Shadow-Bitcoin: Scalable Simulation via Direct Execution of Multi-threaded Applications

Andrew Miller
University of Maryland
amiller@cs.umd.edu

Rob Jansen
U.S. Naval Research Laboratory
rob.g.jansen@nrl.navy.mil

Abstract

We describe a new methodology that enables the direct execution of multi-threaded applications inside of Shadow, an existing parallel discrete-event network simulation framework. Our methodology utilizes function interposition and an application-layer thread library to emulate the ordinary thread interface to the application. Using this methodology, we implement a new Shadow plug-in that directly executes the Bitcoin reference client software. To demonstrate the usefulness of this tool, we present novel denial-of-service attacks against the Bitcoin software that exploit low-level implementation artifacts in the Bitcoin reference client; our deterministic simulator was helpful in developing and demonstrating these attacks. We describe optimizations that enable scalable execution of thousands of Bitcoin nodes on a single machine, and discuss how to model the Bitcoin network for experimental purposes.

1 Introduction

Experimentation testbeds for distributed systems and peer-to-peer networks such as Bitcoin, Bittorrent, and Tor, are beneficial to the scientific community as they simplify the code debugging and testing process, reduce time to deployment of new features and protocol modifications, and promote the research and development of new protocols and architectural modifications. However, testbeds like PlanetLab and the Bitcoin testnet do not scale gracefully, are hard to manage and maintain, and do not offer as much control over experimental topology and node configurations as is possible under alternative experimentation techniques. As a result, developers and researchers are often unable to realize the full potential of the experimental method, and new code and design modifications are often accepted into mainline software without fully understanding their effects on the existing, often critical infrastructure.

Alternative approaches to experimentation offer a unique set of benefits over the use of a distributed testbed. In particular, emulation may provide better scalability and improve management of and control over the network model and node configuration, and simulation may further allow for more efficient execution and repeatable experiments. Shadow [2, 24] provides an interesting and unique alternative to traditional experimentation techniques. At its core, Shadow is a simulator; the operating system, network stack, internetwork topology, and communication between nodes are all simulated using a discrete-event engine. However, each virtual host in Shadow runs real application software, such as the network's official reference client or alternate implementations. This unique simulation/emulation hybrid allows Shadow to provide the most efficient experimentation platform possible while remaining true to application-layer effects of the software executed by the virtual hosts. This unique approach is ideal for experimenting with large distributed systems and peer-to-peer networks.

Unfortunately, Shadow does not yet natively support virtual hosts that *fork processes* or run *multi-threaded* software due to the non-trivial layer of complexity added to Shadow's own internal multi-threaded simulation core. As a result, many distributed systems, including Bitcoin, are not amenable to simulation in Shadow.

In this work, we extend the state of the art in this unique simulation/emulation space by designing and implementing a simulation architecture that allows the direct execution of multi-threaded software. As a proof of concept of the efficacy of our approach, we design, implement, and test a new Shadow plug-in that directly executes the multi-threaded Bitcoin software inside the Shadow simulation framework. Our novel approach utilizes GNU Portable Threads (a.k.a., Pth) [1], an application-layer library that provides non-preemptive priority-based scheduling for multiple threads of execution. Pth runs in a single operating system thread while providing the facilities to emulate the Pthreads (posix threads) interface to the application. We then use *function interposition* to redirect Pthreads function calls made from the virtual host to Pth, while allowing Pthreads calls initiated by Shadow itself to be forwarded to and handled by the Pthreads library. Using our techniques, multi-threaded application software running in virtual hosts will function as intended, while the virtual host threads will not interfere with Shadow's internal threading engine. We envision that our approach will be ported to Shadow core so that all existing and

future Shadow plug-ins may benefit.

Using our new Bitcoin Shadow plug-in[1], we demonstrate and measure the cost and effectiveness of novel vulnerabilities in the Bitcoin software. We also show how to model the Bitcoin network and how to optimize the bootstrapping of a Bitcoin network.

Our major contributions are as follows:

- A new approach that utilizes Pth and function interposition to allow direct execution of multi-threaded applications in the Shadow simulator.
- The design and implementation of a new Shadow plug-in that uses our techniques to directly execute the multi-threaded Bitcoin software.
- The description of new attacks against Bitcoin, and an evaluation and measurement of these attacks using our new techniques and tools in a safe, private Shadow environment.
- A Bitcoin model that can be used to efficiently bootstrap and instantiate a large Bitcoin test network in Shadow.

2 Background and Related Work

This section provides background on the Shadow simulation framework, while outlining related experimentation work for and previous attacks on Bitcoin.

