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: August 15th, 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/

Zusammenfassung

Attacken wie Distributed Denial-of-Service (DDoS) stellen eine 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 Contracts 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.

Abstract

Attacks like Distributed Denial-of-Service (DDoS) pose a growing threat to computer networks and internet services. Existing strategies for mitigating DDoS attacks are inefficient because of lacking resources and inflexibility. Blockchains like Ethereum enable new ways to fight DDoS attacks: Using smart contracts, IP addresses of attackers can be singalized without additional infrastructure. This work documents the development of multiple smart contracts for signalisation of DDoS attacks and compares them, describes the Ethereum environment and its effect on the smart contract architecture, gives information and advice about the performance and costs and evaluates the overall feasability and effectivity of a blockchain-based solution for fighting DDoS attacks.

Acknowledgments

I would like to thank my supervisors, Thomas Bocek and Sina Rafati, for helping me find this topic and continuously 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. This technology is deemed so useful that users are trading Bitcoin tokens on digital exchanges and that its market capitalization of Bitcoin is over 60 billion USD as of August 2017 [1].

Ethereum [2] 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 of 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 allow 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 the scenario where a machine or network resource that should be online is being disrupted [3]. 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 [4].

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.

A DDoS attack can be stopped by blocking the traffic from the attacker. Each traffic package contains information about the source, including an identifier called the IP (Internet Protocol) address. By filtering the traffic based on the source IP address, the attack can be mitigated.

Victims of DDoS attacks can receive help by identifying the source IP addresses of attackers and notifying the upstream providers, so that they can block traffic before it reaches the victims infrastructure.

1.3 Motivation

The amount and intensity of DDoS attacks globally is on the rise and mitigation is happening only with limited success [5]. DDoS protection is a burden for organizations and requires a lot of human and financial resources, such that many organizations are hesitant to invest in protection for theoretical DDoS attacks. 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 decentralized database that is readily available and that can not be taken down [2]. With Solidity, the blockchain is scriptable and interfaces for storing and retrieving IP addresses can be programmed. With Ethereum being an infrastructure independent from web services, it is an interesting candidate for signaling IP addresses.

Existing DDoS signaling systems such as [6] send messages about attack information in key-value form using classical server infrastructure. The Ethereum blockchain allows to send messages in a similar format, but requires no additional infrastructure and will not be affected in case of a DDoS attack.

1.4 Description of Work

The paper "A Blockchain-based Architecture for Collaborative DDoS Mitigation with Smart Contracts and SDN" [7] 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 Internet Service Providers (ISPs) who can filter out the malicious packets before they reach the victim of the attack.

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 and retrieval of a list of IP addresses plus relevant metadata. All variants do 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 [7] (Listing 1-3).

2. A contract that stores an URL pointing to a static web resource containing all the information.

3. A contract that implements a bloom filter for the sake 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 overall. A look into the future, including further work needed and the development of the Ethereum ecosystem is given.

1.5 Thesis Outline

Chapter 2: Discusses related work.

Chapter 3: The smart contracts are being specified and developed. The chapter describes the implementation technique, development process, testing strategy, security considerations 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. [4] 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 [8] 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) [6] 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 via a REST-based API, and is 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 even signaling status updates on the efficacy of the mitigation.

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" [7] (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

3.1 Solidity Primer

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 [9] 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 20-byte address of an Ethereum network user.

This Solidity code can be compiled to bytecode and deployed on an Ethereum blockchain using free tools like solc [9] and web3 [10]. 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 requires dozens of Gigabytes of space [11], and it is ever-growing. 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` (the `address` of the transaction sender), `msg.gas` and `msg.value` (for sending Ether currency).

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

A constant method, like `function greet() constant returns (string)` in the example above 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.

3.2 IPv6 considerations

With IPv4 addresses being 32 bits long, only 2^{32} combinations are possible and the amount of free addresses is almost exhausted [12]. This leads to the situation that there is currently 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 [13]: 80 bits of zeros, 16 bits of ones, followed by 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, most notably, is already supported by the Linux kernel [14]. The Google Chrome browser and the Firefox browser will, when entering `http://[::ffff:2e65:6095]`, display the same website as when entering the IPv4 notation, `http://46.101.96.149`, which is easily verifiable.

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 should be considered an IPv4 address.

3.3 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. This section describes the tools and the workflow used during development.

3.3.1 Blockchain

The main Ethereum blockchain is unsuitable for manual testing during development. There are significant costs for deploying 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 [15]. The Ether needed to deploy contracts is not traded on exchanges and was mined with a much lower difficulty level, so costs of using the Testnet are not a concern. However, the Testnet is also a globally shared blockchain with confirmation times and is not perfect for development either.

`TestRPC` [16] 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.

3.3.2 Compiler

solc [9] is the compiler that comes with the Solidity language itself. Because no specific requirements forbid it, the reference compiler was chosen.

Other compilers exists, such as Remix [17], a compiler which runs in the browser.

