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: 15.8.2017

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

Attacken wie Distributed Denial-of-Service (DDoS) stellen ein immer grösser werdende Gefahr dar für Computernetzwerke und Internet-Services. Existierende Strategien zur Bekämpfung von DDoS-Attacken sind ineffizient aufgrund mangelnder Ressourcen und Inflexibilität. Blockchains wie Ethereum ermöglichen neue Methoden zur Mitigation von DDoS-Attacken. Mittels Smart Contract können IP-Addressen von Attackierern auf einer dezentralisierten Plattform signalisiert werden, ohne zusätzliche Infrastruktur einzusetzen. Diese Arbeit dokumentiert die Entwicklung mehrerer Smart Contracts zur Signalisierung von DDoS-Attacken und vergleicht sie, bespricht die Ethereum-Umgebung und ihre Auswirkungen auf die Architektur, gibt Auskunft über Leistung sowie Kosten und evaluiert die Machbarkeit und Wirksamkeit einer blockchainbasierten Lösung zur Bekämpfung von DDoS-Attacken.

Acknowledgments

I would like to thank my supervisors, Thomas Bocek and Sina Rafati, for helping me find this topic and continually guiding me during this work with ideas and knowledge and even helping me with formal language.

I would also like to thank the other members of the Communication Systems Group doing research on blockchains who have provided me with their input as well.

Finally, I would like to thank Prof. Burkhard Stiller for making it possible to write my bachelor thesis at the Communication Systems Group and for providing feedback as well.

Contents

Chapter 1

Introduction

1.1 Blockchains, Ethereum and Smart Contracts

A Blockchain is a decentralized database consisting of a chain of cryptographically secured units, called 'blocks'. Each block references the previous block and cannot be modified without breaking the subsequent blocks. A blockchain is continuously growing, as new data is inserted at the end of the chain. The most popular application for blockchains are digital cryptocurrencies, the most widely used implementation is Bitcoin. With Bitcoin, which had its breakthrough in 2008, network users can exchange tokens securely over a completely decentralized protocol. Because the technology is useful, the tokens have real money value, the total market capitalization of Bitcoin is several dozen million USD.

Ethereum [1] is a blockchain protocol that is inspired by Bitcoin, but not only allows for sending and receiving of tokens, but also offers a scripting language called Solidity, which allows anyone to write programs which can be run on the blockchain. Examples for applications that could run on Ethereum are games like Tic-Tac-Toe or Poker, finance applications like venture capital funds and Initial Coin Offerings (a company raising funds by selling shares of it to investors).

Smart contracts are contracts expressed in code that can automatically enforce the terms of the contract. Ethereum smart contracts allows for storage of arbitrary information and makes it possible for users to send transactions that mutate the storage. By writing the proper code, the creator of the smart contract can control the permissions of the users and the conditions and behaviors of the mutations. Ethereum enables turing-complete programming on the blockchain, which enables a wide variety of possible applications, including a collaborative DDoS mitigation solution, which this thesis is about.

1.2 Denial of Service and DDoS

A Denial of Service (DoS) is when a machine or network resource that should be online is being disrupted [2]. An attacker can either force a DoS by crafting a request payload

causing a lot of computational work on the target machine or by flooding the target with requests. The motivation behind a DoS attack is that the attacker sees benefit in the victim's service being disrupted, be that disagreement with the service offered (activist attack), that the service is from a competitor, or that taking down a service brings pleasure to the victim [3].

A Distributed Denial of Service (DDoS) attack is a DoS attack where the requests are coming from many different sources. By distributing the requests, a denial of service attack can reach much higher magnitudes in terms of traffic and can become much harder to control. Usually, an attacker takes control of as many internet-connected devices as possible by spreading malware, and then directing these devices to attack the victim.

1.3 Motivation

The amount and intensity of DDoS attacks globally is on the rise [4] and mitigation is happening only with limited success. DDoS protection is a burden for most organizations and requires a lot of human and financial resources, such that it is hard to justify for many organizations to invest in DDoS protection. A standard tool for signaling DDoS attacks that can be used collaboratively would lower the investment needed to prepare for DDoS attacks. The Ethereum blockchain is a database that is already available and that can not be taken down [1]. With Solidity, the blockchain is scriptable and interfaces for storing and retrieving IP addresses can be programmed. With Ethereum being an readily available infrastructure independent from web services, it opens up an opportunity for storing signals of IP addresses.

Existing DDoS signaling systems such as described in [5] send messages about attack information in key-value form using server infrastructure. The Ethereum blockchain allows to signal messages that have a similar format. This presents a chance to decrease the infrastructure

1.4 Description of Work

The paper "A Blockchain-based Architecture for Collaborative DDoS Mitigation with Smart Contracts and SDN" [6] 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 other parties like ISPs who can filter out the malicious packets before they reach the victim of the attack. This eliminates the need for additional architecture.

In this thesis, three variants of a smart contract will be developed and compared to each other. Each smart contract serves the same purpose, the storage of a list of IP addresses plus relevant metadata. All variants accept the input of IP addresses and allow to read from it, although the storage of the information differs.

The three variants are:

1. A smart contract that stores a list of all IP addresses on the blockchain in an ordinary array, similar to the contract shown in the original paper [6] (Listing 1-3).

2. A contract that stores a URL pointing to a static web resource containing all the information. This will also include the design of the additional infrastructure needed for this variant of the contract.

3. A contract that implements a bloom filter as a mean of reducing cost and space.

All contracts should support both whitelists and blacklists. A whitelist, in this case, is a list of IP addresses that are explicitly allowed to access the server, while a blacklist is a list of IP addresses that are explicitly disallowed to access the service. Both IPv4 and IPv6 addresses should be insertable.

It should be made as easy as possible to modify the list. Additionally, the contract should make it possible to easily verify the identity of the reporter and prevent unauthorized modifications of entries on behalf of others.

The smart contracts will be benchmarked for speed and cost. In addition to that, other characteristics will be compared such as accuracy and ease of use.

Based on the benchmarks and general observations, the best contract is chosen and a statement is made whether the experiment was positive. A look into the future, including further work needed and the development of the Ethereum ecosystem is given.

1.5 Thesis Outline

Chapter 1: Discusses related work.

Chapter 2 The characteristics of Ethereum smart contracts and their implications on our implementation are discussed, as well as the properties and mechanics of our solution and a look at security risks.

Chapter 3: The smart contracts based on the planning in chapter 2 are being implemented. The chapter describes the implementation technique, development process, testing strategy and documents the established protocols.

Chapter 4: A generic cost model for smart contracts is being introduced to enable the estimation of costs for the developed smart contracts. This chapter also discusses the pricing mechanism of Ethereum and covers how transaction speed is related to cost.

Chapter 5: The developed smart contracts are benchmarked for cost, speed and accuracy.

Chapter 6: A recommendation is made for a smart contract variant and a statement is made on whether a blockchain-based approach to mitigating DDoS attacks is suitable for real-world use.

Chapter 2

Related Work

There is a wide range of articles discussing DDoS mitigation. Osanaiya et al. [3] have analyzed 96 publications about DDoS. Out of 36 DDoS attack defenses described in the publications, 6 of them are distributed, while 26 of them are installed on the access point.

DefCOM [7] is a peer-to-peer framework for collaborative DDoS defense that is being installed on multiple nodes in one network. The framework has a distributed design with the purpose of splitting up tasks. Each peer in the network can have up to 3 tasks: Classifying, Rate Limiting and Alert Generation. Nodes in a network may therefore perform only the tasks that they are good at. The framework also does support prioritization of messages. DefCOM is intended for use within an organization and does not propose a solution for inter-organization sharing. It does not contain a new kind of DDoS response mechanism, but builds a lightweight framework for communication between nodes.

A proposal submitted to the Internet Engineering Task Force (IETF) [5] describes a peer-to-peer protocol called "DDoS Open Threat Signaling" (DOTS) for signaling source IP addresses of DDoS attacks. The protocol that the authors propose communicates over HTTPS and is REST-API-based, and is also not decentralized, which is the main difference to the solution proposed in this thesis. The communication can be intra- or inter-organizational. DOTS specifies handshake calls, sending mitigation requests with various parameters, and reporting on the efficacy.

This thesis aims to further develop the idea laid out by the paper "A Blockchain-based Architecture for Collaborative DDoS Mitigation with Smart Contracts and SDN" [6] (further called 'Original Paper'). The original paper proposes to use the Ethereum blockchain to signal DDoS attacks and demonstrates a proof of concept smart contract that allows storage of IP addresses.