2.1 Shadow

Shadow [2, 24] is a parallel discrete-event network simulator. Shadow has a modular architecture that is broken into two major components: (1) the core simulator, and (2) software run by virtual hosts, which are dynamically loaded at run time as plug-in libraries.

2.1.1 The Core Simulator

Shadow itself is, at its core, a simulator. In addition to the parallel event engine, Shadow contains the logic required to simulate both the internetwork topology over which its virtual hosts will communicate as well as the operating system base upon which virtual hosts will run.

Event Engine. Because Shadow is a simulator, it replaces the concept of real time with its own simulation time over which it has precise control. Every action that happens in Shadow, such as starting virtual host applications or sending and receiving packets, is initiated from an event that occurs at a precise time instant (with nanosecond granularity). Shadow's event engine runs these events in the correct chronological order, while adhering to real-world characteristics such as network delay and loss. Shadow's event engine can benefit from the use of multiple worker threads.

Topology. Shadow uses the standard GraphML format to represent the connectivity and properties of links between each virtual host running in a simulation. This topology contains both vertices and edges: vertices represent Internet points-of-presence at which virtual hosts can be connected; and edges represent the path between those points-of-presence and their properties, including latency, jitter, and packet loss. Shadow models the Internet using real data available from public sources, like CAIDA and NetIndex. Every packet sent between two virtual hosts will be subject to the properties of the edges over which the packet travels, leading to communication characteristics that are not unlike those that would be realized between those locations on the Internet.

Operating System. Each virtual host in Shadow runs a simulated operating system (OS), including sockets, pipes, network protocols (TCP and UDP), timers, asynchronous event facilities, network interfaces, and various data buffers. These mechanisms are implemented to support the Linux POSIX interface and provide the functionality expected by virtual host software. Note that only the mechanisms that would affect the simulation, such as time or network communication, must be implemented; many system functions, such as file I/O, can be handled directly by `libc` as usual. Shadow uses function interposition to intercept calls made from virtual host software to OS functions, and redirects them to their simulated counterparts as required. In this way, Shadow is emulating a Linux environment to the application, which need not be aware that it is being simulated.

2.1.2 Host Software Plug-ins

As mentioned above, each virtual host contains a simulated OS that emulates a POSIX API to the application. The real software applications that run in Shadow are themselves compiled as Shadow plug-ins and loaded at run-time. During the compilation process, LLVM [4, 31] is used to inject a hook function that is used by Shadow to pass control into the application code in order to, e.g., call the `main` function in the application and notify the plug-in of available input/output on file descriptors.

Whenever Shadow instantiates a new virtual node that runs a particular plug-in, Shadow creates a new copy of the plug-in's memory heap and stores it internally. Shadow expects the plug-in to provide an interface in the form of an `on event(e)` function, which Shadow invokes whenever an IO event `e` is available. The plug-in indicates which events it is interested in by using `epoll` library functions, which Shadow intercepts.

A Shadow plug-in that runs the Tor anonymity software [5] has been created [24], is maintained [3] and is used extensively to help explore Tor research and development problems [14, 19, 22, 23, 25–29]. The largest known Tor test network to date contained 3600 relay nodes and 12000 client nodes [23]. Our work was motivated by the utility of the Tor plug-in.

[1] The code for our simulator is made freely available at https://github.com/shadow/shadow-plugin-bitcoin

2.2 Bitcoin Experimentation

Bitcoin [34] is a peer-to-peer "cryptocurrency" network that functions as a decentralized digital currency. There has been a recent surge of Bitcoin-related research including measurement, new applications, security models, incentive analysis, and attacks (see [11] for a comprehensive survey). We describe some of the existing experimentation frameworks, and outline several areas of research which we believe could utilize our simulator.

Testbeds. Parallel to the actual bitcoin network, there exists a public dedicated "test" network[2] that runs a modified version of the code. The modifications, such as frequently resetting the "mining difficulty", are intended to make it easier for experimentation while also discouraging its use as an actual currency. At the time of writing, we crawled Testnet and determined that it consists of approximately 250 nodes (at least an order of magnitude smaller than the actual network). Alternately, a "testnet-in-a-box" [20] can be run as a local instance of the test network. The main advantages of our Shadow-based simulation over Testnet is that the experimenter is afforded greater control over the network, while providing an accurate simulation of the network structure.

Several other projects, such as Simbit [13], simulate various aspects of the Bitcoin network. However, these do not run the actual bitcoind application code, but rather implement simplified abstractions; these may oversimplify or misrepresent the actual behavior. The attacks we demonstrate in Section 5, in particular, make use of implementation-specific behavior that is not modeled elsewhere.

Another form of testbed is a platform for measuring and interacting with the live network itself, such as Coinseer [30] and Coinscope [33].

