University of Zurich[UZH]

BACHELOR THESIS – Communication Systems Group, Prof. Dr. Burkhard Stiller

Collaborative DDoS Mitigation Based on Blockchains

Jonathan Burger
Zurich, Switzerland
Student ID: 13-746-698

Supervisor: Sina Rafati, Thomas Bocek
Date of Submission: TBD

University of Zurich
Department of Informatics (IFI)
Binzmühlestrasse 14, CH-8050 Zürich, Switzerland

ifi

Bachelor Thesis
Communication Systems Group (CSG)
Department of Informatics (IFI)
University of Zurich
Binzmühlestrasse 14, CH-8050 Zürich, Switzerland
URL: http://www.csg.uzh.ch/

Abstract

...

Contents

Chapter 1

Introduction

1.1 Motivation

1.2 Description of Work

The paper "A Blockchain-based Architecture for Collaborative DDoS Mitigation with Smart Contracts and SDN" proposes to use the Ethereum blockchain as a registry for IP addresses from which attacks are originating from. The data can then be read by ISPs who can filter out the malicious packets before they even reach the victim of the attack. This eliminates the need for additional architecture. We develop three variants of a smart contract and compare them to each other. Each smart contract serves the same purpose, the storage of a list of IP addresses. In some way, all variants accept the input of IP addresses and allow to read from it. Our main objective is to eliminate the need to set up and maintain a database for this purpose. The three variants are: 1. A smart contract that stores a list of all IP addresses on the blockchain, similar to the contract shown in the proposal. 2. A contract that points to a web resource containing the list of all IP addresses. 3. A contract that implements a bloom filter. All contracts should support both whitelists and blacklists. Both IPv4 and IPv6 addresses should be insertable. Furthermore, we would like to make it as easy as possible to modify the list. To round it off, our contract should make it possible to easily verify the identity of the reporter and prevent unauthorized modifications of entries on behalf of others.

1.3 Thesis Outline

In chapter 2, we discuss the characteristics of Ethereum smart contracts and their implications on our implementation. We discuss the properties and mechanics of our solution and look at security risks.

In chapter X, we start implementing the smart contracts based on the planning in chapter 1. The chapter describes the implementation technique, testing strategy and documents the established protocols.

In chapter 3, we benchmark the developed smart contracts for cost and speed and we compare. In addition, a generic cost model for smart contracts is being introduced and optimization possibilities are being explored. In chapter X, we make a recommendation for a smart contract variant and make a statement whether the developed approach based on our research is suitable for real-world use.

Chapter 2

Development

2.1 Contract 1: Native storage

The first variant that will be developed stores all IP addresses in the blockchain natively. The advantage is that all infrastructure is outsourced to the blockchain, making the solution completely decentralized. The disadvantage is that high gas fees make this the most costly solution and that the scale is limited in that the Ethereum blockchain is not designed to store data in the Gigabyte range.

2.2 Contract 2: Pointer to web resource

The second variant of the smart contract works around the space constraints and the big cost of the first variant by storing the list of IP addresses on a web resource and pointing to that on the blockchain. The advantage is that it works on a much larger scale. The disadvantage is that a web server needs to be set up and a separate standard has to be established describing the format of the web resource. This solution is also more prone to connectivity issues and does not take advantage of the decentralization and immutability of the blockchain.

