoded. In the next Section, we will present the implementation of these approaches.

Chapter 5

Design

We have designed the previously presented target tracking approaches with and without reputation.

1. Target Tracking without Reputation
2. Target Tracking with Reputation

In the following, we present them and explain their logic. For the target tracking with reputation approach we designed two algorithms. Therefore, we start this chapter off by explaining how this approach is realized using the two designed algorithms followed by an example showing the functionality of the target tracking without reputation algorithms. After that we explain the modifications made to the two algorithms in order for them to be applicable for the target tracking with reputation approach. In the end of this chapter we will explain advanced approaches contributions made by this thesis.

5.1 Target Tracking without Reputation

Each algorithm is responsible for one part of the target tracking process:

- Algorithm 1 deals with the representation of the sightings as nodes in the graph, creates edges between these nodes and determines the longest path on the graph created by Algorithm 1.
- Algorithm 2 stops the execution of Algorithm 1 and distributes the rewards to all Searchers.

5.1.1 Algorithm 1: Graph Preparation

Our implementation of these approaches finds the longest consistent path by finding the longest path to every node. In order to minimize the time complexity of this solution we utilize the timely ordered locations. We utilize the following aspect: the longest consistent path for the entire graph can be calculated by calculating the longest consistent path to each node. By doing this, our algorithm filters out edges, which will not contribute to any longest path. We have done this to improve the overall performance by reducing the amount of edges the algorithm stores. This causes no problem, since we want to find the longest path, this edge will never contribute to any longest path.

Every time a Searcher makes a sighting the Algorithm graph preparation is executed. So Algorithm graph preparation takes the sightings consisting of an x,y coordinate and a timestamp as input. As Algorithm graph preparation receives these sightings in a timely ordered manner, it processes these sightings in chronological order. It prepares the sightings for the longest path calculation. It creates a graph from these sightings submitted by mapping sightings to nodes in the graph data structure and creating edges between these nodes. Algorithm graph preparation does this by chronologically filling two lists. The two lists represent the following:

1. predecessors: maps for each position of a sighting which predecessor position points via an edge to this position.

2. distances: stores for each position the distance of the longest consistent path to reach this location.

Each sighting represents one node in the graph. The lists distances and predecessors both have an entry for each sighting to store the weight of the edge pointing to that node representing the sighting and the associated predecessors node. For each location, the algorithm checks which of the older sightings could have an edge pointing towards the new sighting. To this end, the algorithm excludes all the old locations, which violate the maximum speed constraint. This is done in line 9 of algorithm graph preparation by utilizing the equation presented in the definition for the maximum speed constraint. Amongst all the old locations left, the algorithm now finds the old location, which has the longest path associated with it, in order to reach the current location. This is achieved in line 11 by taking the Euclidean distance between each old node and the new node and adding it with the distance it takes to reach the old node. The corresponding old node selected then gets stored as the predecessor of the current location in the predecessors list. This indicates that there exists an edge in the graph pointing from the old location to the current. Additionally, the length of that longest path to reach that current location gets stored in the distances list entry for the current location. For example, if the selected edge points from node i to node j with distance $d_{i,j}$, the predecessor entry of node j would be i and the distances entry for j would be $d_{i,j} + distances[i]$. By repeating this logic for each node, the distances field holds the maximum distance it takes to reach each node, while the predecessors field stores for each location which edge should be used to reach a location with the maximum distance.

5.1.2 Algorithm 2: Reward Distribution

Algorithm 2 allows the Mislayer to stop the searching phase and begin processing of the graph created by Algorithm 1. All Searchers who contributed sightings get informed that their search is over. The algorithm distributes the reward to all Searchers according to the two lists created. This includes paying the incentive to each Searcher who submitted a location to the longest consistent path. It does this by first checking which of the sightings in the distances list has the highest distance value associated with it, followed by back-propagating through the predecessor starting with this sightings until it reaches a location entry which has no predecessor. Afterwards, each sender gets informed whether they receive an incentive or not.

5.1.3 Example

In order to illustrate the logic of the presented algorithms, we will go through an example scenario.

Assume we have the topology given in Figure 5.1 on which we would need to find the longest path. l_1 to l_4 represent the nodes for the sightings transmitted. First, l_1 to l_4 were submitted in chronological order. The edges represent connections between the nodes with each number on the edge representing the distance. The longest path in this scenario would be (l_1, l_3, l_4). Now we present how the algorithms would find the longest path.

Algorithm 1 Graph Preparation: Creating Directed Acyclic Graph from Sightings

```
1: procedure createGraph($locations$)                         ▷ The dag for all locations received
2:     $predecessors \leftarrow \{\}$
3:     $distances \leftarrow \{\}$
4:     for all locations $i$ received do
5:         $maxDistance \leftarrow 0$
6:         $currentDistance \leftarrow 0$
7:         $furthestPredecessor \leftarrow -1$
8:         for all locations $j$ received before $i$ do
9:             if euclideanDistance($i, j$) $< v_{max}$*timeDifference($i, j$) then
10:                $currentDistance \leftarrow euclideanDistance(i, j) + distances[j]$
11:                if $maxDistance < currentDistance$ then
12:                    $maxDistance \leftarrow currentDistance$
13:                    $furthestPredecessor \leftarrow j$
14:                end if
15:            end if
16:        end for
17:        $distances[i] \leftarrow maxDistance$
18:        $predecessors[i] \leftarrow furthestPredecessor$
19:    end for
20:    return $distances, predecessors$                      ▷ necessary for longest path calculation
21: end procedure
```

Algorithm 2 Reward Distribution

```
1: procedure createLongestPath($distances, predecessors$)          ▷ Reward Distribution
2:     $longestPathList \leftarrow \{\}$
3:     $lastInserted \leftarrow max_{location}(distances)$
4:     while $lastInserted \neq -1$ do
5:         $longestPathlist.append(lastInserted)$
6:         $lastInserted \leftarrow predecessors[lastInserted]$
7:     end while
8:     $payoutPerLocation \leftarrow \frac{contract.Balance}{length(longestPathList)}$
9:     for all locations $i$ in $longestPathList$ do
10:        transferMoney(to: sender($i$), amount: $payoutPerLocation$)
11:    end for
12:    for all locations $i$ not in $longestPathList$ do
13:        send notification: not part of the longest path no incentive received
14:    end for
15: end procedure
```

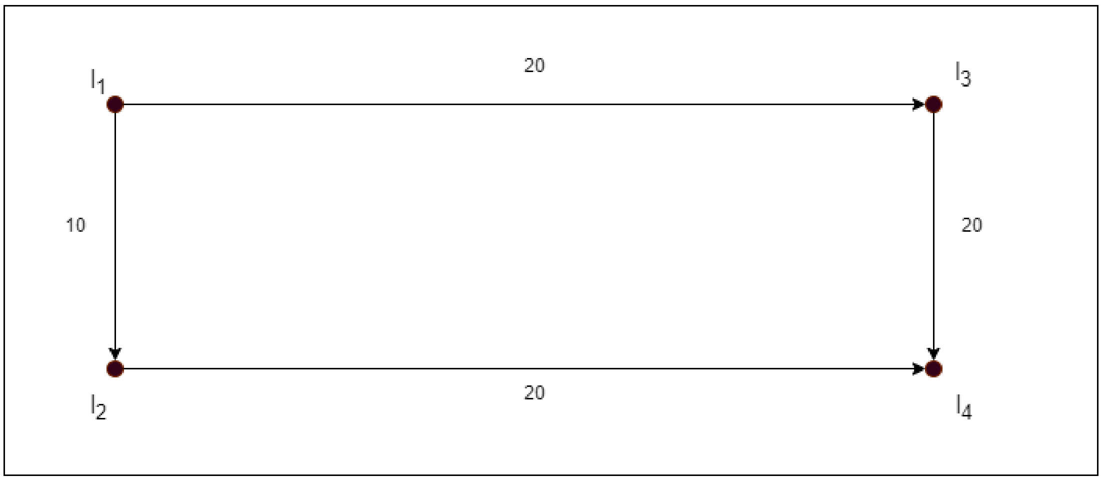


Figure 5.1: Example Graph

In the beginning the scenario looks as seen in Figure 5.2.

Only the sightings have been submitted and the edges are still missing. Algorithm 1 starts by creating the two empty fields distances and predecessors for the internal representation of the graph. Each having one slot for each sighting transmitted. Now when trying to find the edges, in line 4 the algorithm starts with node l_1. It iterates over all the edges which were received before l_1. Since l_1 is the first node, there are no predecessors. Therefore, the distances and predecessor initialized with 0 and -1 (undefined). So the updated fields are as shown in Figure 5.3.

Now the algorithm finds the edges for node l_2. This is done by checking all the previous nodes of l_2, here only l_1. In line 9, the algorithm checks whether l_1 and l_2 violate the maximum speed constraint. In line 10, it calculates the longest path for reaching l_2 by going through l_1, which in this case would be a distance of 10. In line 11, it checks whether among all the possible predecessor nodes for l_2, which one has the longest path for reaching l_2. Here, since l_1 is the only possible predecessor, l_1 becomes the predecessor of l_2, and the distances entry for l_2 is set 10, as seen in Figure 5.4. When processing l_3, the algorithm checks l_1 and l_2 as potential predecessor nodes. As l_2 is out of reach for l_3 and l_1 is not, l_1 becomes the predecessor of node l_3. The longest path over l_1 to l_3 has a distance of 20 and therefore distances for l_3 gets set to 20, as seen in Figure 5.5. The possible predecessor nodes for node l_4 are l_1, l_2, and l_3. When checking which of the nodes violates the maximum speed constraint, the presented topology does not allow an edge between node l_1 and l_4. Therefore, only l_2 and l_3 remain as possible predecessors. Now Algorithm 1 checks amongst all possible predecessors, here l_2 and l_3, which one forms the longest path to node l_4. This means comparing $distance(l_2, l_4) + distances[l_2] = 20 + 10 = 30$ to $distance(l_3, l_4) + distances[l_3] = 20 + 20 = 40$. Since $40 > 30$, l_3 become predecessor of l_4 and distances of l_4 set to 40. There are no locations left, so Algorithm 1 would terminate here. The final graph would be as shown in Figure 5.6. When comparing original topology as seen in Figure 5.1 and the graph that Algorithm 1 created, as shown in Figure 5.6, they do not look the same because the edge between l_2 and l_4 is missing, as Algorithm 1 filters out edges, which will not contribute to any longest path.

As the edges have been created and the longest path is stored in the lists by Algorithm 1, Algorithm 2 deals with distributing the rewards by going through the lists created. First, it finds the maximum entry in the distances field, as seen in Figure 5.7, this would be node l_4 with distance