Attacks. A very large focus of Bitcoin-related research has been on Bitcoin's weak privacy guarantees. Although the reference client takes some measures to maintain privacy, such as creating a new address to store "change," implementation quirks often allows one Bitcoin transaction to be linked to others. [32] The timing of information propagation can often be used to associate transactions with IP addresses [30]. Another vector for deanonymization involves exploiting the mechanisms by which Bitcoin nodes propagate information about their peers. [10]. Bitcoin privacy could be improved through a variety of techniques such as mixing [12, 35] or by upgrading the Bitcoin protocol to support privacy preserving cryptography [9, 36].

A well-known class of attacks involves deviating from the default mining behavior, and can in some cases allow the deviating miner to profit disproportionately. [7,17,18]

Another well-known class of attacks involves "double-spending" by convincing a victim that a payment transaction is (or will be imminently) accepted by the network, when in reality the attacker has ensured that a conflicting transaction will be accepted first [8]. So-called "fast payment" attacks exploit weaknesses of Bitcoin's information propagation mechanism [15]. We illustrate how our simulator can be used to model information propagation in Bitcoin.

Researchers have recently demonstrated that an attack that fills up a node's address list so that it eventually only connects to the attacker's nodes [21]. This attack and the vulnerabilities it exploits are unrelated to ours. They demonstrate and evaluate their attack against a "victim" node that they connect to the live network, and propose but do not evaluate several potential countermeasures; we believe an implementation of this attack in our simulator would be a good way to evaluate potential countermeasures and study their interactions within the entire network.

Another recently published denial-of-service attack involves exploiting the address propagation mechanism to exhaust a node's memory, causing it to crash [10]. The attack we demonstrate uses an entirely different mechanism, but has a similar effect.

3 Direct Execution of Multi-Threaded Applications

Shadow plugins ordinarily use an epoll-based event loop. This works well for systems such as Tor and Bittorrent, which are implemented as a single event loop and exclusively use sockets in non-blocking mode. Essentially, a Shadow plug-in consists of an event loop that is executed by the Shadow framework; each Shadow worker thread delivers one event to a single instance at a time. Shadow uses cooperative scheduling, rather than preemptive scheduling. As such, it's assumed that each plug-in finishes responding to every event within a short time.

This assumption does not hold for typical multi-threaded applications, such as Bitcoin, that create several *OS-level* threads – typically through the POSIX threads (Pthreads) API – and allow each thread to *block* when reading or writing to a socket.

In this section we describe an architecture for directly running such applications as a Shadow plug-in.

Pth. Pth (GNU Portable threads) [1] is a free software library that provides user-space threads. Pth threads are *cooperative* rather than *pre-emptive*. A Pth thread runs until it reaches a `pth_yield` instruction, which transfers control to the scheduler and activates another available thread. The underlying mechanism for switching between threads is fairly intricate [16]; it involves introspection and self-modification of the program stack. Pth

[2] see https://en.bitcoin.it/wiki/Testnet

```
on startup():
 for every event e we're waiting for:
  epoll_add(e)

on event(e):
 handle event e
```

Figure 1: Pseudocode for an ordinary epoll Shadow plug-in

```
on startup():
  pth_create(plugin_main);
  pth_setpriority(LOWEST);
  pth_yield();
  for each event e thread is waiting on:
   epoll_add(e)

on event(e):
  pth_yield();
  epoll_clear()
  for each event e thread is waiting on:
   epoll_add(e)
```

Figure 2: Pseudocode for a Pth-based Shadow plug-in

provides a substitute for the Pthreads api, as well as for the ordinary suite of POSIX I/O operations, such as reading and writing on files and sockets. The Pth version of an I/O operation ensures that the underlying file descriptor is in non-blocking mode; instead of blocking, it uses `pth_yield` to yield to the scheduler and indicates which events it can wait for.

Ordinarily, the Pth scheduler will activate threads until every thread is blocked waiting for an I/O event, and then it will use `select` in blocking mode to actually wait for an event. A plug-in using Pth directly would violate Shadow's assumption that the plug-in processes each event quickly and then returns control back to Shadow.

We take a simple approach that bypasses Pth's blocking call to `select`. Instead, we ensure that the "Shadow thread" is always available to run, but assign it the lowest-priority value so that it is only activated when every other thread is blocked. (The "main thread" of the application code is run in Pth thread with ordinary priority). When the Shadow thread receives an event, it yields to the Pth scheduler which activates any threads that can now run. When no more threads can run, the Shadow thread inspects which events the other threads are waiting for, and translates these into epoll event requests, which Shadow recognizes. Pseudocode for this plug-in architecture is given in Figure 2 (compare with pseudocode for a typical Shadow plugin in Figure 1).

