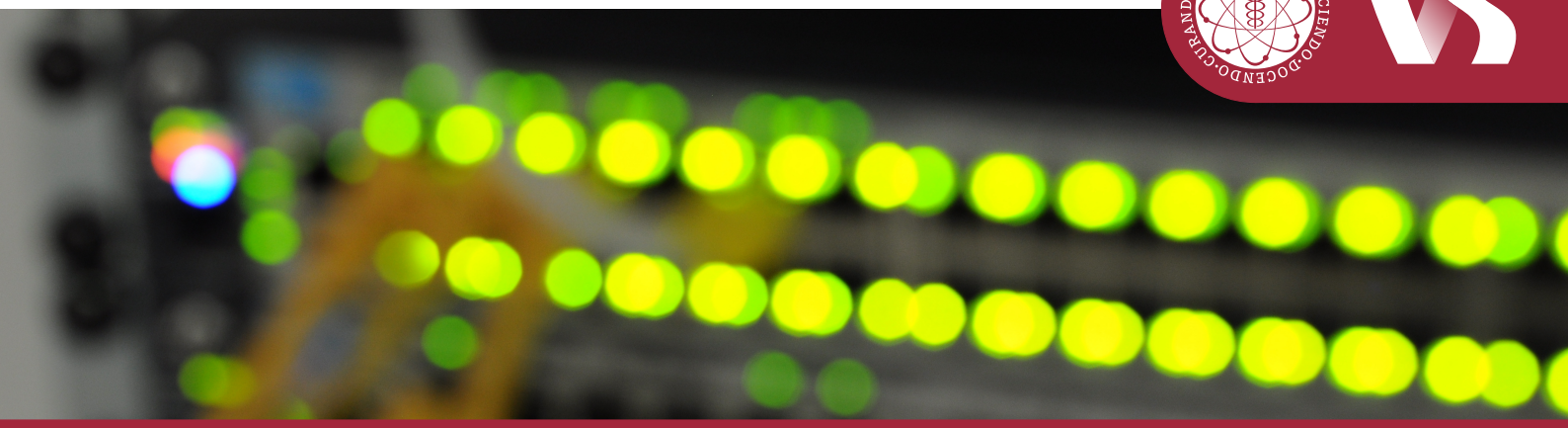

State of the Art in Cryptocurrency Network Simulation

Niklas Haas
Bachelor's Thesis

Supervisor: David Mödinger, Henning Kopp
Examiner: Franz Hauck
VS Number: VS-B24-2017
Submission Date: 3.8.2017

Issued: 3.8.2017

I hereby declare that this thesis titled:

State of the Art in Cryptocurrency Network Simulation

is the product of my own independent work and that I have used no other sources and materials than those specified. The passages taken from other works, either verbatim or paraphrased in the spirit of the original quote, are identified in each individual case by indicating the source.
I further declare that all my academic work has been written in line with the principles of proper academic research according to the official “Satzung der Universität Ulm zur Sicherung guter wissenschaftlicher Praxis” (University Statute for the Safeguarding of Proper Academic Practice).

Ulm, 3.8.2017

Niklas Haas, student number 840177

Abstract

Cryptocurrencies including Bitcoin are the focus of ongoing research, with many practical implementation questions (such as the optimal block size) left unanswered. There exist a number of Bitcoin simulators that simulate various aspects of the network in order to help answer these questions and design new cryptocurrencies.

We perform a comparative study of three simulators (ns-3, Shadow and simbit) and evaluate their user friendliness and ease of use, performance and scaling characteristics, adaptability to other cryptocurrencies and the range of parameters which they can simulate. We present these findings in the form of a detailed description for each simulator as well as a tabular overview. In addition to this, we perform a comparison of the simulation performance and provide the results in the form of a graph.

Contents

1 Introduction

1.1 Motivation

Over the last few years, peer-to-peer cryptocurrencies have gained considerable momentum and interest, with the originator Bitcoin [34] being followed by a plethora of different cryptocurrencies inspired by it. Cryptocurrencies are digital currencies on top of peer-to-peer networks, and in particular share the general characteristic of using a blockchain to publish and synchronize immutable shared records. Aside from simple electronic payments, cryptocurrencies have been adapted to many novel use cases such as decentralized domain name registration and HTTPS key pinning [15], fair distributed backups and file storage [39], turing complete smart contracts [40], censor-resistant message delivery [38] or zero-knowledge transactions [35]. In addition to extending the capabilities of the network, the core protocol has been subject to alterations. example by replacing proof-of-work with proof-of-stake systems. [28]

Within the context of blockchains, there are many parameters which influence the behavior and characteristics of a network, including the number of peers, the block rate, the block size, the transaction size, the transaction fee (where applicable), the number and computational capacity of bad actors, the peer join/leave rate, the average peer bandwidth, the average peer storage capacity, and so on. When developing peer-to-peer networks based on blockchains, it can therefore be of value to be able to simulate the network and evaluate its performance based on adjusting these variables. In doing so, potential problems, inefficiencies or attack vectors can be revealed, and the network can be engineered to respond better to typical real-world use cases or malicious actors. Selection of these parameters are part of ongoing research and disputes, including multiple current attempts to fork the Bitcoin blockchain in order to change variables such as the block size. [3, 25]

To aid with the selection of these parameters, as well as test fundamental properties of the network such as its resistance to bad actors and denial-of-service attacks, there have been a number of simulation tools developed that focus on Bitcoin. However, in practice, we find a lack of consensus or information comparing the different simulation

tools and evaluating their capabilities, strengths and weaknesses. The goal of this thesis is to serve as an evaluation of the most prominent simulators, in order to make it easier for readers to choose the right simulator based on what properties they are trying to examine.

