High Performance Tor Experimentation from the Magic of Dynamic ELFs

Justin Tracey
University of Waterloo
j3tracey@uwaterloo.ca

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

Ian Goldberg
University of Waterloo
iang@cs.uwaterloo.ca

Abstract

The Tor anonymous communication network and Bitcoin financial transaction network are examples of security applications with significant risk to user privacy if they fail to perform as expected. Experimentation on private instances of these networks is therefore a popular means to design, develop, and test improvements before deploying them to real users. In particular, the Shadow discrete-event network simulator is one of the most popular tools for conducting safe and ethical Tor research. In this paper, we analyze Shadow's design and find significant performance bottlenecks in its logging and work scheduling systems stemming from its representation of simulated processes and its use of a globally shared process namespace. We design, implement, and empirically evaluate new algorithms that replace each of these components. We find that our improvements reduce Shadow run time by as much as 31% in synthetic benchmarks over a variety of conditions, and by as much as 73% over small and large experimental Tor networks. Our improvements have been merged into Shadow release v1.12.0 to the benefit of the security and privacy communities.

1 Introduction

Networked applications are a staple of modern computing environments, in nearly all domains. Applications intended for security and privacy protection are no exception, as evidenced by tools such as Tor [6], which allows for greater privacy to clients or servers over the public internet, and Bitcoin [15], which is used to provide secure financial transactions and a public append-only ledger.

Experimentation is an important element of developing any networked application, as it allows for developers to know which components of such applications are in need of improvement or debugging, and allows them to test whether changes truly did improve or fix the application as intended. Experimentation is similarly necessary in research on such applications, as it is what allows for empirical testing of relevant hypotheses. Being able to perform experiments on networked applications that are independent of networks actively used by end users is often necessary, as it grants a greater amount of control over the conditions of the experiment. In the case of programs which have inherent security and privacy implications, it is of even greater importance to use experimentation platforms designated for testing, so as to avoid violating the consent of parties who are assuming privacy and security is being provided. For example, the Tor Project has a series of strict guidelines on what constitutes acceptable research methods on Tor.[1] The first of these guidelines is: *Use a test Tor network whenever possible*. Therefore, developing and testing experimentation platforms for these domains is of central importance.

Conceptually, network testing platforms exist on a spectrum of environmental realism versus environmental control: the most realistic environments are isolated networks that are deployed on many physical machines, while the most controlled environments are produced by network simulators, such as ns-3.[2] The deployed networks are extremely limited by the physically available resources, while network simulators run simplified, representational models of the network under study. Between the extremes on this spectrum are network emulators, which run many instances of the relevant software on a few machines—e.g., the ExperimenTor [3] and NetMirage[3] Tor network emulators. Such tools are able to provide a greater level of control and are less constrained by physical requirements than a deployed network, but are also not as realistic in resource usage as a deployed network or as scalable as a network simulation.

The Shadow discrete-event network simulator [10] breaks out of the simplified realism versus control spectrum. While Shadow is a network simulator and therefore runs representations of the network stack with simulated timescales independent of real world time, Shadow also directly executes compiled application code (e.g., Tor or Bitcoin) instead of simplified representations of applications. This best-of-both-worlds approach has made Shadow one of the most popular tools for running Tor experiments in particular [17].

In this work, we analyze Shadow's design and execution and identify performance bottlenecks in its logging system and its work scheduling system that stem from its representation of simulated processes and its use of

[1] `research.torproject.org/safetyboard.html`
[2] `www.nsnam.org`
[3] `crysp.uwaterloo.ca/software/netmirage`

a globally shared process namespace. These bottlenecks limit the scale of experiments that can be completed in a feasible amount of time or on given hardware, which may lead to research conclusions being drawn from scaled-down networks of questionable accuracy.

From our analysis, we redesigned several components of Shadow's implementation. First, we designed a new thread-aware logging system that eliminates a global lock that was previously acquired during the logging of simulation messages, reducing lock contention and synchronization overhead. Second, we developed and incorporated the use of a custom high-performance dynamic loader called *drow-loader* to replace the standard Linux loader. The new loader allows Shadow to load simulated processes into independent namespaces in order to optimize and improve correctness of execution and eliminate lock contention across instances. Third, we design and implement a dynamic load-balancing algorithm that uses *drow-loader* to dynamically migrate available work tasks to idle Shadow threads.

