University of Zurich^UZH

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

# Collaborative DDoS Mitigation Based on Blockchains

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

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

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

ifi

# Abstract

...

# Contents

# Chapter 1

# Note about this intermediate report

This is a work-in-progress version of my bachelor thesis. I plan on continuing the thesis according to the task description and the previously worked out schedule.

The thesis is accompanied by code, always available under

https://github.com/JonnyBurger/ddos-bachelor-thesis.

For that access, an invitation is required until the thesis is finalized, but a copy from Thursday, June 1st is provided.

# Chapter 2

# Introduction

## 2.1 Blockchains and Ethereum

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 crypto currencies, the most widely used implementation is Bitcoin. With Bitcoin, which had it's breakthrough in 2008, network users can exchange tokens securely over a completely decentralized protocol. Because of this usefulness, these tokens are valued with real money, the total market capitalization of Bitcoin is several dozen millions.

Ethereum [1] is blockchain protocol that is inspired by Bitcoin, but not only allows for sending and receiving of tokens, but also offers a scripting language called Solidity, which allows anyone to write programs which can be run on the blockchain. Examples for applications that could run on Ethereum are games like Tic-Tac-Toe or Poker, finance applications like venture capital funds and Initial Coin Offerings (a company raising funds by selling shares of it to investors). An application is also called a smart contract and allows for storage of arbitrary information and allows for users to send transactions that mutate the storage. The creator of the smart contract controls with code the permissions of the users, the conditions and behaviors of the mutations. This enables a wide variety of possible applications, including a collaborative DDoS mitigation solution, which this thesis is about.

## 2.2 Denial of Service and DDoS

A Denial of Service (DoS) is when a machine or network resource that should be online is being disrupted [2]. An attacker can either force a DoS by crafting a request payload causing a lot of computational work on the target machine or by flooding the target with requests. The motivation behind a DoS attack is that the attacker sees benefit in the victim's service being disrupted, be that disagreement with the service offered (activist attack), that the service is from a competitor, or that taking down a service brings pleasure to the victim [3].

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

## 2.3 Motivation

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

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

## 2.4 Description of Work

The paper ”A Blockchain-based Architecture for Collaborative DDoS Mitigation with Smart Contracts and SDN” [6] proposes to use the Ethereum blockchain as a registry for IP addresses from which attacks are originating from. The data can then be read by parties like ISPs who can filter out the malicious packets before they even reach the victim of the attack. This eliminates the need for additional architecture.

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

The three variants are:

1. A smart contract that stores a list of all IP addresses on the blockchain, similar to the contract shown in the original paper [6] (Listing 1-3). The difference is that it supports more features and is more robust, closer to a prototype from a proof of concept.

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

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

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

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

The smart contracts will be benchmarked for speed and cost. In addition to comparing numbers, opinions about ease of use and general suitability for the job are expressed.

For the conclusions, a winner is picked and a statement is made whether the experiment was positive. A look into the future, including further work needed and the development of the Ethereum ecosystem is given.

## 2.5 Thesis Outline

**Chapter 1:** Discusses related work.

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

**Chapter X:** The smart contracts based on the planning in chapter 1 are being implemented. The chapter describes the implementation technique, development process, testing strategy and documents the established protocols.

**Chapter 3:** The developed smart contracts for cost and speed are benchmarked and compared. In addition, a generic cost model for smart contracts is being introduced and optimization possibilities are being explored. Pros and cons of each variant are discussed.

**Chapter X:** A recommendation is made for a smart contract variant and make a statement whether the developed approach based on our research is suitable for real-world use.

# Chapter 3

# Related Work

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

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

A proposal submitted to the Internet Engineering Task Force (IETF) [5] describes a peer-to-peer protocol called "DDoS Open Threat Signaling" (DOTS) for signaling source IP addresses of DDoS attacks. The protocol that the authors propose communicates over HTTPS and is REST-API-based, which is the main difference to the solution proposed in this thesis. The communication can be intra- or inter-organizational. DOTS implements handshake calls, sending mitigation requests with various parameters, and reporting on the efficacy. The draft expired on June 30th 2017 because of inactivity according to IETF guidelines.