1.2 Background

Bitcoin is a *peer-to-peer consensus network* - meaning it lacks a central authority - which synchronizes to form what's known as a *blockchain*. A blockchain is an linked list of blocks. Each block references the previous block as part of its header, and contains a list of *transactions* moving virtual *coins* from one address to another. These transactions and virtual coins are what enables the Bitcoin blockchain to act as a digital currency. The number of these transactions per block is bound by the *block size*. [34]

To prevent abuse such as trying to spend the same coin twice, all transactions contained in new blocks are validated by each peer in the network, and forming new blocks is intentionally made difficult. For Bitcoin, this block creation difficulty comes from a concept known as *proof-of-work*, in which generating a new block requires brute forcing a cryptographic hash, the contents of which depend on the previous block. The result of this is that each new block takes a more or less consistent amount of computational work to generate, with the amount of work that needs to be performed (the block *difficulty*) being continuously adjusted using what's known as a *difficulty adjustment function* in order to ensure that the average frequency of blocks hits a constant target, the *block frequency*. This entire process of forming new blocks is what's known as *mining*, and the computational work required is compensated by awarding a *block reward* to the address of the peer (the *miner*) that managed to find the block. [34]

Research into Bitcoin alternatives and successors is ongoing, partly due to the fact that a blockchain serves as a convenient synchronized ledger for any sort of fact storage which is difficult to alter or censor, partly because it allows an elegant integration of monetary costs and rewards into peer-to-peer protocols, and partly because of perceived weaknesses of the Bitcoin protocol such as its reliance on the computationally wasteful proof-of-work scheme as the source of its robustness. One of the proposed alternatives is known as *proof-of-stake*, in which new blocks are randomly granted to individual addresses using a deterministic process, based on the amount of wealth that address currently has available to spend. In essence, it rewards those who put large amounts of money into the system (i.e. peers with a big *stake* in the network). [28] That being said, proof-of-stake is currently an unproven technology that has yet to establish itself as working reliably in practice.

1.3 Research Questions

In order to evaluate the strengths and limitations of each simulator, we ask several questions which we will attempt to answer for each studied simulator in a reasonably objective manner. Of particular interest is the ability to adapt the simulator to alternative and domain-specific cryptocurrency, which generally go beyond simply being a Bitcoin clone with trivial changes. The questions were chosen through a mixture of thinking about the useful properties a Bitcoin simulator could have, as well as trying to come up with ways to compare them objectively rather than subjectively. Some questions were taken from [29], but the rest are original work.

1.3.1 Code metrics and user interface

The goal of these questions is to evaluate how difficult it is to get each particular simulator up and running, but also establish metrics for how well-written the simulator's code is. As the testing environment, we are using *Gentoo Hardened unstable*, with up-to-date versions of all involved compilers and dependencies.

- Does the software compile as-is on a modern machine? If not, how difficult is it to modify the software until it compiles? [29]
- Is it portable, and/or are the dependencies easily satisfied?
- How ergonomic is the interface with which the simulator can be programmed and the results presented? How much code needs to be written to automate running multiple tests? [29]
- Does making small changes to the desired network behavior require invasive modifications to the code? How difficult is it to control the number and behavior of peers?
- How large is the codebase, and what is the quality of its documentation? [29]

1.3.2 Adaptability

The goal of this section is to evaluate how difficult it would be to adapt it to alternative or novel cryptocurrencies, rather than simulating the existing Bitcoin network.

- Does the software depend on the official Bitcoin reference implementation[1]?
- How difficult is it to extend the set of transaction types and associated user behavior? How many entrypoints in the code does each transaction have?
- Is it possible to embed the concept of peer-to-peer file storage and associated resources (disk space, bandwidth) into the simulation?
- Is it possible to simulate proof of stake? Is it possible to modify the difficulty adjustment function?

1.3.3 Parametrization

The goal of this section is to evaluate how flexible a simulator is, which determines what kind of real-world questions it could and couldn't answer.

- How flexible is the range of parametrization? What can and can't be varied during the course of a simulation? What assumptions are hard-coded?
- Can the simulator perform automatic optimization of these variables? (For example finding a local or global optimum of multiple variables)
- Can the simulator handle varying parameters (like the average peer bandwidth or the number of bad actors) over the course of a simulation, or is the network entirely static?
- Can the simulator handle joining and leaving nodes?

[1] `bitcoind`, also known as *Bitcoin Core*

1.3.4 Performance

The goal of these questions is to get an approximation of how well a simulator performs, which determines the size and complexity of networks that can be simulated, but also the rate at which blocks can be simulated - which is important for cryptocurrencies involving variable mining parameters. As a test platform for this section, we are using a 2x8-core Xeon E5-2670 machine with 64 GiB of DDR2-1333 RAM.

- What network sizes can the simulator reasonably handle?
- How well does the simulation scale with additional hardware? Can it run on multiple processors and/or machines?
- How much faster than real-time is the simulation on tested hardware?
- Is it capable of nulling out the computational expense of proof-of-work blockchains?

2 Simulators

In this thesis we will be taking a look at three simulators in depth. Two of them, Shadow [27] and ns-3 [24], are generic network simulation programs that have available Bitcoin-specific plugins. The third, simbit [2], is a purpose-built simulator for Bitcoin written in JavaScript. ns-3 and Shadow were chosen for their sheer size and development activity. simbit was chosen because it offers a high amount of flexibility and adaptability, which we felt was unique among the Bitcoin simulators we found.

A fourth simulator, btcsimulator [33], was not included on the grounds of being unable to get it to work properly, and the code itself breaking if one tried modifying e.g. the number of clients in the simulation. It also had a very over-engineered design involving a number of separate build systems. Additionally, it was designed to be used exclusively via a web interface, which we found to exhibit communication failures with the python backend. There was no developer activity [37], no response to our issues on GitHub [23], issues with performance, nonexistence of documentation, and an inspection of the code revealed no interesting features that none of the other three simulators provided.

2.1 Shadow

Shadow [27] is a simulator framework based on the design of "hosting" existing applications, with minor modifications, into a simulation. This is done by stubbing out key system calls such as `epoll`[1] in order to feed it with fake I/O events. Shadow was originally designed as part of a research project to simulate Tor, but it can be adapted to other software by writing a plugin. [27] A plugin which wraps `bitcoind` is already available, forming the basis of our testing. [30, 32] This plugin has been used in the past to demonstrate denial-of-service attacks in the reference Bitcoin software. [31, 32]

[1] `epoll` is a Linux kernel API that forms the basis of most modern asynchronous I/O operations such as waiting for network packets.

2.1.1 Installation

Shadow's documentation consists of a handful of wiki pages documenting the installation, usage and customization processes. [14] We found Shadow to be difficult to install in practice, due to a combination of unusual cmake usage and non-portable build scripts. That said, we managed to overcome these challenges and get Shadow built, in both its `master` and `v1.10.2` branches, although the latter required cherry-picking some fixes.