For each of these newly designed components, we constructed and ran experiments to measure the impact each incrementally had on overall performance. Our experiments demonstrate a considerable improvement both in standard synthetic benchmarks for discrete-event network simulators, as well as in more pertinent simulated Tor networks. Through experimentation, we found that in combination, our improvements reduced Shadow run time by as much as 31% in synthetic benchmarks over a variety of conditions, and by as much as 73% over small and large experimental Tor networks. All of our improvements have been merged into Shadow release v1.12.0 and are publicly available.[4]

2 Background

In this section we provide a brief overview of the design of Shadow as it existed during our Shadow design analysis presented in Section 3 and before our improvements discussed in Section 4. Previous work provides additional background [14, Section 2.1].

Overview: Shadow is a parallel and conservative-time discrete-event network simulator [10] that dynamically loads applications as plugins and directly executes them as native code. Plugins are executed in simulated *processes* running on simulated *hosts* that communicate over a simulated *network*. The types of hosts, the processes they run, and the layout of the network that connects them are specified in a user-supplied XML file, which we will refer to as the *configuration file*. Upon initialization, the hosts are distributed evenly across a user-specified number of system *worker threads*.

[4] `github.com/shadow/shadow`

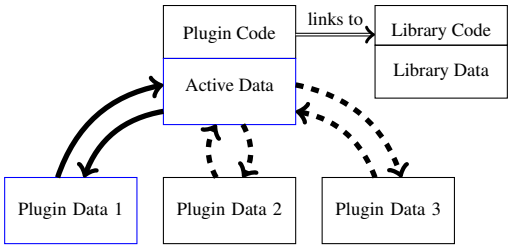


Figure 1: Shadow's state swapping technique. Each instance of a plugin (e.g., Tor) has its data segment's previous state copied into the data segment of the corresponding shared object when executing, then copied back out when another instance executes. All plugins link to a single instance of each library.

Dynamic Loading: Shadow hosts can run multiple simulated processes, each of which represents the Linux process being simulated. To execute a simulated process on a host, Shadow dynamically loads the process executable, known as a *plugin*, as an ELF shared object into memory. This is done using the standard `dlopen()` function provided by libdl (part of glibc). Using features of the LLVM compiler, these plugins have their entire data segment stored in a known location in memory, as a single structure (a technique known as *hoisting*).

Portable Threads: Once the executable plugin is loaded for a process, an instance of a modified version of the GNU Portable Threads (Pth) library is used to execute the process (by calling the `main()` function) and any logical threads it creates. Pth accomplishes this task by keeping multiple stack contexts and switching between them using `setjmp()` and `longjmp()`. All such stacks are executed until they would block (e.g., waiting for I/O), and then execution returns to the Shadow stack.

Function Interposition: Virtual processes run their plugin executable code without explicit knowledge that they are being loaded and run in Shadow. As a result, the plugin code makes calls to libc functions as usual; e.g., to open sockets or to send and receive data to and from other network hosts. Shadow intercepts such functions using *function interposition*; e.g., in order to redirect data over the simulated network to other simulated hosts. Shadow intercepts a large number of functions in order to emulate a Linux environment to the executing plugin.

State Swapping: Upon switching execution from one simulated process to another, Shadow copies the data segment of the previously executing process elsewhere in memory and exchanges it with the stored segment of the newly executing process—a procedure we call *state swapping* (see Figure 1). Using this technique, multiple instances of a process corresponding to a particular plugin may be simulated, despite the dynamic loader only allocating memory for the plugin once.

Time and Events: As a simulator, Shadow has its own representation of time which allows control over the pro-

cess execution; Shadow interposes all time-related functions and returns the simulated time to the calling process. Major simulator tasks that should occur at a future time, such as packets arriving at another host, are called *events* and stored in a min-heap sorted by time. This allows Shadow to execute events in order and guarantee causality (i.e., that time only moves forward). To support parallel execution, each worker thread has its own queue of events, and a barrier is placed on the earliest event time that can affect the behavior of a host associated with another worker thread. The span of events that take place between two such barriers is called a *round*.