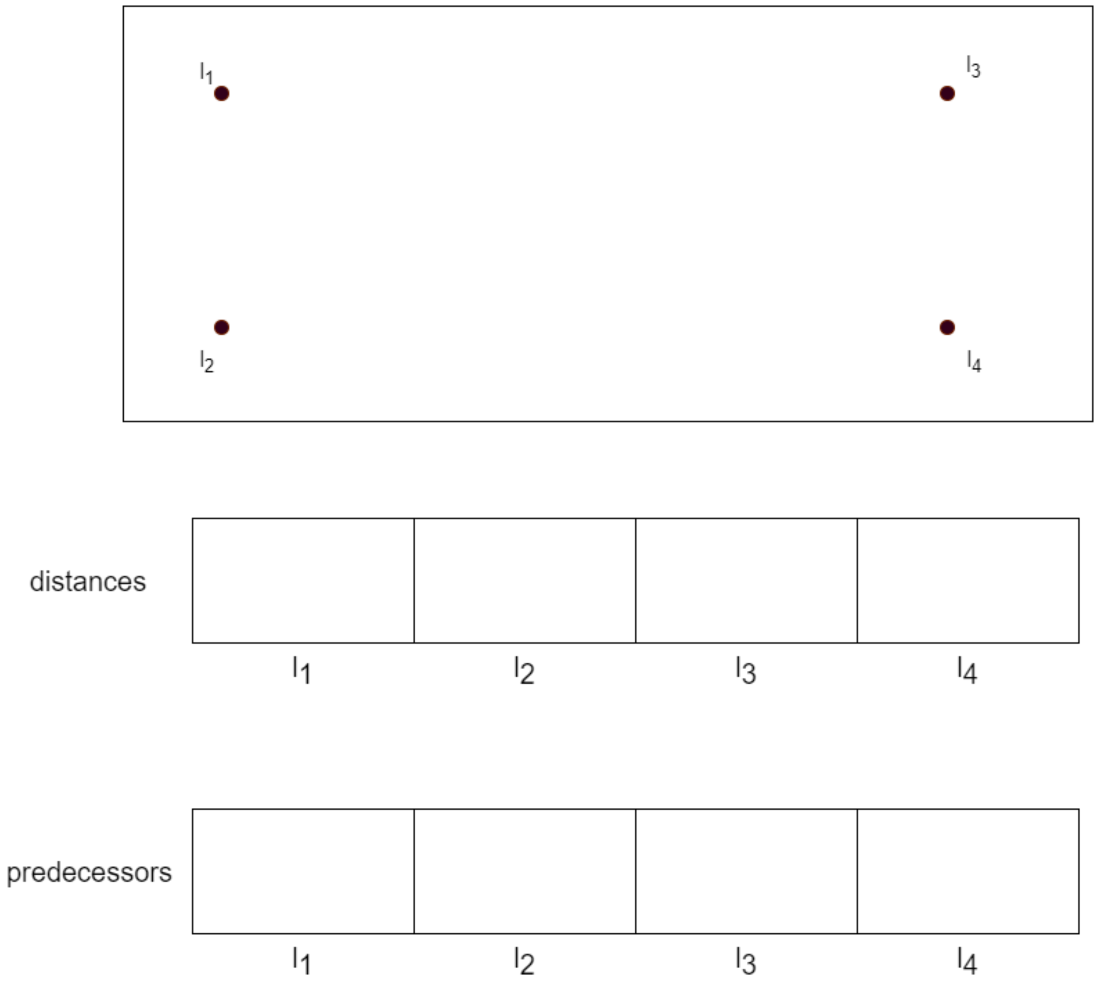


Figure 5.2: Example: Initial State of Algorithm 1

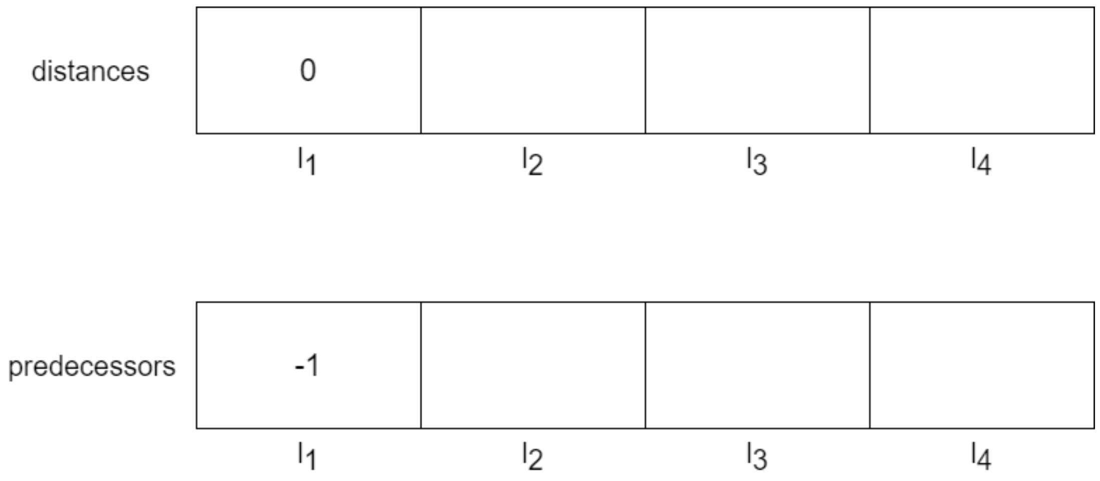


Figure 5.3: Example: Predecessors and Distances Fields after Checking Node l_1

40. This means that the longest path on the graph has to end with node l_4. Secondly, the algorithm back-propagates through the predecessors field until it reaches an entry with value -1. This would mean it has reached the root of the longest consistent path. As seen in Figure 5.7, the longest path found is l_4, l_3, l_1. In the end, the algorithms pays the incentive to each of the senders of the nodes on the longest and informs the rest of the Searchers that they receive no incentive.

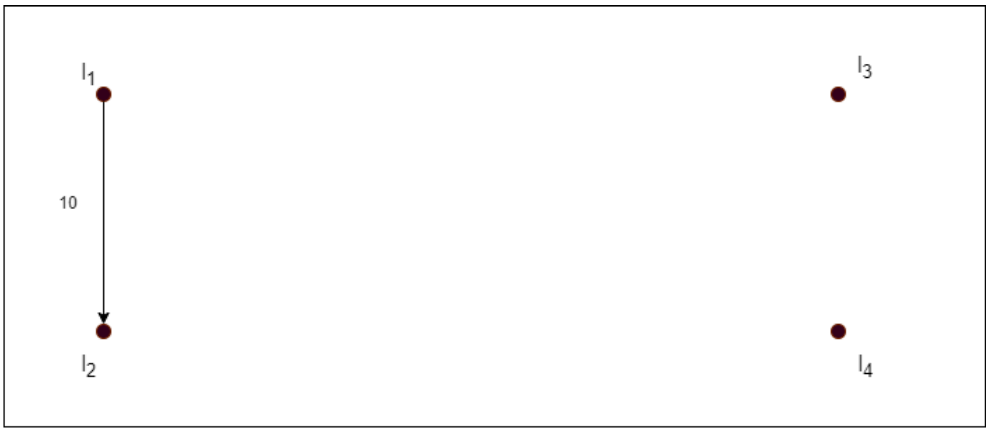
5.2 Target Tracking with Reputation

We have seen how Algorithm 1 and 2 together find the longest consistent path upon receiving sightings submitted by Searchers. As presented in the problem statement this approach does not prevent sybil attacks which lead to the security of the system model definition being violated. In this section, we present what modifications to the previous two algorithms need to be integrated in order to prevent sybil attacks using the RDP presented in the previous chapter, while still allowing for a successful longest consistent path calculation. Recall the basic idea of this is to weight the distance between two nodes by the reputation of these two nodes.

To this end, now we assumed that submitting a sighting consists of: the sighting and **a reputation value of the sender of this sighting**. Inserting the newly added reputation aspect into the logic of Algorithms 1 and 2 required modifying Algorithm 1 graph preparation as the RDP only changes the graphs weight. As a result, the RDP formula was integrated into Algorithm graph preparation as follows:
We modified the calculation done when trying to find the predecessor node amongst all predecessor nodes, which are within reach of the investigated node. Previously, for each node the Algorithm would only consider the Euclidean distance it would take to reach said node, as a weight. For two consistent sightings s_1 and s_2 with timestamp $t(s_1) < t(s_2)$, the RDP is integrated into Algorithm graph preparation as follows:

$$dist(s_1, s_2) * reputation(sender(s_1)) * reputation(sender(s_2)) + distances[s_1] \tag{5.1}$$

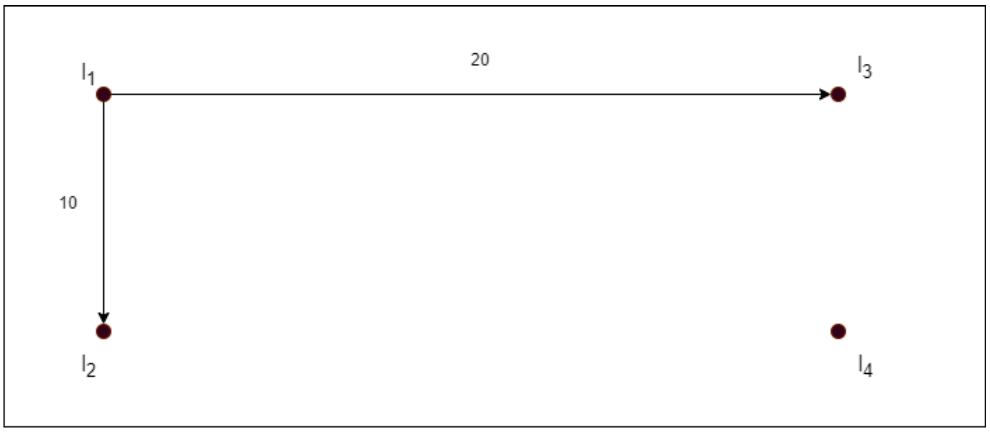
The basic idea is, for a potential edge between s_1 and s_2, set the RDP as the weight of that edge and add it to it total RDP weight of a path leading to s_1. By doing this the edges created only the



	l_1	l_2	l_3	l_4
distances	0	10		

	l_1	l_2	l_3	l_4
predecessors	-1	l_1		

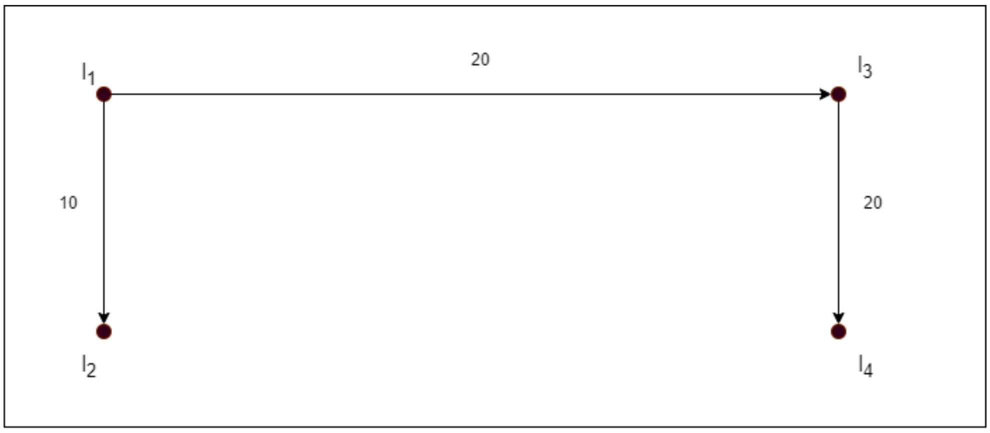
Figure 5.4: Example: After Processing Node l_2



distances	0	10	20	
	l_1	l_2	l_3	l_4

predecessors	-1	l_1	l_1	
	l_1	l_2	l_3	l_4

Figure 5.5: Example: After Precessing Node l_2



	l_1	l_2	l_3	l_4
distances	0	10	20	40

	l_1	l_2	l_3	l_4
predecessors	-1	l_1	l_1	l_3

Figure 5.6: Final Representation after Checking all Nodes

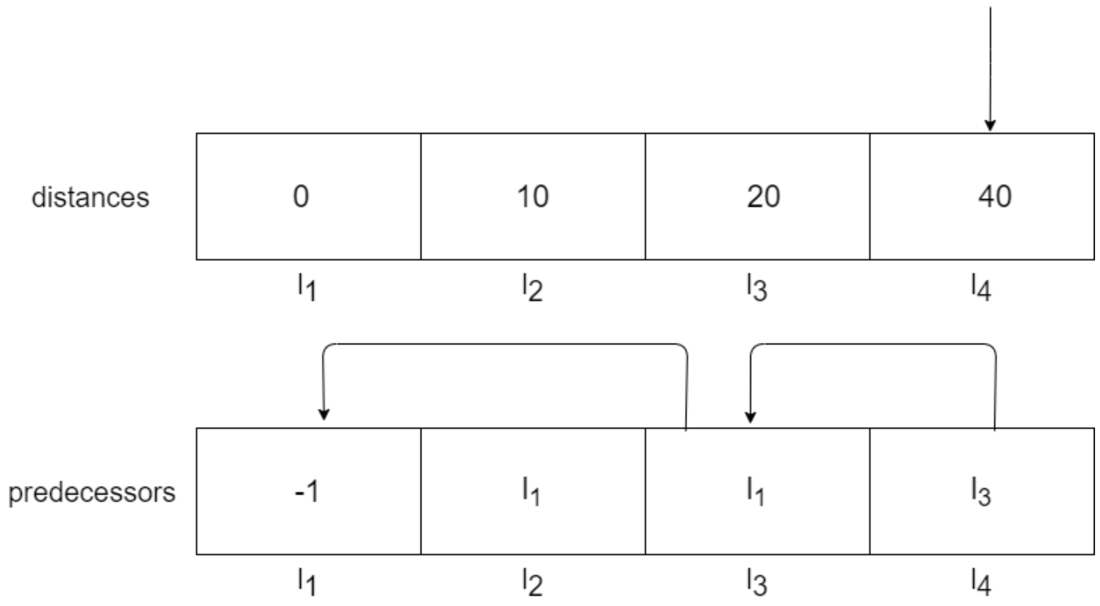


Figure 5.7: Example: Algorithm 2 Finding Longest Path

path selected as the longest consistent path gets changed while the edges created by the approach with and without reputation stay the same.