Chapter 3

Development

In the following, three variants of a DDoS attack signaling protocol are being developed. For that, a smart contract is being written in the Soldity programming language [8] for each variant. Code-wise, a smart contract strongly resembles a 'class' that is known from object-oriented programming. The following is a 'Hello World' smart contract from the Ethereum website (https://www.ethereum.org/greeter).

```
pragma solidity 0.4.9;

contract mortal {
    address owner;

    function mortal() { owner = msg.sender; }

    function kill() { if (msg.sender == owner) selfdestruct(owner); }
}

contract greeter {
    string greeting;

    function greeter(string _greeting) public {
        greeting = _greeting;
    }

    function greet() constant returns (string) {
        return greeting;
    }
}
```

Instead of the `class` keyword, solidity uses a `contract` keyword. Inheritance is possible using the `is` (rather than `extends` in Java) keyword. A contract can, like a class, be instantiated. The constructor is defined by the method within the contract that has the same name as the contract - in this case, `function greeter(string greeting)` is the constructor of the `greeter` contract. Methods can be declared public or private. They

are, similar to Javascript, prefixed with the `function` keyword. A special type in Solidity is the `address` type, which can hold an 40-byte address of an Ethereum network user.

This Solidity code can be compiled to bytecode and deployed on an Ethereum blockchain. When deployed, the contract is stored in a block, alongside with other data that users committed to the Blockchain, and synced to all users of the network. Downloading the complete public Ethereum Blockchain uses dozens of Gigabytes. Once the deployment is finished, Ethereum users can instantiate the smart contract. If they choose to do so, they send a transaction to the Ethereum network and the instance of the contract is registered on the Blockchain. Methods can also be executed by sending a transaction to the Ethereum Blockchain.

In each method body, a `msg` global object is available, containing information about the transaction being executed, including `msg.sender` (an `address`), `msg.gas` and `msg.value` (for sending Ether).

In addition to that, a second global variable `block` gives information about the current block, including `block.number` and `block.timestamp`.

A constant method, like `function greet() constant returns (string)` is a special function that does not trigger a transaction. Instead, it is a getter function that only executes locally. Constant functions provide convienient interfaces for reading data, but all data should be considered public.

3.1 IPv6 considerations

With IPv4 addresses being 32 bits long, only 2^{32} combinations are possible and the amount of free addresses is almost exhausted [9], so we are currently in a transition phase from IPv4 to IPv6. Therefore it makes sense to support both formats.

An IPv4 address can be represented in IPv6 using a format that is defined in RFC 3493: 80 bits of zeros, 16 bits of ones, followed the IPv4 address. For example, the IPv4 address `46.101.96.149` would be `0:0:0:0:0:ffff:2e65:6095` in IPv6 hex representation. This notation is supported by the Linux kernel and macOS natively. Most web browsers will, when entering `http://[::ffff:2e65:6095]`, display the same website as when entering `http://46.101.96.149`.

This makes it possible to greatly simplify support for both IPv4 and IPv6, with no flag needed to indicate which version of the protocol is meant. All addresses can be stored in an IPv6 format (using an `uint128` type) and if the bits 81-96 are all ones, it indicates that it is an IPv4 address.

3.2 Workflow considerations

Developing a smart contract requires a compiler and a blockchain on which the developer can execute the smart contract. Additional tools can be used to improve development

speed, code quality and developer experience.

3.2.1 Environment

The main Ethereum blockchain is unsuitable for validating the correctness of the code manually. There are significant costs to deploy a contract to the main blockchain, also it is not possible for a developer to get immediate feedback because of transaction processing times. It is also insensitive to use the main blockchain for testing purposes, as all network participants need to download all blocks.

The 'Testnet' is a separate Ethereum blockchain for testing purposes. The Ether needed to deploy contracts is not traded on exchanges and was mined with a much lower difficulty level, so usage of the Testnet is basically free. However, the Testnet is also a globally shared blockchain with confirmation times and is not perfect for development.

`TestRPC` [10] is a library that makes it possible to set up a local blockchain for testing purposes. 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, making it simple to switch the message sender address and test whether the access control features of the developed smart contract are working as intended. This makes `TestRPC` the ideal blockchain for developing.

3.2.2 Testing

A compiler, such as solc [8], will warn about syntax errors and does not compile invalid Solidity code, indicating to the developer that there is an error in the code.

`solc` does refuse to compile code that has operations of incompatible types, invalid variable redeclarations, invalid return types and incorrect syntax. It does however not give an error for unused variables, dead code, missing arguments and does not completely protect against runtime errors or gas limit errors.

While compilers provide a first layer of assurance by only compiling valid contracts with valid syntax, it is still possible to write code with bugs and security vulnerabilities.

Testing is a technique used in almost all fields of software development to reduce unintended regressions introduced when modifying code. A set of test cases is defined which a testing framework can run through and determine whether all assertions still pass. It is the automation of manual quality assurance.

Cases that can be tested to improve robustness are: Intended smart contract use, malicious smart contract use, and edge cases. For example: 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. A robust contract should include unit tests validating that normal use of the contract features result in correct behavior, as well as tests for edge cases and abuse of the contract features.

Usually testing frameworks are written in the language that they are made for testing. There is no testing framework in Solidity, however there is`web3.js` [11], which exposes a Javascript interface for creating contracts, reading from contracts, and calling transactions. This makes it possible to select from an array of available Javascript testing frameworks. In this section, it is described how to test Solidity code with the `ava` framework. However, this choice is arbitrary and not important, as testing with another framework will work similarly.

With a macro function, it is possible to create an isolated blockchain for each tdst scenario. The header for each test file is:

```
const test = require('ava');
const Web3 = require('web3');
const TestRPC = require('ethereumjs-testrpc');

const makeBlockchain = () => {
    const provider = TestRPC.provider({total_accounts: 2});
    const {unlocked_accounts} = provider.manager.state;

    return {
        web3: new Web3(provider),
        accounts: Object.keys(unlocked_accounts).map(acc =>
            unlocked_accounts[acc])
    };
};

test.beforeEach(t => {
    const {web3, accounts} = makeBlockchain();
    t.context.web3 = web3;
    t.context.accounts = accounts;
});
```

The dependencies `ava`, `web3`, `ethereumjs-testrpc` are imported at the top of the file. [1]. The function `makeBlockchain` creates a testing blockchain using `TestRPC`, and returns an interface for interacting with it, as well as a list of Ethereum account addresses that were generated. 2 accounts were created in this instance, which is sufficient to test from the perspective of a contract owner and an non-provileged user.

For each test, `makeBlockchain()` is called beforehand and separate blockchain interface and list of addresses is generated and made available. This prevents test cases from interfering with each other.

Consider the 'Hello World' contract from the beginning of this chapter. A simple test would be to create an instance of the contract and test if the `greet()` function would return the string that was passed to the constructor as the first argument. This test can be implemented in `ava` with the following code:

[1] They need to be installed first using the command `npm install ava web3 ethereumjs-testrpc`, assuming node.js is installed on the computer.

```
const solc = require('solc');
const path = require('path');
const fs = require('fs');

test.cb('Greeter should greet', t => {
    t.plan(1); // Expect 1 assertion
    const code = fs.readFileSync(path.join(__dirname,
        '../contracts/Greeter.sol'), 'utf8');
    const compiled = solc.compile(code, 1);
    const contract = compiled.contracts[':greeter'];
    const Greeter =
        t.context.web3.eth.contract(JSON.parse(contract.interface));

    let i = 0;
    Greeter.new(
        'Hello!',
        {
            from: t.context.accounts[0].address,
            data: contract.bytecode,
            gas: 1000000
        },
        (err, myContract) => {
            // The callback happens twice:
            // Once when submitted, once when mined
            i++;
            if (err) {
                throw err;
            } else if (i === 2) {
                // Call a method
                myContract.greet((err, greeting) => {
                    t.is(greeting, 'Hello!');
                    t.end()
                });
            }
        }
    )
});
```

At first, the contract code gets read from a file and compiled. The compiler, `solc`, returns the bytecode of the contract, as well as an 'Application Binary Interface' (ABI) in JSON format, which contains information about which methods are available. The ABI is necessary because that information cannot be inferred from the bytecode.

Then, the contract gets instantiated with a 'Hello!' string as the first argument. The address from which this transaction is sent, the bytecode and the amount of gas also has to be specified. In normal circumstances, a password for the address also has to be provided, but in a `TestRPC` environment, it can be disabled. The provided gas in this example is hard-coded for simplicity – a more sensible solution would be to estimate gas

using the `estimateGas` helper function provided by `web3` [2].

The final argument is a 'callback function'. `web3` provides a non-blocking asynchronous interface, which means instead getting a return value, a function is called [3]. `web3` oddly calls the callback function twice – only after the second time the transaction is committed to the blockchain.

Once the contract instance is created and on the blockchain, methods of the contract can be called. `myContract.greet()` takes another callback function which is called with the return value of the method. The return value in this case is expected to be 'Hello!'. If this assumption is not true, `ava` would throw an error here.

When running the test [4], the test framework should give `1 passed` as the output and exit with code 0.

In addition to testing, critical contracts should be audited by computer security professionals before being deployed. As this thesis does not yet aim to provide a production-ready platform, no external audits were performed.

3.3 Contract 1: Native storage

The first variant stores all reports in the blockchain natively. No optimizations regarding speed and cost are being made, all IP addresses and metadata are simply stored in an array.

All the code in this section is assumed to be in the contract body:

```
pragma solidity 0.4.9;

contract ArrayStore {
    // Contract body
}
```

Inside it, some structs are defined, with syntax resembling that of the C programming language:

```
struct IPAddress {
    uint128 ip;
    uint8 mask;
}

struct Report {
```

[2] In the project files, a helper function is defined under `lib/estimate-gas.js`, which handles gas estimation for the whole project.