Use Cases and Constraints: Although originally designed to run Tor [10], Shadow is theoretically able to run any TCP-based application with slight modifications and constraints—the most notable being that I/O events are polled using supported interposed functions such as `poll()`, `epoll()`, and `select()`, the application can be made not to use `fork()` or `exec()`, and the application can be compiled as a shared object or position-independent executable (which most applications can). While Shadow has been used for simulating Bitcoin [14], Tor remains Shadow's primary use case and Shadow is currently one of the most popular means of performing Tor experiments [17].

3 Design Analysis

While using Shadow for Tor research, we found that a small, 234-host experiment finished more quickly on an four-core (eight-thread) desktop machine with eight worker threads than on a machine in the CrySP RIPPLE Facility[5] with 80 cores (160 threads) (irrespective of the number of Shadow worker threads). We believe this is due to a marginally higher CPU clock speed on the desktop machine. Following this initial finding, we reviewed Shadow's architecture and identified significant limitations in its design. In the remainder of this section, we provide a summary of the major contributors to Shadow's poor multi-threading scalability.

GLib Message Logging: Shadow uses GLib, an open-source library that provides common core application building blocks for libraries and applications written in C. While this has aided in the speed of Shadow development, it has come at a performance cost because GLib was not written with high-performance computing in mind. We reviewed the source code of the GLib message logger that is used by Shadow and found that GLib uses a single global lock for all messages logged using the logger. Contention on GLib's global message lock will significantly reduce multi-threaded performance.

[5] `ripple.uwaterloo.ca`

To understand how this affected Shadow, we ran two Tor experiments with ~700 relays and ~20,000 clients: in one version we commented out all GLib logging in Shadow. As shown in Figure 2, a simulation configured to run for 35 simulated minutes took 115 hours in the original Shadow, and only 9 hours after removing GLib logging (the main workload starts at 20 minutes simulation time). Clearly, Shadow could benefit from replacing GLib's logger with a thread-aware logger.

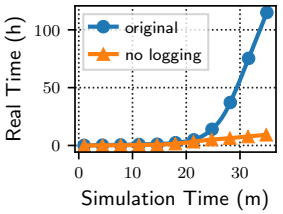


Figure 2: We found that message logging significantly increased Shadow runtime (configured to use 24 threads).

Loading and Running Plugins: As mentioned previously, Shadow creates multiple instances of a simulated process from a single plugin by swapping state. However, this technique comes with some drawbacks. The most obvious is that copying the entire state of a plugin can be expensive, depending on the size of the data segment. Doing so every time a different simulated process runs adds some overhead over directly running it.

Another drawback is that this state swapping technique cannot work for any shared object that was not built using the LLVM pass that moves the entire state into a single, movable structure. This means there is an assumption made that every system library used by a simulated process is effectively stateless, or that there is no effect of sharing this state between multiple simulated processes between events. In the instances where state does impact operation (as is the case for the OpenSSL library used by Tor), a lock is required to ensure only one host is using that library at any time (a workaround that does not actually solve the underlying problem, but in practice was found to be sufficient). As a contrived but demonstrative example, suppose the SSL library used stored state in the form of which cipher suite was currently being used. When there is only one process making use of this state, it operates correctly. But when multiple simulated processes, all operating from the same actual process, attempt to use two different cipher suites, this state would be corrupted, and the simulated behavior would not match the behavior of the actual application.

Load Balancing: Another drawback of Shadow's loading technique is that it greatly complicates any attempts to migrate hosts or simulated processes from one worker thread to another. Because each thread has its own distinct copy of the plugin, each with its own associated memory where the active state is located (which must be contiguous with the rest of the executable in memory), any pointer variables that store an address within the active state would no longer point to the correct address af-

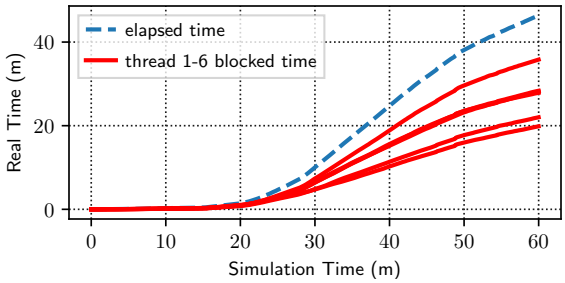


Figure 3: The amount of time that each worker thread has spent blocked compared to the total elapsed time of the experiment.

ter the state was migrated to another thread's copy of the plugin. Because of this, Shadow's scheduler only assigns hosts to worker threads once, when Shadow is preparing the experiment. Since there is no adjustment of the initial scheduling, some worker threads will frequently run considerably longer than most of the others before hitting the round barrier. As such, the simulation fails to make effective use of the multithreaded environment it runs in, with initial tests showing some threads remaining idle for upwards of 80% of an experiment. This effect can be seen in Figure 3, which compares the simulation time to the real time that each worker thread has spent idle (as well as the total elapsed time across all threads). We see that the same threads are finishing early throughout, and could therefore handle more load than assigned.