3.3.3 Testing

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.

solc for example 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 in the Ethereum context 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` [10], 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` [18] framework. However, this choice is arbitrary, testing with another framework will work similarly.

With a macro function, it is possible to create an isolated blockchain for each test 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, which does generate a separate blockchain interface and list of addresses. This prevents test cases from interfering with each other — with `ava` it is even possible to run tests in parallel.

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:

```
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()
                });
```

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

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

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 to 1'000'000 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 of returning a value, a user-defined function is called[3]. `web3` oddly calls the callback function twice – only after the second time it is safe to assume the transaction is committed to the blockchain.

Once the contract instance is created and on the blockchain, methods of the contract can be called. The method `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 and indicate to the developer that the code does not behave as expected.

When running the test[4], the test framework should give `1 passed` as the output and exit with code 0 to indicate that all tests passed. Exit code 1 is used if there is at least one problem.

During the development of the smart contracts, 20 test cases were written[5], testing the normal use and edge cases of the contracts using dummy data. In addition, 3 tests were intentionally marked as known failures to demonstrate that a certain approach did not work. These failed tests cover code that was not used in the final contracts.

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.

[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.

[5]See Installation instructions in the appendix to run all tests.

3.4 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 the contract body, two structs are defined, with syntax resembling that of the C programming language:

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

struct Report {
    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. For example, '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 IPv4, the maximum value for a mask is 32, 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 variable.

The next part of the contract is 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. This concept is taken from the original paper [7].

The address of the creator of the contract instance gets stored 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 a 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. A user of the smart contract could inadvertently give permission to an user to register reports for the whole range of IP addresses by forgetting a method argument, hence the check using the modifier.

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

The following method is 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.

The method also checks if the supplied IP range is outside the IP boundary and throws if this is the case. For that, if the mask of the IP boundary is `n`, the last 128 - `n` bits of both IP addresses are set to 0 and then both IP addresses are compared to each other. 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. An additional check has the be made whether the supplied mask has a smaller numerical value than the mask of the IP boundary. If this is the case, then it is automatically a violation because it cannot be a subset of the IP boundary.

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 was registered using the `createCustomer` method.

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 a technicality, the reports array can not be marked as `public`, because public nested structs are not supported in Solidity at the time of writing. For programming with Solidity, it is generally advised to keep the data structure as flat as possible to avoid this problem. Although all data in a contract is technically public, it is complicated to access, as disassembly of native blockchain data is required. In order to create an interface where blockchain users can read nested structs, it is necessary to flatten the structure into one-dimensional 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);
}
```

The `blocked` function 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. No lambda functions or arrays of dynamic size are supported in Solidity. 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 the methods are marked as `internal` and are only invoked 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 `web3.js` (the Javascript client), or using a graphical interface like Mist [19].

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 [20] allows to easily convert a string representation of an IP address to a big integer and vice versa:

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

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

3.5 Contract 2: Pointer to web resource

The main disadvantage of storing 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.

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 prone to connectivity issues and does not take full advantage of the decentralization and immutability of the blockchain.

3.5.1 Smart Contract

The basic data structure of the smart contract is similar to the array store contract. The difference is that the items do not contain IP addresses, but URLs which point to lists of IP addresses. 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 no array is needed and 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 destinationIp;
    uint8 destinationMask;
    bytes32 _hash;
}

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, bytes32 _hash) {
    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,
        destinationIp: subnetIp,
        destinationMask: subnetMask,
        _hash: _hash
    });
}
```

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>` type in Java, 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` does not nest the `IPAddress` struct inside the `Report` struct like in the previous contract, but instead stores IP address 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, therefore this type of structure is not supported.

For this reason, the contract is programmed to do without nested structs, but with 'mappings' instead, another Solidity language feature. The benefit of mappings is that no helper methods are needed to retrieve the data.

Verifying that the sender of the transaction is a customer works by comparing `members[msg.sender].ip` to 0. In many other programming 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'. Instead, accessing a primitive value that has not been set returns zero and accessing a struct that has not been set returns a struct where all values are zero.

3.5.2 Web resource

Several design aspects need to be considered for the web resource. 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.

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, since there are a selection of clients for many languages available and because they are heavily standardized.

The use of an established syntax increases the **Portability** of the protocol. An essay by Nicolas Seriot [21] shows the challenges of covering edge cases in syntax 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 JSON and XML syntaxes allow to extend a schema by adding more keys to an object (JSON) or by adding more allowed tags to the schema (XML). **Upgradeability** is a desired property of the protocol, because it allows the web resource format 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.

With the expectation that lists of reports can 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**.

Streaming in CSV is simple: 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 [22] provides a reasonable tradeoff. Its syntax is a list of items that are delimited using a newline character (`\n`), where each item is a valid JSON object according to the JSON standard RFC 7158 [23]. 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. Unlike in a blockchain, a web resource can change its content 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 when the entry into the blockchain was created. To avoid 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 themselves and should reject reports that do not have matching hashes. This technique

is inspired by 'Subresource integrity' [24], which validates resources like stylesheets and scripts in browsers and is already widely deployed.