[3] By using 'Promises', a syntactic sugar language feature in Javascript, callback functions can be avoided. Promises are used in the actual project, however in this example classic callback functions are used to avoid confusion.

[4] Run the test using `./node_modules/ava/cli.js test/greeter.js` in the project

```
    uint expirationdate;
    IPAddress sourceIp;
    IPAddress destinationIp;
}
```

For each IP address, a mask can be specified. This makes it possible to specify a range of IP addresses with no redundancy. When discussing masks, the notation of '127.0.0.0/24' is used, where everything before the slash represents the IP address base and the number behind the slash represented the mask. Assuming IPv4, a mask of '/32' means specifically and only that IP, while '/0' means the whole range of IP addresses possible. '127.0.0.0/24' means all IP addresses from 127.0.0.0 to 127.0.0.255 (all addresses that match the first 24 bits if the IP address base). In IPv6, the maximum value for a mask is 128.

An entry that can be added to the smart contract is a composite of 3 values: The 'victim IP' or destination IP, the 'attacker IP' or source IP and an expiration date. Expired reports can be filtered by comparing to the `block.timestamp` global.

The next step is to write the constructor function.

```
address owner;
IPAddress ipBoundary;

modifier needsMask(uint8 mask) {
    if (mask == 0) {
        throw;
    }
    _;
}
function ArrayStore(uint128 ip, uint8 mask) needsMask(mask) {
    owner = msg.sender;
    ipBoundary = IPAddress({
        ip: ip,
        mask: mask
    });
}
```

The constructor function takes two arguments, an IP address and a mask, which form the 'IP Boundary'. The boundary makes it possible for the creator of the smart contract to restrict the destination IP addresses that can be added to only a certain range.

The address of the creator of the instance gets saved in the `owner` property. This makes it possible to write access control logic in other parts of the contract.

The constructor uses a 'modifier' called `needsMask`. A modifier is piece of code that is being run before the method body. The modifier `needsMask` simply throws when the user calls the constructor without the second argument (which the language itself allows). If the second argument is missing, the mask would default to 0, encapsulating all IP addresses possible.

The underscore statement in Solidity can only be used in modifiers. It's effect is that it jumps to the main method body immediately.

In the contract, there is a method for adding a 'customer'.

```
function createCustomer(address customer, uint128 ip, uint8 mask)
    needsMask(mask) {
    if (msg.sender != owner) {
        throw;
    }
    if (!isInSameSubnet(ip, mask)) {
        throw;
    }
    customers[customer] = IPAddress(ip, mask);
}

function isInSameSubnet(uint128 ip, uint8 mask) constant returns (bool) {
    if (mask < ipBoundary.mask) {
        return false;
    }
    return int128(ip) & -1<<(128-ipBoundary.mask) == int128(ipBoundary.ip) &
        (-1<<(128-ipBoundary.mask));
}
```

Only the owner of the contract can add a customer, otherwise the method throws an error. In addition to checking ownership of the contract, the contract also checks if the mask argument was supplied using the previously discussed `needsMask` modifier.

Also, the method checks if the supplied IP range is outside the IP boundary and throws if this is the case. For that, if the IP boundary is `n`, the last 128 - `n` digits of both IP addresses are set to 0 and they should match up. For example, to find out if `::127.0.200.20/120` is in the `::127.0.0.1/112` boundary, the last 16 bits (128 - 112) are set to zero in both addressed. Then, because `::127.0.0.0 = ::127.0.0.0`, it is true that the first IP range is included in the second one.

The following method provides an interface for registering a report:

```
function block(uint128[] src, uint8[] srcmask, uint expirationDate) {
    if (src.length != srcmask.length) {
        throw;
    }
    IPAddress destination = msg.sender == owner ? ipBoundary :
        customers[msg.sender];
    if (destination.ip == 0) {
        throw;
    }
    for (uint i = 0; i < src.length; i++) {
        if (!isInSameSubnet(src[i], srcmask[i])) {
            throw;
        }
```

```
        reports.push(Report({
            expirationDate: expirationDate,
            sourceIp: IPAddress({ip: src[i], mask: srcmask[i]}),
            destinationIp: destination
        }));
    }
}
```

Two cases are distinguished: If the owner of the smart contract calls the method, the rule gets applied to the whole IP boundary. Otherwise, the rule gets applied to the range of IP addresses that were registered using the `createCustomer` method. So the creator of the smart contract can restrict for which IP address ranges the customer can add reports, but the customer can add reports in that range without contacting the smart contract owner. The correct permissions are verified by the other blockchain users who are executing the transaction also and updating their state of the blockchain.

This code is enough to allow for customers adding reports to the contract. Because of technicalities, the reports can not be marked as `public`, because public nested structs are not supported in Solidity. It is generally advised to keep the data structure as flat as possible to avoid this problem. All data in a contract is technically public, but in machine code that has to be deassembled. In order to create an interface where blockchain users can read nested structs, it is necessary to flatten the structure into plain arrays.

```
function blocked() constant returns (uint128[] sourceIp, uint8[] sourceMask,
    uint128[] destinationIp, uint8[] mask) {
    Report[] memory unexpired = getUnexpired(reports);

    uint128[] memory src = new uint128[](unexpired.length);
    uint8[] memory srcmask = new uint8[](unexpired.length);
    uint128[] memory dst = new uint128[](unexpired.length);
    uint8[] memory dstmask = new uint8[](unexpired.length);

    for (uint i = 0; i < unexpired.length; i++) {
        src[i] = unexpired[i].sourceIp.ip;
        srcmask[i] = unexpired[i].sourceIp.mask;
        dst[i] = unexpired[i].destinationIp.ip;
        dstmask[i] = unexpired[i].destinationIp.mask;
    }
    return (src, srcmask, dst, dstmask);
}
```

This code calls helpers functions which are also declared in the contract.

```
function filter(Report[] memory self, function (Report memory) returns (bool)
    f) internal returns (Report[] memory r) {
    uint j = 0;
    for (uint x = 0; x < self.length; x++) {
        if (f(self[x])) {
            j++;
```

```
        }
    }
    Report[] memory newArray = new Report[](j);
    uint i = 0;
    for (uint y = 0; y < self.length; y++) {
        if (f(self[y])) {
            newArray[i] = self[y];
            i++;
        }
    }
    return newArray;
}

function isNotExpired(Report self) internal returns (bool) {
    return self.expirationDate >= now;
}

function getUnexpired(Report[] memory list) internal returns (Report[] memory)
    {
    return filter(list, isNotExpired);
}
```

The helper functions `filter` and `isNotExpired` composed together form the `getUnexpired` function which returns all reports that are not yet expired. Passing functions to other functions is possible in Solidity, but the filtering is not as straight-forward as it is in most languages. There is not native `filter` function like in Python or Javascript, and no arrays of dynamic size can be created inside a function body. Therefore, two for-loops are needed, the first to determine the size of the array that should be created, and the second one to fill an array of that size. This does not make the contract more expensive to operate, since all these methods are marked as `internal` and are only run locally.

The contract now has all methods needed to write and read reports. These methods can be called programmatically using a client library, like `geth` (the Go client) or `the Javascript client`. For inserting IPv6 addresses into the contract from a client interface, it needs to be converted into a 128-bit integer. Helper libraries exist for this task, for example the `ip-address` package on the npm (Node package manager) registry allows to easily convert a string representation of an IP address to a big integer:

```
const {Address6} = require('ip-address');

const stringRepresentation = new Adress6('::123.456.78.90');
const bigIntegerRepresentation = Address6.fromBigInteger(stringRepresentation);
assert(stringRepresentation === bigIntegerRepresentation);
```

3.4 Contract 2: Pointer to web resource

The main disadvantage of storing the reports directly in the contract is that the cost of the data entry scales linearly with the number of reports. What is just a few cents in gas fees for a few reports, can grow expensive at scale. Ethereum disincentivizes the storage of large data because each node needs to keep track of a whole blockchain by downloading it. Therefore it is also a concern for the Ethereum ecosystem to store large amounts of data.

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 it on the blockchain. The advantage of this variant is that it works on a much larger scale and is expected to be cheaper in the long-term. The disadvantages are that a web server needs to be set up and a separate specification has to be defined for the format of the web resource. This solution is also more prone to connectivity issues and does not take full advantage of the decentralization and immutability of the blockchain.