Supporting `select`. Pth uses `select`-based tools for manipulating file descriptors. These crucially assume that the file descriptor is less than 1024. However, epoll-based programs do not make this assumption. Shadow is currently unfriendly to such programs, since the virtual mapped file descriptors may be any large number, and in fact every "instance" in Shadow has a unique number.

To fix this, we added an extra layer of mapping between file descriptors. For each instance, Shadow maintains a mapping between the local file descriptor number (which is typically less than 1024) and actual file descriptors on the host (which will be unique among all instances, and therefore typically greater than 1024 in number).

Interposition of Pthreads. While the approach described above is suitable for writing new Pth-based applications, most existing application code is written to depend on the Pthreads api. Our solution is to intercept calls intended for I/O or Pthreads library, and route them to the appropriate Pth functions. Fortunately, Pth provides an emulation of the Pthreads interface, which we were able to use mostly intact.

Ease of Adding New Applications. Although we focus on Bitcoin-related plug-ins in this paper, we believe our framework can easily be used to simulate most other single- or multi-threaded applications as well, with little or no application-specific customization required in general. A limitation is that any program relying on busy-loops or expensive computations to avoid deadlock or races must be modified; our framework assumes each activated application thread eventually reaches a blocking I/O call, waits for a lock, or `sleep`s. Computations in Shadow all occur in one virtual time instant, so an application thread performing long computations will appear to run fast. Our implementation does not support multi-processing (i.e., `fork()` or `exec()`).

4 Simulating Bitcoin in Shadow

We implemented the architecture above as a reusable plug-in "template" for simulating arbitrary multi-threaded applications; the template simply calls the application's "main" function to create a new instance. As a proof-of-concept, we used this framework to build a Shadow plug-in for bitcoind. We now describe the architecture of bitcoind and several further changes we needed to make to Shadow to support it.

The Satoshi Client. While the Bitcoin network comprises dozens of different client implementations, the *de facto* standard is the "reference client" (also known as, `bitcoind`, *mainline*, or the *Satoshi* client). There is arguably no authority to define an "official" client; regardless, as `bitcoind` remains far and away the most widely used client,[3] other alternative clients generally

[3]The reference client is the most popular among "reachable" nodes that receive incoming connections. According to https://getaddr.bitnodes.io/, which performs daily crawls of the network, recent versions of Satoshi accounts for 83% of the reachable nodes. BitcoinJ is likely more popular among mobile clients, which are often behind a firewall and do not receive incoming connections.

aim for full compliance with its behavior.

The reference client was originally written by the pseudonymous author, Satoshi Nakamoto, and published to a cryptography mailing list. Since then, it has been maintained as a free software project. The reference client is written in C++, and uses a heterogeneous multi-threaded architecture. The basic architecture of `bitcoind` has remained unchanged, despite frequent version updates with optimizations and new features. It contains dependencies on several libraries such as LevelDB and Boost.[4]

The reference client interacts with the rest of the network by sending and receiving messages. It uses one connection handler thread for each connected peer, but a single main thread that processes messages arriving in a queue. It maintains eight outgoing connections by default, and handles up to 117 incoming connections. This multithreaded architecture makes use of both blocking and non-blocking behavior. For example, although each peer connection uses a non-blocking socket, the connection thread performs a blocking `sleep` for 100 milliseconds in between polling the socket.

Supporting C++ in Shadow. Shadow supports ordinary static variables by initializing them once, then memoizing the initialized state to reuse later for other instances. This is insufficient for C++, since static objects may execute arbitrary code in their constructors. We modified Shadow with an extra LLVM pass that executes all necessary constructors each time an instance is loaded.

Injector. In addition to the `bitcoind` plugin, we used our multi-threaded framework to easily build a special purpose "injector" plug-in for our experiments. The plug-in connects to a single node and performs only the minimal handshake required before sending a payload of messages from a file. This plug-in shares no common codebase with `bitcoind` whatsoever, and instead uses a free library made by Bitcoin core developer Jeff Garzik called PicoCoin that provides C routines for manipulating Bitcoin protocol messages. Using this library, we made a small application, the injector, that communicates with the Bitcoin network in a very limited way. Effectively, it connects to a node, performs the `VERSION`/`VERACK` handshake, delivers a payload of blocks and transactions, and then quits. It requires under 250 lines of code.

Local Sockets. We've taken initial steps towards supporting simulations of Bitcoin network measurement platforms, such as Coinscope [33], within our framework. Coinscope consists of multiple processes (each of which becomes a single plugin) that coordinate using local unix domain sockets (which are not currently supported within Shadow). We implemented unix domain sockets as a new socket type in Shadow and are now able to directly execute Coinscope code.