Immutability also brings another advantage: If an asset is static, then it never needs to be generated on-the-fly when requesting it and it can be easily deployed to a Content Distribution Network (CDN), which is harder to deny using a DDoS attack.

To enable immutability, the `Pointer` struct in the smart contract simply contains another property, a `hash` with a type of `string`. 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 these 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.

In the following, a few rules are established to avoid ambiguity and security vulnerabilities: 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 [25].

whitelist can either be `true` or `false`. The default is `false`. When this flag is set to `true`, 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 [6] 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=` [6]. This value is set for the `hash` field when registering the web resource in the smart contract.

3.6 Contract 3: Embedded 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 within the smart contract.

A bloom filter is a probabilistic data structure that is very space efficient [26]. 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.

A false positive is the condition where a function erroneously returns a positive result, where it should have returned a negative result. A false negative is the condition where a function erroneously returns a negative result, where it should have returned a positive result. In a bloom filter, when checking whether a string has been inserted before, it is possible to get false positives, but not false negatives [26].

The way a bloom filter works is by creating a fixed-length array that initially only contains zeroes. When adding a string to the bloom filter, it gets hashed and the hash determines which array items get shifted to ones. The string itself does not get stored. Therefore it is not possible to read all entries that have been added, but it is possible to check if a

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

given string has been added by re-hashing the string and checking if the array contains ones at the calculated indices.

This variant is space-efficient and unlike variant 2 which needed a web resource extension, all logic is self-contained, meaning the contract can run standalone.

However retrieving data with perfect accuracy cannot be guaranteed anymore. Also, additional parameters such as a blacklist/whitelist flag, expiration dates, IP boundaries can not be stored in a bloom filter, as it is impossible to retrieve the information that the bloom filter was fed without testing.

It is however possible to store additional information in the contract outside the bloom filter, 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.

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.

3.6.1 Hashing function

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 should use a hash function that produces as uniform results as possible. However, usually more uniform results also means slower hashing [27].

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 hard to port with identical behvaior because of differences between programming languages.

According to [27], cryptographic hashes are a suboptimal choice for bloom filter hashing, because they are relatively slow — simple hashing algorithms with enough independence like Murmur or Fowler-Noll-Vo (FNV) are preferred [28].

Therefore, in an first attempt, the popular hashing library `imurmurhash` [29] was taken and ported Solidity with as identical behavior 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 and Javascript environments. Otherwise, the Solidity version of the `imurmurhash` hash function is identical in its mechanics.

However, this hash function did not work — when later using the ported hash function in a bloom filter, it would return many false positives[7].

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

In a second attempt, the `bloomfilter.js` [30] library was ported to Solidity. 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. This led to the assumption that the `>>>` operator is not a requirement for it to work. 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 big 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 exists 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[8].

Porting existing hash functions to Solidity provides great challenges. Two hashing algorithm functions are globally available in every Solidity contract, `sha256()` and `sha3()`. For the third attempt, bloom filter implementations were searched that used one of the hash functions already implemented in Solidity. The 'Simple Bloom Filter' [31] code snippet that used SHA-256 hashing 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. However, as mentioned above, cryptographic hash functions are not ideal as they have worse performance [28].

3.6.2 Hashing parameters

Instead of only hashing an input value once, it usually is hashed multiple times. This reduces the chance of collision [32]. 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 [31].

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) [33]. Since every node in the network needs to download every transaction, there is an artificial cap on how computationally expensive a transaction may be.

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

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. Beyond that, this specific contract would not be able to have a transaction executed.

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 in the future without hitting the gas threshold.

3.6.3 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 [31].

```
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;
}
```

3.6.4 IPv6 format ambiguity consideration

An IPv6 address can be formatted in more than one way [34]. 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 [34] mentions. Passing the non-shortened IP does not result in higher storage costs, as a SHA256 hash is always only 256 bits long.

3.7 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, it was suggested by the original paper [7] there should be a way to automatically verify the ownership of the destination IP addresses, leaving the implementation open.

This problem 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 [35]), 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 domains and issue certificates for it. CAs need to fulfill a wide range of requirements [36] 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 [37].

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 an issuance fee for domains. The notable exception is "Let's Encrypt" [38], which makes money through sponsorships.

Although it is technically possible to issue an SSL certificate for an IP address, it is very uncommon. GlobalSign [39] is the only provider whose certificates are trusted by Firefox and that issues certificates for IP addresses. To obtain a certificate, it is a requirement that the IP address is registered in the RIPE database [40], which most are not, and a certificate starts at $349.

Concluding the overview of the certificate issuance process, there are no suitable 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 would be possible to obtain and validate certificates for IP addresses, it is a computationally expensive task that would likely reach the gas limit of Ethereum. This is just a hypothesis however, as 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 [41]. 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 — everything in the blockchain is public including stored certificates and IP addresses, it can also be done off the blockchain.