Lack of Compiler Optimizations: Finally, Shadow in its original state must be compiled with the LLVM/Clang C compiler in order to utilize the hoisting technique described earlier. More importantly, it can only be compiled with the default set of compiler optimizations. Attempting to set the optimizations to a higher level causes segmentation faults during the course of an experiment for reasons that were never fully determined by us or the Shadow developers; we believe the problem is caused by the hoisting technique.

4 Design Improvements

In this section, we describe how we improved Shadow's design by developing a new thread-aware logging system and a new custom dynamic loader. The new loader allows us to load compiler-optimized virtual processes and their dependencies into independent namespaces and utilize new scheduling policies to dynamically balance load across worker threads at runtime.

Logging: In order to reduce lock contention while logging messages, we designed a new logging system and modified Shadow to use it instead of GLib's logger.

Our thread-aware logging system works as follows. Each Shadow worker thread generates formatted log messages as it executes code, and first buffers them in a thread-local *message queue*. Periodically or when a simulation round ends, each Shadow worker thread passes ownership of its message queue to a new single-purpose logger thread using an asynchronous *communication queue*. There are w such communication queues in the simulation, one to connect each of the w worker threads to the logger thread. The logger thread receives the w message queues through the respective communication queues, sorts the messages by time using a min-priority heap, and then writes out all sorted messages.

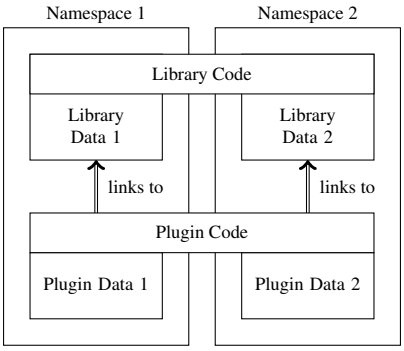


Figure 4: Our new design. Each instance of the plugin has its own dedicated namespace, which includes its code, data, and linked libraries. Read-only sections such as code, however, are mapped to the same physical memory. Contrast with Figure 1.

Our design is more scalable than the GLib logger since, unlike the GLib logger, it does not require any global locks. Buffering messages locally in each worker thread and writing all messages with a single logger thread further reduces synchronization overhead.

Loading Independent Namespaces: We developed a dynamic loader that we call *drow-loader*[6] (which we forked from elf-loader [19]) and redesigned Shadow to use it to load and directly execute applications (see Figure 4 and contrast with Figure 1). For more details on *drow-loader*, see the first author's Master's thesis [20].

Instead of using `dlopen()` to open the shared object, we use `dlmopen()` to load the shared object into its own namespace, once per instance of that plugin in the network. Because each plugin is run in its own namespace, it obviates the need for state swapping. Instead, each instance of a plugin has its own copy of the entire executable associated with it, including code and data, ready for execution at any time. The lack of state swapping means there is less overhead from copying these states every time a different host is run, simulated process execution is simpler, and the build process is decoupled from LLVM/Clang to allow for use of other compilers and optimization levels.

In addition to the advantage of independent namespaces, all of a plugin's dependencies are also loaded into

[6] "Drow" (rhymes with "now") are a race of shadow elves from the game Dungeons and Dragons.