We decided to multiply the reputation values in the RDP formula for several reasons:

1. We want the distance of an edge to be weighted by both nodes involved in an edge. This way nodes with a low reputation value could not profit by being connected to nodes with a high reputation value.

2. Doing something else rather than multiplying the reputation values before multiplying the result with distance, like taking the average, minimum or maximum reputation of the nodes, either helps nodes with low reputation connected to high reputation nodes or reduces the impact which high reputation nodes have.

3. Nodes which have miserable reputation values like zero will drastically reduce the weight of an edge and therefore are more likely not to be considered.

The modified version of Algorithm 1 graph preparation using the RDP is presented through Algorithm 3 graph preparation using RDP.

5.3 Advanced Attacks and Approaches

In this section, we identify advanced security threats that are still possible even when using the presented basic reputation-based approach and we will present counter-measures to these attacks. In the implementation and evaluation, we only consider the previous approaches because of time restrictions.

Algorithm 3 Graph Preparation using RDP

```
1: procedure createGraph($locations$)    ▷ The dag for all locations received
2:     $predecessors \leftarrow \{\}$
3:     $distances \leftarrow \{\}$
4:     for all locations $i$ received do
5:         $maxDistance \leftarrow 0$
6:         $currentDistance \leftarrow 0$
7:         $furthestPredecessor \leftarrow -1$
8:         for all locations $j$ received before $i$ do
9:             if euclideanDistance$(i, j) < v_{max}*$timeDifference$(i, j)$ then
10:                $currentDistance \leftarrow euclideanDistance(i, j) *$ reputation(sender($(i)$)) $*$ reputation(sender($(j)$)) $+ distances[j]$
11:                if $maxDistance < currentDistance$ then
12:                    $maxDistance \leftarrow currentDistance$
13:                    $furthestPredecessor \leftarrow j$
14:                end if
15:            end if
16:        end for
17:        $distances[i] \leftarrow maxDistance$
18:        $predecessors[i] \leftarrow furthestPredecessor$
19:    end for
20:    return $distances, predecessors$    ▷ necessary for reward distribution
21: end procedure
```

5.3.1 Reputation Farming Attack

One advanced attack could be the so-called reputation farming attack.

As mentioned before, the reputation-based approach prevents sybil attacks by weighting each sighting submitted by a Searcher according to the reputation value of that Searcher. Therefore, sybil attacks do not benefit from multiple (fake) identities, as these identities could not outweigh submissions made by honest Searchers.

However, the attacker could still try to aggregate reputation in order to later perform sybil attacks. To this end, the attacker starts off by increasing his reputation through valid sightings to a longest consistent path. This could be done in two ways: either the attacker performs search requests correctly, and submits valid sightings, or the attacker creates search requests himself and let his own Searcher device contribute to the longest path. The later is referred to as artificial reputation increase:

Definition 5.3.1 ***Artificial Reputation Increase*** *If a Searcher or a group of Searchers increase their associated reputation value by acting as a Mislayer creating their own search requests multiple times and create the longest consistent path themselves, this is referred to as artificial reputation increase.*

After the Searcher devices have built a solid reputation, they can start to perform sybil attacks on real tracking jobs, which compromises the reputation-based security approach. Possible countermeasures to this attack are:

- **Utilizing the fact that sending sightings to the smart contract costs money.** As we will see in the evaluation chapter, submitting sightings to the smart contract using the RDP takes between 80 and 90% of the total cost for a single search request. This means for a single search request, 80 to 90% of the cost that the Mislayer puts into the request does not get distributed to any Searcher who contributed a sighting to the longest consistent path. So, if an attacker tries to aggregate reputation through an artificial reputation increase, the attacker loses a lot of money. Therefore, accumulating reputation requires a huge upfront investment to perform reputation farming attacks. However, this still does not prevent previously honest Searchers who have a large reputation to turn malicious and perform this attack.

- **Aging of reputation to decrease the effectiveness of aggregating reputation.** Aging means that the weight of reputation values gradually decreases over time. Possible aging implementations could be:

  - Use a time window in order to allow only reputation gained recently to be considered.
  - Calculate the moving average. For instance, with an exponential moving average, the weight of old reputation values would decrease exponentially. As all the reported sightings are stored on the blockchain, the complete history of gaining reputation is recorded on the blockchain. Therefore, aging can be implemented based on this historic information.

  The idea is that Searchers who own a large amounts of reputation cannot rely on reputation aggregated a long time ago. Thus, an attacker constantly needs to generate new reputation in order to prove that he did not turn into a malicious Searcher. The problem with this approach is how to implement and tune the function defining the aging of reputation.

- **Testing Searchers using the Ground Truth** In the previous approaches, we have used the maximum speed constraint as a consistency criterion for defining the longest consistent path. However, although the maximum speed constraint ensures that paths are plausible, i.e., they could exist in reality, there is still no guarantee that the longest consistent path actually only consists of valid sightings that really exist as reported in reality. In order to very that Searchers actually only reported valid sightings, one needs to the ground truth, i.e., the actual position of the mobile object in reality to compare this real position to the reported position. This can be achieved by proposing test cases as search requests, where the Mislayer actually knows the mobile object position a priori. For instance, the Mislayer could simply search for a mobile object that does not exist. If a Searcher reports a sighting for this non-existing object, it gets a reputation value of zero.

  The drawback of this approach is that it relies now on the trustworthiness of the Mislayer proposing the test cases and defining the ground truth, which contradicts the nature of the blockchain approach with unknown individuals.

5.3.2 Copy Cat Attack

In this section, we show how the full disclosure of information stored in the blockchain can also lead to an attack called copy cat attack.

As mentioned before one property of blockchains is that everything that is stored in the blockchain is visible to everyone. This lack of information hiding leads to an attack, which we called the *copy cat attack*.

The attacker does the following: the attacker spies on the blockchain looking at every sighting reported to the smart contract representing the search request. So an attacker can see, which sightings are submitted. The attacker contributes invalid sightings by copying or slightly adapting valid sightings sent by honest Searchers to become a contributor to the longest consistent path with only limited effort. This is possible because every transaction is made public to everyone.

Two possible counter-measures to this attack are:

- Let Searchers submit sightings for a search request only in a short submission time window, and not accepting sightings after that time window. By doing this, attackers could spy on the blockchain only in a short time window. The submission time window needs to be set small enough not allowing for a successful submission of copied sightings, not giving the Searchers enough time to spy on the submitted sightings. The submission of copied sightings would then occur after the time window is closed

  For example, setting the submission time window could be as follows: the current average time for a transaction to be verified on the Ethereum is six minutes (10.8.2020) [Eth20]. This would mean the submission time window should be set to twelve minutes as spying could only happen after the first six minutes of that window and the time until this transaction is six minutes meaning they could not verify the spied sighting in time. This would mean if honest sightings have to be submitted in the first six minutes or more after the submission time window is closed the sighting would not be accepted.

  The problem of this approach is that Searcher device could not have Internet access at the submission time window preventing a successful transmission of a valid sighting. Another problem, is that many transactions are issued during that time window on the Ethereum blockchain, the submission of locations could be delayed due to Ethereum accepting requests only slowly. This could lead to a transaction not getting verified during that time window, leading to rejecting valid sightings. Furthermore a problem w.r.t the example presented before would be that spied sightings after the six minutes still could be verified before the twelve minutes are over, as they could pay more gas for faster verification of the sighting.

- Alternatively, Searchers could first transmit sightings in an encrypted form until a deadline, including a secure hash-value of the (unencrypted) sighting. After the deadline is over, the Searchers disclose how to decrypt encrypted sightings by publishing the encryption key. The Mislayer can verify that the encrypted sightings are consistent with the decrypted sightings by calculating the hash of the decrypted sightings and comparing it to the hash of the submission,. A matching hash also proves that the Searcher actually knew the sightings already before the deadline. So spied sightings could only occur after the deadline is closed.

Chapter 6

Implementation

In this chapter, we have presented the challenging and "interesting" parts of the implementation for the approach presented in this work.

6.1 Integer Representation of Coordinates

In this section, we highlight a limitation which had to be made when implementing the smart contract in the Solidity programming language.

In this work sightings consist of an x,y coordinate and a timestamp. As Solidity does not support integer numbers, sightings send our contracts have to contain only integer values. Therefore, a real world sensing mobile application of this system which senses the sightings should transform the sightings into integers. For instance, the positions of sighting could be mapped to UTM coordinates which maps locations to integer values.

6.2 Allowing Smart Contracts to receive Money

As defined in the system model the Mislayer must set the reward (in Ether) he is willing to pay for the tracking of his mobile object. This is implemented into the smart contract by distributing for each sighting on the longest consistent path: the reward divided by the number of sightings on the longest consistent path. The Mislayer does this by sending a transaction containing the reward in Ether to the contract. But in order for a contract to receive Ether the contract has to be allowed to receive Ether. As a default a contract is not allowed to receive Ether. This has been done in order to prevent unintentional Ether transfer to a contract. This reason being, if a contract receives Ether and the contract has no function to distribute Ether, it would be impossible to retract that Ether. A contract has to have at least one function containing the keyword *payable* to receive Ether. We used the so-called fallback function as this is the function which is called if the contract receives a transaction without triggering a function in the contract. As seen in the following:

```
//fallback function
function () external payable {}
```

the fallback function is always the function in the contract without a name.

6.3 Sighting Submission Function

In order for our smart contract to handle a sighting to be processed, we have implemented a contract function (a potential mobile application on the Searcher's device has to call). As seen below:

```
// processes a submitted sighting by a Searcher
function newLocation(int _x, int _y, int _time, int _reputation) public{
  appendLocation(_x, _y, _time, _reputation);
  findLongestPathToNode(_x, _y, _time, _reputation);
}
```

this function takes as parameter the x,y coordinates and the timestamp of a sighting, and forwards these values to the *appendLocation* and *findLongestPathToNode* function. The *appendLocation* stores the sightings and the address of the Searcher in the associated data structures, and the *findLongestPathToNode* finds the predecessor sighting which has the longest consistent path to that sighting according to the algorithms presented the previous chapter.

For the reputation-based approach using the RDP, our implementation required the Searchers to include in the sighting the reputation as an additional parameter.

6.4 Submission Deadline Function

In order for a Mislayer to stop the submission of sightings and start the reward distribution the *createLongestPath* function has to be called. This function can only be called by the creator of the smart contract which is the Mislayer. As seen below:

```
// Mislayer stops submission of sightings and distributes reward
function createLongestPath() public{
  uint maxElement;
  uint maxValue=0;
  uint lastInserted;
  for (uint i = 0; i < distances.length; i++){
    if (distances[i] > maxValue){
      maxValue = distances[i];
      maxElement = i;
    }
  }
  lastInserted= maxElement;
  while(true){
    if (lastInserted != 99999){
      lastInserted = appendNodeNumberToLongestPathList(lastInserted);

    }
    else{
        break;
    }
  }
  payout();
  selfdestruct(owner);
}
```

first this function find the sightings on the longest consistent path. Then, it calls the payout function which takes care of the reward distribution for all the Searchers on the longest consistent path. Finally, it calls the *selfdestruct* function leading. It stops transaction from being sent to the contract forever.

Additionally this function makes sure only the Mislayer can call the *createLongestPath* function as it requires to be called by the owner of the contract which is the Mislayer. If any one but the Mislayer calls the *createLongestPath* function, it fails because the *selfdestruct* throws an error, leading to the entire transaction being aborted.