To propose an alternative solution for identity verification, each Ethereum transaction is signed by the user and the `msg.sender` value is verified by the network, guaranteeing the authenticity of the sender address. By pre-validating the destination IP addresses before a customer is added, no reports can be added to the contract on others behalf.

3.8 Security considerations with Solidity

Applications written in Solidity deserve special security considerations for several reasons. Unlike classical applications, code for transactions runs on every node in the network and any user can call any method. Therefore it is necessary to implement proper access control for each method.

The source code of smart contracts usually is made public after the deployment of it to allow users to verify the behavior of the contract before they interact with it. This increases the chance of bugs being found. Examples of it are TheDAO contract and the Parity multisig wallet, both of which were hacked because a vulnerability was found in the source code [42] [43].

All data stored in the smart contract has to be considered public. A rock-paper-scissor contract which people used for gambling turned out to have a trivial flaw where the first move could be extracted from the blockchain - rock would have the value `0x60689557` and scissor and paper would have a different one [44].

The main takeaway for the contracts described in this contract 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 if they know the contract address.

The creator of Solidity, Christian Reitwiessner, advocates for implementing a 'fail safe' mode that can be activated in case of a hack that will turn the contract into a read-only, withdraw-only mode [45].

The creator of Ethereum, Vitalik Buterin, has compiled a list of vulnerabilities based on real-world exploits. To be safe against the most common vulnerabilities, the following practices should be followed [44]:

Constructor naming The constructor needs to have the same name as the contract. If by mistake, the constructor has a different name, it is not called on deployment, but can be called as a transaction. In many contracts, including the variants in this thesis, calling the constructor makes the caller the owner of the contract. The 'Rubixi' contract suffered from this bug and led to a takeover vulnerability [44].

Loops Loops are susceptible to gas limit failures and can be stalled. Therefore, the number of iterations in a loop should not be controlled by transaction parameters. Additionally, the `var` keyword should not be used within a `for` statement, as `var` is interpreted as `uint8`, which would lead to an overflow if the loop exceeds 256 iterations.

Call stack depth The 'call stack depth' describes the amount of nested function calls — it increases when a function calls another function, and if a function returns, the call stack depth decreases. 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. This limit is 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. Therefore contracts must be designed to not expose vulnerabilities when a function that is susceptible to a call stack depth attack gets only partially executed. The core development team of Ethereum proposes to solve this problem on a language-level in EIP #150 [46].

Chapter 4

Cost model

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.

This chapter describes the internal pricing mechanism of smart contracts and develops a formula to calculate usage costs given certain variables.

4.1 Cost variables

The costs of a single transaction on the Ethereum blockchain is dependent 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 the cost model, gas is treated as a real money cost.

The 5 variable factors are:

1. The operations being done: Each smart contract instance and transaction consists under the hood of a set of 31 possible assembly operations [47]. 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 [47], a fee for each operation is defined. For example, a transaction costs 21000 gas and a SSTORE operation costs 20000 gas.

In general, it can be stated that more complex contracts and transactions cost more gas. Interestingly, the different fees for operation types 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 operations 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 [46]. For example, the cost of a CALL operation will be increased from 700 to 4000 gas with the Metropolis upgrade. 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 hard-coded into Ethereum clients, but determined by the users dynamically.

Each Ethereum client 'signals' a gas price. The default configuration of go-ethereum [48], 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 [49] shows that the clients of the network mostly leave the default configuration value unchanged, 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 is filled first. 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 [49]. 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 [50] 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

[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.

to the same sites gas-time calculator [51], 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.

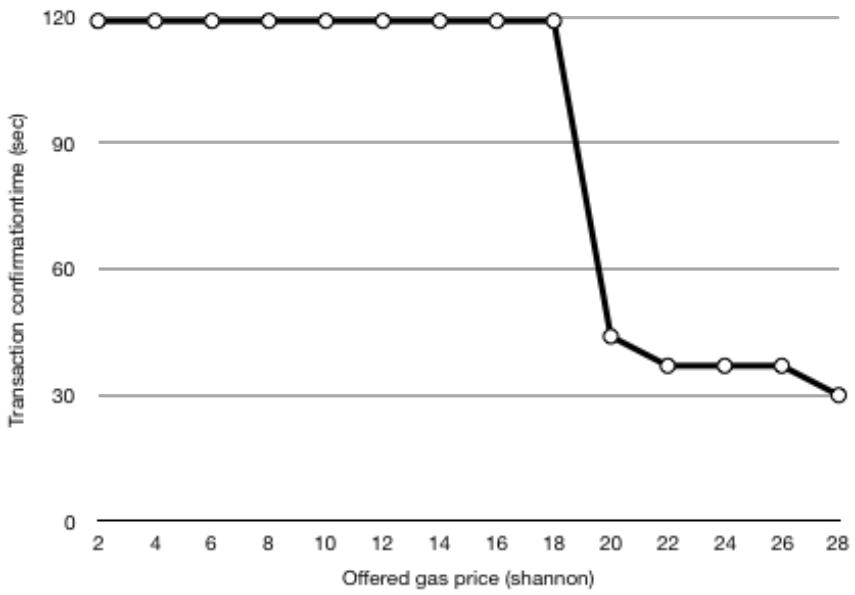

Figure 4.1: Average time until transaction is confirmed. Made with data from ethgasstation.info on May 8th.