2.2.1 Web resource

When we point to a web resource, we need to think about the design of the architecture that we have off the blockchain. By storing a URL in the blockchain, it is already implied that we plan to connect to the resource using HTTP(S), which is a suitable protocol for our needs. The format of the web resource we point to could be in any format imaginable. There are a few properties which are desirable: 1. Portability: To increase the adoption of our solution, use an already existing format like XML, CSV or JSON. 2. Upgradeability: The format should be able to be developed further in the future to enable more features, with backwards compatibility (which is hard if the format is CSV-like). 3. Streamability:

As we expect these lists to become very large, it would be useful to design the format in a way that it can already be partially evaluated while not yet fully loaded. Downloading a resource can take quite some time and we would like to not have to load the full list into memory. This is easy with CSV, as the file can be read line by line, but hard with XML or JSON, which needs to be fully loaded in order to be valid. This is a classic 'Pick two out of three'-scenario, as these properties are conflicting with each other. Finally, we need to think about whether the web resource should be immutable or not. Immutable means that the content of the web page can never change. Additions or deletions to the list are not supported, if the list had to be updated, it would have to move to a different URL and the entry in the blockchain has to be updated as well. This is not only a bad thing, as otherwise the client who accesses the resource has to worry about checking for updates. A hash can be generated for an immutable file, which could be stored in the blockchain as well and allow the clients to validate that the list has not been altered since it was registered in the smart contract, which is more in the spirit of the blockchain. If an asset is immutable, it can be easily outsourced to a CDN, which are harder to take down using a DDoS attack. In summary, immutability makes it slower (and potentially more expensive) to register changes, but has some advantages.

2.3 Contract 3: Bloom filter

The third variant of the smart contract is a standalone contract that solves the scalability issue by storing not the whole list of IP addresses but only limited information from which it can only be said with a likelihood whether an IP address was blocked. The bloom filter contract is a nice balance between leveraging the blockchain and using little space, however it gives up perfect accuracy.

2.4 IPv6 support

With IPv4 addresses being 32 bits long, only 232 combinations possible and the amount of free addresses is almost exhausted, which is the reason that we are currently in a transition phase from IPv4 to IPv6. Therefore it makes sense to support both formats. Two ways of supporting both IPv4 and IPv6 could be considered:

With IPv4 addresses being 32 bits long, only 232 combinations possible and the amount of free addresses is almost exhausted, which is the reason that we are currently in a transition phase from IPv4 to IPv6. Therefore it makes sense to support both formats. Two ways of supporting both IPv4 and IPv6 could be considered: The first is to represent IPv4 addresses as IPv6 addresses. The most widespread format is defined in RFC 3493: 80 bits of zeros, 16 bits of ones and then the IPv4 address. So for example the IPv4 address `46.101.96.149` would be `0:0:0:0:0:ffff:2e65:6095` in IPv6 hex representation. In fact, this notation is supported by the Linux kernel and macOS natively, in a web browser `http://[::ffff:2e65:6095]` will display the same website as `http://46.101.96.149`. The other option would be to use a flag indicating whether an address is IPv4 or IPv6.

If mainly IPv4 addresses are stored in a contract, this would yield a space benefit. Given that there is already a standardized solution for notating IPv4 in IPv6 format, we are proceeding with the first option.

2.5 IP verification

IP verification In all variants of the smart contract, owners of destination IP addresses can store source IP addresses they want blocked in the smart contract. In order to establish trust, we are interested in automatically verifying the ownership of the destination IP addresses. This is a challenging task, as is explained in this section. Using certificates to validate IP ownerships in Solidity is currently not practical for at least two reasons. It is a computationally expensive task that would require more gas than the gas limit allows and there is no implementation of certificate verification in Solidity. Certificate verification, as it is most commonly done with OpenSSL, would have to be ported ported to Solidity, which is complex. There is however a proposal to add certificate validation on a language-level (Ethereum Improvement Proposal #74). As of writing, it looks like the intent is to add BigInt to the Ethereum Virtual Machine, which could enable certificate validation. Since everything in the blockchain is public including stored certificates and IP addresses, it is not required that certificates are validated in Solidity, it can be done off the blockchain. The second challenge is obtaining a certificate. As there is no way to mathematically or logically prove that somebody is the owner of an IP (it can be spoofed), the certificate process of domains could be applied to IPs: There are certificate agencies (CA) whose business is to provide a service validating the ownership of a domain. These CAs need to make investments in establishing a system that securely validates a domain and manages the certificates that are issued. Only if a CA has high standards their certificates are trustworthy. That is why nearly all CAs for domains issue certificates for a fee or need to generate revenue by sponsorships. Although it is technically possible to issue a SSL certificate for an IP address, it is very uncommon. GlobalSign[1] is the only provider known to us that does this, and requires that the IP is registered in the RIPE database. A system that currently exists for domain could be introduced by the industry for IP addresses, but is very complicated and defeats the purpose of using the blockchain as the original idea was to remove the need for additional infrastructure.