3.4.1 Smart Contract

The basic data structure of the smart contract is similar to the array store contract. The difference is that it should accept URLs which point to lists of reports rather than reports themselves. Since it is possible to include as many reports as desired in a web resource, the assumption is made that only 1 pointer is needed per customer. This assumption can simplify the contract, so that is not necessary anymore to include a helper function that flattens the data.

```
mapping (address => bool) public members;
mapping (address => Pointer) public pointers;

struct Pointer {
    uint expirationDate;
    string url;
    uint128 ip;
    uint8 mask;
}

struct IPAddress {
    uint128 ip;
    uint8 mask;
}

function createCustomer(address customer, uint128 ip, uint8 mask) {
    if (msg.sender != owner) {
        throw;
    }
    if (isInSameSubnet(ip, mask, ipBoundary.ip, ipBoundary.mask)) {
        members[customer] = IPAddress({
```

```
            ip: ip,
            mask: mask
        });
    }
}

function setPointer(string url, uint128 subnetIp, uint8 subnetMask, uint
    expirationDate) {
    if (members[msg.sender].ip == 0) {
        throw;
    }
    if (!isInSameSubnet(subnetIp, subnetMask, members[msg.sender].ip,
        members[msg.sender].mask)) {
        throw;
    }
    if (expirationDate < now) {
        throw;
    }
    pointers[msg.sender] = Pointer({
        expirationDate: expirationDate,
        url: url,
        ip: subnetIp,
        mask: subnetMask
    });
}
```

Instead of using an array to store the pointers, a `mapping(address => Pointer)` is being used. It is the equivalent of a `Map<address, Pointer>` in Java with it having types, but it uses a 'Javascript object like' syntax for getting, setting and deleting values.

With this design, each user of the smart contract can have one pointer at a time. With each transaction the previous pointer is overwritten, hence the method name `setPointer`.

The struct Pointer `Pointer` does not nest another struct `IPAddress` like in the previous contract, but instead stores base and mask directly. Although it would be a cleaner design, it would trigger an error message `Internal type is not allowed for public state variables`. The reason for this that each Ethereum contract has a JSON interface called Application Binary Interface (ABI) in which a structure like this cannot be represented at the moment.

For this reason, the contract is programmed to do without nested structs. The benefit is that mappings can be used which means that no helper method is needed to retrieve the data. Verifying that the sender of the transaction is a member works by comparing `members[msg.sender].ip` to 0. In many other languages, `members[msg.sender]` would be compared to `null` and the comparison the above code would be susceptible to a null-pointer exception. However, in Solidity, there is no concept of 'null'. Accessing a pure value that has not been set returns zero, accessing a struct that has not been set returns a struct where all values are zero.

3.4.2 Web resource

The web resource has several design aspects that need to be considered. By storing a URL in the blockchain, it is already implied that the connection to the resource is made using HTTP(S), which is a suitable protocol for our needs.

A syntax and a data structure that the reports are represented in needs to be selected. It is desired to use an already common syntax like XML, CSV or JSON, because there are clients for many languages out there and and they are heavily standardized. An essay by Nicolas Seriot [12] shows the challenges of covering edge cases in standards by showing inconsistencies in trailing commas, unclosed structures, duplicate keys and white space in JSON, making a case against developing an additional format. The use of an established format increases the **Portability** of the protocol.

JSON and XML allow to extend the schema by adding more keys to the object or by adding more allowed tags to the schema. **Upgradeability** is a desired property of the protocol, it allows it to be developed further in the future to enable more features with backwards compatibility. This is more difficult with a two-dimensional design like CSV, because the format can only be extended by adding more columns.

As one can expect that the lists to become very large, it is beneficiary to have the format in a way where it can already be partially evaluated while not yet fully loaded. This is called **Streaming**. Network latency and throughput rates do influence the speed in which the files will be downloaded. Streaming in CSV is easy, as soon as a newline character is detected, the client can safely assume that an entry has been fully loaded and that it can be processed. On the contrary, with XML or JSON, who have closing tags, streaming is not possible.

Neither CSV, JSON or XML have both good Streamability and Upgradeability, and other formats don't have good Portability.

'Line delimited JSON streaming' [13] provides a reasonable tradeoff. It's syntax is a list of items that are delimited using a newline character (`\n`), where each item has a valid JSON object according to RFC 7158 [14]. Packages for Line-delimited JSON streaming exist for at least node.js and Python in their respective registries, so client libraries are available. As a fallback, it is always possible to download the full file, split it by newlines and pass each item into one of the many JSON parsers. With these properties, line-delimited JSON streaming is a suitable syntax the web resource.

With the web resource and the smart contract being disconnected, a set of rules has to be defined to ensure interoperability. In contrast to the blockchain, a web resource can change as many times as needed. This makes it hard for the users of the blockchain to keep track of updates, and to verify whether the content of the webpage is still the same as it was the entry into the blockchain was created. To mitigate these issues, a new rule is added to the protocol: The contents of a web resource must be immutable and can never be changed. If a customer desires to update the data, a new resource must be created under a different URL and the smart contract needs to be updated with the new pointer.

To prevent customers from modifying the content of their web resources (which they can), a SHA256-hash of the body of the resource needs to be included in the report that gets added to the smart contracts. Clients should generate hashes of the web resources themselved and should reject reports that do not have matching hashes. This technique is inspired by 'Subresource integrity' [15], which validates resources like stylesheets and scripts in browsers and is therefore already widely deployed.

This also brings another advantage: If an asset is immutable, it can be easily deployed to a CDN, which is harder to deny using a DDoS attack.

To enable this feature, the `Pointer` struct in the smart contract simply contains another property, `bytes32 hash`. Accordingly, the `setPointer()` method gets updated as well.

In variant 1 of the contract, a report object contained 1. an IP address of the source with optional mask 2. an IP address of the destination with optional mask 3. an expiration date.

Therefore, the web resource contains there fields as well. Since multiple reports can be stored under one URL, an array is used and stored under the `reports` key.

```
{
    "reports": [
        {
            "expirationDate": 1502755200000,
            "sourceIp": {
                "ip": "::ffff:2e65:6095",
                "mask": 120
            },
            "destinationIp": {
                "ip": "::ffff:1234:abcd",
                "mask": 120
            }
        },
        {
            "expirationDate": 1502755200000,
            "sourceIp": {
                "ip": "::ffff:efab:4321",
                "mask": 80
            },
            "destinationIp": {
                "ip": "::ffff:efab:4321",
                "mask": 80
            }
        }
    ]
}
```

Code snippet 3.1: Example web resource content

Instead of using an array, it would also be possible to create two separate JSON reports and delimit them with a newline (`\n`) to enable streaming.

The timestamps should be UNIX timestamps with miliseconds. IPs should be in IPv6 format (short notations are allowed), masks should be values from 0-128. Clients should check and reject the reports if they are not in the IP boundary set by the smart contract.

In addition to the `reports` field, other fields are supported for context and metadata:

version A string specifying the used version of the protocol to make the protocol future-proof. The versioning should follow Semantic Versioning [16]

whitelist can either be `true` or `false`. The default is `false`. When this flag is set, all reports in the `reports` array should be considered whitelist entries.

The specification can be expanded in the future by adding more fields and increasing the version number. DOTS [5] allows more fields, such as limiting a report to specific port numbers, adding metadata about the attack (duration, attack type, registration time, mitigation status). These information can be considered for addition to the format as well in a future version.

The SHA256 hash of code snippet 3.1 is `xZ9hL0AColp7EQ82H/LuAGrGjr5fA60K/vXMjISqnIA=` [5]. This value is set as the `hash` when registering the web resource in the smart contract.

3.5 Contract 3: Bloom filter

While variant 1 of the smart contract is space-inefficient and variant 2 needs additional infrastructure, variant 3 is an attempt to strike a balance. It is a standalone contract that solves the scalability issue by using a bloom filter.

A bloom filter is a probabilistic data structure that is very space efficient. In the context of large amounts of IP addresses, it can be tested if an IP address has been inserted into the bloom filter beforehand with constant space requirements. False positives are possible, false negatives are not.

This variant is space-efficient and the logic is self-contained, however perfect accuracy cannot be guaranteed anymore. Also, additional parameters like a blacklist/whitelist flag, expiration dates, ip boundaries can not be stored in a bloom filter, as it is impossible to fully retrieve the information that the bloom filter was fed.

It is however possible to store additional information in the contract, and use multiple contracts if the parameters differ for reports.

No implementation of a bloom filter in Solidity could be found online, therefore the approach taken to implement this variant was to convert one of the countless examples written in other programming languages into Solidity.

[5] On macOS or Linux a hash can be generated with the following command: `curl $RESOURCEURL | openssl dgst -sha256 -binary | openssl enc -base64 -A`

One part of a bloom filter is the hash function that takes any string and converts it into a fixed-length hash. The other part is managing the store, and exposing interfaces for adding entries and checking if an entry has been added.

Different hash functions are available and the choice of the hash function influences the properties of the bloom filter, mainly accuracy and speed. A bloom filter becomes more accurate if the hash function it uses produces more uniform hashes. However, usually more uniform results also means slower hashing.

3.5.1 Hashing function

The hashing part is the more complex code to convert to Solidity, since hashing functions use big numbers and bit-shifting to generate their hashes, which both are very prone to differences in different environments.

In an first attempt, the popular hashing library `imurmurhash` [17] was taken and rewritten in Solidity as close as possible. This library is using the `>>>` (logical right shift) operator which is absent in Solidity. By using a `>>` (arithmetic right shift) operator instead, the hashes could not be reproduced. It also generated big numbers in the process whose behavor was inconsistent between Ethereum VM and Javascript environments. When later using the ported hash function in a bloom filter, it would return many false positives[6].