Remark 1 *In Ethereum, after a smart contract is deployed on the blockchain, every node in the Ethereum network can call every function of a smart contract, unless the contract has access restrictions for functions implemented into it.*

This means for our implementation, that it needs to limit the access to functions which only the Mislayer should have access to. In contrast, our implementation should not limit the access to the submission of sightings, allowing any possible Searcher to submit sightings.

6.5 One-Dimensional Arrays for Storage of Edges

Traditional graph representations in algorithms often use adjacency arrays representing the outgoing edges for each node represented through two-dimensional arrays. As the cost of the execution of smart contracts is affected by the runtime and space complexity of that contract, we only used one-dimensional arrays for the storage of the edges. Additionally, instead of creating objects or structures representing each sighting, we again used one-dimensional arrays. As using two-dimensional arrays or objects or structures has higher access times compared to one-dimensional arrays we decided to use them to reduce the overall workload.

Remark 2 *Storing all possible outgoing edges for a node in a graph removes all the benefits gained by using an one-dimensional array. As the longest consistent path calculation does not require to store all outgoing edges for each node at the same time, each node stores the predecessor node of the edge on the longest path to that respective node.*

In order to understand how the sightings are stored in a one-dimensional, we present how our implementation iterates through the sightings having reputation values associated. If a sighting and the reputation is submitted they all get appended to the locations array. First the x then the y coordinate, followed by the timestamp, and in the end the reputation value. This means every four elements in the array belong to one sighting. As seen below:

```
// iteration head for going over all the old sightings
for (uint i = 0; i < newLocationNumber; i++){
    xOld = locations[4*i];
    yOld = locations[(4*i)+1];
    timeOld = locations[(4*i)+2];
    reputationOld = locations[(4*i)+3];
```

in order to access the i-th sighting, the locations array from index $4 * i$ to index $4 * i + 3$ have to be used.

As a sighting in the approach without reputation consists only of 3 parameters (x,y-coordinate and time), the one dimensional array stores the sightings in blocks of three. So in order to access the i-th sighting there, index $3 * i$ to index $3 * i + 2$ have to be accessed.

6.6 Deployment of the Implementation

In this section we present, how to deploy Smart Contracts via the Truffle" framework onto the associated "Ganache" development blockchain.

Truffle and Ganache can be described them as follows:

- **Truffle** is a development environment for the creation and deployment of smart contracts to the Ethereum blockchain. It allows to simulate issuing transaction to a smart contract [Gro20].
- **Ganache** is a local development blockchain, which is used to test the behaviour of smart contracts on the Ethereum blockchain. It emulates the behaviour of the Ethereum blockchain, meaning smart contracts get deployed on the Ganache Blockchain. Allowing to create Ethereum like external accounts in order to interact with a smart contract [Gro20].

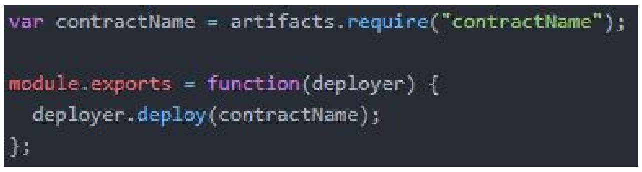
Together Truffle and Ganache can be used to test how a smart contract would perform on the Ethereum blockchain.

To deploy a smart contract onto the Ganache test blockchain using Truffle, the prerequisites are:

- Windows 10
- Truffle version 5.1.12
- Solidity version 0.5.16
- Node Package Manager version 12.16.0
- Web3 version 1.2.1

In order to create a project representing a smart contract on the Ganache blockchain with the Truffle environment the following steps are necessary:

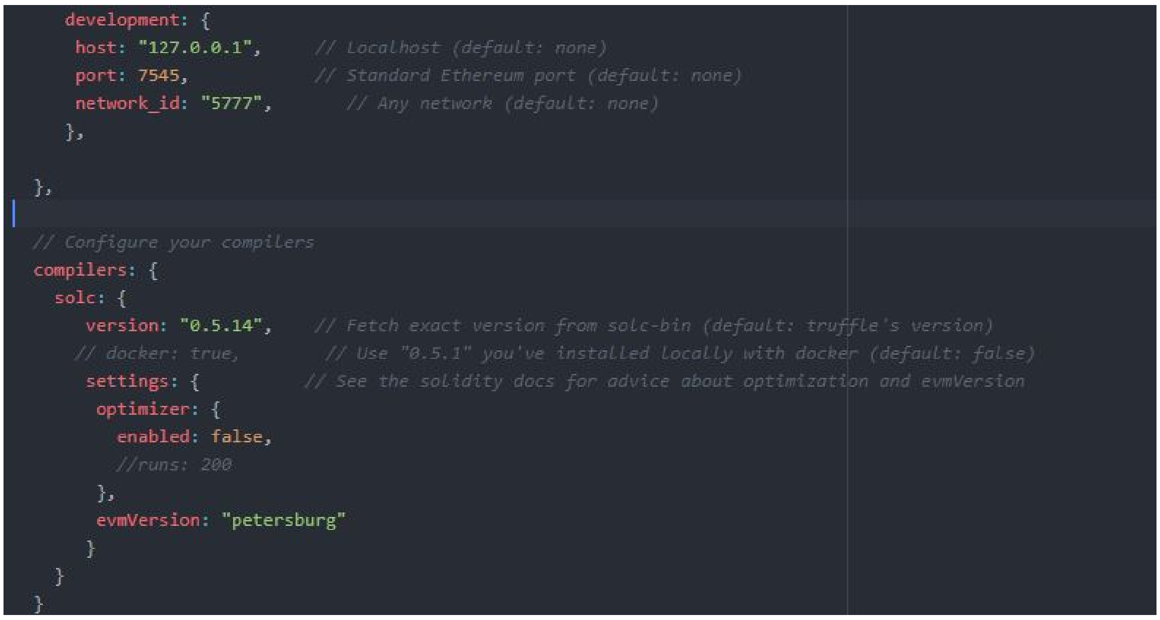
1. Create a folder for the project.
2. Open the command line terminal and direct it to the project folder.
3. In there execute the command "truffle init". This creates a bare bone Truffle project including the required folders.
4. Write a smart contract and insert the smart contract into the "contract" folder.
5. In the "migrations" folder create a JavaScript file with the name "2_deploy_contracts". This file needs to include the code seen in Figure 6.1.This file is necessary for the deployment of the created smart contract to the Ganache development blockchain.
6. Modify the truffle-config.js file: under development you need to set the network_id to 5777 and the port to 7545. Additionally under compilers you need to change the compiler version to 0.5.14 and the EVM version to petersburg, as seen in Figure 6.2.
7. Open the Ganache application and create a new Workspace. This will prompt you to link it to the truffle-config file created one step earlier.
8. Now execute "truffle migrate" in the command line terminal in order to deploy the smart contract on the Ganache blockchain workspace created.



```
var contractName = artifacts.require("contractName");

module.exports = function(deployer) {
  deployer.deploy(contractName);
};
```

Figure 6.1: Example Deployment Code for Smart Contracts



```
    development: {
     host: "127.0.0.1",     // Localhost (default: none)
     port: 7545,            // Standard Ethereum port (default: none)
     network_id: "5777",       // Any network (default: none)
    },

  },

  // Configure your compilers
  compilers: {
    solc: {
       version: "0.5.14",    // Fetch exact version from solc-bin (default: truffle's version)
      // docker: true,        // Use "0.5.1" you've installed locally with docker (default: false)
       settings: {          // See the solidity docs for advice about optimization and evmVersion
        optimizer: {
          enabled: false,
          //runs: 200
        },
        evmVersion: "petersburg"
       }
    }
  }
```

Figure 6.2: Example Modified Truffle Config File

9. Execute ”truffle console” in the command line terminal in order to start interacting with the smart contract.

This is how we deployed our smart contract implementation on the test environment for our evaluation.

This chapter presented the implementation of our approaches and how to deploy the smart contracts containing them to the blockchain with these approaches on the test environment. The next chapter will evaluate the monetary cost of running it on a blockchain via the Ganache test blockchain.

Chapter 7

Evaluation

The purpose of this chapter is to evaluate the previously presented contracts. In order for the approach to be accepted by both, Searchers and the Mislayers, the monetary cost is crucial. Different types of scenarios allow to evaluate the effects of additional costs caused. To this end, we created different scenarios and monitored the monetary cost of running these contracts on them. We want to evaluate the overhead with respect to the monetary cost of our proposed security concepts to counter attacks ("price of security").

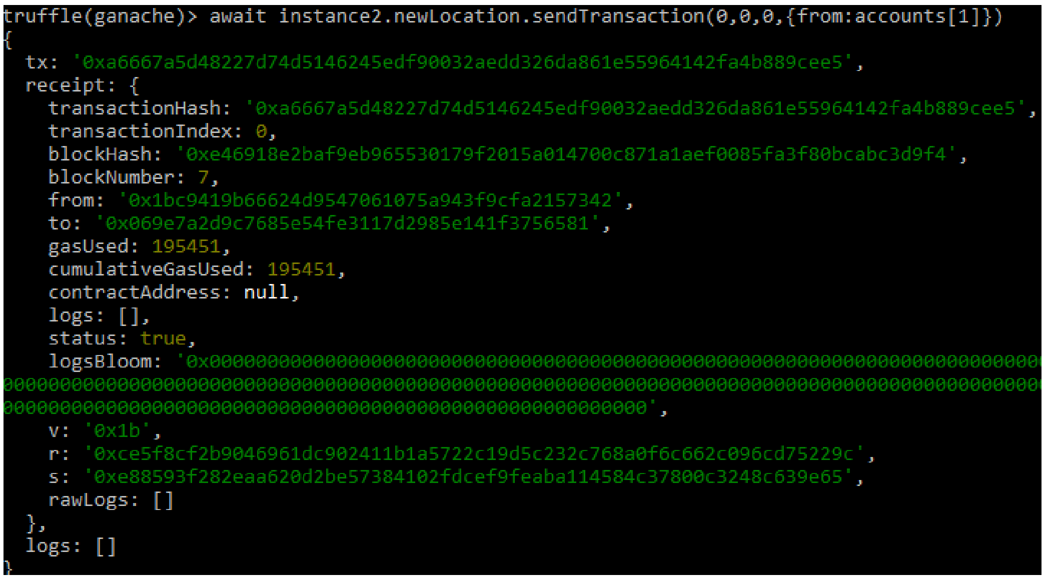
7.1 Evaluation Setup

Before presenting the evaluation, we present our evaluation setup, including the test scenarios and the performance metric to measure the cost.

The test environment has been setup as described in Section 6.6. Truffle allows for interaction with smart contracts via JavaScript code. In order to interact with our smart contract and recreate our evaluation scenarios the following commands have to be performed in the Windows Command Prompt:

```
\\ evaluation in the Windows Command Prompt
truffle console
let instance = await contractName.new()
let accounts = await web3.eth.getAccounts()
await instance.sendTransaction({from: accounts[0]}, value:
    1000000000000000000)
await instance.newLocation.sendTransaction(x_0,y_0,timestamp_0, {from:
    accounts[1]})
...
await instance.newLocation.sendTransaction(x_n, y_n,timestamp_n, {from:
    accounts[n+1]})
await instance.createLongestPath.sendTransaction({from: accounts[0]})
```

First the Truffle development environment has to be entered by using the command "truffle console". After that a new instance of our smart contract is stored in the variable "instance". Truffle allows for smart contracts to be interacted with just like using objects in object-oriented languages. In order to issue a transaction from these accounts, the next step is to store an array containing all the accounts that exist on the Ganache blockchain in the variable accounts. Among



```
truffle(ganache)> await instance2.newLocation.sendTransaction(0,0,0,{from:accounts[1]})
{
  tx: '0xa6667a5d48227d74d5146245edf90032aedd326da861e55964142fa4b889cee5',
  receipt: {
    transactionHash: '0xa6667a5d48227d74d5146245edf90032aedd326da861e55964142fa4b889cee5',
    transactionIndex: 0,
    blockHash: '0xe46918e2baf9eb965530179f2015a014700c871a1aef0085fa3f80bcabc3d9f4',
    blockNumber: 7,
    from: '0x1bc9419b66624d9547061075a943f9cfa2157342',
    to: '0x069e7a2d9c7685e54fe3117d2985e141f3756581',
    gasUsed: 195451,
    cumulativeGasUsed: 195451,
    contractAddress: null,
    logs: [],
    status: true,
    logsBloom: '0x000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000',
    v: '0x1b',
    r: '0xce5f8cf2b9046961dc902411b1a5722c19d5c232c768a0f6c662c096cd75229c',
    s: '0xe88593f282eaa620d2be57384102fdcef9feaba114584c37800c3248c639e65',
    rawLogs: []
  },
  logs: []
}
```

Figure 7.1: Example Transaction Receipt for Evaluation

these accounts, the account which represents the Mislayer (accounts[0]) has to send a transaction to the smart contract paying the reward to be distributed to all the Searchers who submitted sightings to the longest consistent path. The amount of Ether has to be given in the unit wei. In this example one Ether (10^{18} wei = 1000000000000000000 wei) is transferred. After that sightings can be submitted to the smart contract. Therefore accounts call the "newLocation" function on the contract instance and set the parameters according to their sightings. For each sighting one transaction has to be issued. In the end, the Mislayer calls the "createLongestPath" function on the contract, which leads to every account who submitted a sighting successfully to the longest consistent path receiving some Ether in return. For example, if ten sightings from ten different accounts are on the longest consistent path each account would have received 0.1 Ether.