On top of this, we had immense difficulty building the Shadow Bitcoin plugin. The build process is documented on a wiki page [14], but following them resulted in several hundred build errors; ranging from usage of outdated and no longer existing library functions to internal C++ errors such as references to ambiguous types. Fixing the former issue requires downgrading several system libraries, including Shadow and `bitcoind` themselves, to very specific, older versions - which by itself created some build difficulties. In addition, several fixes were made to patch the source code into a working state, but even after manually resolving all of the C++ build errors, we ran into LLVM internal errors that prevented it from linking correctly on our test platform. Multiple upstream bug reports exist, and at least one of them has been inexplicably closed unanswered by the author despite the bug persisting in the code. [16] Despite these difficulties, we were able to successfully build and run the Shadow Bitcoin plugin inside a virtual machine running Ubuntu 14.04, which was the newest version of Ubuntu that did not exhibit the LLVM linking errors.

2.1.2 Code Metrics

The development of Shadow is somewhat active. The project uses git, and at the time of writing, the most recent commit was 7 hours ago, but it has only received a few dozen commits in the past year. [13] The overall codebase excluding dependencies is about 25k SLOCs[2] of mostly C with some amounts of C++. The code is styled according to a variant of K&R style, and this style is used consistently throughout. Comments are used sparingly, with particularly difficult or pitfall-laden parts of the code being briefly commented, but there is generally no documentation of the API of internal functions. That said, the code is fairly comprehensible.

The Shadow Bitcoin plugin consists of around 2k SLOCs of C, with some C++ wrappers, and it was reasonably understandable and well-documented. It follows the same coding conventions as Shadow. It has not been updated to the latest version of Shadow

[2] Source Lines of Code, which excludes blank or commented lines

(v1.11.2), and has not received any commits in the past 2 years, although the maintainer still occasionally replies to issues.

2.1.3 User Interface

Shadow is designed to be used primarily via the command line. Configuration of the simulation network is done via an XML file which describes the network's nodes and edges alongside their attributes such as packet loss, bandwidth, latency or jitter. The documentation includes examples and explanations of the configuration format, and the Bitcoin plugin includes a simplified example XML which only includes a single region (marked "US") and does not otherwise attempt to approximate the real Bitcoin network, although this would be possible in principle. Shadow also comes with a set of python tools to ease the creation and modification of these XML files, although the documentation for this is incomplete or missing. [14] To perform simulations on more complicated networks, the XML file describing the entire network, including all connections, must be generated independently and passed to Shadow. The authors of the Shadow Bitcoin plugin used a special script for this purpose, but this script has not (yet) been made public. [26]

Any output messages generated internally by Shadow or the plugin during the simulation will be appended to a text log file, containing a timestamp (real and simulated), pseudo-hostname, function name and message body. This includes information about the TCP/IP stack as well as the process's external metrics (CPU time and memory usage). In addition to this, the `stdout` of each simulated process is dumped into a separate log file as-is. By default, the Bitcoin plugin installs a hook that wraps `bitcoind`'s internal log callback with Shadow's logging functions. Nonetheless, to gather blockchain-specific statistics of interest that occur as part of the simulation, `bitcoind` itself must be modified to log these. [14]

Shadow includes some helper scripts that can parse the Shadow output messages and chart them in various ways, but it only recognizes the messages built into Shadow - meaning it will not be of much use when trying to plot custom user statistics, unless modified to do so. [14]

2.1.4 Adaptability

Shadow is inherently independent of Bitcoin and can work with a wide range of peer-to-peer software, due to its design of merely simulating the OS network stack and physical network while running the program code unmodified. This makes porting new applications to Shadow considerably easier than writing all of the simulation logic from scratch. Unfortunately, for the use case of simulating Bitcoin adaptations, this requires making those modifications on the `bitcoind` codebase - which involves a considerable amount more work than simply writing a "dry" simulator, if the protocol in question has not yet been implemented. On the other hand, this makes it easy to test `bitcoind` forks that have already been written.

Since Shadow does not stub out file I/O at all, this means that the simulated Bitcoin nodes actually store their data on disk. For efficiency, the plugin can be configured such that the Bitcoin nodes share the common parts of the blockchain, owing to the internal design of `bitcoind`, while only the new blocks and the database storing indices into the blockchain must be kept separate. [32] The authors claim to have successfully tested a network involving 6000 nodes each storing a full copy of the real blockchain (which was around 100 GB in size at the time) while only using 300 GB of storage overall. [32]

Finally, Shadow has no built-in model of client behaviors, so simulating an attack on the Bitcoin network would require writing a real proof of concept attack program and linking it into the plugin (in addition to `bitcoind`). This is also required to simulate anything involving block mining or transaction generation, since the Shadow Bitcoin nodes themselves do nothing except sending keep-alive pings. The Shadow Bitcoin plugin comes with some examples of how this can be accomplished, but we found them mostly unusable: they relied on hard-coded paths on the author's machine, uncommitted modifications, and unpublished scripts for their basic operation. [30]

While we've made the claim that simulating a new cryptocurrency requires modifying `bitcoind`, this is not actually true, if one is willing to discard the Shadow Bitcoin plugin. A high-level simulation *could* be written as a simple program that passes a mock protocol over the wire, and given to Shadow so it can run many instances with a simulated network topology. Shadow shouldn't care whether the plugin it's running is a real world program or a "fake" test program. This would essentially give it much of the same power as a more abstract simulator like ns3, in that the simulation and protocol can be simplified to the point where adding a new transaction is easy. It also allows the ultimate in language flexibility, since all Shadow cares about is whether or not the simulated program uses the right I/O syscalls. So in principle, the simulation could be written in, say, Haskell; although we have not verified whether this actually works.

2.1.5 Parametrization

Shadow does not allow changing the network configuration or topology at runtime, so simulations with e.g. a changing hash rate, or adding and removing nodes is not directly possible. That said, it would be possible to modify `bitcoind` to support nodes that do nothing until a certain block age. Outside of existing `bitcoind` runtime arguments, the Shadow Bitcoin plugin does not support a whole lot of customization; there's no simple way to, for example, adjust the block rate or block size; except by actually modifying the relevant constants in the `bitcoind` code. The only thing Shadow can directly influence as a result of the test configuration are the physical features of the network: bandwidth, jitter, packet loss and so on, but also the CPU power associated with each node - which means that simulations involving big and small miners are inherently supported. Finally, Shadow does not perform automatic optimization of any kind, although based on the lack of support for directly modifying the blockchain parameters this is understandable.