[4] We omit BerkeleyDB by compiling bitcoind without the "wallet" functionality that depends on it. A BerkeleyDB version change was involved in an accidental "fork" disaster [6] where non-updated nodes temporarily diverged from the network.

5 The mapOrphans Attack

In this section, we present novel denial-of-service attacks that exploit vulnerabilities in the `bitcoind` implementation. We describe how we used our simulator to implement and evaluate these attacks, demonstrating that our simulator framework is useful for practical research.

The mapOrphans Vulnerability. Bitcoin transactions form a directed graph; each transaction spends some previously available "input" coins, and creates several new "output" coins that can be spent by subsequent transactions. Consider a pair of related transactions: one transaction (the "child") spends a transaction output created by the other (the "parent"). If a node receives these transactions out of order (i.e., first the child and then the parent), the child transaction can not be validated until the parent is received. To help with out-of-order arrivals (e.g., due to varying latency or a dropped connection), the reference client maintains a buffer called mapOrphans. [5] Transactions with unknown parents are placed in this buffer, and are not validated until the parent is received.

This mechanism can be exploited to circumvent `bitcoind`'s defenses. The most computationally expensive step in validating a transaction is checking the ECDSA signature. To prevent exhaustion attacks, signature checking is deferred until all other validation steps are complete, and a node bans any peer that sends transactions with invalid signatures. However, when a node places transactions in mapOrphans, it forgets which peer relayed it. Thus by sending a set of (invalid) transactions out-of-order (as illustrated in Figure 3), an attacker can (at low cost to itself) cause a node to perform a large number of signature checks. Since `bitcoind` processes all transactions in a single main thread, the net result is that a victim node can be frozen until all the signatures have been processed.

The primary constraints on this attack are the maximum size of mapOrphans (10k transactions) and the maximum size of an orphan transaction (5KB, enough to hold 40 signatures). On a test machine (Intel Core i7, 1.73Ghz), each signature verification took 1.7 milliseconds; hence an attacker could plausibly freeze a node for over 10 minutes.

Evaluation in Shadow-Bitcoin. We implemented this attack in our simulator to confirm its effectiveness. We generated a payload of transactions as described above, and used the `injector` plug-in to deliver it to an instance of `bitcoind` (version 0.9.2).

[5] We use mapOrphans to abbreviate mapOrphanTransactions

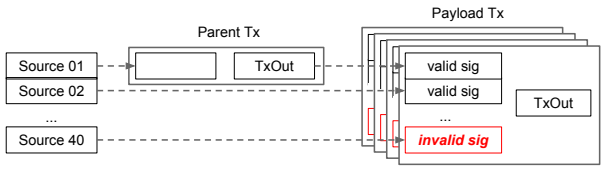

Figure 3: Transactions used in the mapOrphans DoS Attack. The payload transactions are invalid and mutually conflicting. The attacker delivers these transactions to the victim out-of-order: first the payloads, and then the parent.

It was necessary to modify the `bitcoind` code to simulate the time delay of signature validation, since Shadow models all computation as occurring instantaneously. Therefore we inserted a `sleep` function after signature verification based on the amount of time our measurements indicate the computation should take.

We used our simulator to observe the effects of a frozen message queue on a node's connections. We experimented with this by configuring the Shadow experiment script to launch nodes and form new connections at various moments before, during, and after an attack. Connections that were established prior to the attack are still serviceable after the attack subsides. Although the stalled message queue prevents the node from responding to messages from it peers, the separate connection threads prevent the socket buffers from overflowing. However, new incoming connection attempts are dropped, since a peer times out if the initial handshake is not completed within 60 seconds.

Memory Consumption Extension. While the victim's main thread is busy processing invalid transactions, the connection handler threads continue to receive and buffer input from each of its peers. Each connection buffers up to 5 megabytes of messages; if this limit is reached, the connection is dropped. By using up the maximum available connections (i.e., 125), and filling up these buffers to the maximum limit while a mapOrphans attack is underway, an attacker can consume up to 500+ megabytes of RAM. This can crash nodes with a limited (but plausible) amount of memory.

Mitigations. We reported these vulnerabilities to the Bitcoin developers, who deployed a mitigation in version 0.9.3. The mitigation reduces the size of mapOrphans to 500, down by a factor of 100. Also, whenever a transaction is placed in mapOrphans, the identity of the peer who sent it is stored alongside the transaction itself. If a mapOrphans transaction turns out to be invalid, then that peer is disconnected. Whenever a peer disconnects, any items in mapOrphans associated with that peer are discarded without inspection.