In order to measure the cost of processing sightings and distributing the rewards for our smart contracts, Truffle offers to monitor the gas consumption a transaction had. As shown in Figure 7.1, each time a transaction is issued, Truffle outputs the so-called transaction receipt. The entry *gasUsed* shows how much gas has been used to run that transaction with the associated contract code.

Different sightings have been send to the contract in order to create certain scenarios. These scenarios will be referred to as topologies from now on. Each topology is presented as a graph. The size of the topology gets changed by the number of sightings it receives.

As presented in the previous chapter, we evaluate both approaches: tracking with and without reputation. In the following, the smart contract representing the approach: tracking without the reputation-based is referred to as *contract without security* and to the other one as *contract with security.*

In order to evaluate the scalability of these contracts, each topology has been tested with three sizes:

- small: a topology consisting of 10 sightings

- medium: a topology consisting of 25 sightings
- large: a topology consisting of 50 sightings

7.2 Gas Consumption

In this section, we want to briefly review what affects the gas consumption of a smart contract in Ethereum to provide a common understanding to interpret the following results.

In Ethereum, if an address sends a transaction to a smart contract, it triggers the execution of the associated code. An operation in a smart contract has different types of execution cost. Generally, two factors impact the gas consumption: the amount of processing required and the storage consumption of that code. Operations in Solidity, either are read-only operations or they manipulate the storage of the smart contract. Read-only operations consume a smaller amount of gas than operations which manipulate the storage. Therefore, modifying arrays consumes a large amount of gas. This is important to know, since our graph is represented through arrays and the biggest storage consumers are the arrays for storing the locations and edges of the corresponding graph. Since Ethereum penalizes large memory and runtime usage, frequent manipulation of the arrays can impact the gas consumption negatively. Therefore, when evaluating our algorithms we need to create topologies, which require from our contracts to store varying numbers of sightings leading to different costs.

7.3 Execution Cost

This section presents the main evaluation.

7.3.1 Performance Metric

Before explaining the results, we explain what we measured and the different topologies in more detail.

For each topology we measure:

- The *total gas cost* of executing the contract with security and without security on the topology, because this information provides knowledge about the entire cost of running the contract as it is.
- The *gas cost of sending sightings to the smart contract.* This information indicates how much each Searcher would need to pay when submitting a location to the tracking process.
- The *gas cost of distributing rewards distribution* for the sightings submitted. This shows how much the Mislayer would need to pay.

We present our results by creating one plot for each topology according to one of the three listed considerations above. Each plot contains the results for the contract using the reputation-based approach and the approach without reputation in order to detect the price of security. For the cost of distributing the rewards and the total cost, we created bar diagrams showing the different sizes of topologies. For the cost of sending a sighting, we created a graph.

In the following sections, we present the topologies and the associated results.

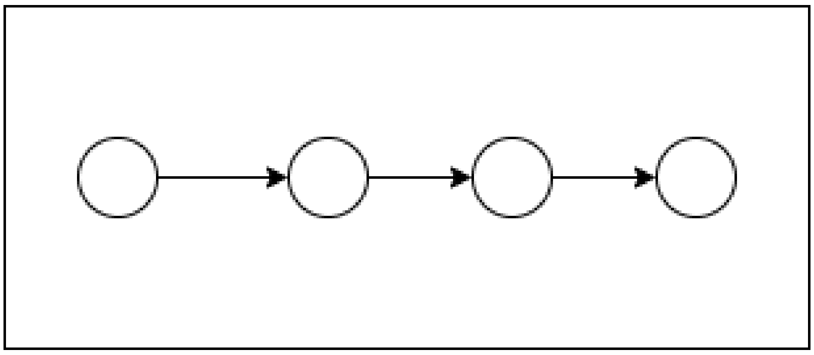


Figure 7.2: Example Topology: Single Longest Consistent Path

Remark 3 *All the absolute costs were measured in gas, as gas consumption solely depends on the EVM specification, enabling comparability of future research. Gas is paid in Ether. In order to illustrate the effective real-world cost associated, in addition we present the absolute cost in Ether and Euro. Due to the volatile exchange rate when converting gas into Ether, and Ether into Euro we used the exchange rate as of 10.08.2020 [Eth20]:*

- gas price*: 20 Gwei*
- Ether in Euro*: 1 Eth = 337 Euro*

7.3.2 Execution Cost of Deployment

The cost of deploying the smart contracts we evaluated is:

- contract without security: 0.0235 Eth (7,90 Euro)
- contract with security: 0.0241 Eth (8,10 Euro)

So when comparing the deployment cost, we see that only 2.5% more cost is required to deploy a contract with security compared to a contract without security.

7.3.3 Execution Cost of Single Longest Consistent Path Topology

This topology acts as a best-case scenario of the sightings a contract can receive. The graph created by the algorithms would contain only a single unambiguous path. This means, the algorithm needs to create and store the fewest amount of possible edges, which minimizes the cost. This is achieved by sending sightings in a manner that each node has at most a single incoming and outgoing edge, as shown in Figure 7.2.

Results

By applying both smart contracts to the topology having a single longest consistent path, we get the following results.

First, we present a comparison of the results for the total gas cost of the tracking procedure, as shown in Figure 7.3. As we can see, both smart contracts consume more in total when the size of the topology increases. As expected, as the topologies get bigger, each contract needs to store more information, which increases its cost. Comparing the cost performance of each size for each contract showed the following:

- **smart contract without security:** The total absolute cost for all topology sizes are: small topology 1797893 gas (0.036 Eth, 12 Euro), medium topology 4783988 gas (0.097 Eth, 32 Euro), and large topology 11063317 gas (0.22 Eth, 74 Euro). The additional relative gas cost from the small to the medium topology with 2.5 times more locations is 2.8 times as much. When comparing the small topology to the large topology, the topology is 5 times bigger and the cost increases 6.2 times.

- **smart contract with security:** The total absolute cost for all topology sizes are: small topology 2520834 gas (0.05 Eth, 17 Euro), medium topology 7123194 gas (0.14 Eth, 48 Euro), and large topology 16421464 gas (0.328 Eth, 111 Euro). The additional relative gas cost from the small to the medium sized topology is also 2.8 times as much consumption. When comparing the small topology to the large topology, here the cost increases 6.5 times.

So the relative cost increase of both contracts, which is occurring due to the topology getting bigger, is almost similar. In regards to further results for the other topologies, the contracts cost always scales a similar amount independent of the contract used.

When comparing the difference in cost between both contracts, we see that the contract with security consumes 45% more gas than the other contract for all sizes of the single longest consistent path topology. This is due to the fact that when the topology scales up, both contracts need to process and store equally many more locations. The only difference is that the contract with reputation needs to store the reputation, which leads to the array containing the reputation consuming a constant factor more storage and increasing the number of operations performed. As our results will show, 45% is the largest percentage increase for adding security to the contract across all the topologies. This is due to the overall consumption for this topology being comparably low, because there are not many edges, which can be created by the contract. As a result, the overhead created by the insertion of security has a large relative impact on the gas consumption.

The total gas consumption is made up by the gas cost of submitting a sighting performed by the Searchers, and the cost of distributing the rewards, paid by the Mislayer. By cumulating all the gas cost of processing sightings and the cost for reward distribution, the distribution is.

- **smart contract without security:** for the small scenario, 92% of the total cost impact the Searchers and 8% the Mislayer. For the large topology up to 97% of the cost are performed by the Searcher.

- **smart contract with security:** for all sizes of the topology, 81% of the total cost impact the Searchers and 19% the Mislayer.

Most of the computation necessary for tracking an object is performed by the Searchers since the creation of the edges in the graph is performed immediately after a submission of a sighting. This is an optimization step chosen in the implementation to reduce the overall storage consumption of the contract in order to decrease the overall cost for tracking a mobile object. Additionally, in this topology only a few edges need to be compared when creating the longest consistent path, which leads to the Mislayer's cost being relatively low.

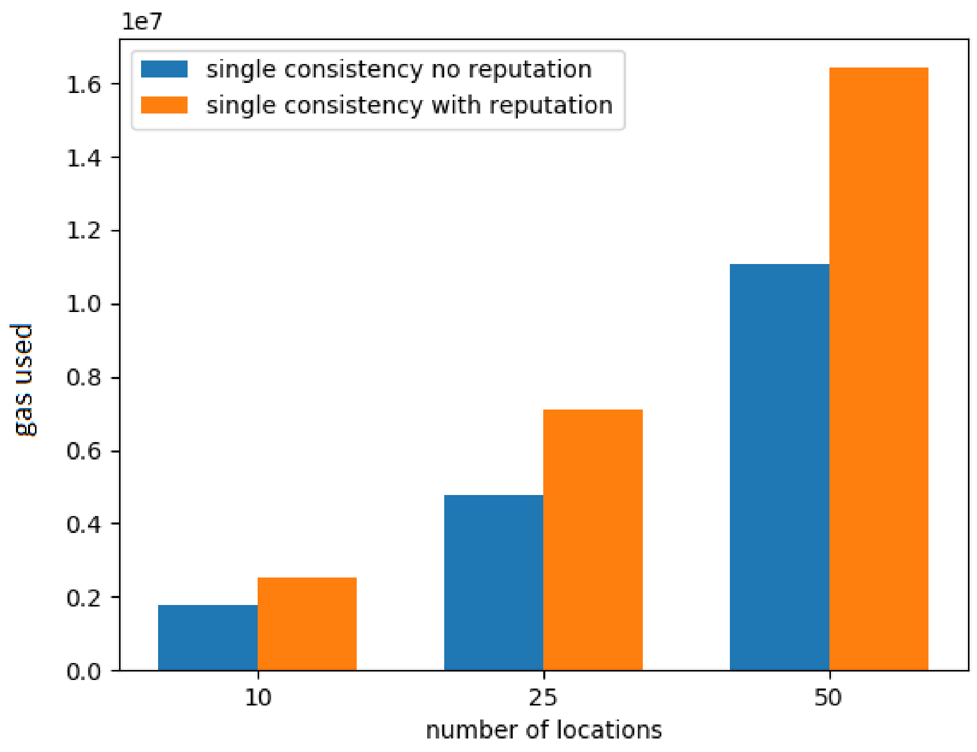


Figure 7.3: Total Cost of Tracking: Single Consistent Path Topology

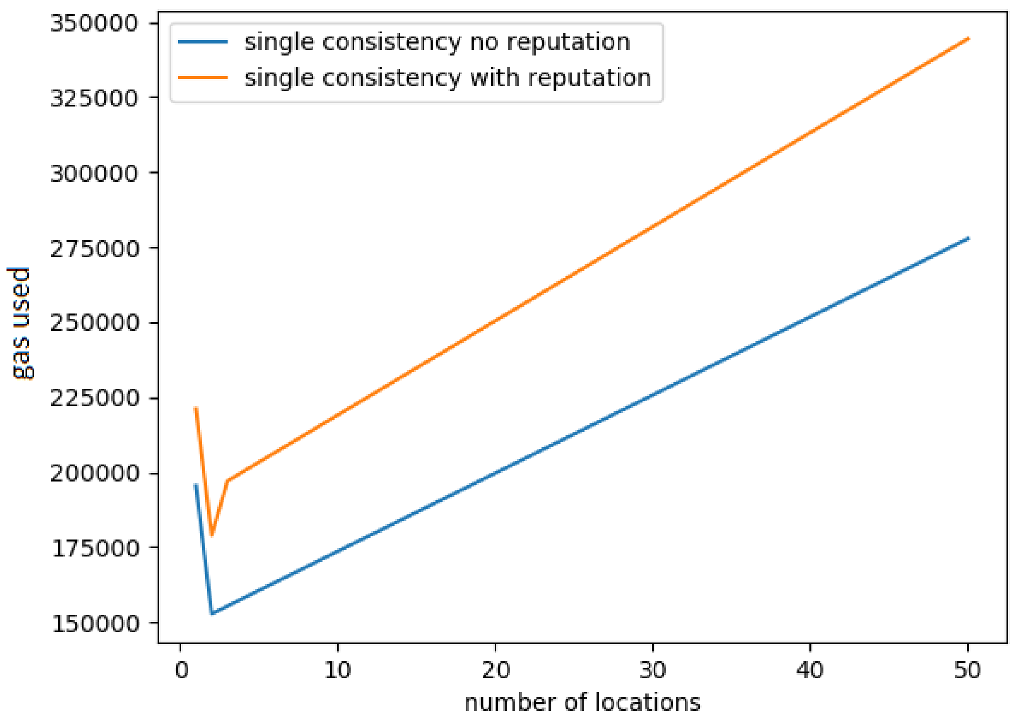


Figure 7.4: Cost of Inserting new Location into Graph: Single Consistent Path Topology