2.1.6 Performance

Shadow is capable of simulating faster than real-time (despite the fact that it ostensibly runs black box software), limited only by processing speed. It does this by intercepting sleep calls and instead advancing the virtual time if all nodes are sleeping. Due to the difficulty of generating a working example of a large Bitcoin network, [26] we were unable to give exact performance numbers - but for the example network consisting of two nodes, the simulation performed at 105x realtime. The authors claimed the ability to simulate networks involving as many as 6000 nodes in practice. [32] In order to help run simulations involving actual block generation, the version of `bitcoind` used is patched to disable verification of blocks, thus allowing clients to bypass proof-of-work. [30]

2.2 ns-3

ns-3 is a simulator framework intended to help users write purpose-built simulators by calling library functions that it provides. It's the third version of the *ns* family of discrete event network simulators, and it was developed for the purpose of research and teaching, with funding from the U.S. National Science Foundation. [24] A Bitcoin-specific plugin for ns-3 is already available, which is what we will be focusing our testing on. [18]

2.2.1 Installation

ns-3 comes with extensive documentation spanning several hundred pages [7] as well as an online wiki containing a similar number of articles. [12] The build instructions were detailed and following them was straightforward. We successfully built the project unmodified. What's worth pointing out about ns-3, however, is that the build process for ns-3 plugins (such as the Bitcoin plugin we were testing) is a bit unusual: Plugins are not built separately, but they must be copied into the ns-3 source code and built alongside it. That being said, this plugin-specific installation procedure was also documented and the instructions as presented worked without major deviations. The ns-3 Bitcoin plugin also included a helper shell script to automate this process. The build system is based on *waf*, a build framework written in Python.

2.2.2 Code Metrics

ns-3 uses mercurial. [4] The development of ns-3 is very active: At the time of writing, the most recent change was 15 hours ago, and it has received several hundred commits in the past year. [4] The overall codebase excluding dependencies is approximately 400k SLOCs of mostly C++, with some python helper scripts here and there. The code is styled according to GNU standards, which seems to be consistently applied throughout. Particularly difficult or pitfall-laden sections of the code appear to be commented, as well as all public functions using the Doxygen syntax convention, but apart from that most code is relatively uncommented. That being said, we had no issues understanding ns-3 internal code.

The ns-3 Bitcoin plugin consists of around 8k SLOCs of C++, formatted in what appears to be Allman style. The last non-documentation commits were over a year ago. It has not been updated for the latest version of ns-3 (3.26). It uses Git, but doesn't follow any of the usual git conventions - and instead routinely commits commented out code, "work in progress" code, and dead code. [17] The code is collected into few, large files - each with several thousand lines of code, and logic is mostly clumped into a few big functions containing anywhere from hundreds to over a thousand lines of code. Comments appear strewn throughout, but even with them we found it difficult to understand the code mostly due to the deeply nested switch statements and lack of abstraction into components.

2.2.3 User Interface

ns-3 is designed as an extensible, general-purpose network simulator, and all functionality is organized into individual libraries that users can either choose to use or not. As such, the primary interface to ns-3 is in the form of a program which depends on it. This program will invoke ns-3 helper functions to create nodes, configure their topology, assign them properties and behaviors, and ultimately run the simulation while printing interesting metrics. ns-3 provides about a hundred examples of such programs [5], and the ns3 Bitcoin plugin includes one which sets up a topology seemingly modeled after the real Bitcoin network. [20, 21] These *scripts* are then built and executed by the ns-3 build framework, via `./waf --run <scriptname>`. [9]

There are at least three ways in which ns-3 scripts can produce output for analysis: The tracing framework, the logging framework and at the end of the simulation. The tracing framework is designed to present a flexible network of *sources* (of events), and *sinks* (consumers of events) which can be connected to handle events one is interested in certain ways - for example by attaching a logging sink to the "packet received" event in order to print out interesting bits of the packet, or otherwise increment a counter or similar. [11] The logging framework is designed to be structured in a hierarchical way that lets users quickly enable or disable logging of components, as well as set their log level on a per-component basis. Out of the box, pretty much everything in ns-3 can be extremely verbose if one wishes it to be, by setting that component's log level to `LOG_ALL` - which will log everything from individual function calls to the results of switch statements or other control flow within a function. [10] Finally, the script can perform detailed inspection of the final state of the network after the simulation halts, which the ns-3 Bitcoin plugin uses to print overall stats like the average block size and the block propagation time histogram. [19] ns-3 also includes libraries to help users take various gathered statistics and turn them into *gnuplot* graphs.

2.2.4 Adaptability

ns-3 is inherently independent of Bitcoin and can simulate any sort of conceivable network protocol, at the cost of the user having to specify the application logic for any simulated protocol.[3] Writing a simulated model of a protocol is generally much easier than

[3] Incidentally, ns-3 also supports a mode called *DCE* (Direct Code Execution) which aims to wrap real applications inside ns-3 in order to simulate their logic directly, similar to Shadow, but this seems to be somewhat obscure and less-used so we mention it only in passing. Its use also defeats some of ns-3's advantages, such as the ability to test protocol modifications without fully implementing them in `bitcoind`.

actually implementing it, because one can make assumptions about the clients (due to controlling them all), and stub out parts of the logic that are not relevant to the simulation. The simulated protocol can also be limited to a subset of the full protocol, focusing only on the behavior one is interested in.

To simulate protocols, ns-3 simulates the network stack to the level of individual packets, which allows the most flexibility in terms of simulating things like TCP overhead due to packet loss, jitter, latency and other sources of real-world headaches. ns-3 can simulate Ethernet/CSMA, WiFi, IPv4+ARP, IPv6+ICMPv6, TCP, UDP and others. [6, 8] That being said, applications don't have to implement the real wire protocol they are simulating. The ns-3 Bitcoin plugin, for example, passes all messages in a pseudo-protocol based on JSON, thus making it easier to parse. One could also directly serialize a constant-sized `struct`, which reduces parsing to a pointer cast.

The plugin in particular supports has built-in tables of region-specific hash rates and node counts for Bitcoin, Litecoin and Dogecoin. It also supports some existing, novel (at the time) technologies such as SPV[4] and BlockTorrent[5]. The definition of the protocol itself is largely contained inside a single function, which is essentially a single 1500-line switch statement containing the implementation of every protocol message. In practice, we found reading, understanding and extending it to be very difficult due to high levels of nesting, lack of abstraction and mixture of business logic with parsing logic.

It's also worth pointing out that the bitcoin plugin does not implement the full Bitcoin protocol - it restricts itself to the subset of messages concerning itself with block generation and transmission. There is no implementation whatsoever, for example, for individual transactions. (For statistics, the simulation simply assumes the number of transactions is equal to the block size divided by the average transaction size). This means that adding support for the simulation of new transaction types would depend on first implementing support for sending transactions at all.