2.6 Validating Smart contracts during development

Developing a smart contract requires a compiler and a blockchain on which the developer can test whether the smart contract behaves as desired. A compiler, such as solc, will warn about syntax errors and does not compile invalid Solidity code, indicating to the developer that there is an error in the code. While compilers provide a first layer of assurance by only compiling valid contracts with valid syntax, it is still possible to write code with behavioral bugs and security vulnerabilities. As with any software, developers

[1] https://support.globalsign.com/customer/portal/articles/1216536-securing-a-public-ip-address—ssl-certificates

need a workflow which allows them to test their changes fast and efficiently. The main ethereum blockchain is unsuitable for validating the correctness of the code manually. There are significant costs to deploy a contract to the blockchain, also it requires some time to receive the confirmations. It is also frowned upon to use the main blockchain for testing purposes, as all network participants need to download all blocks. This is not the case in the testnet, which is a seperate blockchain for testing purposes. Even better, there is a test framework called TestRPC that makes it possible to set up a local blockchain for testing. Using TestRPC, it is possible to simulate deployments and transactions of smart contracts with no confirmation delay. Out of the box, TestRPC sets up multiple ethereum accounts, so it easy to switch the message sender address and test whether the security features of the developed smart contract are working as desired. This makes TestRPC the ideal blockchain for developing. In addition to manual testing, the robustness should be validated by unit tests, just like in any other piece of software. This allows to automatically validate that all assertions are still true when a change is introduced to the contract. A robust contract should have unit tests validating that normal use of the contract features result in correct behaviour, as well as edge cases and abuse of the contract features handled correctly. Examples of that are: Calling a function without permission, calling a function more often than expected, calling a function with unexpected arguments, such as null arguments, wrong types or a big payload. I have used TestRPC, the Javascript "solc" compiler, and the "ava" testing framework to set up a test environment. Each test is completely atomic and independent from other ones, that means that for each test, a separate TestRPC blockchain is created and the scenario is run on it. After the test is finished, the blockchain and the accounts get destroyed. This complete isolation of each test is good practice, as side-effects can be ruled out. With "ava", the tests can be run in parallel, making it faster to run the whole test suite. However, there are many other generic test frameworks that work as well and have their own advantages and disadvantages. One interesting framework is Truffle[2], which provides helpers for developing and testing Solidity contracts. Truffle tries to make the process described in this section easier and looks very promising, but is still in beta. For the reason that the framework is still very much in development and changing, and that I have only discovered it after already having a solution working, I did not use it. For real-world smart contract applications, testing is not enough, critical contracts should be audited by computer security professionals before being deployed. As the aim of this thesis is to document a proof-of-concept DDoS mitigation application, no audit will be conducted.

2.7 Security considerations with Solidity

Code vulnerabilities are more common in Ethereum than in other environments. The whole blockchain data is public, and serious contracts are made open source in order to create trust for the people who use it. Therefore applications are auditable and bugs are more likely to be found.The application must also regulate itself and can in most cases not recover from an attack. In addition to that, the language Solidity aims to have a

[2]http://truffleframework.com/

syntax similar to JavaScript and is therefore quite loose with enforcing the correct type. Static code analysis tools (such as Solium) are also not yet up to par with other languages. In this section, we will explore some common "traps" that can lead to security exploits, partially taken from a list compiled by the co-founder of Ethereum [1].