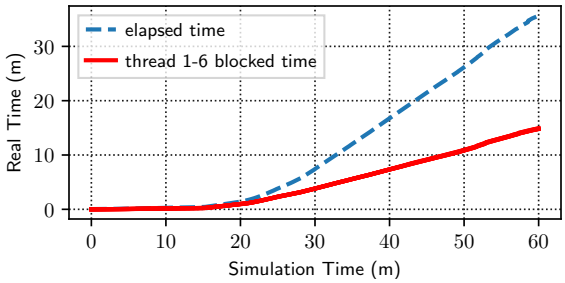


Figure 5: Amount of time each worker thread has spent blocked over the course of an experiment when using work stealing. Individual threads cannot be readily seen because their blocked times are nearly identical. Contrast with Figure 3.

its instance namespace. This isolation improves upon Shadow's original design, where only a single instance of each library is loaded for each worker thread, irrespective to the number of simulated processes making use of it. We benefit from greater assurances of correct behavior of applications by isolating each dependency's state, and we run Tor and OpenSSL code in parallel by removing the now-unnecessary global lock that was used to prevent parallel access from Tor to the OpenSSL library.

To reduce the additional memory overhead from these independent instances, all read-only sections of the executable make use of shared memory mappings. By doing so, the code of two processes that correspond to the same executable are backed by the same physical memory. This would occur automatically from Linux's Copy-on-Write memory mapping feature under normal circumstance, but we must do it explicitly because our dynamic linker must modify these sections in libraries that were linked to glibc for technical reasons [20].

Scheduling: One of the primary motivations for implementing the new dynamic loader functionality was to allow for better scheduling policies than those that Shadow previously supported. With the additional features provided by *drow-loader*, we were able to implement a means of dynamically migrating hosts from one thread to another to improve load balancing during execution.

The new scheduling algorithm is a form of work stealing [4, 22]. Just as with Shadow's original design, all hosts are distributed evenly across the available worker threads at the start of the experiment. As in Shadow's default scheduler, each worker thread executes every host assigned to it, until none of the hosts has any events left to process in this round. However, unlike in the original scheduler, once a worker thread has finished executing all hosts assigned to it, instead of blocking on the round barrier, it queries the lists of hosts assigned to other worker threads. If it finds there is a host on another worker thread that has yet to begin executing this round, it will remove the host from that list of hosts, insert it into its own, and begin executing its events for this round. This continues until every worker thread finds that no host has yet to begin execution this round and the round barrier is reached.

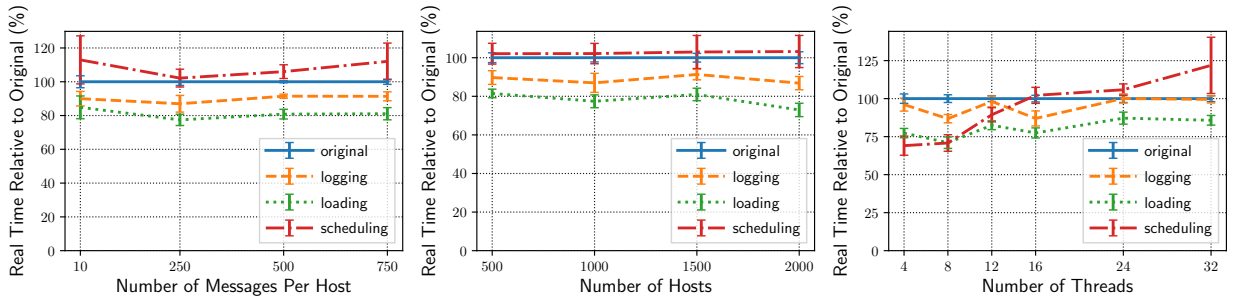
In this manner, we ensure that work is evenly distributed across worker threads; as can be seen by comparing Figure 5 with Figure 3, no thread spends significantly more time blocked than any other. Because hosts are migrated only when a worker thread is otherwise idle, we avoid much of the cost of migration and cache invalidation that would come with an algorithm more akin to a thread pool, where hosts are not *a priori* affiliated with any particular thread by the scheduler. This algorithm also naturally lends itself to the changing behavior of the simulation over time. For example, hosts that are particularly active at infrequent or irregular intervals will not skew the work distribution in chaotic ways.

5 Performance Benchmark

The parallel hold model (PHOLD) [7] is a standard parallel event model that is commonly used to benchmark discrete-event simulators. Each host in a traditional PHOLD benchmark enqueues a new event after a time that is usually drawn from an exponential distribution, and the destination host for the generated event is selected uniformly at random from all connected hosts. As a result, many performance studies that use PHOLD to evaluate symmetric networks lead to well-balanced workloads across hosts and are not closely representative of real-world networks [5]. Therefore, we use a variation on the standard PHOLD benchmark in our evaluation since we expect uneven workloads across Shadow hosts in practical use cases (e.g., Tor).