There is no built-in support for proof-of-stake whatsoever, so simulation of this would have to be implemented from scratch. On the other hand, the ns-3 Bitcoin plugin does implement some rudimentary support for naive proof-of-work attackers, e.g. a double-spending attacker that tries to fork the blockchain (with a configurable hash rate).

[4] Simple Payment Verification (SPV) is a technique for running lightweight Bitcoin clients that can verify transactions without storing an entire copy of the blockchain. [34]

[5] BlockTorrent combines Bitcoin with techniques from the BitTorrent protocol to speed up the propagation of new blocks. [36]

2.2.5 Parametrization

The Bitcoin plugin supports a handful of parameters which are designed to be configurable at runtime as command line arguments. These include configuration of the overall network (number of nodes/miners) as well as some fixed static assumptions (block size, block timeout, block rate) and associated protocol usage patterns (blocktorrent, block broadcast type). However, these parameters are designed to be constant throughout the course of a simulation. There's no built-in ability to adjust the block rate as a function of the block age, for example. Such logic would have to be hard-coded into the simulation, e.g. by modifying the difficulty adjustment function in the Bitcoin miner definition.

ns3 supports no built-in ability to perform automatic optimization of runtime configurable parameters, so any such optimization would have to be done using additional client software. ns3 also does not inherently support nodes joining and leaving the network as part of the simulation, but it would be technically possible to have "conditional nodes" which are initially part of the network but only enable their functionality past a certain block age, to simulate the effect of joining and leaving nodes to some degree. This would, however, also have to be done by modifying the simulation source code.

2.2.6 Performance

The Bitcoin simulation does not actually perform proof of work; blocks are instead generated by "sleeping" for a certain amount of time (calculated based on the current block difficulty and hash rate). This is done by using the ns3 simulator's built-in scheduling capabilities, which can schedule events on a simulated real-world timeframe (but which may actually run at several times normal speed). [22]

ns3 supports distributed computing based on the MPI standard, which the simulation can take advantage of. The ns-3 Bitcoin plugin supports this out of the box, and in practice we were able to scale the simulation up to all 32 cores of the test machine. In principle, MPI also allows horizontal scaling across machines, but this was not tested. In practice, we were able to obtain a 17x speedup over real-time on a network consisting of 10,000 nodes and 16 miners, with an average of 5–10 connections per node, and a 48x speedup over real-time on a network with 1,000 nodes. This makes ns-3 very viable for simulations involving thousands of nodes.

2.3 simbit

simbit [2] is a standalone simulator written in JavaScript which was specifically written for the purpose of simulating consensus networks including Bitcoin. It is designed for both simulation and visualization (by running inside the browser). [2] simbit appears to be a hobby project by a single developer, with no academic publication tied to its creation.

2.3.1 Installation

Despite running in the browser unmodified, simbit comes with a few dependencies when running offline (via Node.js). One of these dependencies relied on a deprecated API call that was removed in Node 0.12, and therefore did not work out of the box on newer machines. That said, patching this was a trivial one-line change, and we have submitted the fix upstream.

2.3.2 Code Metrics

The project uses git, but has not seen any activity within the past year. The overall codebase excluding dependencies is about 1500 SLOCs of JavaScript. The code is consistently styled, and contains a reasonable amount of internal comments. The project comes with a README file detailing the usage of the simulator, the API, and a handful of examples providing a guideline on how to simulate different scenarios including a selfish mining attack, and a simulation that measures the broadcast latency within the network. [1]

Being written in JavaScript, simbit makes heavy use of a callback-oriented programming style, which we found somewhat difficult to follow in places. That said, due to the relatively small size of the project, it was still possible to quickly establish an overview of the architecture and work from there.

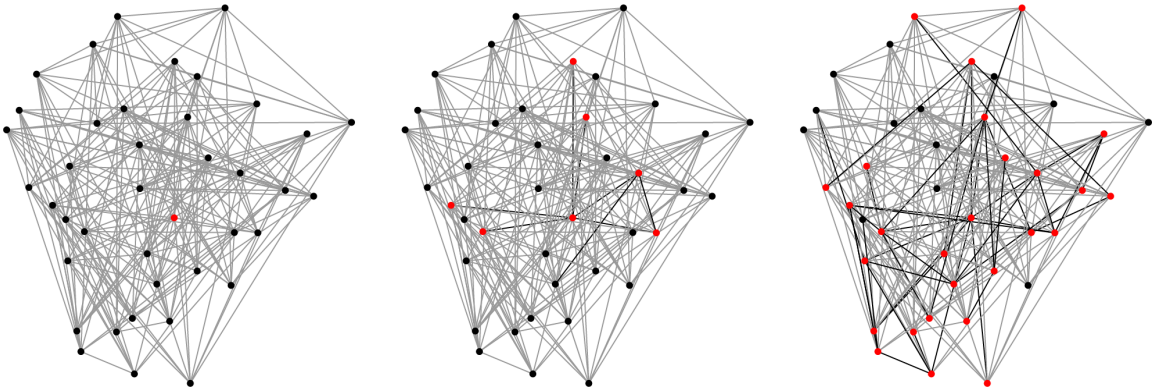

Figure 2.1: Example of the simbit network visualization. This is a series of screenshots of a running simulation illustrating broadcast latency. The nodes colored in red have already received the broadcast, and are in the process of sending it to their neighbours. The connections with current activity are highlighted in black.

2.3.3 User Interface

The simulator is presented as a JavaScript library which can be included in a standalone program (such as the provided `sim.js`) and used to run simulations. There is no concept of command-line parameters or other sources of configuration except for the contents of the JavaScript program using it. As such, modifications to the network, hash rate, or number of peers require adjusting the source code; while automated experiments would require programming the automation into the JavaScript file.

In terms of output, the code has some diagnostic `console.log` calls strewn throughout the implementation, and users are encouraged to do the same for their own statistics. Apart from this, the library comes with a built-in network visualization (see Figure 2.1), which users can interact with by programmatically changing the color of nodes.

2.3.4 Adaptability

simbit is purpose-built for Bitcoin, but is technically organized in a way that would allow the more abstract components to be reused for other consensus networks. Fundamentally, it provides abstractions for peer-to-peer networks and swarm consensus, and then implements Bitcoin as an extension of this interface. Modifying it to support other peer-to-peer networks, including any sort of Bitcoin variant, would be doable.