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

# Chapter 4

# Development

In the following, three variants of a DDoS attack signaling protocol are being developed. For that, a smart contract is being written in the Soldity programming language [8] for each variant. 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).

```
contract mortal {
    address owner;

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

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

contract greeter is mortal {
    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 `greeter`. Methods can be declared public or private. They are, like in Javascript, prefixed with the `function` keyword. A special type in Solidity is the `address` type, which can hold an 40-byte address of an Ethereum network user.

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

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

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

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

## 4.1 IPv6 considerations

With IPv4 addresses being 32 bits long, only $2^{32}$ combinations possible and the amount of free addresses is almost exhausted [9], there currently is 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 RFC 3493: 80 bits of zeros, 16 bits of ones, followed the IPv4 address. For example, the IPv4 address `46.101.96.149` would be `0:0:0:0:0:ffff:2e65:6095` in IPv6 hex representation. This notation is supported by the Linux kernel and macOS natively, in a web browser `http://[::ffff:2e65:6095]` will display the same website as `http://46.101.96.149`.

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

## 4.2 Contract 1: Native storage

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

```
pragma solidity 0.4.9;

contract ArrayStore {
}
```

Inside the contract body, some 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 added. This makes it possible to specify a range of IP addresses with no redundancy. When discussing masks, a notation such as '127.0.0.0/24' is used, where everything before the slash is the IP address base and the number behind the slash is the mask. A mask of '/32' means specifically and only that IP, while '/0' means the whole range of IP addresses possible. '127.0.0.0/24' means all IP addresses from 127.0.0.0 to 127.0.0.255. Since the contract works with IPv6 addresses, the maximum value for mask is 128.

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

The next step is to write the constructor function.

```
address owner;
IPAddress ipBoundary;

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

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

The address of the creator of the instance gets saved in the `owner` property. This allows to define that the owner of the contract can call more methods than other users.

The constructor also has a 'modifier'. A modifier is piece of code that is being run before the method body. The modifier `needsMask` simply throws when the user calls the constructor without the second argument (which the language itself allows). The underscore statement in Solidity can only be used in modifiers and its effect is that it jumps to the main method body immediately.

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

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

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

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

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

The following method provides an interface for registering a report:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

## 4.3 Contract 2: Pointer to web resource

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

### 4.3.1 Web resource

When pointing to a web resource, the design of the architecture that is not on the blockchain needs to be considered. By storing a URL in the blockchain, it is already implied that the connection to the resource is made using HTTP(S), which is a suitable protocol for our needs.

The format of the web resource pointed to could be in any format imaginable. A syntax and a data structure has to be settled on. In this section, various aspects of the web resource format are discussed and a format will be developed.

Among the desirable properties that are:

**1. Portability:** An already common syntax like XML, CSV or JSON is desired because client libraries are already available and no further specifications are needed.

**2. Upgradeability:** The format of the web resource should be able 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 rows.

**3. Streamability:** As one can expect these lists to become very large, it is beneficiary to have the format in a way where it can already be partially evaluated while not yet fully loaded. Downloading a resource fully will take time and it is desirable to not have to load the full list into memory. This is easy with CSV, as the file can be read line by line, but hard with XML or JSON, which needs to be fully loaded in order to be valid.

These properties are conflicting with each other, as no mentioned format supports upgradeability or streamability at the same time.

Additionally, it needs to be decided whether the web resource should be immutable or not. Immutable means that the content of the web page can never change. Additions or deletions to the list are not supported, if the list had to be updated, it would have to move to a different URL and the entry in the blockchain has to be updated as well. This is not only a bad thing, as otherwise the client who accesses the resource has to worry about checking for updates.

A hash can be generated for an immutable file, which could be stored in the blockchain as well and allow the clients to verify that the list has not been altered since it was registered in the smart contract, which is more inline with the rest of architecture that resides in the blockchain. If an asset is immutable, it can be easily outsourced to a CDN, which is harder to deny using a DDoS attack. In summary, immutability makes it slower and more expensive to register changes, but has the advantage of making it easy to manage changes.

## 4.4 Contract 3: Bloom filter

The third variant of the smart contract is a standalone contract that solves the scalability issue by using a bloom filter.

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

The bloom filter contract is a balance between leveraging the blockchain and using little space, however perfect accuracy cannot be guaranteed anymore.