As seen in Figure 7.4, inserting a new location into graph costs a Searcher different amount of gas depending on how many locations have been transmitted before already. This shows to be the same for both contracts. Because as more locations get inserted into the graph, the corresponding data structures get bigger in size. This leads to either accessing elements or storing new elements consuming more gas. Therefore, the sender of the newest location needs to pay more than the Searchers before. The cost of inserting is linearly growing as more locations get submitted. An exception being the first location submitted which causes the dent in the graph, as seen in Figure 7.4. The reason being, the first location submitted to the contract initializes all the data structures required for the graph representation. This means, as all the other Searchers, the first Searcher needs to pay for the insertion of his sighting into the graph. Additionally, she needs to pay for the initialization of all associated data structures of the contract.

As seen in Figure 7.5 the cost of calculating the longest path and distributing the rewards is relatively low, when compared to the cost of submitting the sightings:

- **smart contract without security:** The longest path calculation and distribution costs for all topology sizes are: small topology 134209 gas (0.003 Eth, 0.9 Euro), medium topology 204169 gas (0.004 Eth, 1.4 Euro), and large topology 320273 gas (0.006 Eth, 2.2 Euro). The additional relative gas cost from the small to the medium sized topology is, with 2.5 times more locations, 1.5 times as many gas. When comparing the small sized topology to the large

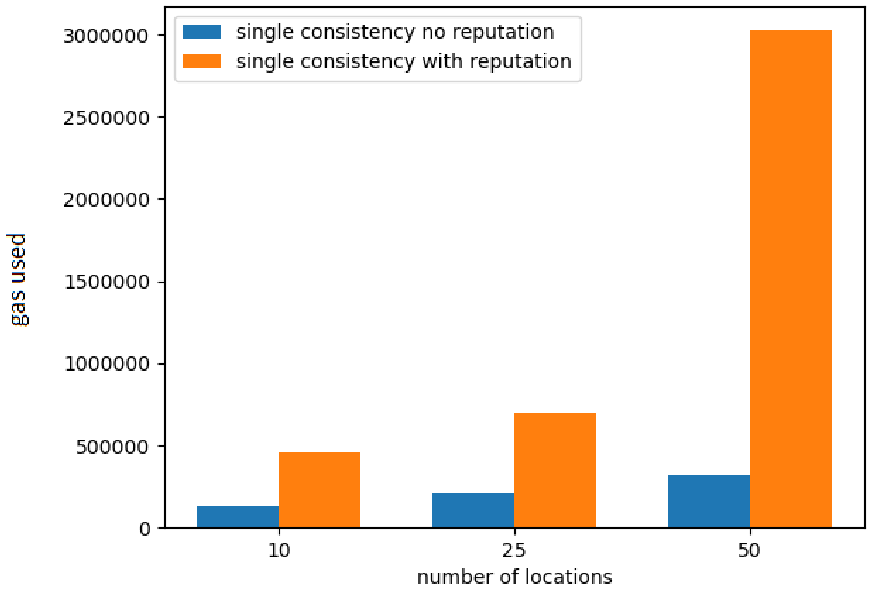


Figure 7.5: Cost of Finding Longest Path and Reward Distribution: Single Consistent Path Topology

sized topology, the topology is 2 times bigger and the cost increases 2.4 times.

- **smart contract with security:** The absolute cost for all topology sizes are: small topology 456924 gas (0.009 Eth, 3 Euro), medium topology 1397709 gas (0.03 Eth, 9.4 Euro), and large topology 2024234 gas (0.06 Eth, 20 Euro). The additional relative gas cost from the small to the medium sized topology consumes also 3 times as many gas. When comparing the small sized topology to the large sized topology, the cost increases 6.6 times.

Without reputation, the creation of the longest path requires a small amount gas. With reputation it requires large amount and the factor of increase is equivalent to the factor with which the total gas consumption increased.

7.3.4 Execution Cost of Full Consistency Topology

This topology acts as a worst-case scenario for the sightings a contract receives. The graph created contains multiple possible paths. Therefore, the contracts need to store a lot of edges, which increases the cost of storage. We achieved this by making each node having an edge pointing to its successor nodes, as seen in Figure 7.6.

Results

Now we present the results of the topology with full consistency.

The distribution between cost of Searcher and cost of Mislayer is:

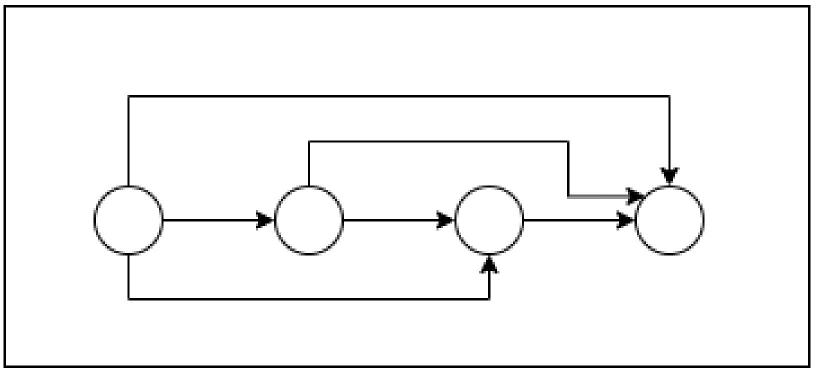


Figure 7.6: Example Topology: Fully Connected Graph

- **smart contract without security:** for all the sizes of the topology, 77% of the cost affect the Searchers and 23% the Mislayer.
- **smart contract with security:** for all sizes of the topology, 81% of the cost affect on the Searchers and 19% the Mislayer.

As the contracts need to compare many more different paths when creating the longest path, more cost is required to be performed by the Mislayer when calculating the longest path.

The cost of appending locations again is linearly increasing with the typical dent occurring after the first location, as seen in Figure 7.8 due to initialization. This time the difference between inserting locations to the graph is smaller, since appending locations requires a lot of comparison to other locations and this topology has all locations in reach of each other. This causes the additional computation for the security having a smaller impact compared to without security.

An interesting result is seen in Figure 7.9. The calculation of the longest consistent path even cost less without the reputation than with reputation. This occurred with no other topology we investigated. Our investigation of that phenomena found that applying the reputation-based contract on this topology leads to fewer locations which would be part of the longest path. Therefore, the calculation of the longest path consumed less gas.

When comparing the total gas consumption presented in Figure 7.7, the gas consumption is increasing with the size of the topology, as in the previous topology. Here the difference is that the contract with security costs only 10% more than without security compared to the 45% difference in the topology with a single consistent longest path. This is due to the full consistency topology having many edges which need to be created. This leads to storage and processing of the reputation having a smaller overall impact on the total gas consumption of inserting the locations.

When comparing the cost increase between each size of this topology, the factor of increase is almost similar to the other topologies:

- **smart contract without security:** The absolute cost for all topology sizes are: small topology 2251739 gas (0.04 Eth, 15 Euro), medium topology 6328411 gas (0.127 Eth, 43 Euro), and large topology 15223073 gas (0.3 Eth, 101 Euro). The additional relative gas cost from the small to the medium sized topology is with 2.5 times more locations it consumes 2.8 times as many gas. When comparing the small topology to the large topology, the topology is 5 times bigger and the cost increases 6.8 times.