To simulate protocols, simbit uses a very high level approach involving events and callbacks. There's no simulation of a protocol per se, nor is there any concept of a network stack - but messages do arrive with a simulated delay based on the randomized distance between peers. There are no higher-level simulations of different regions, or of inter-regional and intra-regional latencies, although modifying the simulator to support this should be no more than a few lines of code.

There is no-built in support for alternative technologies like proof-of-stake, so the code would definitely have to be modified to simulate anything other than the naive Bitcoin network, but it does include a handful of simulations for attack scenarios including selfish mining, as well as a simulation of the broadcast propagation time.

Unlike both ns-3 and Shadow, simbit performs a full simulation of the blockchain including transactions (and transaction validation). This would make it a suitable host for any simulations involving interactions on the transaction level.

2.3.5 Parametrization

simbit has no runtime-configurable parameters at all, but a few numbers are easily modified in the simulation file. These include the block rate, the number of peers, the overall hash rate, and the simulation duration. Changing anything else would require more invasive changes to the code, although due to the high level nature and small codebase, we found it easy to quickly identify the right places to make such changes.

The simulator does not support any form of automatic optimization, although there's a proof-of-concept script that dispatches a bunch of simulators with slightly altered parameters on EC-2 nodes and collects the results.

simbit fully supports nodes joining and leaving the network at arbitrary points in time; and it is possible to interleave simulation ticks with network updates without resetting the blockchain, a feature which made it stand out among the alternatives. On the other hand, simbit has no model of regions, nor does it try distinguishing between mining peers and non-mining peers - every simbit node is a miner.

2.3.6 Performance

simbit is written entirely in single-threaded JavaScript, and does therefore not benefit from horizontal scaling of any kind, but it does stub out proof-of-work. In practice, we found that the performance decreases significantly with both the total number of nodes and the total length of the simulated blockchain. The networks we were able to simulate comfortably were only about 100 nodes large.

3 Evaluation

We have found that the different available Bitcoin simulators differ in both their design and their capabilities, which influences the choice of a Bitcoin simulation software depending on the requirements of the project. For a detailed list of differences between simulators, see Table 3.1. Shadow is mostly geared towards the simulation of the underlying network and software implementation, and will gather statistics related to network bandwidth, client processing time or propagation latency. It is useful for verifying that your software is resistant to floods or denial of service attacks, but is less useful for establishing the behavior of your blockchain. Adapting Shadow to new cryptocurrencies requires modifying `bitcoind` directly.

ns-3 is mainly focused on efficiently simulating a static network modeled after the real Bitcoin network. It can efficiently simulate large numbers of nodes, including simulated miners, and scales horizontally very well. It is useful for evaluating a network's resistance to mining attacks, but also gathering high-level network statistics such as block propagation times in large networks. As a downside, the ns-3 Bitcoin plugin is unable to simulate transactions. Adapting ns-3 to new cryptocurrencies requires modifying a C++ plugin.

simbit is mainly focused on allowing dynamic network behavior including transactions and joining/leaving nodes. It can be used to model per-transaction behavior, which may be important when developing domain-specific cryptocurrencies. As a downside, it is fairly slow compared to the alternatives, and does not parallelize. It also does not approximate the real Bitcoin network. Adapting simbit to new cryptocurrencies requires modifying its JavaScript source code.

Examined metric	**Shadow**	**ns-3**	**simbit**
Installation	Difficult	Easy	Easy
Dependencies easily satisfied?	No	Yes	Yes
General ease of use	Difficult	Easy	Easy
Language	C	C++	JavaScript
License	BSD-3	GPL-2	MIT
Lines of code (approximate)	27,000	408,000	1,500
Documentation	Moderate	Ample	Lacking
Depends on `bitcoind`	Yes	No	No
Can import real blockchain data	Yes	No	No
Protocol modifications	Difficult	Difficult	Easy
Simulates low-level protocol	Yes	No, but possible	No
Simulates proof-of-stake	No	No	No
Simulates block creation	Possible	Yes	Yes
Simulates transactions	Possible	No	Yes
Modification of parameters	Invasive	Simple	Simple
Automatic optimization	No	No	No
Simulates latency	Yes	Yes	Yes
Simulates regions	Possible	Yes	No
Simulates full network stack	Yes	Yes	No
Joining and leaving nodes	No	No	Possible
Typical network size	5,000–10,000 [27]	10,000–20,000[1]	500–1,000
Multi-core scaling	Yes	Yes	No
Multi-machine scaling	No	MPI	No
Nulls out proof-of-work	Yes	Yes	Yes

Table 3.1: An overview of the capabilities of each tested simulator. An entry marked as "Possible" means that the simulator design supports the feature, but it would take some additional work to enable.

3.1 Performance Comparison

Figure 3.1 illustrates the scaling and performance differences of ns-3 and simbit when generating blocks. Shadow was not included in this comparison due to the Shadow Bitcoin plugin's inability to simulate block creation without further modifications. The ns-3 tests were run using multi-threading across 32 threads, except for the smallest two networks, for which we used 8 and 16 threads respectively because there were not enough nodes to evenly distribute across 32 threads. All tests were run roughly 5 times and the

[1] Assuming scaling across multiple machines

results averaged.[2] Care needs to be taken when directly comparing ns-3 and simbit due to their different internal network design. In simbit, every node is a miner, but ns-3 distinguishes between miners and non-miners. If we try and make every ns-3 node a miner, we run into problems because ns-3 always fully connects miners. This leads to the number of simulated connections scaling quadratically with the number of nodes, which is unrealistic for large networks. Modifying this logic was intrusive and difficult, so we decided to instead cap the number of active miners at around 10% of the total network size. At that point the effect of the number of total nodes dominated, and the number of connections simulated overall was comparable to simbit networks of equal size. By doing so, we managed to normalize both networks to have an average of 10 connections per node, which we feel makes the results directly comparable.

An examination of the results reveals that ns-3 exhibits almost no speed loss as a result of increasing the simulation duration, whereas simbit seems to lose performance steadily as the number of mined blocks increases. In terms of the network scaling, both ns-3 and simbit scaled linearly with the number of nodes in the network. In theory, as the network size tends towards infinity, ns-3's performance should scale quadratically with the number of nodes - due to to its hard-coded behavior of fully connecting all miners with every other miner. In practice, however, we find this effect negligible for the tested network sizes. In summary, ns-3 is several orders of magnitude faster than simbit, due primarily to its strong support for horizontal scaling, implementation in C++ as opposed to JavaScript, and improved internal design.