2.7.1 Constructor naming exploit

Simple human mistakes can lead to the contract being vulnerable. The 'Rubixi' contract has a different constructor name than contract name. Therefore the constructor does not get called on deployment, but can be called as a transaction. If done so, the caller of the transaction becomes the owner of the contract. Because of more unfortunate design of the contract, the bug became not immediately apparent and the Solidity contract was correct.

```
contract Rubixi {
    function DynamicPyramid() {
        creator = msg.sender
    }
}
```

2.7.2 Public data

A misconception is that stored data can be made private on the blockchain. A rock-paper-scissor contract which people used for gambling turned out to have a trivial flaw where the first move could be seen - rock would have the value `0x60689557` and scissor and paper have a different one. The main takeaway is that in our case, the stored IP addresses will be public (although maybe obfuscated) to everybody. Even attackers can determine the list of IPs that are blocked. The contract should be created in a way that minifies the usefulness for attackers, for example the pattern of the IPs should not make the way how IPs are being blocked predictable.

2.7.3 Loops

The section "Security Considerations" of the Solidity documentation warns about loops that have a non-fixed amount of iterations. With a high number of iterations, the block gas limit could be reached and according to the Solidity documentations "cause the complete contract to be stalled at a certain point". In the reference contract, the function blockIPv4() has a loop whose amount of iterations are dependant of a transaction argument. The contract has to be tested with a big payload and a solution needs to be developed. Furthermore, even for constant functions (which are executed locally and have no gas limit), there can be bugs. If, in the reference contract there would be a slight change:

```
function blockedIPv4() {
    uint32[] memory src;
-    for (uint i = 0; i < drop_src_ipv4.length; i++) {
+    for (var i = 0; i < drop_src_ipv4.length; i++) {
        src[src.length] = drop_src_ipv4[i].src_ip;
    }
    return src;
}
```

We would introduce a bug. `var` would be interpreted as `uint8`, and if there are more than 255 entries in the `drop_src_ipv4` array, there would be an overflow leading to an infinite loop. This bug can be simply avoided by disallowing `var` in the code or by being aware of the issue.

2.7.4 Re-Entry bugs using .call()

The `.call()` method in Solidity is dangerous as it can execute code from external contracts. This weakness was first exploited in the DAO smart contract, which had over 50 million USD stored in it. The main takeaway from this incident is that `.call()` can call any public function, even the function from which it was called from, leading to recursion. The following example is not safe:

```
function withdraw(amount) {
    if (balances[msg.sender] < amount) throw;
    msg.sender.call.value(amount);
    balances[msg.sender] -= amount;
}
```

Let's say that the balance of an account is 10 and that account calls the withdraw method. The account can execute arbitrary code when `msg.sender.call()` is called on line 3, so `withdraw()` can also be called again before line 4 of the above snippet is reached. An attacker could withdraw more than what its balance is. The vulnerability could be mitigated by using `.send()` instead of `.call()` in the example above.

2.7.5 Call stack depth attack

The call stack depth gets increased when a function calls another function, and if a function returns, the call stack depth gets decreased. A long call stack can be produced by excessive recursion. Solidity has a 1024 call stack depth limit - this means that Solidity could for example not compute the 1025th fibonacci number using recursion. The deterministic depth limit of Solidity proves to be an attack vector. An attacker could craft a function that calls itself 1023 times and on the 1024th time calls another, vulnerable function, that stops execution as soon as a subfunction is called, because the maximum call stack size is reached. An attacker might force a function to only partially execute, which is a problem,

if for example a withdraw function is composited of a subfunction that sends funds and a function that decreases the users balance. The core development team of Ethereum wants to solve this problem on a language-level in EIP #150 soon, therefore it will not be further discussed here.