Figure 4.1 shows 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, as on January 1st 2017, the price of an ether was $8.22 on the Coinbase exchange [52] and on On August 14th 2017, it was $300.48. Ether has seen a price drop of over 60% when a smart contract called "TheDAO" was hacked in June 2016 [42]. In July 2017, the price dropped from over $400 to $135, causing Ether to lose two thirds of its value temporarily with no hacking incident causing it [52]. 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 [53]. The reason for this is that not enough buy orders were placed to fill the sell order. This shows that low volume on Ethereum exchanges is a major cause of price volatility and 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 is 318'552 gas when compiled with Version 0.4.8 of the Solidity Compiler (solc), but slightly different in other compilers (Table 4.1).

```
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

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).

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 [47]. 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 [33] (set to be raised to 5'500'000 in the next fork [46]). At the moment, this means that the condition in Equation 4.2 is always fulfilled.

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

The gas price is denoted as the variable α. Since according to ethgasstation.info (Figure 4.1, [51]), 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 $ETH = \$50$, the cost to deploy that contract is \$0.05 (Equation 4.7).

$$500'000 \cdot 30 \cdot 10^{-9} \cdot \$50 = \$0.05 \tag{4.7}$$

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 dependent 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 4.

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`.

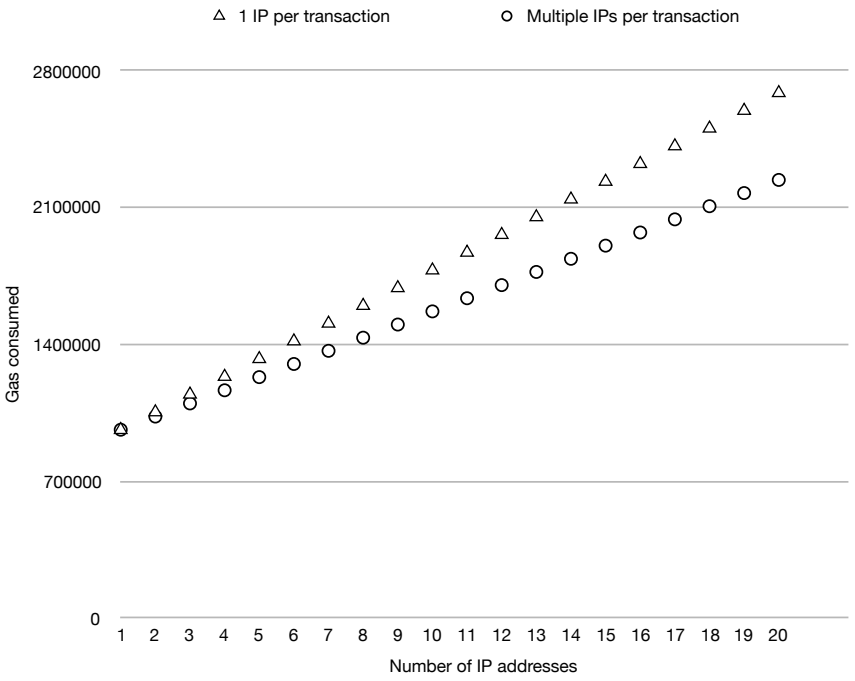

Figure 5.1: Gas cost incurred using variant 1

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. Assuming 20 reports, and ranges of $\alpha = [2, 28]$ and $ETH = [\$100,\ \$300]$, the amount of gas consumed corresponds to real-money costs between \$0.45 and \$18.83 according to the cost model, however linearly increasing with the amount of reports.

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 is consumed. This makes variant 2 the cheapest contract in terms of gas. Using the same values for the number of reports, α and ETH as for the calculation in variant 1, the real-world costs are between \$0.15 and \$6.3, however they do not increase with the amount of IP addresses stored.

[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`.

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 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.

5.1.3 Variant 3

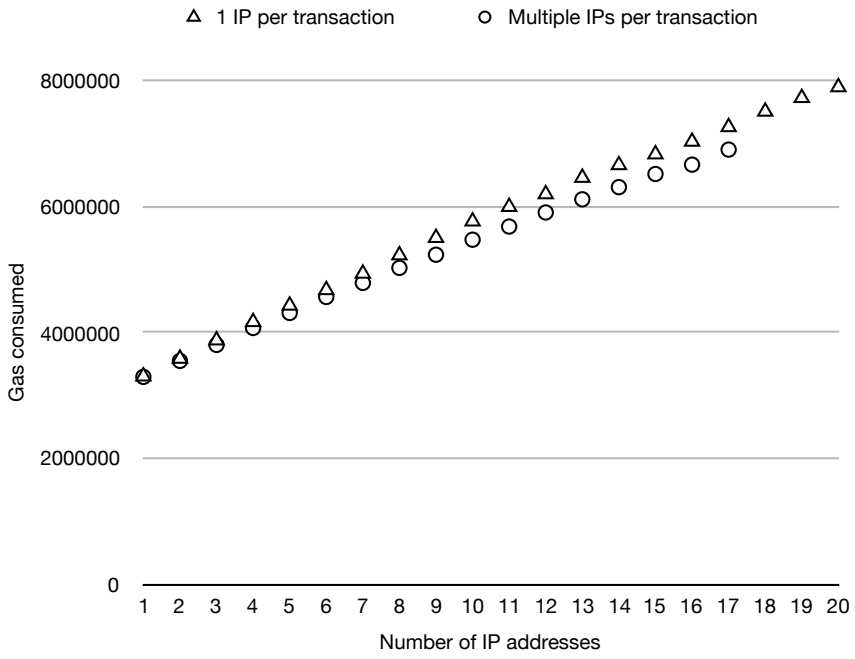

Figure 5.2: Gas cost incurred using variant 3

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.

[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`.

Using the worse case (since 20 IPs cannot be stored using just 1 transaction) and the same values and methodology as in variant 1 and 2 of, the real-world costs are between $1.58 and $66.33. This cost is also linearly growing with the amount of IP addresses inserted.

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 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, which has a size of multiple gigabytes [11], needs to be downloaded to a client anyway — the only reason to optimize for storage is to get cheaper costs.

5.2 Speed

Current speed is dependent on several network parameter and can change over time. Additionally, by tweaking the amount of gas provided to a transaction the speed of inserting reports is modifiable.