[2] The longest tests were repeated less often due to the time required, down to a minimum of 3, and the shortest tests were repeated more often.

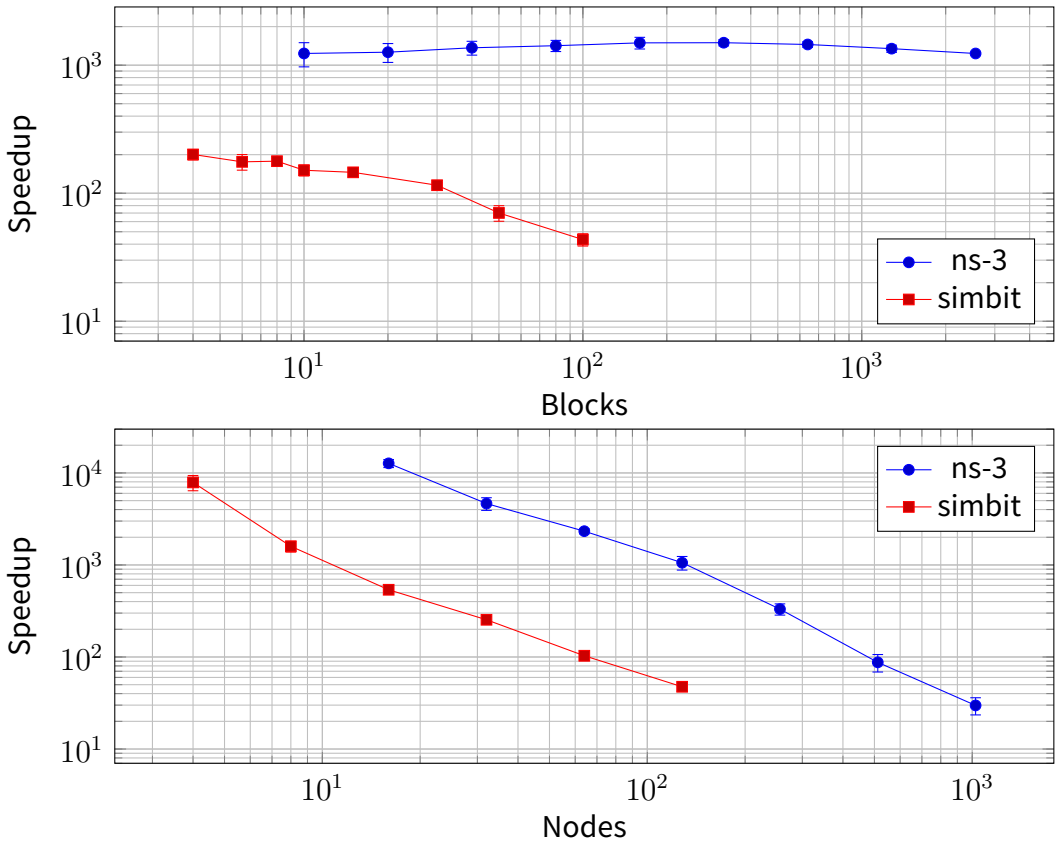

Figure 3.1: The top graph shows the speedup compared to real-time as function of the number of blocks mined, with the network size fixed to 100 nodes. The bottom graph shows the speedup compared to real-time as a function of the network size, with the simulation duration fixed to 50 blocks. Both graphs show averaged figures, with the error bars indicating the standard deviation.

4 Conclusion and Future Work

We find that all simulators examined have different areas of interest, strengths and weaknesses. Shadow excels at testing for attacks on real-world implementations of cryptocurrencies, but is difficult to set up and use. ns-3 offers high performance and many features, but the Bitcoin plugin is poorly written and very difficult to work with. simbit is high level and easy to quickly modify, and therefore suitable for prototyping new ideas, but it suffers from poor performance and a very simplistic network architecture. None of the tested simulators stood out as being obviously better than the rest, instead each showed their own problems.

Writing a simulator capable of both protocol flexibility and high scalability remains to be done. We also find that none of the simulators or simulation frameworks supported automatic optimization of parameters to find a global minimum. As such, picking these parameters by hand to minimize bandwidth, CPU time, propagation latency and so on remains an open problem.

Finally, it's worth mentioning that all of the tested Bitcoin simulators have been essentially abandoned by their authors, receiving no updates to modern versions of their dependencies.[1] This resulted in a fair amount of difficulty getting them to work. In addition to writing a new simulator from scratch, it would be useful to take over maintainership of one of these abandoned projects and keep it up to date.

[1] For ns-3 and Shadow, the simulation frameworks themselves are still being actively developed - just not the Bitcoin-specific plugins.

Bibliography

[1] Bowe, Sean. *simbit Documentation*. URL: https://github.com/ebfull/simbit/blob/master/README.md (visited on 07/06/2017).

[2] Bowe, Sean. *simbit: javascript p2p network simulator*. URL: https://github.com/ebfull/simbit.git (visited on 12/01/2016).

[3] Jean-Pierre Buntinx. *BTCFork Developer Will Soon Release a Bitcoin UAHF Client Capable of Increasing Block Size to 16 MB*. URL: http://www.newsbtc.com/2017/06/30/btcfork-developer-will-soon-release-bitcoin-uahf-client-capable-increasing-block-size-16-mb (visited on 06/30/2017).

[4] ns-3 developers. *ns-3 Development Tree*. URL: http://code.nsnam.org/ns-3-dev/shortlog (visited on 07/06/2017).

[5] ns-3 developers. *ns-3 development tree: examples*. URL: http://code.nsnam.org/ns-3-dev/file/8dbfaa3bc882/examples (visited on 07/06/2017).

[6] ns-3 developers. *ns-3 development tree: src/internet/model*. URL: http://code.nsnam.org/ns-3-dev/file/8dbfaa3bc882/src/internet/model (visited on 07/06/2017).

[7] ns-3 developers. *ns-3 Documentation*. URL: https://www.nsnam.org/docs/release/3.26/tutorial/html/index.html (visited on 06/07/2017).

[8] ns-3 developers. *ns-3 Documentation: Building Topologies*. URL: https://www.nsnam.org/docs/release/3.26/tutorial/html/building-topologies.html (visited on 06/07/2017).