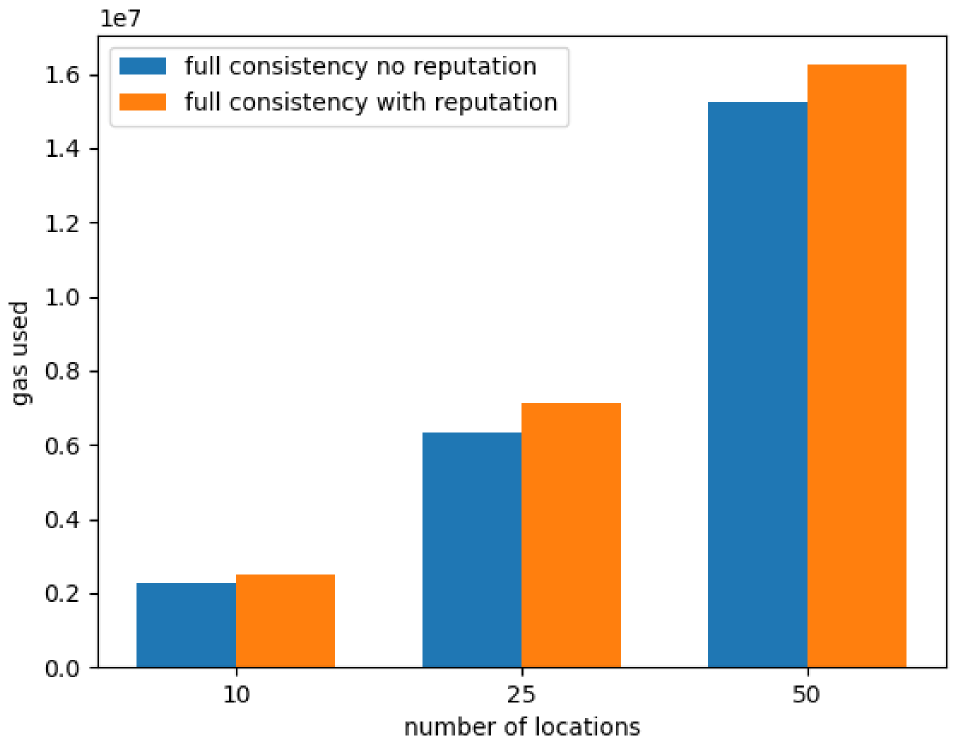


Figure 7.7: Total Cost of Tracking: Full Consistency Topology

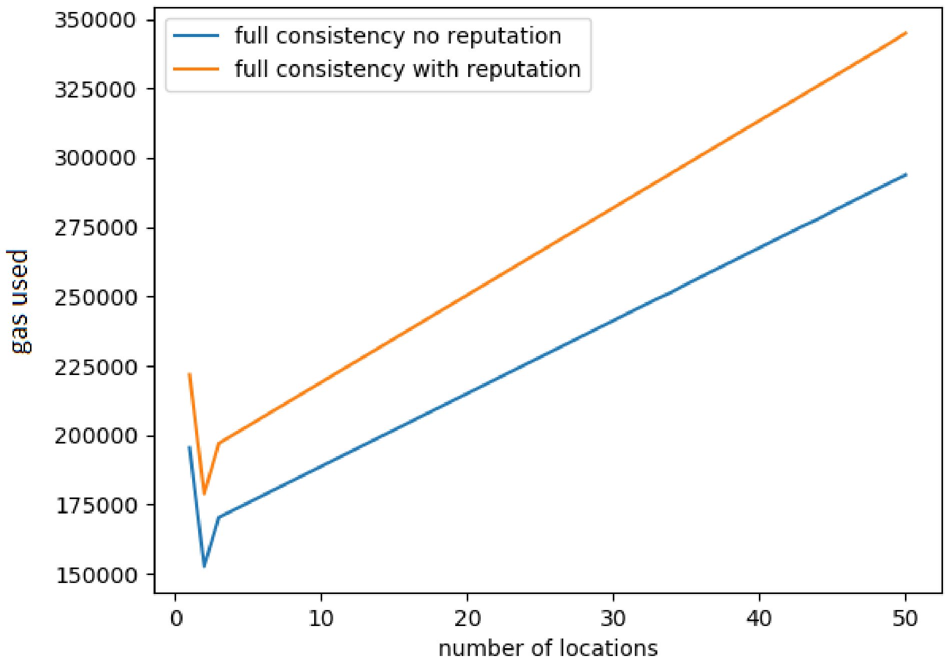


Figure 7.8: Cost of Inserting new Location into Graph: Full Consistency Topology

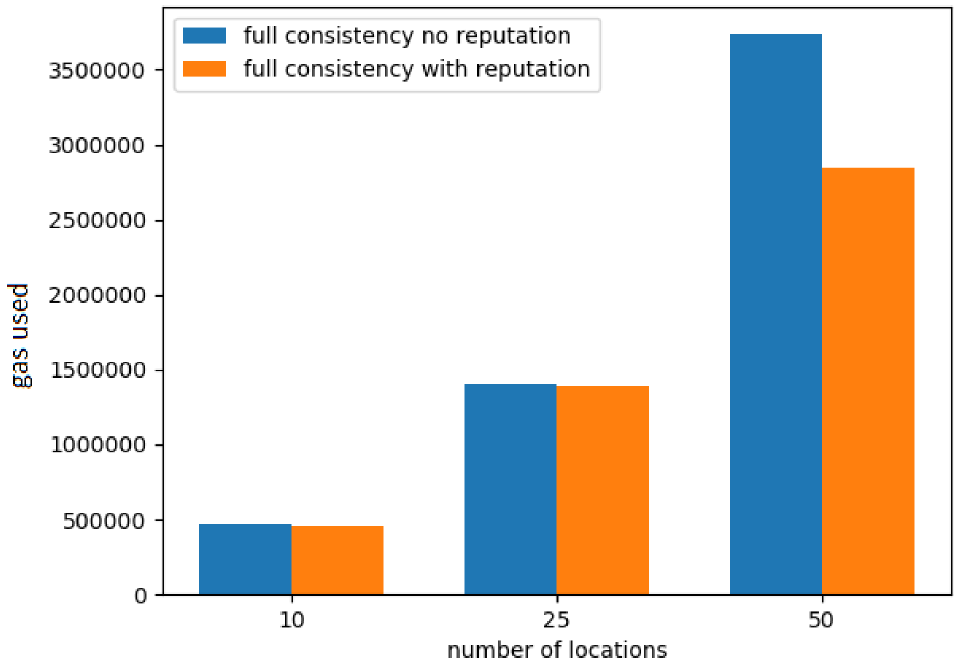


Figure 7.9: Cost of Reward Distribution: Full Consistency Topology

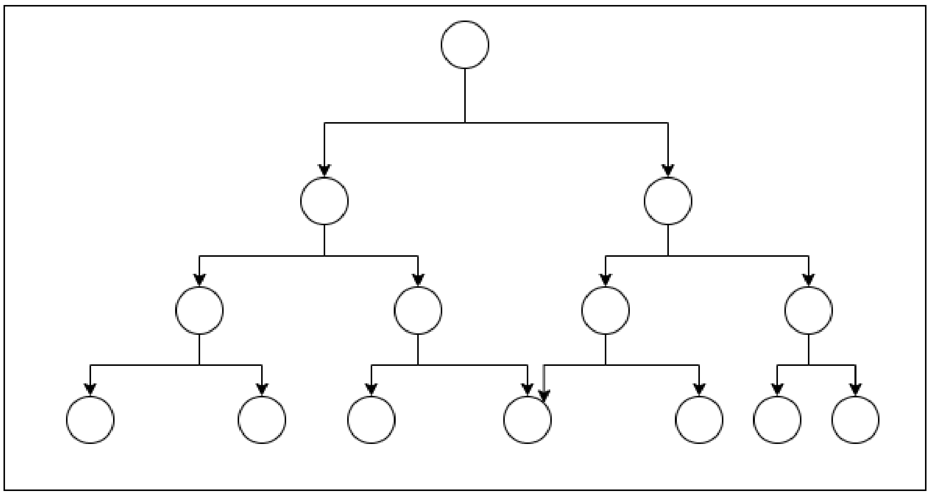


Figure 7.10: Example Tree Topology

- **smart contract with security:** The absolute cost for all topology sizes are: small topology 2521754 gas (0.05 Eth, 17 Euro), medium topology 7125404 gas (0.143 Eth, 48 Euro), and large topology 324963020 gas (0.35 Eth, 116 Euro). The additional relative gas cost from the small to the medium sized topology it consumes also 2.8 times as many gas. When comparing the small topology to the large topology, here the cost increases 6.5 times.

This indicates that independent of the topology, the total gas cost of tracking a mobile object scales with number of locations transmitted equally.

7.3.5 Execution Cost Tree Topology

This topology acts as a possible real world example, where the graph created would look like a tree. We decided to create a topology, which is a balanced binary tree for this. Each node has two similarly weighted successor nodes with an edge pointing towards them, as seen in Figure 7.10.

Results

When comparing the total cost, as shown in Figure 7.11, for the tree like topology we observe the following:

- **smart contract without security:** The absolute cost for all topology sizes are: small topology 1977961 gas (0.04 Eth, 13 Euro), medium topology 5358451 gas (0.12 Eth, 36 Euro), and large topology 12115812 gas (0.24 Eth, 81 Euro).
- **smart contract with security:** The absolute cost for all topology sizes are: small topology 2452181 gas (0.05 Eth, 17 Euro), medium topology 6451364 gas (0.13 Eth, 43 Euro), and large topology 1468341 gas (0.31 Eth, 103 Euro).
- The contract with security only consumed 20% more gas than the contract without security across all sizes of the topology.

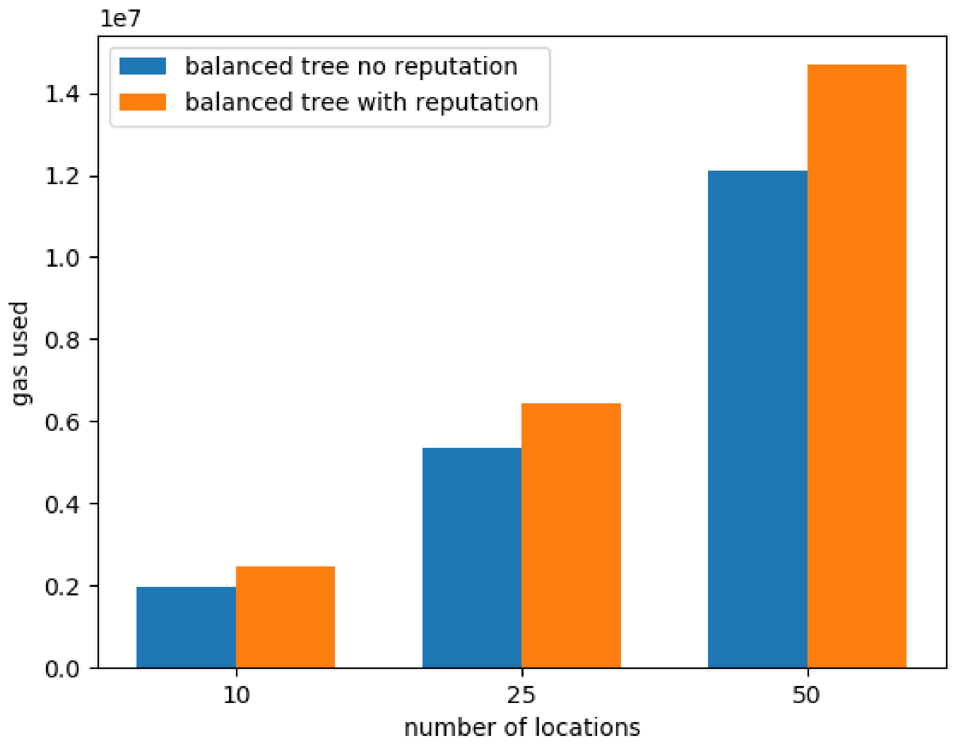


Figure 7.11: Total Cost of Tracking: Tree Topology

- The gas consumption increases by same factor for each size as with the previous topologies independent of which version of the contract is used.

When taking a look at the distribution between inserting locations cost and creating longest path cost, around 90% of the total gas consumption for this topology is on the Searchers accounts for appending locations for both versions of the smart contract. To understand this, we have a look at the cost, as depicted in Figure 7.12, showing the cost for inserting a location into the graph representation. This graph has the typical dent after the first location but a very much irregular linear growth with many multiple short drops. This happens due to the implementation of the Euclidean distance function which calculates the distance between locations submitted. When calculating the distance between two vectors, first the vectors get subtracted from another and from that result the absolute value is taken. In our implementation of the Euclidean distance function, if this subtraction leads to a negative result the calculation requires more computation. It flips the subtraction around to receive positive values. This extra step requires the euclidean distance to perform more computation thus, requiring more gas. Setting up this topology in our test environment requires the Euclidean distance function implemented to perform this extra computation for most sightings. Every time a drop occurred, this extra computation was not required, leading to the irregular growth. Whereas the previous topology has a constant linear growth due sightings requiring the same amount of computation when calculating the Euclidean distance. Thus having most of