These factors make speed hard to quantify without some assumptions about the network and the user preference. In chapter 4, the range of possible speeds is calculated and mapped to the range of prices. The range of speeds at the time of writing is 30 seconds to 120 second and is visualized in Figure 4.1. All Ethereum transactions fall inbetween these transaction times, therefore all variants are equally fast on average given the same gas price offered.

5.3 Accuracy

Variant 1 and 2 store the reports and the IP addresses in a lossless format, guaranteeing perfect accuracy in that the retrieved data is identical with the inserted data. Variant 3 however does lose accuracy because of the bloomfilter, the exact amount of which is calculated in this section.

Given a number of items that are indexed, and a number of hash functions, the ideal array size and number of hash functions for a bloom filter can be calculated. According to [54], 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 5.1 and the number of hash functions k using equation 5.2.

$$m = \lceil \frac{n * log(p)}{ln(\frac{1.0}{2.0^{ln(2)}})} \rceil \tag{5.1}$$

$$k = \lfloor ln(2) * m/n) \rceil \tag{5.2}$$

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 5.1 for p gives equation 5.3.

$$p = e^{\frac{m \cdot ln(\frac{1}{2^{ln(2)}})}{n}} \tag{5.3}$$

The probability of at least one false positive is shown in table 5.1 and figure 5.3. It assumes the number of **bits** in the filter is 4'096 (512 $bytes \cdot 8$) as it is in variant 2 of the contract.

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 5.1: Probability of false positives, using equation 5.3 and $m = 4'096$

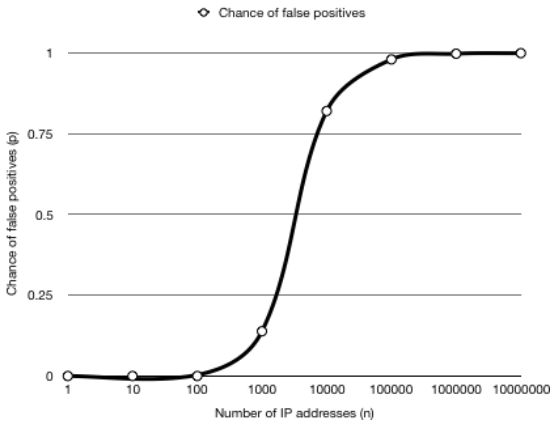

Figure 5.3: Probability of false positives, using data from Table 5.1 (logarithmic scale)

The bloom filter does have a chance of a false positive 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 constraint 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.

Chapter 6

Summary and Conclusions

Ethereum provides a new platform for decentralized applications of many kinds. Using the turing-complete programming language Solidity, smart contracts can be programmed to solve a wide variety of problems.

In order to enable decentralized applications, Ethereum must make some tradeoffs by disincentivizing computation-heavy or space-inefficient applications with cost and limitations.

Several factors decide the cost of Ethereum applications. The complexity of the application, parameters of Ethereum clients and Ether price influence the cost. Speed is correlated to cost with faster transactions for higher prices.

Three protoypes of a smart contract for signaling DDoS attacks for the Ethereum platform were developed, tested and benchmarked. In general, all variants are functional and can be used to store IP addresses. For a small number of IP addresses, the smart contracts are reasonable solutions that are relatively cheap to implement. However, storing more than a few hundred IPs directly in the contract causes serious scalability issues that are hard to overcome on the Ethereum blockchain.

The approach to directly store all IP addresses in an array results in high costs as expected. The supposed solution, the bloom filter, only increased the cost while also introducing imperfect accuracy and eliminating the possibility to obtain a full list of stored IP addresses.

The approach of pointing to a list of IP addresses on the web is the most scalable approach of the three, outsourcing the big data to an established protocol. By testing the integrity of the resources using a hash, the immutability properties of the blockchain can be extended to the web resource as well. On the contrary, this variant is the least ambitious of the solutions and does not fully deliver on the promise of a blockchain-based solution.