Setup: We write a benchmark application as a Shadow plugin. In our benchmark experiments, we configure a number of threads t, a number of hosts h, and a message load m. Each host starts a simulation by sending m messages to the other hosts, where the choice of the destination host is made according to a globally defined set of weights that are sampled from a Pareto distribution. Whenever any host receives a message, it generates a new message whose destination host is chosen according to the same set of globally defined weights. The t, h, and m parameters allow us to adjust the per-thread event density in Shadow, while the weights and the message reply logic ensure unbalanced workloads across hosts. We configure each experiment to run for 60 simulated seconds, and we track the real time to run each simulation.

We run our benchmark across a range of parameters with each of the *logging*, *loading*, and *scheduling* improvements (see Section 4) as they were cumulatively added to the *original* Shadow (in the respective order), and we repeat each experiment on three distinct machines with identical hardware profiles. The application



(a) Messages m (b) Hosts h (c) Threads t

Figure 6: Real time to complete our variation of the PHOLD benchmark experiments relative to the original Shadow version (with 95% confidence intervals) while varying the number of messages m in Figure 6a, the number of hosts h in Figure 6b, and the number of threads t in Figure 6c. The default values for the parameters not presented in each figure are: $m = 250$, $h = 1000$, and $t = 16$.

logic in the benchmark is minimal by design: the benchmark is meant to test the performance of the simulation engine in isolation and avoid any complexities introduced by the logic of applications being run in Shadow. While our benchmark allows us to better understand how our design changes affect performance in general, note that we also evaluate performance while running a private Tor network as a practical case study in Section 6.

Results: Figure 6 shows the benchmark performance of our improvements, relative to the performance of the *original* Shadow design, with 95% confidence intervals.

We find that our improvements reduce the real time to complete experiments relative to *original* similarly across the tested message parameter settings (Figure 6a), which is intuitive since an increase in simulation workload should equally affect each Shadow version. Increasing the number of hosts (Figure 6b) also has the effect of increasing the total number of messages (i.e., workload), and we observed a similar performance effect as expected. Varying the number of threads (Figure 6c) also shows similar results for all but *scheduling*, which showed a 31% performance increase with 4 threads (the best we measured) but a 22% performance decrease with 32 threads. We hypothesize that our scheduling improvement performed worse than the others because the overhead associated with migrating hosts to different threads was greater than any gain obtained from parallel execution: it is faster to run a host than migrate it because the computation in the benchmark plugin is minimal. This is not the case in more realistic applications (e.g., Tor), as we shall see in Section 6.

We find that *logging* and *loading* both significantly reduce the real time to complete experiments relative to *original* across all parameter settings. We observed no case that *logging* reduced performance, and in most cases saw an improvement of about 15%. Similarly, we observed that *loading* always outperforms *logging*, and that *loading* reduces real time by about 25% relative to *original* across parameter settings (and 29% in the best case).

6 Case Study: Tor

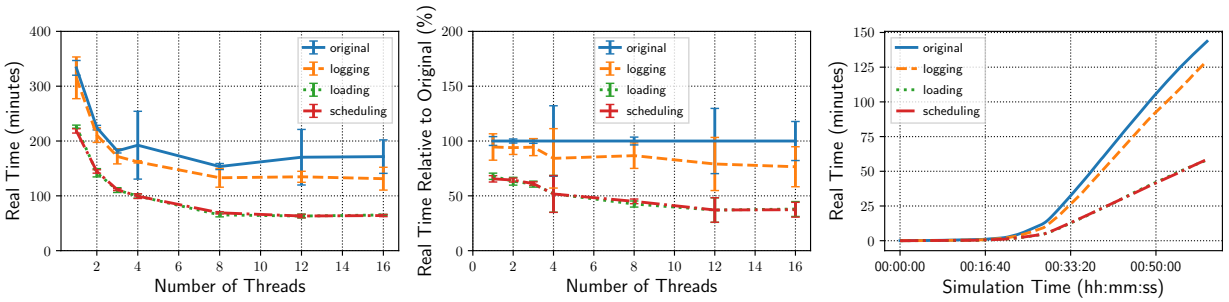
We now evaluate the performance of our Shadow improvements in both small- and large-scale Tor networks.