## 4.5 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, there should be a way to automatically verify the ownership of the destination IP addresses. This is a challenging task, as is explained in this section. Using certificates to validate IP ownerships in Solidity is currently not practical for at least two reasons.

It is a computationally expensive task that would require more gas than the gas limit allows and there is no implementation of certificate verification in Solidity. Certificate verification, as it is most commonly done with OpenSSL, would have to be ported ported to Solidity, which is complex. There is however a proposal to add certificate validation on a language-level [10]. 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.

However, since everything in the blockchain is public including stored certificates and IP addresses, it is not required that certificates are validated in Solidity, it can also be done off the blockchain.

The second challenge is obtaining a certificate. As there is no way to directly mathematically or logically prove that somebody is the owner of an IP (it can be spoofed), an indirect solution is required. It was considered that the certificate process of domains could be applied to IPs: There are Certificate Authorities (CA) who verify and validate the ownership of a domain.

CAs need to make investments in establishing a system that securely validates a domain and manages the certificates that are issued. CAs need to fulfill a wide range of requirements [11] to be considered reputable and be included in the root key store of operating system and browser vendors. Nearly all CAs for domains issue certificates for a fee or need to generate revenue by sponsorships.

Although it is technically possible to issue an SSL certificate for an IP address, it is very uncommon. GlobalSign [12] is the only provider known to us that issues certificates for IP addresses, and requires that the IP is registered in the RIPE database.

The same system that currently exists for domain owner verification could be introduced by the industry for IP addresses verification, but is very complicated and defeats the purpose of using the blockchain as the original idea was to remove the need for additional infrastructure.

## 4.6 Validating Smart contracts during development

Developing a smart contract requires a compiler and a blockchain on which the developer can execute the smart contract. A compiler, such as solc [8], will warn about syntax errors and does not compile invalid Solidity code, indicating to the developer that there is an error in the code.

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. As with any software, developers need a workflow which allows them to test their changes fast and efficiently.

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

This is not the case in the Testnet, which is a separate blockchain for testing purposes. Still, the Testnet is a globally shared blockchain and is not perfect for development.

Instead, a test framework called `TestRPC` that makes it possible to set up a local blockchain for testing is used. 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 desired. This makes `TestRPC` the ideal blockchain for developing.

In addition to manual testing, the robustness should be validated by unit tests, just like in any other piece of software. This allows to automatically validate that all assertions are still true when a change is introduced to the contract. A robust contract should 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.

Examples of cases that should be tested are: Calling a function without permission, calling a function more often than expected, calling a function with unexpected arguments, such as null arguments, wrong types or a big payload.

`TestRPC`, the `solc-js` compiler, and the `ava` testing framework to set up a test environment. Each test is completely atomic and independent from other ones, that means that for each test, a separate `TestRPC` blockchain is created and the scenario is run on it. After the test is finished, the blockchain and the accounts get destroyed.

The complete isolation of each test is good practice, as side-effects can be ruled out. With `ava`, the tests can be run in parallel, making it faster to run the whole test suite. However, there are many other generic test frameworks that work just as well and have their own advantages and disadvantages.

One interesting specialized framework is Truffle [13], which provides helpers for developing and testing Solidity contracts. Truffle attempts to make the process described in this section easier and looks very promising, but is still in beta. For the reason that the framework is still very much in development and changing, a generic testing framework was used.

For real-world smart contract applications, testing is not enough, critical contracts should be audited by computer security professionals before being deployed. As the aim of this thesis is to document a proof-of-concept DDoS mitigation application, no audit will be conducted.

## 4.7 Security considerations with Solidity

Code vulnerabilities are more common in Ethereum than in other environments. The whole blockchain data is public, and serious contracts are made open source in order to create trust for the people who use it. Therefore applications are auditable and bugs are more likely to be found.

The application must also regulate itself and can in most cases not recover from an attack with an update.

In addition to that, the language Solidity aims to have a syntax similar to JavaScript and is therefore quite loose with enforcing the correct type. External static code analysis tools such as Solium [14] do not yet have as many rules included as comparable tools for more mature languages.

In this section, some common mistakes that can lead to security exploits will be explained, partially taken from a list compiled by the co-founder of Ethereum [15] and based on real-word vulnerabilities.