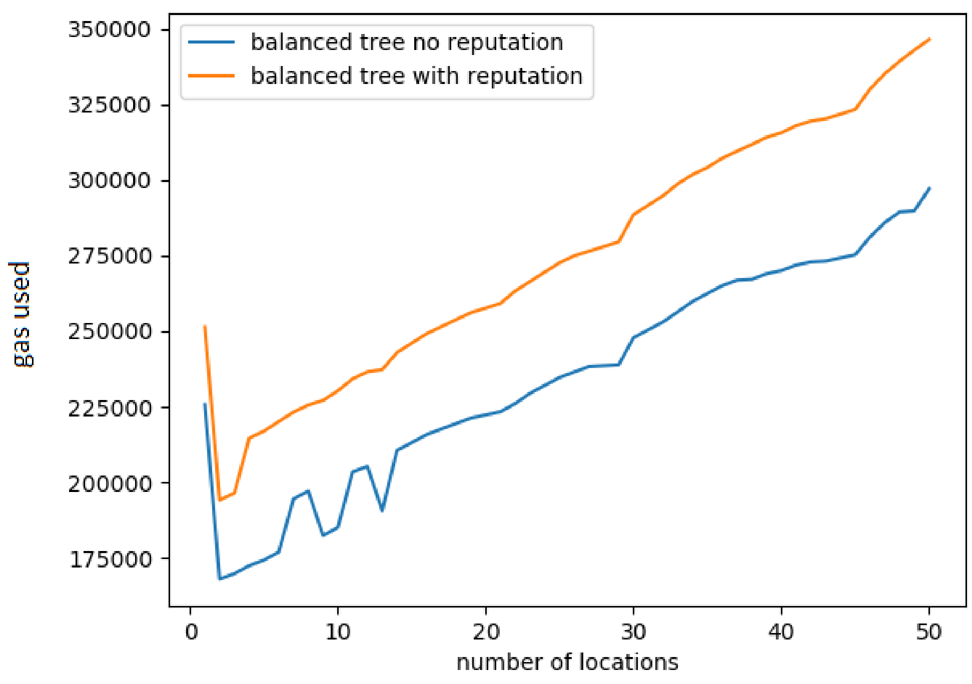


Figure 7.12: Cost of Inserting new Location into Graph: Tree Topology

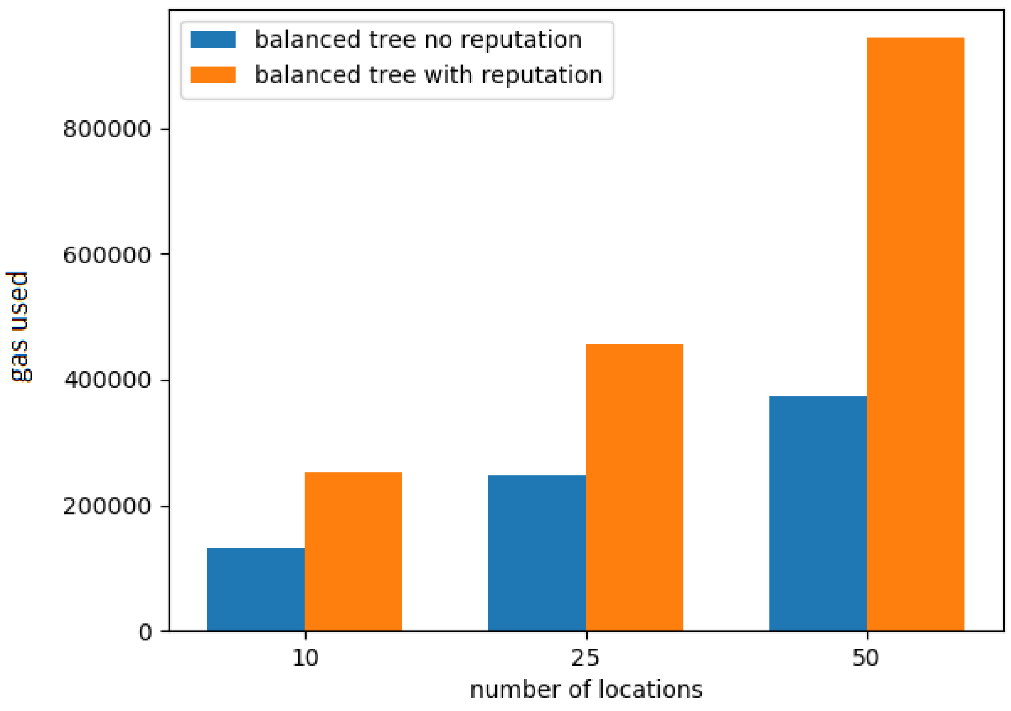


Figure 7.13: Cost of Finding Longest Path and Reward Distribution: Tree Topology

the distances being of different sign leads to an extra cost of approximately 10% . The cost of distributing the rewards behaved unsurprisingly, as shown in Figure 7.13. This is due to the cost of distribution of rewards being unaffected by the phenomena during the submission of sightings.

Another interesting finding is that storing zeros requires less storage in Solidity. In order to create the previous topologies only x-coordinates of a sighting needed to be changed. Therefore, all the y-coordinates were set to zero. When creating the tree topology all the sighting's x and y-coordinates were non-zero. This leads to the overall cost of submitting sightings to the longest path calculation path being higher, which in result leads to the overall distribution between the total cost of submitting sightings being 90%.

7.4 Summary

We summarize our findings in the following.

- Adding security to the tracking of mobile objects with our reputation-based approach can cost between 10% and 45% money for each tracking job depending on the sightings made. A general assumption of a 20% more cost showed to be appropriate.

- The Euclidean distance implemented depending on the sightings can increase the overall cost up to 10% more.
- As more sightings get submitted the overall cost is increasing, but the factor by which it increases is always the same independent of topology. Doubling the number of submitted sightings approximately multiplies the overall cost 2.5 times.
- Most of the cost for tracking a mobile object (around 80-90%) are caused during the submission of sightings, so they are paid by Searchers.
- Running the smart contract on the same topology but with different sightings delivers similar cost results.
- Transmitting sightings increases in price as more sightings have been transmitted before. The general cost increase is linear as more sightings get transmitted.
- The first sighting always costs more than the following few sightings because the first sighting initializes the contracts data structures.

As we have seen in the evaluation, the cost of submitting a location as a Searcher increases linearly. In our approach, each submission contributing to the longest consistent path the Searcher receives the same amount of Ether as an incentive. The problem being that some Searchers receive less than the fair share of the reward. Therefore, it is important to make sure that each Searcher, who contributed a location to the longest path, receives a fair share of the reward.

The amount of Ether spent for the submission of a location by each Searcher is publicly visible. An improved fair incentive mechanism could be implemented by looking up the gas expenses for each Searcher who contributed to the longest path and calculating individually the overall gas consumption for submitting locations. This requires the Mislayer to pay enough money upfront to the contract so that each successful Searcher receives the same reward. This requires the Searchers to trust the Mislayer that he acts honestly and estimated the cost correctly. If she does not want to spent too much on finding the location, he could limit the number of locations to be submitted, which however could decrease the effectiveness of tracking.

Chapter 8

Summary and Future Work

Finally, we summarize our findings and contributions in this chapter and give an outlook on future work.

8.1 Summary

The goal of this thesis was to design and implement a secure smart contract for a decentralized mobile target tracking application dealing with sybil attacks, and to evaluate the cost of executing these contracts with and without security features. First, we motivated the use of smart contracts for mobile crowdsensing. Especially the functionality of these concepts has been highlighted. Additionally we described and discussed related work. After presenting the background information necessary, in the main part of the work, we designed the target environment for our system and defined the problem to be solved. In particular, we identified sybil attacks as a crucial threat.

A reputation-based approach was designed as counter-measure and integrated into the Ethereum system. As an additional contribution of this work, we analysed and described specific attacks targeting our specific approach. In our implementation, we presented the challenges of the implementation in Solidity of our approaches and highlight decisions in the implementation made to decrease to overall cost of running both approaches.

We evaluated the different topologies we used to compare the cost of running the smart contracts with and without the reputation based approach to prevent sybil attacks. This has shown us that the overall cost of the proposed security mechanisms would cause between 10 and 45% more cost.

8.2 Future Work

The security of mobile target tracking smart contracts is a relevant topic for both, mobile crowdsensing and blockchain-based smart contract applications. This thesis is a contribution to both research fields, and it provides a conceptual basis, as well as algorithms as a starting point for future work. The following extensions might be of particular interest in the future.

In order to improve the reputation-based approach, it is important to implement concrete algorithms and formulas to calculate reputation as presented in this work. To this end, it would be interesting to investigate how reputation could be automatically inferred from previous actions of participants recorded in the blockchain history.

For the overall success of decentralized smart contracts, more efficient testing capabilities for smart contracts need to be developed, as currently the evaluation is very cumbersome.

Moreover, to enforce security for smart contracts, concrete implementations and evaluation of the advanced attacks presented is necessary.

Implementing a mobile application, which acts as an interface between the smart contract and the mobile sensor objects, is an important extension for the implementation of the overall system.

The operation of blockchain-based applications is paid in the associated cryptocurrency. Cryptocurrencies currently have a high fluctuation in value, which leads to make planning the monetary operational cost of that application very hard, or even impossible. Without knowledge of how much the operation of an application costs in the future, one interesting approach could be to define the reward in terms of ”hard” (stable) currencies, e.g., by automatically adding the current change rate to the smart contract. We see this as one of the biggest challenges, which blockchain-based applications need to overcome.

Finally, it is anticipated that due to its novel approach and early development, the field of decentralized mobile crowdsensing applications will continue to offer challenging open problems for a long time in the future.

Bibliography

[But20] Vitalik Buterin. A next-generation smart contract and decentralized application platform, 2020.

[Com19] Christina Comben. Three huge names that are making Ethereum their own, May 2019.

[CV17] Christian Cachin and Marko Vukolic. Blockchains consensus protocols in the wild. 07 2017.

[Doe17] Alex Doe. Transaction starvation in ethereum. `https://ethereum.stackexchange.com/questions/28590/transaction-starvation-in-ethereum`, September 2017.

[Eth20] Etherscan. The Ethereum blockchain explorer, 2020.

[Gha20] Roham Gharegozlou. Cryptokitties. `https://www.cryptokitties.co/`, 2020.

[Gro20] Truffle Blockchain Group. Sweet tools for smart contracts, 2020.

[GYL11] R. K. Ganti, F. Ye, and H. Lei. Mobile crowdsensing: current state and future challenges. *IEEE Communications Magazine*, 49(11):32–39, 2011.

[JCK15] R. John, J. P. Cherian, and J. J. Kizhakkethottam. A survey of techniques to prevent sybil attacks. In *2015 International Conference on Soft-Computing and Networks Security (ICSNS)*, pages 1–6, 2015.

[Kas17] Preethi Kasireddy. How does ethereum work, anyway? `https://medium.com/@preethikasireddy/how-does-ethereum-work-anyway-22d1df506369`, September 2017.

[Kra18] Peter M. Krafft. Focus: An experimental study of cryptocurrency market dynamics. In *CHI '18 Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems Paper No. 605*. ACM Press, April 2018.

[LSZ16] J. Liu, H. Shen, and X. Zhang. A survey of mobile crowdsensing techniques: A critical component for the internet of things. In *2016 25th International Conference on Computer Communication and Networks (ICCCN)*, pages 1–6, 2016.

[LYL$^+$18] J. Lin, D. Yang, M. Li, J. Xu, and G. Xue. Frameworks for privacy-preserving mobile crowdsensing incentive mechanisms. *IEEE Transactions on Mobile Computing*, 17(8):1851–1864, 2018.

[Mar19] Coin MarketCap. Top 100 cryptocurrencies by market capitalization. `https://coinmarketcap.com`, August 2019.

[Mor20] Henrique Moreira. Smart contracts and Solidity, 2020.

[RH18] Aniket Kate Ryan Henry, Amir Herzberg. Blockchain access privacy: Challenges and directions. `https://ieeexplore.ieee.org/document/8425613`, August 2018.

[Sai18a] Vaibhav Saini. ConsensusPedia: An Encyclopedia of 30+ Consensus Algorithms. `https://hackernoon.com/consensuspedia-an-encyclopedia-of-29-consensus-algorithms-e9c4b4b7d08f`, June 2018.

[Sai18b] Vaibhav Saini. Consensuspedia: An encyclopedia of 30+ consensus algorithms. `https://hackernoon.com/consensuspedia-an-encyclopedia-of-29-consensus-algorithms-e9c4b4b7d08f/`, June 2018.

[Scr19] First Scribe. Why are big corporations minting their own cryptocurrencies? `https://www.computenorth.com/why-big-corporations-are-minting-their-own-cryptocurrencies/`, June 2019.

[SLD15] C. Song, M. Liu, and X. Dai. Remote cloud or local crowd: Communicating and sharing the crowdsensing data. In *2015 IEEE Fifth International Conference on Big Data and Cloud Computing*, pages 293–297, 2015.

[Sta18] Stefan Stankovic. Cryptocurrency regulation in the european union. `https://unblock.net/cryptocurrency-regulation-in-the-european-union/`, July 2018.

[Vit17] Vitalik Buterin. A Prehistory of the Ethereum Protocol. `https://vitalik.ca/general/2017/09/14/prehistory.html`, September 2017.

[WCMA14] X. Wang, W. Cheng, P. Mohapatra, and T. Abdelzaher. Enabling reputation and trust in privacy-preserving mobile sensing. *IEEE Transactions on Mobile Computing*, 13(12):2777–2790, 2014.

[Woo20] DR. Gavin Wood. Ethereum: A secure decentralised generalised transaction ledger eip-150 revision. `http://gavwood.com/Paper.pdf`, 2020.

[WWDR11] H. Weinschrott, J. Weisser, F. Dürr, and K. Rothermel. Participatory sensing algorithms for mobile object discovery in urban areas. In *2011 IEEE International Conference on Pervasive Computing and Communications (PerCom)*, pages 128–135, 2011.

[WYJ+19] X. Wei, Y. Yan, W. Jiang, J. Shen, and X. Qiu. A blockchain based mobile crowdsensing market. *China Communications*, 16(6):31–41, 2019.

[XJX+18] L. Xiao, D. Jiang, D. Xu, W. Su, N. An, and D. Wang. Secure mobile crowdsensing based on deep learning. *China Communications*, 15(10):1–11, 2018.

[YY18] Fei-Yue Wang Yong Yuan. Blockchain and cryptocurrencies: Model, techniques, and application. `https://ieeexplore.ieee.org/document/8425613`, July 2018.

Declaration

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

place, date, signature