[9] ns-3 developers. *ns-3 Documentation: Running a Script*. URL: https://www.nsnam.org/docs/release/3.26/tutorial/html/getting-started.html#running-a-script (visited on 06/07/2017).

[10] ns-3 developers. *ns-3 Documentation: Using the Logging Module*. URL: https://www.nsnam.org/docs/release/3.26/tutorial/html/tweaking.html#using-the-logging-module (visited on 06/07/2017).

[11] ns-3 developers. *ns-3 Documentation: Using the Tracing System*. URL: https://www.nsnam.org/docs/release/3.26/tutorial/html/tweaking.html#using-the-tracing-system (visited on 06/07/2017).

[12] ns-3 developers. *ns-3 Wiki*. URL: https://www.nsnam.org/wiki/Main_Page (visited on 06/07/2017).

[13] Shadow Developers. *Shadow Development Tree*. URL: https://github.com/shadow/shadow/commits/master (visited on 07/06/2017).

[14] Shadow Developers. *Shadow Wiki*. URL: https://github.com/shadow/shadow/wiki (visited on 07/06/2017).

[15] Durham, Vincent and Kraft Daniel and others. *Namecoin: a trust anchor for the internet*. URL: https://namecoin.org/ (visited on 01/12/2017).

[16] Frankenmint. *GitHub issue: make fails*. URL: https://github.com/shadow/shadow-plugin-bitcoin/issues/3 (visited on 07/06/2016).

[17] Arthur Gervais. *Bitcoin Simulator*. URL: https://github.com/arthurgervais/Bitcoin-Simulator/commits/master (visited on 06/07/2017).

[18] Arthur Gervais. *Bitcoin Simulator: An open-source bitcoin simulator built on NS3*. URL: http://arthurgervais.github.io/Bitcoin-Simulator/ (visited on 12/01/2016).

[19] Arthur Gervais. *Bitcoin Simulator: scratch/bitcoin-test.cc:767*. URL: https://github.com/arthurgervais/Bitcoin-Simulator/blob/master/scratch/bitcoin-test.cc#L767 (visited on 06/07/2017).

[20] Arthur Gervais. *Bitcoin Simulator: scratch/bitcoin-test.cc:79*. URL: https://github.com/arthurgervais/Bitcoin-Simulator/blob/master/scratch/bitcoin-test.cc#L79 (visited on 06/07/2017).

[21] Arthur Gervais. *Bitcoin Simulator: src/applications/model/bandwidth-distributions.h*. URL: https://github.com/arthurgervais/Bitcoin-Simulator/blob/master/src/applications/model/bandwidth-distributions.h (visited on 06/07/2017).

[22] Arthur Gervais. *Bitcoin Simulator: src/applications/model/bitcoin-miner.cc*. URL: https://github.com/arthurgervais/Bitcoin-Simulator/blob/master/src/applications/model/bitcoin-miner.cc (visited on 06/07/2017).

[23] Niklas Haas. *GitHub issue: 'npm install' fails building 'bufferutil' and 'utf-8-validate'*. URL: https://github.com/tagoro9/btcsimulator/issues/1 (visited on 06/19/2016).

[24] Henderson, Tom and Riley, George and Floyd, Sally and Roy, Sumit et al. *ns-3: a discrete-event network simulator for Internet systems*. URL: https://www.nsnam.org/ (visited on 12/14/2016).

[25] Alyssa Hertig. *Explainer: What Is SegWit2x and What Does It Mean for Bitcoin?* URL: http://www.coindesk.com/explainer-what-is-segwit2x-and-what-does-it-mean-for-bitcoin (visited on 07/12/2017).

[26] Rob Jansen. *GitHub issue: Examples for more complicated networks?* URL: https://github.com/shadow/shadow-plugin-bitcoin/issues/12 (visited on 07/06/2016).

[27] Rob Jansen and Nicholas Hopper. "Shadow: Running Tor in a Box for Accurate and Efficient Experimentation". In: *Proceedings of the 19th Symposium on Network and Distributed System Security (NDSS)*. Internet Society, Feb. 2012.

[28] Sunny King and Scott Nadal. "Ppcoin: Peer-to-peer crypto-currency with proof-of-stake". In: *self-published paper, August* 19 (2012).

[29] Johannes Lessmann, Peter Janacik, Lazar Lachev, and Dalimir Orfanus. "Comparative study of wireless network simulators". In: *Networking, 2008. ICN 2008. Seventh International Conference on*. IEEE. 2008, pp. 517–523.

[30] Miller, Andrew and Jansen, Rob. *shadow-plugin-bitcoin: A Shadow plugin that runs the Bitcoin Satoshi reference software*. URL: https://github.com/shadow/shadow-plugin-bitcoin (visited on 12/01/2016).

[31] Andrew Miller. *GitHub issue: Simulate block creation??* URL: https://github.com/shadow/shadow-plugin-bitcoin/issues/13 (visited on 07/14/2016).

[32] Andrew Miller and Rob Jansen. "Shadow-Bitcoin: Scalable Simulation via Direct Execution of Multi-threaded Applications." In: *IACR Cryptology ePrint Archive* 2015 (2015), p. 469.

[33] Mora Afonso, Víctor. *Bitcoin network protocol simulator*. URL: https://github.com/tagoro9/btcsimulator (visited on 12/01/2016).

[34] Satoshi Nakamoto. *Bitcoin: A peer-to-peer electronic cash system*. 2008.

[35] Eli Ben Sasson, Alessandro Chiesa, Christina Garman, et al. "Zerocash: Decentralized anonymous payments from bitcoin". In: *Security and Privacy (SP), 2014 IEEE Symposium on*. IEEE. 2014, pp. 459–474.

[36] Jonathan Toomim. *Torrent-style new-block propagation on Merkle trees*. URL: http://toom.im/blocktorrent/ (visited on 07/03/2017).

[37] Victor. *btcsimulator - Commits*. URL: https://github.com/tagoro9/btcsimulator/commits/develop (visited on 07/14/2017).

[38] Jonathan Warren. "Bitmessage: A peer-to-peer message authentication and delivery system". In: *white paper (27 November 2012), https://bitmessage. org/bitmessage. pdf* (2012).

[39] Shawn Wilkinson, Tome Boshevski, Josh Brandoff, and Vitalik Buterin. "Storj A Peer-to-Peer Cloud Storage Network". In: (2014).

[40] Gavin Wood. "Ethereum: A secure decentralised generalised transaction ledger". In: *Ethereum Project Yellow Paper* (2014).