In conclusion, Ethereum as a general-purpose blockchain is not the ideal technology to run DDoS signaling applications. However, most of the issues come down to scalability and the developed solutions are viable for signaling small amounts of data. To further advance the idea of decentralized DDoS signaling, specialized blockchains can be developed which are better optimized for this type of application.

Bibliography

[1] *CryptoCurrency Market Capitalizations*. https://coinmarketcap.com/. Last visited 08/12/2017.

[2] *Ethereum: Blockchain App Platform*. Last visited 08/08/2017. URL: https://ethereum.org/.

[3] US-CERT. *Understanding Denial-of-Service Attacks*. Last visited 08/08/2017. Nov. 4, 2009. URL: https://www.us-cert.gov/ncas/tips/ST04-015.

[4] 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.

[5] 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.

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

[7] 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.

[8] 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.

[9] *The Solidity Contract-Oriented Programming Language*. Last visited 08/08/2017. URL: https://github.com/ethereum/solidity.

[10] *ethereum/web3.js: Ethereum JavaScript API*. https://github.com/ethereum/web3.js/. Last visited 07/30/2017.

[11] *What is the data size of the Ethereum blockchain? : ethereum*. https://www.reddit.com/r/ethereum/comments/6kvzvp/what_is_the_data_size_of_the_ethereum_blockchain/. Last visited 08/13/2017.

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

[13] R. Gilligan. *https://curl.haxx.se/rfc/rfc3493.txt*. https://curl.haxx.se/rfc/rfc3493.txt. Last visited 08/13/2017.

[14] *IPv6 readiness IPv6 Wikipedia*. Last visited 08/13/2017. URL: https://en.wikipedia.org/wiki/IPv6#IPv6_readiness.

[15] *Test Networks. Ethereum Homestead 0.1 documentation*. http://ethdocs.org/en/latest/network/test-networks.html. Last visited 08/13/2017.

[16] *ethereumjs/testrpc: Fast Ethereum RPC client for testing and development*. https://github.com/ethereumjs/testrpc. Last visited 07/30/2017.

[17] *ethereum/remix: Ethereum IDE and tools for the web*. https://github.com/ethereum/remix. Last visited 08/13/2017.

[18] *avajs/ava: Futuristic JavaScript test runner*. https://github.com/avajs/ava. Last visited 08/13/2017.

[19] *ethereum/mist: Mist. Browse and use Dapps on the Ethereum network*. https://github.com/ethereum/mist. Last visited 08/13/2017.

[20] *ip-address*. https://www.npmjs.com/package/ip-address. Last visited 08/13/2017.

[21] Nicolas Seriot. *Parsing JSON is a Minefield*. Last visited 08/08/2017. Oct. 26, 2016. URL: http://seriot.ch/parsing_json.php.

[22] Wikipedia. *JSON Streaming - Line delimited JSON*. Last visited 08/08/2017. URL: https://en.wikipedia.org/wiki/JSON_Streaming#Line_delimited_JSON_2.

[23] IETF. *RFC 7158*. Last visited 08/08/2017. URL: https://tools.ietf.org/html/rfc7159.

[24] Mozilla. *Subresource Integrity*. Last visited 08/08/2017. URL: https://developer.mozilla.org/en-US/docs/Web/Security/Subresource_Integrity.

[25] Tom Preston-Werner. *Semantic Versioning 2.0.0*. Last visited 08/08/2017. URL: http://semver.org/.

[26] *Bloom filter - Wikipedia*. https://en.wikipedia.org/wiki/Bloom_filter. Last visited 08/11/2017.

[27] *Bloom Filters by Example*. https://llimllib.github.io/bloomfilter-tutorial/. Last visited 08/11/2017.

[28] Adam Kirsch and Michael Mitzenmacher. "Less hashing, same performance: Building a better bloom filter". In: *In Proc. the 14th Annual European Symposium on Algorithms (ESA 2006*. 2006, pp. 456–467.

[29] *jensyt/imurmurhash-js: An incremental implementation of MurmurHash3 for JavaScript*. https://github.com/jensyt/imurmurhash-js. Last visited 07/25/2017.

[30] *jasondavies/bloomfilter.js: JavaScript bloom filter using FNV for fast hashing*. https://github.com/jasondavies/bloomfilter.js. Last visited 07/25/2017.

[31] *A Simple Bloom Filter in Python*. https://gist.github.com/josephkern/2897618. Last visited 07/25/2017.