In an second attempt, the `bloomfilter.js` [18] library was ported to Solidity as close as possible. This library uses the Fowler-Noll-Vo hash function instead. The commit history shows that it had once worked without the `>>>` operator, but that operator was added later. Yet, the Solidity implementation did still not yield the same hashes for the same input as the Javascript equivalent of the code. By testing subfunctions of the hash function, it was discovered that the inconsistency is caused by bug numbers. In one hash operation, the statement `2166136261 ^ 65 & 0xff` would be executed. It is easily verifiable that this expression evaluates to `-2128831100` in Javascript and to `2166136196` in Solidity Version 0.4.9.

This difference results because Solidity uses types for numbers such as `int`, while Javascript only supports floating-point math. With the different hashes being generated, the contract that resulted from the second attempt also led to false positives. [7]

Porting existing hash functions to Solidity provides great challenges. For the third attempt, bloom filter implementations were searched that used one of the hash functions already implemented in Solidity. Two hashing algorithm functions are globally available in every Solidity contract, `sha256()` and `sha3()`. For the third attempt, a 'Simple Bloom Filter' [19] code snippet from Github was used as the basis for the smart contract.

This third attempt proved successful as the SHA-256 hashes are reproducible and false positives did not occur anymore. The smart contract that resulted from this attempt has also the most concise and easiest to read code.

[6] A test case showing the false positive is available under `test/hash-function.js` in the project files.

[7] A test case showing the false positive is available under `test/hash-function-2.js` in the project files.

3.5.2 Hashing parameters

Instead of only hashing an input value once, it usually is hashed multiple times. This reduces the chance of collision [20]. The amount of hash iterations is also variable. In this simple bloom filter, additional hashes beyond the first one are just bit-shifts of the first hash.

The bloom filter takes two parameters: numbers of bits in the filter and number of hash functions.

The bigger the amount of the array items and the bigger the amount of the hash iterations, the more accurate a bloom filter gets. The default parameters of the Python bloom filter were an array size of 1024 bytes and 13 hash iterations. [19].

In Solidity however, this configuration would raise an 'out of gas' error when adding a string to the filter. The maximum amount of gas (also called gas limit) that can be spent on a transaction is 3,141,592 (pi million). Since every node in the network needs to download every transaction, there is an artificial cap on how computationally expensive a transaction may be.

By decreasing the array size to 512 bytes, the cost stays below the gas limit and the bloom filter works. Testing different values determined that an array size of 836 bytes is the maximum before this specific contract would not be able to have a transaction executed. However this thesis continues with the assumption of a 512 byte array size to stay well below the gas limit — this way more features can be added without hitting the gas threshold.

3.5.3 Accuracy

Given a number of items that are indexed, and a number of hash functions, the ideal array size and number of hash functions can be calculated. According to http://hur.st [21], given n = number of items in the filter and p = probability of false positives, the number of bits in the filter m can be calculated using equation 3.1 and the number of hash functions k using equation 3.2.

$$m = \lceil \frac{n * log(p)}{ln(\frac{1.0}{2.0^{ln(2)}})} \rceil \tag{3.1}$$

$$k = \lfloor ln(2) * m/n) \rceil \tag{3.2}$$

For example: Given that the magnitude of DDoS attacks is in the millions (n = 1'000'000) and assuming that a 5% chance of false positives is good enough (p = 0.05), an array size of 14357134 bits (1.71 MB) and 4 hash functions will do.

Solving equation 3.1 for p gives equation 3.3.

$$p = e^{\frac{m \cdot ln(\frac{1}{2^{ln(2)}})}{n}} \tag{3.3}$$

The probability of at least one false positive is shown in table 3.1 and figure 4.1. It assumes the number of **bits** in the filter is 4'096 (512 $bytes \cdot 8$).

IP addresses (n)	Probability
1	10^{-855}
10	10^{-86}
100	10^{-9}
1'000	0.14
10'000	0.82
100'000	0.98
1'000'000	0.998
10'000'000	0.9998

Table 3.1: Probability of false positives, using equation 3.3 and $m = 4'096$

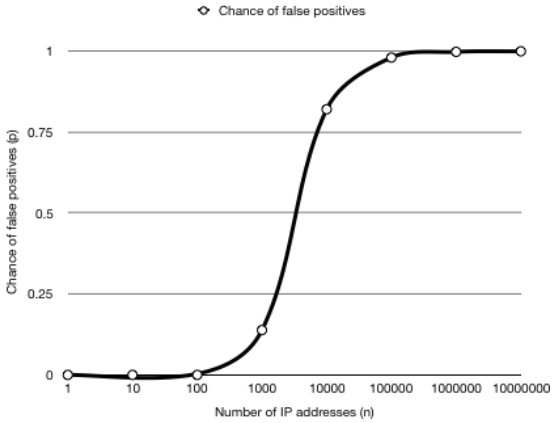

Figure 3.1: Probability of false positives, using data from Table 3.1 (logarithmic scale)

The bloom filter does have a false positive chance of under 1% for up to 425 IP addresses. Inserting more IP addresses after that decreases the accuracy of the bloom filter dramatically. With 1'000 insertions, variant 3 would lead to 14% of legimitate traffic being blocked. With 3'000 insertions, over half of the legitimate traffic would be falsly blocked.

With the gas limit constaint in place and the probabilities in mind, it is recommended to use more than one contract if the probability of false positives would be higher than acceptable otherwise. The acceptable probability of false positives is individual for each user.

3.5.4 State management and interface

A bloom filter creates a fixed length array that initially only contains zeroes.

```
contract BloomFilter {
    int array_size = 512;
    int bits_per_entry = 8;
    int bitcount = 512 * bits_per_entry;
    int hashes = 13;
    uint8[] filter;
    bool whitelist;
    address owner;
    function BloomFilter3(bool white) {
        owner = msg.sender;
```

```
        filter = new uint8[](512);
        whitelist = white;
    }
}
```

When adding an entry, it gets hashed and some items in the array get bit-shifted from '0' to '1'. The algorithm is taken from the original Python implementation [19].

```
function add(string ipAddress) {
    if (owner != msg.sender) {
        throw;
    }
    var digest = int(sha256(ipAddress));

    for (var i = 0; i < hashes; i++) {
        int a = digest & (bitcount - 1);
        filter[uint8(a / bits_per_entry)] |= (2 ** uint8(a % bits_per_entry));
        digest >>= (bits_per_entry / hashes);
    }
}
```

For testing whether an IP address has been inserted, the input string gets hashed again and the same positions in the array are calculated. If all the values at the array positions are 1's, the bloom filter assumes the string has been inserted before (of course false positives are possible).

```
function test(string ipAddress) constant returns (bool) {
    var digest = int(sha256(ipAddress));
    for (var i = 0; i < hashes; i++) {
        int a = digest & (bitcount - 1);
        if ((filter[uint8(a / bits_per_entry)] & (2 ** uint8(a %
            bits_per_entry))) <= 0) {
            return false;
        }
        digest >>= (bits_per_entry / hashes);
    }
    return true;
}
```

3.5.5 IPv6 format ambiguity consideration

A IPv6 address can be formatted in more than one way. [22] For instance, leading zeroes in one block can, but don't have to, be omitted. Also, multiple subsequent blocks containing only zeroes can, but don't have to be replaced by two colons. For example, `0000:0000:0000:0000:0000:ffff:2e65:6095` can also be formatted as `0:0:0:0:0:ffff:2e65:6095` or even as `::ffff:2e65:6095`. Passing to the hash function the same

IP address, but in different formats, will result in vastly different hashes, assuming an uniform hash function.

To prevent false negatives, the possibility of multiple representations per IP addresses has to be eliminated. Therefore, it makes sense to forbid shorthand notations as well as IPv4-in-IPv6 notations such as `::ffff:46.101.96.149` and any other alternative notations that RFC 4291 [22] mentions. This does not result in higher storage costs, as a SHA256 hash is always only 256 bits long.

3.6 IP address ownership verification

In all variants of the smart contract, owners of destination IP addresses can store source IP addresses they want to be blocked in the smart contract. In order to establish trust, one might suggest there should be a way to automatically verify the ownership of the destination IP addresses. This approach was explored during the development of the prototype, however it turns out to be a challenging task. Using certificates to validate IP address ownership in Solidity is currently not practical for at least two reasons.

The main issue is obtaining a certificate. As there is no way to directly mathematically or logically prove that somebody is the owner of an IP address (it can be spoofed), an indirect solution is required. Theoretically, the certificate process of domains could be applied to IP addresses.

Certificate Authorities (CA) are institutions whose business is to issue and manage certificates. They verify and validate the ownership of a domain and issue an certificate for it. CAs need to fulfill a wide range of requirements [23] to be considered reputable and be included in the root key store of operating system and browser vendors. Currently, only 156 certificates from 60 different owners are trusted in Firefox [24].

To fulfill the strict requirements, CAs need to make investments in establishing a system that securely validates a domain and manages the certificates that are issued. To make money, nearly all CAs that are trusted in Firefox charge a issuance fee for domains. The notable exception is ”Let’s Encrypt” [25], which makes money through sponsorships.