Chapter 3

Cost model

3.1 Cost variables

Computing costs occur for blockchain applications, and because the blockchain is decentralized, there needs to be a reward system for the people that provide the service of confirming the transactions. Therefore, a blockchain application can be significantly more expensive than a comparable centralized service. The costs of a single transaction on the Ethereum blockchain is dependant on at least 5 variable factors: 1. The operations being done 2. The gas costs assigned to those operations 3. The desired speed of which transactions are executed 4. The value of ether 5. The compiler being used.

When a node wants to submit a transaction to the blockchain or create a new contract, a certain amount of 'gas' has to be offered to so called 'block miners'. For our purpose, we model gas as a real money cost and treat it as a currency. We just care about the conversion rates and the amount we have to pay.

1 The operations being done: The amount of gas that has to be paid is determined by the assembly code of the transaction. There are currently 31 possible operations, each one has a gas price tag attached to it.

2 The gas costs assigned to those operations: Gavin Wood has created a list of 31 operations and their prices, and these are in use today. For example, a transaction costs 21000 gas and a SHA3 hash operation costs 30 gas. In general, we can state that more complicated contracts and transactions cost more gas. Interestingly, the different gas prices for operations are not necessarily proportional of the actual computational work needed, but were set by the Ethereum core team and accepted by the majority Ethereum users. Because of the non-proportionality, some gas costs will change in the future for rebalancing purposes when the majority of nodes upgrade to the Metropolis hard fork, which will likely happen as this hard fork is pushed by Ethereum inventor Vitalik Buterin. For example, the cost of a CALL operation will be increased from 700 to 4000 gas in the foreseeable time. In summary, gas prices are determined by the community and are heavily influenced by the Ethereum Foundation. Gas prices can change over time, so that the very same transaction can cost more or less gas depending on the time when it is committed.

3 The desired speed of which transactions are executed: How much a gas costs is dependant on the users of the Ethereum network and unlike the previously mentioned operation price table, the gas-to-ether conversion rate is not hardcoded into the Ethereum, but determined by the users dynamically. The default configuration of go-ethereum[1], which is the standard implementation for Ethereum, sets a gas price of 20 shannon[2], although it is configurable. When the gas price is set to 20 shannon in a miners client, the miner will only mine transactions that offer a gas price of 20 or more. When the gas price is set to 20 shannon in a client that wants to send a transaction, the client will automatically offer that amount. The average gas price chart from etherscan.io shows that the clients of the network in general go with the default configuration value, 1 gas equals to approximately 23 shannon on average. The reason that the real average gas price is a bit higher than 20 shannon is because clients can voluntarily offer a higher price for a transaction, like for example a gas price of 24 shannon. This has the advantage that the transaction likely will be fulfilled quicker. The mechanics of mining are similar to those of a stock market. Let's say that a stock is worth 100 USD and is being actively traded on an exchange. If a buyer is willing to buy the stock at an overvalued price of for example 102 USD, his offer will jump to the top of the order book and will be filled the quickest. For our application, it could be desirable to pay a premium for faster transaction mining. The default price of gas is regularly debated in the Ethereum community and was already changed once in March 2016 when the price for Ether soared. The gas price was then reduced from 50 shannon to 20 shannon. Now that the Ether price is in a different magnitude, another correction might be included in the next hard fork. At the time of writing, the gas price equilibrium would be 16 shannon according to the calculation from etherchain.org. On May 7th 2017, ethgasstation.info announced[3] that 10% of the network hashpower is accepting a gas price of only 2 shannon. The site is promoting lower gas prices and is encouraging Ethereum users to change the settings to allow lower gas prices. According to the same site, a transaction that offers 2 shannon per gas, the mean transaction confirmation time is 119 seconds. For 20 shannon and 28 shannon, the transaction confirmation time is 44 and 30 seconds respectively.

[1] https://github.com/ethereum/go-ethereum/blob/fff16169c64a83d57d2eed35b6a2e33248c7d5eb/eth/config.go#L45

[2] Shannon is 10^{-9} Ether. A shannon is also known as a 'Gwei'. In the context of gas prices, 'shannon is more commonly used.