[32] John Kurlak. *Why do bloom filters have multiple hash functions?* Last visited 08/08/2017. URL: https://www.quora.com/Why-do-bloom-filters-have-multiple-hash-functions/answer/John-Kurlak.

[33] Xawery Wisniowiecki. *What is Gas Limit in Ethereum?* Last visited 08/08/2017. URL: https://bitcoin.stackexchange.com/a/42629.

[34] *RFC 4291 - IP Version 6 Addressing Architecture*. https://tools.ietf.org/html/rfc4291. Last visited 08/02/2017.

[35] *IP Source Address Spoofing BCP38*. http://www.bcp38.info/index.php/IP_Source_Address_Spoofing. Last visited 08/13/2017.

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

[37] *https://ccadb-public.secure.force.com/mozilla/CACertificatesInFirefoxReport*. https://ccadb-public.secure.force.com/mozilla/CACertificatesInFirefoxReport. Last visited 08/02/2017.

[38] *Let's Encrypt*. Last visited 08/02/2017. URL: https://letsencrypt.org/.

[39] *Securing a Public IP Address - SSL Certificates*. Last visited 08/08/2017. URL: https://support.globalsign.com/customer/portal/articles/1216536-securing-a-public-ip-address---ssl-certificates.

[40] *Database Query —RIPE Network Coordination Centre*. https://apps.db.ripe.net/search/query.html. Last visited 08/02/2017.

[41] Alex Beregszaszi. *Ethereum Improvement Proposal 74: Support RSA signature verification*. Last visited 08/08/2017. URL: https://github.com/ethereum/EIPs/issues/74.

[42] Wikipedia. *The DAO (Organization)*. Last visited 08/08/2017. URL: https://en.wikipedia.org/wiki/The_DAO_(organization).

[43] Ethereum\x{fffd}\x{fffd}\x{fffd}sParityUsersLoseMillionsinaMulti-SigHack. Last visited 08/14/2017.

[44] Vitalik Buterin. *Thinking about Smart contract security*. Last visited 08/08/2017. 2016. URL: https://blog.ethereum.org/2016/06/19/thinking-smart-contract-security.

[45] Russ Harben. *Smart Contract Security in Ethereum: Lessons learned from The DAO and the future of safe smart contracts*. Last visited 08/14/2017. URL: https://docs.google.com/presentation/d/1kS9mVOQNieloYByGQw3P-Yyup2BYE5tg7jOItMNnR0A/edit#slide=id.g15480448e8_0_185.

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

[47] Gavin Wood. *Ethereum: A secure decentralised generalised transaction ledger*. Last visited 08/08/2017. URL: http://gavwood.com/paper.pdf.

[48] *go-ethereum Github Repository. File* ***eth/config.go***. Last visited 08/08/2017. URL: https://github.com/ethereum/go-ethereum/blob/fff16169c64a83d57d2eed35b6a2e33248eth/config.go#L45.

[49] etherscan.io. *Average Gas Price*. Last visited 08/08/2017. URL: https://etherscan.io/chart/gasprice.

[50] @ethgasstation. *The Safe Low Gas Price*. Last visited 08/08/2017. URL: https://medium.com/@ethgasstation/the-safe-low-gas-price-fb44fdc85b91.

[51] ETH Gas Station. *Gas Time Calculator*. Last visited 08/14/2017. URL: http://ethgasstation.info/calculator.php.

[52] Coinbase. *Bitcoin*. Last visited 08/08/2017. URL: https://www.coinbase.com.

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

[54] *Bloomfilter calculator*. https://hur.st/bloomfilter. Last visited 08/05/2017.

Abbreviations

ABI	Application Binary Interface
CA	Certificate Authority
CDN	Content Distribution Network
CSV	Comma separated values
DoS	Denial of Service
DDoS	Distributed Denial of Service
ETH	Ether (currency of Ethereum)
FNV	Fowler-Noll-Vo
HTTP	Hypertext Transfer Protocol
HTTPS	Hypertext Transfer Protocol Secure
IP	Internet Protocol
IPv4	Internet Protocol Version 4
IPv6	Internet Protocol Version 6
ISP	Internet Service Provider
JSON	Javascript Object Notation
npm	Node package manager
REST	Representational State Transfer
SHA	Secure Hash algorithm
solc	Solidity compiler
URL	Uniform Resource Locator
XML	Extensible Markup Language

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.

The test suite can be run using the following command:

```
npm test
```

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.