Shadow's role. Our simulator's faithful modeling of application-level behavior helped us notice errors in our initial implementation. For example, `bitcoind` processes signatures in a deterministic order, which we exploit to incur the greatest cost on the victim; also, `bitcoind` maintains a cache of previously-validated signatures, hence the attack must consist of entirely distinct valid signatures. Our simulator allowed us to make rapid iterations while developing and testing the implementation. Additionally, the deterministic network-schedule simulation simplified debugging our experiment involving the victim's peer connections.

6 Bitcoin Network Model

We now describe how to run thousands of simulated instances of `bitcoind` to create a realistic, full-scale model of the Bitcoin network.

Bitcoin network topology. Although Shadow already supports existing datasets for the Internet topology, we must model the Bitcoin network overlay topology.

A list of the reachable IPs on the Bitcoin network can be imported from publicly available snapshots from `getaddr.bitnodes.io`. We used data obtained through our own crawls using Coinscope [33]. Our network model consists of 6081 nodes; roughly 40% of these are from the US, and a nearly equal amount from Europe. Our data only includes IPV4 addresses, although IPv6 nodes are supported by Bitcoin; according to `getaddr.bitnodes.io` data, at the time of writing less than 4% of nodes use IPV6.

The actual Bitcoin network forms its overlay topology through an intricate mechanism [33]. Information about potential peers propagates throughout the network through a gossip protocol. Each node maintains a list of peers it knows about, and tries to maintain exactly eight outgoing connections; when an outgoing connection is dropped, the node selects a random peer from the set it knows about and attempts to make a new connection. When a new node first joins the network, it queries several hardcoded "seed" nodes for a small starting list. After forming initial connections from this list, nodes learn about each other by relaying "ADDR" messages containing the IP and port of themselves and their peers.

For simplicity, we bypassed this procedure by using existing `bitcoind` configuration options to force node connectivity. The data from our Coinscope crawls give us a set of known peer IP addresses to which each node in our model has connected. To downscale our network, we start with this 6081 node connectivity model and then repeatedly remove the least connected node (the node with the least number of edges) as well as the edges to and from that node until reaching a network with the desired number of nodes. Finally, we configure each node with 8 connections from the remaining set of edges.

The least connected node was chosen in order to minimize the number of edges that get removed from the original connectivity graph. We acknowledge that this over-simplified connectivity model likely affects the accuracy

of our simulated network. For example, it has been shown that the information propagation effectiveness can be influenced by even a single well-connected node [15]. Preliminary measurements of the Bitcoin network have provided evidence of many such well-connected nodes and that random graph models do not account for the observed network structure [33]. However, we stress that our primary goal is to demonstrate the flexibility we have in creating a topology of our choosing, and we believe that it is more important to understand how changes in a given network affect behavior than it is to precisely model the real network.

Providing initial blockchain state. Each node in the Bitcoin network typically maintains its own copy of the entire blockchain. In our model network, we begin with all the nodes "in sync" to some prior blockchain state.

To reduce the storage cost, we allow the simulated nodes to *share* a single copy of state files whenever possible. The `bitcoind` data directory primarily consists of a set of *block files*, each of which stores up to 128 megabytes worth of blocks; and a LevelDB database that maintains an index into the block files and is used to lookup individual blocks or transactions from disk.

The block files are append-only rotated logs; once a block file reaches 128MB, it is finalized and never written to again. Therefore, each node only ever needs to write to the "newest" block file. By choosing our initial blockstate to correspond to the first block after a block file is completed, we minimize the amount of data that must be copied rather than aliased. Similarly, the LevelDB database consists of a number of append-only files that, once full, can also be aliased.

Overall, we are able to run a 6000 node simulation using less than 350 gigabytes of RAM and less than 300 gigabytes of storage.

Transaction Propagation Experiment. We now explore how information propagates in Bitcoin using simulated networks at various scales.

Transactions propagate through the network using a three-round protocol. When a node receives a valid transaction from one of its peers, it sends an `INV` message containing the transaction's hash to each of its peers. When a peer receives an `INV` containing a transaction it does not know about, it requests the transaction by sending `GETDATA`. Finally, a node responds to `GETDATA` with the actual `TX` data.

`INV` messages aren't sent immediately; instead, `INV` messages are buffered for each peer, and every tenth of a second, one peer is selected at random and the corresponding buffer is flushed. If a node has N connections, then for a given peer, it takes on average $10/N$ seconds before the `INV` message is received.

We instantiated a model network at block height 120594, which corresponds to April 2011. In order to simulate spending coins mined then, we modified the client to recognize a hard-coded public key and replace it with a default public key for which we know the corresponding private key.