Although it is technically possible to issue an SSL certificate for an IP address, it is very uncommon. GlobalSign [26] is the only provider whose certificates are trusted by Firefox and that issues certificates for IP addresses. To obtain a certificate, it is also a requirement that the IP address is registered in the RIPE database [27].

Concluding the overview of the certificate issuance process, there are no good providers offering certificates for IP addresses and it is an expensive endeavor to build a certificate authority whose complexity quickly becomes bigger than the one of the scope of the thesis.

Assuming it is possible to obtain and validate certificates for IP addresses, 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 [28]. As of writing, the exact implementation is not clear, with parts of the community vouching for direct RSA signature verification, and other parts wanting BigInt Support which could then enable certificate validation.

Certificate validation in Solidity is not a hard requirement though — since everything in the blockchain is public including stored certificates and IP addresses, it can also be done off the blockchain.

3.7 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 with an update.

In addition to that, the language Solidity aims to have a syntax similar to JavaScript and is therefore quite loose with enforcing the correct type. External static code analysis tools such as Solium [29] do not yet have as many rules included as comparable tools for more mature languages.

In this section, some common mistakes that can lead to security exploits will be explained, partially taken from a list compiled by the co-founder of Ethereum [30] and based on real-word vulnerabilities.

3.7.1 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
    }
}
```

3.7.2 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 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 minimises the usefulness for attackers, for example the pattern of the IPs should not make the way how IPs are being blocked predictable.

3.7.3 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;
}
```

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

3.7.4 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 takeaway from the 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;
}
```

For illustrating the vulnerability, consider 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.

3.7.5 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 [31] soon, therefore it will not be further discussed here.

Chapter 4

Cost model

4.1 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.

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 the purpose of a cost model, gas is as a real money cost and should be treated as a currency.

The 5 variable factors are:

1. The operations being done: Each smart contract consists under the hood of a set of 31 possible assembly operations [32]. Such operations can for example be `ADD` which adds numbers together or `SHA3` which generates a hash.

2. The gas costs assigned to those operations: Referred to as the 'Fee schedule', in the Ethereum yellow paper [32] a fee for each operation is defined. For example, a transaction costs 21000 gas and a SHA3 hash operation costs 30 gas.

In general, it can be stated 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 developers and accepted by the majority of Ethereum users.

Because of the non-proportionality, some assembly instructions will change their gas prices in the future for rebalancing purposes when and if the majority of nodes upgrade to the Metropolis hard fork, the next version of Ethereum. For example, the cost of a CALL operation will be increased from 700 to 4000 gas with the Metropolis upgrade [31]. The fact

that gas prices can change is the reason the two above mentioned variables are separated in this model.

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. Gas price and desired speed: 1 'gas' corresponds to a specific amount of Ether. The price of a gas is formed by the users of the Ethereum network and unlike the previously mentioned gas fee schedule, the gas-to-ether conversion rate is not hardcoded into the Ethereum, but determined by the users dynamically.

Each Ethereum client 'signals' a gas price. The default configuration of go-ethereum [33], which is the standard implementation for Ethereum, sets a gas price of 20 shannon[1], 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 shannon or more. When the gas price is set to 20 shannon in a client, the client will offer this amount for a transaction and on the other hand mine transactions that offer at least this amount.

The average gas price chart from etherscan.io [34] shows that the clients of the network mostly go with the default configuration value, with the average price being 23 shannon.

The reason that the real average gas price is slightly 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 effect that the transaction will be mined faster.

The mechanics of transaction mining are similar to those of a stock market: If a stock is worth 100 USD, is being actively traded on an exchange and 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. In Ethereum, the best gas price offers will be mined first. For our application, it could be desirable to pay a premium for faster transaction mining.

The default price of gas is 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 has reached 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 [35] 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, for a transaction that offers 2 shannon per gas, the mean transaction confirmation time is 119 seconds. For 20 shannon and 28 shannon, the average transaction confirmation time is 44 and 30 seconds respectively.

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

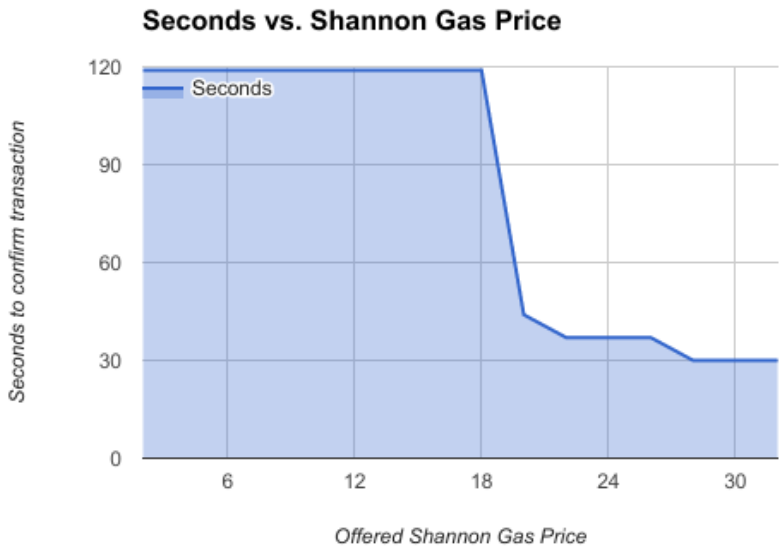

Figure 4.1: Average time until transaction is confirmed. Made with data from ethgasstation.info on May 8th.

In Figure 4.1, it can be seen that a node that offers a higher reward for a transaction will get confirmed 4 times faster on average. The client has to decide based on that information, how much gas it wants to offer.

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 the Coinbase exchange [36]. On July 20th 2017, it was $205.00. Ether has seen a price drop of over 60% when a smart contract called "TheDAO" was hacked in June 2016 [37]. In July 2017, the price dropped from over $400 to $135, causing Ether to lose two thirds of its value temporarily with no incident causing it [36]. Also, in July 2017, on GDAX (an exchange operated by Coinbase), a multi-million market sell order caused the price of Ethereum to drop to $0.10 for a moment [38]. The reason for this is that not enough buy orders were placed to fill the sell order. Low volume on Ethereum exchanges is a major cause of price volatility. Bigger adoption of Ethereum is necessary to stabilize the price.

5. The compiler being used: The final variable is the deviation of gas estimates when using different compilers. This was discovered when receiving different gas estimates for the same contract when upgrading the compiler. To illustrate the effect, consider the smart contract in code snippet 4.1.

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 (Table 4.1).

Even when removing just whitespace from the contract, the gas cost can, but does not have to change. On the other hand, changing variable names does not result in an estimate change (Table 4.2).