[3] https://medium.com/@ethgasstation/the-safe-low-gas-price-fb44fdc85b91

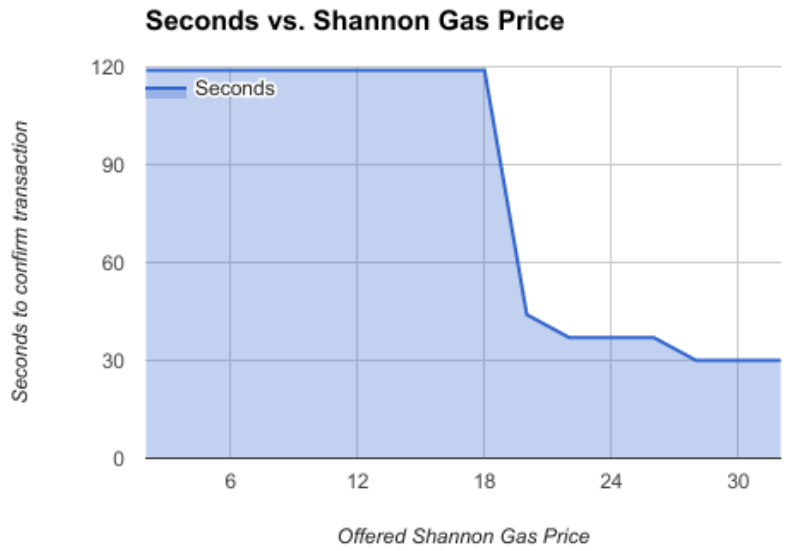

Figure 3.1: Average time until transaction is confirmed. Made with data from ethgasstation.info on May 8th.

In Figure 3.1, we can see that a node that offers a higher reward for a transaction will get confirmed 4 times faster on average. The user has to decide based on that information, how much he wants to pay.

4 Price of Ether: Ether can either be mined or can be purchased on an exchange. The Ether price on exchanges is highly volatile. On January 1st 2017, the price of an ether was $8.22. on May 7th 2017, it was $96.24, according to Coinbase. Ether has also already seen a price drop of over % when a smart contract called "TheDAO" was hacked.

5 The compiler being used: The final variable is the deviation of gas estimates when using different compilers. This was discovered when we received different gas estimates for the same contract when upgrading the compiler. To illustration the effect, let's consider the following smart contract:

```
pragma solidity 0.4.8;

contract DdosMitigation {
    struct Report {
        address reporter;
        string url;
    }

    address public owner;
    Report[] public reports;

    function DdosMitigation() {
        owner = msg.sender;
    }
```

```
    function report(string url) {
        reports.push(Report({
            reporter: msg.sender,
            url: url
        }));
    }
}
```

The gas estimate was 318'552 gas when compiled with Version 0.4.8 of the Solidity Compiler (solc), but slightly different in other compilers:

Change, *ceteris paribus*	**Cost**
Base case	318552 gas
Compiling with solc 0.4.9 instead of solc 0.4.8	329054 gas
Estimate given by Ethereum Wallet 0.8.9	318488 gas
Deployed in Main Network (actual cost)	318487 gas

Even when removing just whitespace from the contract, sometimes the gas cost can change:

Change, *ceteris paribus*	**Cost**
Changing the name from "DdosMitigation" to "Ddos"	318552 gas (no change)
Removing line 14 (whitespace)	318488 gas
Removing line 10 (whitespace)	318552 gas (no change)

Given that all the deviations that we have discovered skewed the total gas cost by not more than 4%, we are omitting this variable from our cost model for simplicity.

3.2 Cost model

We model the set of operations that a transaction executes as $\sigma = (\sigma_1, ..., \sigma_n)$.