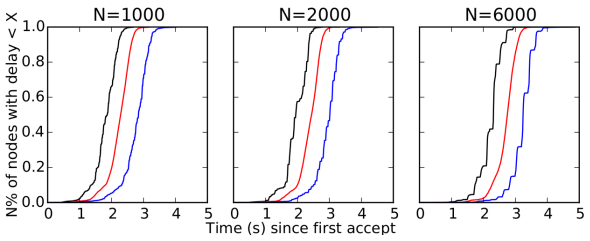

Figure 4: Transaction propagation in our simulated Bitcoin network. (Black: minimum, red: mean, blue: maximum). Each experiment was averaged over 10, 3, and 1 runs respectively).

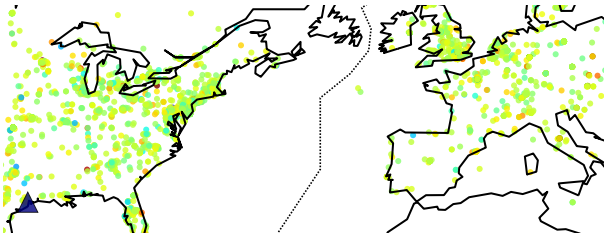

Figure 5: Transaction propagation in our 6000-node simulated Bitcoin network (zoomed to show Europe and the eastern coast of North America). The large triangle (Houston) indicates the transaction origin. The color of each point indicates the time to receive a transaction (averaged over 100 trials) (blue is faster, green and yellow are longer).

We experimented with overlay topologies containing 1000, 2000, and 6000 nodes to determine the effect of network size on transaction propagation times. For each experiment, we generated 100 transactions, and relayed them through a randomly chosen entry node. The results from these three experiments are shown in Figure 4. In our model, on average, transactions take longer to propagate in a larger network.

In Figure 5, we overlay the data for our 6000 node experiment on a map. There appears to be little geographic correlation with transaction propagation time, suggesting that application delays and the structure of the overlay network have a greater impact.

Limitations. While these experiments demonstrate the versatility of our simulation framework, we stress that our initial network model does not capture many salient aspects of the Bitcoin network. For example, the presence of even a small number of very well-connected Bitcoin nodes can have measurable impact on information propagation [15]. Since the network is widely known to indeed have well-connected nodes [33], the accuracy of our network model likely suffers. Improving our network model and validating its accuracy is ongoing work.

7 Conclusion

In this paper, we introduced a new methodology that enables virtual hosts in the Shadow parallel discrete-event simulator to run multi-threaded applications. Using this methodology, we designed and developed a Shadow plug-in that runs the Bitcoin reference software, explained how we model a Bitcoin network for testing purposes, and describe optimizations that enable us to run thousands of Bitcoin nodes in a private test network. Finally, we demonstrated the efficacy of our plug-in through transaction propagation experiments, and by demonstrating novel denial of service attacks based on the mapOrphans transaction processing queue.

Lessons Learned. Through this work, we have realized the benefit of having access to a simulation environment that runs real software. Not only does our Bitcoin simulator allow us to scale to the largest Bitcoin test-network known to date, but it also enables rapid prototyping of new features and fixes. In fact, when the topology size is small, our experiments run in *faster than real time*.

Accurate simulators, rather than simplified abstractions, are ideal tools for studying the nuances of distributed system software. Our work has contributed to our understanding that the Bitcoin peer-to-peer protocol is flawed and highly vulnerable, and that many potential vulnerabilities and exploits lie within the low level details of the Bitcoin implementation.

Future Work. Although we have demonstrated the usefulness of our approach, and provided initial steps towards, it still remains for us to *validate* the accuracy of our network model by comparing it with measurements. We also hope to work with the Shadow developers to merge our work into Shadow core so that other multi-threaded applications can more easily run in Shadow. Finally, we intend to continue studying the Bitcoin implementation for vulnerabilities and hope to help improve the software through mitigation techniques that we can show through simulation to be effective.

References

[1] GNU Portable Threads. http://www.gnu.org/software/pth/.

[2] Shadow homepage and code repositories. https://shadow.github.io, https://github.com/shadow.

[3] shadow-plugin-tor code repositories. https://github.com/shadow/shadow-plugin-tor.

[4] The LLVM Compiler Infrastructure. http://llvm.org.

[5] The Tor Project. https://www.torproject.org/.

[6] ANDRESEN, G. March 2013 Chain Fork Post-Mortem. BIP 50.

[7] BAHACK, L. Theoretical Bitcoin Attacks with less than Half of the Computational Power (draft). Tech. Rep. abs/1312.7013, CoRR, 2013.

[8] BAMERT, T., DECKER, C., ELSEN, L., WATTENHOFER, R., AND WELTEN, S. Have a snack, pay with Bitcoins. In *IEEE P2P* (2013).

[9] BEN-SASSON, E., CHIESA, A., GARMAN, C., GREEN, M., MIERS, I., TROMER, E., AND VIRZA, M. Zerocash: Decentralized anonymous payments from Bitcoin. In *IEEE S&P* (2014).