```
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
        }));
    }
}
```

Code snippet 4.1: A simple smart contract with predictable behavior - results in different gas estimates

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

Table 4.1: Gas estimate deviations of compilers

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)

Table 4.2: Gas estimate deviations of code changes

Given that all the deviations discovered skewed the total gas cost by not more than 4%, this variable is omitted from the cost model for simplicity.

4.2 Cost model

Define 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 shown in Equation 4.1:

$$C = \sum_{i=1}^{n} \sigma_i \cdot f_i \tag{4.1}$$

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 [39] (set to be raised to 5'500'000 in the next fork [31]). At the moment, this means that the condition in Equation 4.2 is always fulfilled.

$$C = [21000, 3141592] \tag{4.2}$$

The price is denoted as the variable α. Since according to ethgasstation.info, there are 3 possible speeds, constants are defined for them representing the minimum cost to pay in order to get that speed (Equations 4.3 - 4.5).

$$\alpha_{cheap} = 2 \cdot 10^{-9} Ether \tag{4.3}$$

$$\alpha_{average} = 20 \cdot 10^{-9} Ether \tag{4.4}$$

$$\alpha_{fast} = 28 \cdot 10^{-9} Ether \tag{4.5}$$

The price of an ether is noted as ETH. As mentioned previously, the deviations of different compiler are negligible.

Multiplying these values together results in the total cost (Equation 4.6).

$$C \cdot \alpha \cdot ETH \tag{4.6}$$

For example, a contract that costs 500'000 gas to deploy and is priced at $\alpha = \alpha_{average}$ per gas, with the price for Ether being \$50, the cost to deploy that contract is \$0.025 (Equation 4.7).

$$500'000 \cdot 10^{-9} \cdot \$50 = \$0.025 \tag{4.7}$$

4.3 Cost estimation for DDoS mitigation

The price that to pay for contracts varies as the contracts are developed and the gas and ether prices change, but it is possible to estimate the magnitude.

For a contract that stores IP addresses directly, the gas estimate was $Deploy\ cost = 700113\ gas$. The cost of adding 1 IP in a transaction is 104046 gas. When adding 2 IP addresses in the same transaction, 1 transaction gas cost of 21000 is saved among other minor savings. The maximum is 46 IP addresses, after that 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. The price to store x IP addresses is in Equation 4.8.

$$c = Deploy\ cost + (x \cdot Add\ IP\ Cost) + \lceil \frac{x}{2} \rceil \cdot Transaction\ cost) \tag{4.8}$$

In a expensive case where the Ether price is expensive $ETH = \$100$ and the user wants fast transactions $\alpha = \alpha_{fast}$, it costs \$3.82 to do one transaction with 20 IP addresses ($C \approx 1364194$).

In a cheap case where Ether is reasonably cheap $ETH = \$50$ and most nodes have started accepting cheaper transactions $a = a_{cheap}$, the transaction costs \$0.13.

So far users of the Ethereum network have not reacted to the higher Ether price by lowering gas costs. With activist sites like ethgasstation.info spreading awareness, it is realistic that users over time react and 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 as of writing) and an Ether price of $ETH = \$90$ (the price at the same time), the result is 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* still applies. The transaction cost for adding a URL with 64 characters was $Transaction\ cost = 121466$ in our test. This gives Equation 4.9.

$$C = Deploy\ cost + \#\ of\ transactions \cdot Transaction\ cost \tag{4.9}$$

Chapter 5

Evaluation

5.1 Cost Benchmark

This section aims to provide a rough guidance on expected cost for each variant of the smart contract.

Actual costs will vary over time as the Ether-to-USD exchange rate as well as the network gas price will change. It is also dependant of the desired transaction confirmation speed. To avoid this section becoming inaccurate in the future, only the amount of gas consumed is benchmarked. The actual cost can be calculated using the cost model described in chapter 5.

5.1.1 Variant 1

For variant 1, a benchmark script was created[1]. The benchmark calculates the cumulative gas spent on contract creation and IP address insertion. Two cases were calculated: For the worst case, each IP address was inserted in a separate transaction. For the best case, reports were bundled into one transaction. The benchmark was executed in both cases for 1 IP address, then 2 IP addresses and so forth up to 20 IP addresses. The resulting costs are displayed in Figure 5.1.

[1]The source code can be found under `v1-gas-benchmark.js`. The benchmark can be executed with `node index v1-gas-benchmark -count=5`.

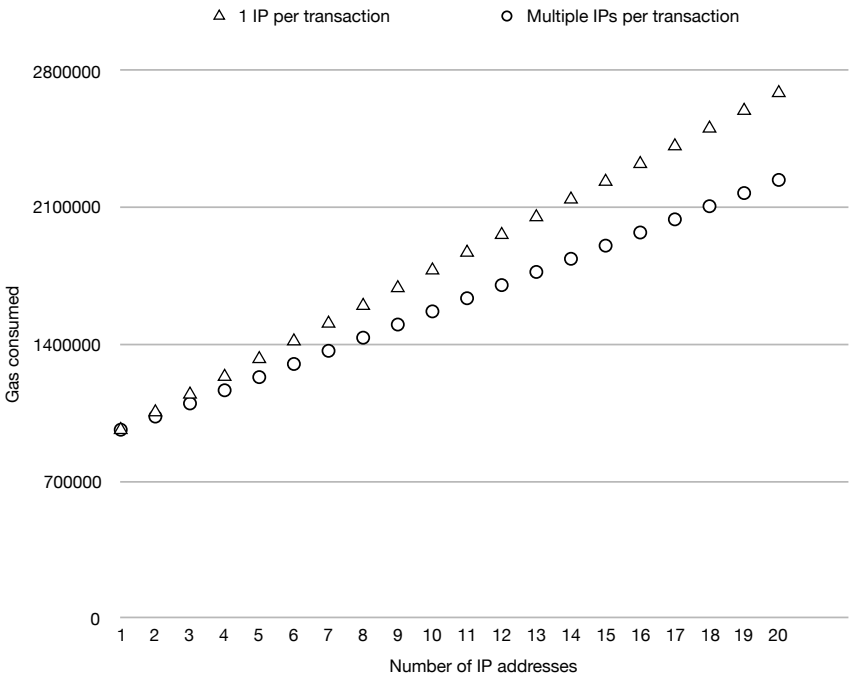

Figure 5.1: Gas cost incurred using variant 1

For only a few reports, the major cost is the initial deployment of the contract, costing approximately 858'000 gas. Adding reports costs approximately 150'000 - 200'000 gas each. It can be observed that bundling reports into one transaction can save 24% of the gas cost beyond the initial cost (this is however not always practical, as new reports have to be accumulated before they can be bundled, which might make the insertion slower).

Even in the best case, the costs grow linearly ($\mathcal{O}(n)$) as new reports are added. This makes variant 1, as predicted, expensive and unsuitable for big amounts of IP addresses.

5.1.2 Variant 2

For variant 2 (web resource pointer), the cost benchmark of the Solidity contract is trivial, however additional infrastructure is needed whose cost is hard to quantify.

For the gas cost, the `estimate-gas` script[2] was used to estimate the deploy cost. For the transaction cost a benchmark script was created[3]. For the deployment, 600'000 gas is consumed and per update 150'000 gas. This makes variant 2 the cheapest contract in terms of gas.

The cost of the additional infrastructure cannot be measured as the specification only requires some format contraints to be fulfilled and leaves many implementation details up to the user, including which hosting solution to use. However, this should not be a

[2]The script can be executed using the command `node index estimate-gas IpPointerContract`

[3]The source code can be found under `v2-gas-benchmark.js`. The benchmark can be executed with `node index v2-gas-benchmark`.

significant cost, as many providers such as Amazon S3, Github pages or `now.sh` allow easy deployment of static files for free or for just a few cents.

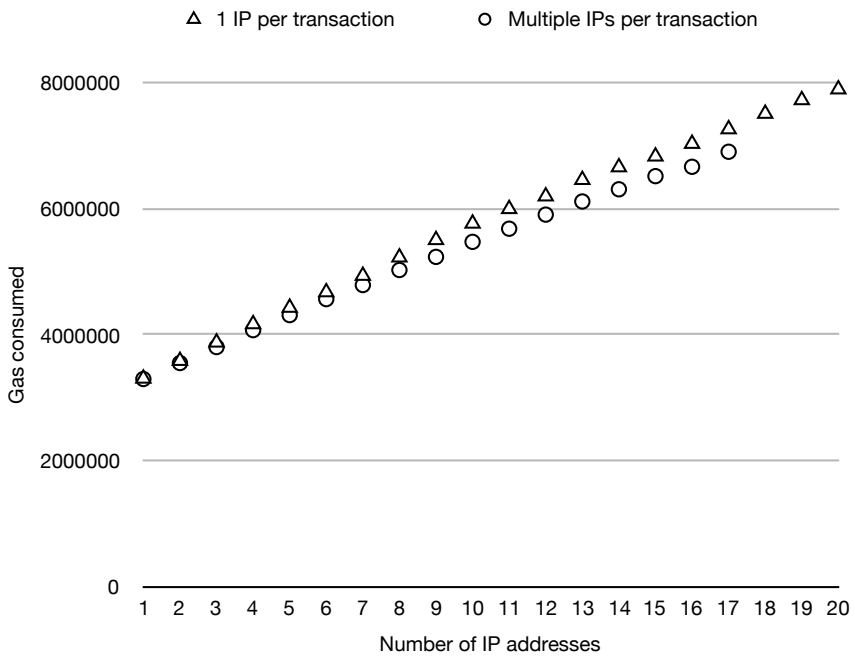

Figure 5.2: Gas cost incurred using variant 3

5.1.3 Variant 3

For variant 3 (bloom filter variant), another benchmark script was created[4]. Like in variant 1, a worst case and a best case scenario was tested, where in the worst case one report gets added at a time, while in the best case, reports could be bundled into one transaction to save gas.

This variant was expected to perform better than variant 1, as less storage is required to store all IP addresses. However, the benchmark does not confirm the expectation. The bloom filter does actually incur more gas costs than storing a full-sized array of IP addresses.

For adding one report, approximately 290'000 gas is consumed, which is almost three times more than in variant 1 (105'000 gas). There are also no economies of scale, as the cost goes up linearly after the first report. Bundling reports into one transaction does save approximately 40'000 gas per report (13%), but is less effective than it is for variant 1. Also, it is not possible to add more than 17 reports in one transaction, as the block gas limit is reached.

It becomes clear that Solidity charges more heavily for expensive computation like hashing than it does for storage. In addition to much higher costs, the bloom filter variant can

[4]The source code can be found under `v3-gas-benchmark.js`. The benchmark can be executed with `node index v3-gas-benchmark -count=5`.

also give false positives, and metadata like expiration date, whitelist/blacklist etc. has to be stored separately.

If less required storage space does not result in lower prices, then there is no advantage in it at all. The whole blockchain, multiple gigabytes needs to be downloaded to a client anyway — the only reason to optimize for storage is to get cheaper costs.