The corresponding fees are $f = (f_1, ..., f_n), \forall f_i \in G$, where G is the fee schedule defined in the Ethereum Yellow paper. The amount of gas a transaction consumes is:

$$C = \sum_{i=1}^{n} \sigma_i \cdot f_i$$

The value of C is best determined by using the `estimateGas()` function made available by Ethereum clients.

A transaction uses at least 21'000 gas and the maximum amount of gas that can be used in a transaction is 3'141'592 (set to be raised to 5'500'000[4] in the next fork). At the moment, this means that $C = [21000, 3141592]$ is always fulfilled.

[4]https://github.com/ethereum/EIPs/issues/150

We model the price as the variable α. Since, according to ethgasstation.info, there are 3 possible speeds, we define constants representing the minimum cost we have to pay in order to get that speed: $\alpha_{cheap} = 2 \cdot 10^{-9} Ether$, $\alpha_{average} = 20 \cdot 10^{-9} Ether$, $\alpha_{fast} = 28 \cdot 10^{-9} Ether$

We denote the price of an ether as ETH. As mentioned above, the deviations of different compiler are negligible.

Putting these values together gives us the total cost $C \cdot \alpha \cdot ETH$. For example, if we create a contract that costs 500'000 gas to deploy and pay averageper gas, and the price for Ether is \$50, the price to deploy that contract is $500'000 \cdot 10^{-9} \cdot \$50 = \$0.025$.

3.3 Cost estimation for DDos mitigation

The price that we pay for our contracts varies as we develop the contracts and the gas and ether prices changes, but we can estimate the magnitude.

For a contract that stores IP addresses directly, the gas estimate was $Deploy\ cost = 700113$. The cost of adding 1 IP in a transaction is 104046. When adding 2 IP addresses in the same transaction, we save 1 transaction gas cost of 21000 among other minor savings. The maximum is 46 IP addresses, then the block gas limit is reached (which, theoretically, users of the network can increase) and another transaction has to be done. In our case, the cost is $Add\ IP\ Cost = 66322$ per IP and $Transaction\ cost = 22660$ per transaction. So the price to store x IP addresses is $c = Deploy\ cost + (x \cdot Add\ IP\ Cost) + \lceil \frac{x}{2} \rceil \cdot Transaction\ cost)$

In a bad case where Ether is expensive $ETH = \$100$ and we want fast transactions $\alpha = \alpha_{fast}$, it costs \$3.82 to do one transaction with 20 IP addresses ($C \approx 1364194$). In a good case where Ether is reasonably cheap $ETH = \$50$ and most nodes have started accepting cheaper transactions $a = a_{cheap}$, the transaction costs \$0.13. Even though the average gas price so far is approximately the default value set by clients, it seems realistic that users over time manually adjust the gas price according to the Ether price. If ether is higher, the gas price should be lower. Using $\alpha = 0.16$ as the gas price (the value that etherchain.org deems as the equilibrium) and a Ether price of $ETH = \$90$ (the price at the same time), we get a transaction cost of \$1.96.

For a contract that stores a pointer to a web resource containing IP addresses, there is a flat cost whenever the list is updated. The *Deploy cost* Deploy coststill applies. The transaction cost for adding a URL with 64 characters was $Transaction\ cost = 121466$ in our test. This gives $C = Deploy\ cost + \#\ of\ transactions \cdot Transaction\ cost$.

Chapter 4

...

Chapter 5

Evaluation

Chapter 6

Summary and Conclusions

Bibliography

[1] Autor: Vitalik Buterin, https://blog.ethereum.org/2016/06/19/thinking-smart-contract-security, 19. Juni 2016.

Abbreviations

AAA	Authentication, Authorization, and Accounting

Glossary

Authentication

Authorization Authorization is the decision whether an entity is allowed to perform a particular action or not, e.g. whether a user is allowed to attach to a network or not.

Accounting

List of Figures

List of Tables

Appendix A

Installation Guidelines

Appendix B

Contents of the CD