[10] BIRYUKOV, A., KHOVRATOVICH, D., AND PUSTOGAROV, I. Deanonymisation of clients in bitcoin p2p network. In *ACM CCS* (2014).

[11] BONNEAU, J., MILLER, A., CLARK, J., NARAYANAN, A., KROLL, J. A., AND FELTEN, E. W. Research Perspectives and Challenges for Bitcoin and Cryptocurrencies (Extended Version). Cryptology ePrint Archive, Report 2015/452, 2015.

[12] BONNEAU, J., NARAYANAN, A., MILLER, A., CLARK, J., KROLL, J. A., AND FELTEN, E. W. Mixcoin: Anonymity for Bitcoin with accountable mixes. In *FC* (2014).

[13] BOWE, S. simbit - p2p network simulator. https://bitcointalk.org/index.php?topic=603171.0, May 2014.

[14] CONRAD, B., AND SHIRAZI, F. Analyzing the effectiveness of dos attacks on tor. In *SIN* (2014), pp. 355:355–355:358.

[15] DECKER, C., AND WATTENHOFER, R. Information propagation in the bitcoin network. In *IEEE P2P* (2013).

[16] ENGELSCHALL, R. S. Portable multithreading: The signal stack trick for user-space thread creation. In *USENIX ATC* (2000).

[17] EYAL, I. The Miner's Dilemma. In *IEEE S&P* (2015).

[18] EYAL, I., AND SIRER, E. G. Majority is not enough: Bitcoin mining is vulnerable. In *FC* (2014).

[19] GEDDES, J., JANSEN, R., AND HOPPER, N. How low can you go: Balancing performance with anonymity in Tor. In *PETS* (2013).

[20] HEARN, M. Testnet in a box. https://bitcointalk.org/index.php?topic=4483.0, March 2011.

[21] HEILMAN, E., KENDLER, A., ZOHAR, A., AND GOLDBERG, S. Eclipse attacks on bitcoins peer-to-peer network.

[22] JANSEN, R., BAUER, K., HOPPER, N., AND DINGLEDINE, R. Methodically modeling the Tor network. In *CSET* (2012).

[23] JANSEN, R., GEDDES, J., WACEK, C., SHERR, M., AND SYVERSON, P. Never been KIST: Tor's congestion management blossoms with kernel-informed socket transport. In *USENIX Security* (2014).

[24] JANSEN, R., AND HOPPER, N. Shadow: Running Tor in a box for accurate and efficient experimentation. In *NDSS* (2012).

[25] JANSEN, R., JOHNSON, A., AND SYVERSON, P. LIRA: Lightweight incentivized routing for anonymity. In *NDSS* (2013).

[26] JANSEN, R., SYVERSON, P., AND HOPPER, N. Throttling Tor bandwidth parasites. In *USENIX Security* (2012).

[27] JANSEN, R., TSCHORSCH, F., JOHNSON, A., AND SCHEUERMANN, B. The Sniper Attack: Anonymously deanonymizing and disabling the Tor network. In *NDSS* (2014).

[28] JOHN GEDDES, R. J., AND HOPPER, N. Tor IMUX: Managing connections from two to infinity, and beyond. In *WPES* (2014).

[29] JOHNSON, A., WACEK, C., JANSEN, R., SHERR, M., AND SYVERSON, P. Users get routed: Traffic correlation on tor by realistic adversaries. In *ACM CCS* (2013).

[30] KOSHY, P., KOSHY, D., AND MCDANIEL, P. *An analysis of anonymity in bitcoin using p2p network traffic*. Springer, 2014.

[31] LATTNER, C., AND ADVE, V. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)* (Mar 2004).

[32] MEIKLEJOHN, S., POMAROLE, M., JORDAN, G., LEVCHENKO, K., MCCOY, D., VOELKER, G. M., AND SAVAGE, S. A Fistful of Bitcoins: Characterizing Payments Among Men with No Names. In *IMC* (2013).

[33] MILLER, A., LITTON, J., PACHULSKI, A., GUPTA, N., LEVIN, D., SPRING, N., AND BHATTACHARJEE, B. Discovering bitcoin's public topology and influential nodes, May 2015.

[34] NAKAMOTO, S. Bitcoin: A Peer-to-Peer Electronic Cash System. http://bitcoin.org/bitcoin.pdf, 2008.

[35] VALENTA, L., AND ROWAN, B. Blindcoin: Blinded, Accountable Mixes for Bitcoin. In *Workshop on Bitcoin Research* (2015).

[36] VAN SABERHAGEN, N. Cryptonote v 2.0. *https://cryptonote.org/whitepaper.pdf* (2013).