### 4.7.1 Constructor naming exploit

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

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

### 4.7.2 Public data

A misconception is that stored data can be made private on the blockchain. A rock-paper-scissor contract which people used for gambling turned out to have a trivial flaw where the first move could be seen - rock would have the value `0x60689557` and scissor and paper have a different one.

The main takeaway is that the stored IP addresses will be public (although maybe obfuscated) to everybody. Even attackers can determine the list of IPs that are blocked. The contract should be created in a way that minimises the usefulness for attackers, for example the pattern of the IPs should not make the way how IPs are being blocked predictable.

### 4.7.3 Loops

The section "Security Considerations" of the Solidity documentation warns about loops that have a non-fixed amount of iterations. With a high number of iterations, the block gas limit could be reached and according to the Solidity documentations "cause the complete contract to be stalled at a certain point".

In the reference contract, the function `blockIPv4()` has a loop whose amount of iterations are dependant of a transaction argument. The contract has to be tested with a big payload and a solution needs to be developed.

Furthermore, even for constant functions (which are executed locally and have no gas limit), there can be bugs. If, in the reference contract there would be a slight change:

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

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

### 4.7.4 Re-Entry bugs using .call()

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

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

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

### 4.7.5 Call stack depth attack

The call stack depth gets increased when a function calls another function, and if a function returns, the call stack depth gets decreased. A long call stack can be produced by excessive recursion. Solidity has a 1024 call stack depth limit - this means that Solidity could for example not compute the 1025th fibonacci number using recursion.

The deterministic depth limit of Solidity proves to be an attack vector. An attacker could craft a function that calls itself 1023 times and on the 1024th time calls another, vulnerable function, that stops execution as soon as a subfunction is called, because the maximum call stack size is reached.

An attacker might force a function to only partially execute, which is a problem, if for example a withdraw function is composited of a subfunction that sends funds and a function that decreases the users balance. The core development team of Ethereum wants to solve this problem on a language-level in EIP #150 [16] soon, therefore it will not be further discussed here.

# Chapter 5

# Cost model

## 5.1 Cost variables

Computing costs occur for blockchain applications, and because the blockchain is decentralized, there needs to be a reward system for the people that provide the service of confirming the transactions. Therefore, a blockchain application can be significantly more expensive than a comparable centralized service.

The costs of a single transaction on the Ethereum blockchain is dependant on at least 5 variable factors.

When a node wants to submit a transaction to the blockchain or create a new contract, a certain amount of 'gas' has to be offered to so called 'block miners'. For the purpose of a cost model, gas is as a real money cost and should be treated as a currency.

The 5 variable factors are:

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

**2. The gas costs assigned to those operations:** Gavin Wood has created a list of 31 operations and their prices, and these are in use today. For example, a transaction costs 21000 gas and a SHA3 hash operation costs 30 gas.

In general, it can be stated that more complicated contracts and transactions cost more gas. Interestingly, the different gas prices for operations are not necessarily proportional of the actual computational work needed, but were set by the Ethereum developers and accepted by the majority Ethereum users.

Because of the non-proportionality, some assembly instructions will change their gas prices in the future for rebalancing purposes when the majority of nodes upgrade to the Metropolis hard fork. This hark fork is not yet certain to happen, but will likely do so as this hard fork is pushed by Ethereum inventor Vitalik Buterin. For example, the cost of a CALL operation will be increased from 700 to 4000 gas with the Metropolis upgrade.

In summary, gas prices are determined by the community and are heavily influenced by the Ethereum Foundation. Gas prices can change over time, so that the very same transaction can cost more or less gas depending on the time when it is committed.

**3. The desired speed of which transactions are executed:** How much a gas costs is dependant on the users of the Ethereum network and unlike the previously mentioned operation price table, the gas-to-ether conversion rate is not hardcoded into the Ethereum, but determined by the users dynamically.

The default configuration of go-ethereum [17], 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 or more. When the gas price is set to 20 shannon in a client that wants to send a transaction, the client will automatically offer that amount.

The average gas price chart from etherscan.io shows that the clients of the network in general go with the default configuration value, 1 gas equals to approximately 23 shannon on average.