Chapter 6

Summary and Conclusions

Bibliography

[1] *Ethereum: Blockchain App Platform*. (Accessed on 08/08/2017). URL: https://ethereum.org/.

[2] US-CERT. *Understanding Denial-of-Service Attacks*. (Accessed on 08/08/2017). Nov. 4, 2009. URL: https://www.us-cert.gov/ncas/tips/ST04-015.

[3] Opeyemi Osanaiye, Kim-Kwang Raymond Choo, and Mqhele Dlodlo. "Distributed denial of service (DDoS) resilience in cloud: Review and conceptual cloud DDoS mitigation framework". In: *Journal of Network and Computer Applications* 67 (2016), pp. 147–165. ISSN: 1084-8045. DOI: http://dx.doi.org/10.1016/j.jnca.2016.01.001. URL: http://www.sciencedirect.com/science/article/pii/S1084804516000023.

[4] Steve Mansfield-Devine. "The growth and evolution of DDoS". In: *Network Security* 2015.10 (2015), pp. 13–20. ISSN: 1353-4858. DOI: http://dx.doi.org/10.1016/S1353-4858(15)30092-1. URL: http://www.sciencedirect.com/science/article/pii/S1353485815300921.

[5] K. Nishizuka et al. *Distributed-Denial-of-Service Open Threat Signaling (DOTS) Architecture*. https://tools.ietf.org/html/draft-ietf-dots-architecture-04. (Accessed on 08/08/2017).

[6] B. Rodriguez et al. "A Blockchain-based Architecture for Collaborative DDoS Mitigation with Smart Contracts and SDN". In: (2017). URL: http://doi.org/10.1007/978-3-319-60774-0_2.

[7] George Oikonomou et al. "A framework for a collaborative DDoS defense". In: *Computer Security Applications Conference, 2006. ACSAC'06. 22nd Annual*. IEEE. 2006, pp. 33–42. DOI: 10.1109/ACSAC.2006.5. URL: https://doi.org/10.1109/ACSAC.2006.5.

[8] *The Solidity Contract-Oriented Programming Language*. (Accessed on 08/08/2017). URL: https://github.com/ethereum/solidity.

[9] RIPE Network Coordination Centre. "IPv4 Exhaustion". In: (2015). (Accessed on 08/08/2017). URL: https://www.ripe.net/publications/ipv6-info-centre/about-ipv6/ipv4-exhaustion.

[10] *ethereumjs/testrpc: Fast Ethereum RPC client for testing and development*. https://github.com/ethereumjs/testrpc. (Accessed on 07/30/2017).

[11] *ethereum/web3.js: Ethereum JavaScript API*. https://github.com/ethereum/web3.js/. (Accessed on 07/30/2017).

[12] Nicolas Seriot. *Parsing JSON is a Minefield*. (Accessed on 08/08/2017). Oct. 26, 2016. URL: http://seriot.ch/parsing_json.php.

[13] Wikipedia. *JSON Streaming - Line delimited JSON*. (Accessed on 08/08/2017). URL: https://en.wikipedia.org/wiki/JSON_Streaming#Line_delimited_JSON_2.

[14] IETF. *RFC 7158*. (Accessed on 08/08/2017). URL: https://tools.ietf.org/html/rfc7159.

[15] Mozilla. *Subresource Integrity*. (Accessed on 08/08/2017). URL: https://developer.mozilla.org/en-US/docs/Web/Security/Subresource_Integrity.

[16] Tom Preston-Werner. *Semantic Versioning 2.0.0*. (Accessed on 08/08/2017). URL: http://semver.org/.

[17] *jensyt/imurmurhash-js: An incremental implementation of MurmurHash3 for JavaScript*. https://github.com/jensyt/imurmurhash-js. (Accessed on 07/25/2017).

[18] *jasondavies/bloomfilter.js: JavaScript bloom filter using FNV for fast hashing*. https://github.com/jasondavies/bloomfilter.js. (Accessed on 07/25/2017).

[19] *A Simple Bloom Filter in Python*. https://gist.github.com/josephkern/2897618. (Accessed on 07/25/2017).

[20] John Kurlak. *Why do bloom filters have multiple hash functions?* (Accessed on 08/08/2017). URL: https://www.quora.com/Why-do-bloom-filters-have-multiple-hash-functions/answer/John-Kurlak.

[21] *Bloomfilter calculator*. https://hur.st/bloomfilter. (Accessed on 08/05/2017).

[22] *RFC 4291 - IP Version 6 Addressing Architecture*. https://tools.ietf.org/html/rfc4291. (Accessed on 08/02/2017).

[23] *Baseline Requirements Certificate Policy for the Issuance and Management of Publicly-Trusted Certificates*. (Accessed on 08/08/2017). URL: https://cabforum.org/wp-content/uploads/CA-Browser-Forum-BR-1.4.5.pdf.

[24] *https://ccadb-public.secure.force.com/mozilla/CACertificatesInFirefoxReport*. https://ccadb-public.secure.force.com/mozilla/CACertificatesInFirefoxReport. (Accessed on 08/02/2017).

[25] *Let's Encrypt*. (Accessed on 08/02/2017). URL: https://letsencrypt.org/.

[26] *Securing a Public IP Address - SSL Certificates*. (Accessed on 08/08/2017). URL: https://support.globalsign.com/customer/portal/articles/1216536-securing-a-public-ip-address---ssl-certificates.

[27] *Database Query —RIPE Network Coordination Centre*. https://apps.db.ripe.net/search/query.html. (Accessed on 08/02/2017).

[28] Alex Beregszaszi. *Ethereum Improvement Proposal 74: Support RSA signature verification*. (Accessed on 08/08/2017). URL: https://github.com/ethereum/EIPs/issues/74.

[29] *Solium Framework*. (Accessed on 08/08/2017). URL: https://github.com/duaraghav8/Solium.

[30] Vitalik Buterin. *Thinking about Smart contract security*. (Accessed on 08/08/2017). 2016. URL: https://blog.ethereum.org/2016/06/19/thinking-smart-contract-security.

[31] V. Buterin. *Ethereum Improvement Proposal 150: Long-term gas cost changes for IO-heavy operations to mitigate transaction spam attacks*. (Accessed on 08/08/2017). URL: https://github.com/ethereum/EIPs/issues/150.

[32] Gavin Wood. *Ethereum: A secure decentralised generalised transaction ledger*. (Accessed on 08/08/2017). URL: http://gavwood.com/paper.pdf.

[33] *go-ethereum Github Repository. File* ***eth/config.go***. (Accessed on 08/08/2017). URL: `https://github.com/ethereum/go-ethereum/blob/fff16169c64a83d57d2eed35b6a2e`
`eth/config.go#L45`.

[34] etherscan.io. *Average Gas Price*. (Accessed on 08/08/2017). URL: `https://etherscan.io/chart/gasprice`.

[35] @ethgasstation. *The Safe Low Gas Price*. (Accessed on 08/08/2017). URL: `https://medium.com/@ethgasstation/the-safe-low-gas-price-fb44fdc85b91`.

[36] Coinbase. *Bitcoin*. (Accessed on 08/08/2017). URL: `https://www.coinbase.com`.

[37] Wikipedia. *The DAO (Organization)*. (Accessed on 08/08/2017). URL: `https://en.wikipedia.org/wiki/The_DAO_(organization)`.

[38] CNBC. *Ethereum briefly crashed from $319 to 10 cents in seconds on one exchange after multimillion dollar trade*. (Accessed on 08/08/2017). URL: `http://www.cnbc.com/2017/06/22/ethereum-price-crash-10-cents-gdax-exchange-after-multimillion-dollar-trade.html`.

[39] Xawery Wisniowiecki. *What is Gas Limit in Ethereum?* (Accessed on 08/08/2017). URL: `https://bitcoin.stackexchange.com/a/42629`.

Abbreviations

CA	Certificate Authority
DoS	Denial of Service
DDoS	Distributed Denial of Service
IP	Internet Protocol
ETH	Ether (currency of Ethereum)
solc	Solidity compiler
URL	Uniform Resource Locator

Glossary

DoS attack Denial of Service attack. An attack which has the goal of taking down a service that should be available. Usually performed by creating a malicious request payload exploiting the service and triggers big computational work.

DDoS attack Distributed Denial of Service attack. An attack designed to take down a service, by flooding the service with many requests from different sources, for example by controlling botnets.

List of Figures

List of Tables

Appendix A

Installation Guidelines

Using a UNIX terminal, navigate to the source code folder on the CD:

```
cd code
```

If no CD is available, the project can be cloned from the public Github repository (`git` required):

```
git clone https://github.com/JonnyBurger/ddos-bachelor-thesis && cd code
```

If the `node` command is not installed, install it from `https://nodejs.org`. The project has been tested with version 8.1.1 of node.js (`node -v` to see the version).

As the last step, the dependencies need to be installed:

```
npm install
```

The successful installation can be verified using:

```
node index
```

The above command also gives a help message to which commands can be used.

Appendix B

Open Source statement

The source code of this project is open source and available under `https://github.com/JonnyBurger/ddos-bachelor-thesis`.

All code is licensed under the MIT license, whose full text can be found under `https://github.com/JonnyBurger/ddos-bachelor-thesis/blob/master/code/LICENSE`.

Appendix C

Contents of the CD

code/: The source code of the project.

Raw data.xls: Excel spreadsheet containing all the raw data and calculations needed to create the figures.

Related Work/: A collection of related work papers.

tex/: The source code for this document.