The reason that the real average gas price is 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 advantage that the transaction likely will be fulfilled quicker.

The mechanics of mining are similar to those of a stock market: If a stock is worth 100 USD and is being actively traded on an exchange and a buyer is willing to buy the stock at an overvalued price of for example 102 USD, his offer will jump to the top of the order book and will be filled the quickest. For our application, it could be desirable to pay a premium for faster transaction mining.

The default price of gas is debated in the Ethereum community and was already changed once in March 2016 when the price for Ether soared. The gas price was then reduced from 50 shannon to 20 shannon. Now that the Ether price has reached a different magnitude, another correction might be included in the next hard fork. At the time of writing, the gas price equilibrium would be 16 shannon according to the calculation from etherchain.org.

On May 7th 2017, ethgasstation.info announced [18] that 10% of the network hashpower is accepting a gas price of only 2 shannon. The site is promoting lower gas prices and is encouraging Ethereum users to change the settings to allow lower gas prices. According to the same site, a transaction that offers 2 shannon per gas, the mean transaction confirmation time is 119 seconds. For 20 shannon and 28 shannon, the transaction confirmation time is 44 and 30 seconds respectively.

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

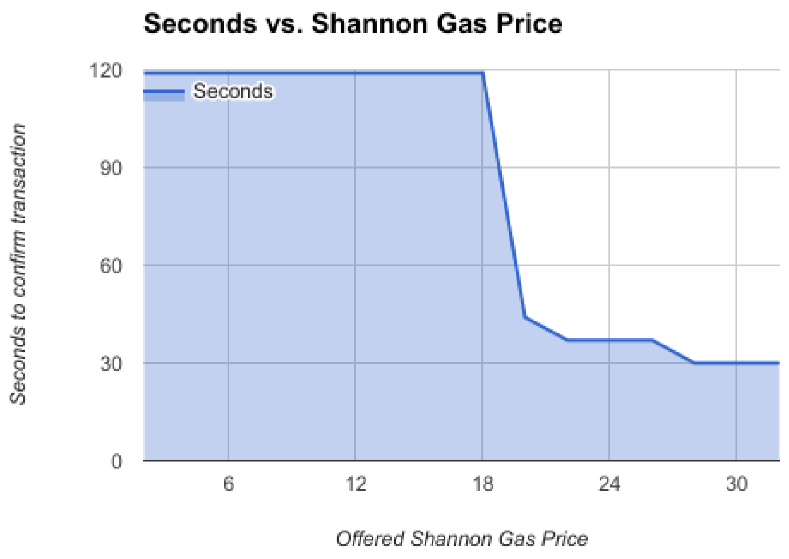

Figure 5.1: Average time until transaction is confirmed. Made with data from ethgasstation.info on May 8th.

In Figure 5.1, it can be seen that a node that offers a higher reward for a transaction will get confirmed 4 times faster on average. The user has to decide based on that information, how much he wants to pay.

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

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

```
pragma solidity 0.4.8;

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

    address public owner;
    Report[] public reports;

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

| **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 5.1: 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 5.2: Gas estimate deviations of code changes

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

The gas estimate was 318’552 gas when compiled with Version 0.4.8 of the Solidity Compiler (solc), but slightly different in other compilers (Table 5.1).

Even when removing just whitespace from the contract, the gas cost can, but does not have to change. On the other hand, changing variable names does not result in an estimate change (Table 5.2).

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

## 5.2 Cost model

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

The corresponding fees are $f = (f_1, ..., f_n), \forall f_i \in G$, where $G$ is the fee schedule defined in the Ethereum Yellow paper. The amount of gas a transaction consumes is shown in Equation 5.1:

$$C = \sum_{i=1}^{n} \sigma_i \cdot f_i \tag{5.1}$$

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 (set to be raised to 5'500'000 in the next fork [16]). At the moment, this means that the condition in Equation 5.2 is always fulfilled.

$$C = [21000, 3141592] \tag{5.2}$$

The price is denoted as the variable $\alpha$. Since according to ethgasstation.info, there are 3 possible speeds, three constants are defined representing the minimum cost to pay in order to get that speed (Equations 5.3 - 5.5).

$$\alpha_{cheap} = 2 \cdot 10^{-9} Ether \tag{5.3}$$

$$\alpha_{average} = 20 \cdot 10^{-9} Ether \tag{5.4}$$

$$\alpha_{fast} = 28 \cdot 10^{-9} Ether \tag{5.5}$$

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 5.6).

$$C \cdot \alpha \cdot ETH \tag{5.6}$$

For example, a contract that costs 500'000 gas to deploy and is priced at $\alpha = \alpha_{average}$ per gas, with the price for Ether being \$50, the cost to deploy that contract is \$0.025 (Equation 5.7).

$$500'000 \cdot 10^{-9} \cdot \$50 = \$0.025 \tag{5.7}$$

## 5.3 Cost estimation for DDoS mitigation

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

For a contract that stores IP addresses directly, the gas estimate was $Deploy\ cost = 700113\ gas$ . The cost of adding 1 IP in a transaction is 104046 gas. When adding 2 IP addresses in the same transaction, 1 transaction gas cost of 21000 is saved among other minor savings. The maximum is 46 IP addresses, after that the block gas limit is reached (which, theoretically, users of the network can increase) and another transaction has to be done. In our case, the cost is $Add\ IP\ Cost = 66322$ per IP and $Transaction\ cost = 22660$ per transaction. The price to store $x$ IP addresses is in Equation 5.8.

$$c = \mathit{Deploy\ cost} + (x \cdot \mathit{Add\ IP\ Cost}) + \lceil \frac{x}{2} \rceil \cdot \mathit{Transaction\ cost}) \tag{5.8}$$

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

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

Even though the average gas price so far is approximately the default value set by clients, it seems realistic that users over time react and adjust the gas price according to the Ether price. If ether is higher, the gas price should be lower.

Using $\alpha = 0.16$ as the gas price (the value that etherchain.org deems as the equilibrium as of writing) and an Ether price of $ETH = \$90$ (the price at the same time), the result is a transaction cost of \$1.96.

For a contract that stores a pointer to a web resource containing IP addresses, there is a flat cost whenever the list is updated. The *Deploy cost* still applies. The transaction cost for adding a URL with 64 characters was $\mathit{Transaction\ cost} = 121466$ in our test. This gives Equation 5.9.

$$C = \mathit{Deploy\ cost} + \#\ \mathit{of\ transactions} \cdot \mathit{Transaction\ cost} \tag{5.9}$$

# Chapter 6

## ...

# Chapter 7

# Evaluation

# Chapter 8

# Summary and Conclusions

# Bibliography

[1] *Ethereum: Blockchain App Platform.* URL: `https://ethereum.org/`.

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

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

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

[5] K. Nishizuka et al. *Inter-organization cooperative DDoS protection mechanism.* URL: `https://tools.ietf.org/html/draft-nishizuka-dots-inter-domain-mechanism-02` (visited on 12/27/2016).

[6] B. Rodriguez et al. "A Blockchain-based Architecture for Collaborative DDoS Mitigation with Smart Contracts and SDN". In: (2017).

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

[8] *The Solidity Contract-Oriented Programming Language.* URL: `https://github.com/ethereum/solidity`.

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

[10] Alex Beregszaszi. *Ethereum Improvement Proposal 74: Support RSA signature verification.* URL: `https://github.com/ethereum/EIPs/issues/74`.

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

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

[13] *Truffle Framework.* URL: `http://truffleframework.com/`.

[14] *Solium Framework.* URL: `https://github.com/duaraghav8/Solium`.

[15] Vitalik Buterin. *Thinking about Smart contract security.* 2016. URL: https://blog.ethereum.org/2016/06/19/thinking-smart-contract-security (visited on 05/18/2017).

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

[17] *go-ethereum Github Repository. File `eth/config.go`.* URL: https://github.com/ethereum/go-ethereum/blob/fff16169c64a83d57d2eed35b6a2e33248c7d5eb/eth/config.go%5C#L45.

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

# Bibliography

# Abbreviations

| | |
|---|---|
| CA | Certificate Authority |
| DoS | Denial of Service |
| DDoS | Distributed Denial of Service |
| IP | Internet Protocol |
| ETH | Ether (currency of Ethereum) |
| solc | Solidity compiler |

# 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