Small-scale Tor Network: One important use of Tor simulation is the testing of changes to the Tor code itself. These tests may be for basic correctness and performance consistency checks in order to ensure that such changes do not have unintended consequences. Such test networks will generally be smaller, and able to run on typical laptops at simulation rates that should match or exceed real time for fast turnaround of results. Here, we show the performance and memory impact of our changes on such networks.

The network was originally constructed such that one hour of simulation time took approximately one hour of real time in an older version of Shadow. It simulates one hour of simulation time for 268 hosts: 20 web servers, 18 Tor relays, and 230 Tor clients. Each client uses one of the preconfigured traffic behaviors that comes with the Shadow Tor plugin, designed to simulate web browsing, bulk file downloading, etc. To be clear, this network is only intended to represent the aforementioned consistency tests, and not any rigorous experimentation representative of the live Tor network.

The first 30 minutes of simulation time is designated for bootstrapping the network, during which each of the Tor directory authorities, relays, and clients start their simulated Tor process and clients start their file downloading processes. The start of each such process is distributed over time to avoid overloading the directory authorities, which are contacted by every Tor process as it starts. We consider that the network is in a ready state starting at simulated minute 30.

As in Section 5, we run the *original* version of Shadow as well as a version where each of the *logging*, *loading*, and *scheduling* improvements (see Section 4) were cumulatively added to Shadow (in the respective order). Each Shadow version was run by varying the number of



(a) Threads vs. Real Time (b) Threads vs. Real Time (Fraction) (c) Simulation Time vs. Real Time

Figure 7: Results from the small-scale Tor network with 268 hosts. Figure 7a shows the amount of time to complete the experiment on a varying number of threads. Figure 7b shows the same results, normalized to the *original* performance. Figure 7c shows the progression of a single run using 12 worker threads. The *loading* and *scheduling* lines are almost coincident on each of the plots.

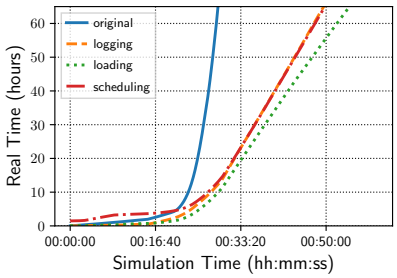


Figure 8: Amount of real time to reach a given simulation time, from the large-scale Tor network with 22,499 hosts. The *logging* and *scheduling* lines are almost coincident.

threads from 1 to 16, once on each of six machines with identical hardware specifications. The results are shown in Figure 7 with 95% confidence intervals.

As shown in Figure 7, our improvements significantly reduce the real time to complete the small-scale experiments. While the *logging* improvement alone improves performance by up to 30%, we measured the largest improvement of up to 73% from both the *loading* and *scheduling* versions. Based on these results, we found that the incremental improvement of *scheduling* over *loading* is insignificant in small-scale Tor networks.

Large-scale Tor Network: The other major use of Tor simulation is for testing design changes and research questions, where using networks that scale as closely as possible to the real, deployed Tor network is desirable. Towards that end, we measure the performance of a larger Tor network designed to accurately represent the performance characteristics of the real Tor network, establishing greater feasibility of running such experiments in an acceptable amount of time and memory overhead.

Our large Tor experiment consists of 22,499 hosts, including 1,500 web servers, 699 Tor relays, and 20,300 clients. We configure the experiment to run for 60 simulated minutes, where bootstrapping occurs during the first 30 minutes as in the small-scale experiments. The experiment was run for 65 hours of real time, at which point the progress was measured.

The results in Figure 8 show that our improvements significantly reduce the real time to run the large-scale Tor experiments. As in our benchmark experiments, even *logging* alone increases performance and *loading* provides a larger improvement than *scheduling*, though here only slightly.

7 Related Work

We focus our discussion in this section on related Tor research and development, as that is Shadow's primary use case. In Tor's early years, experimentation was done using single-use simulators that were written to analyze Tor performance when changing some specific part of Tor's design, such as adding a relay incentive scheme [11, 16]. Recognizing a need for a more general yet safe approach for conducting Tor research, Bauer *et al.* developed the ExperimenTor emulation testbed [3] and Jansen and Hopper developed the Shadow discrete-event network simulator [10]. Models of the Tor network that were useful for conducting Tor experiments with both ExperimenTor and Shadow were also developed soon thereafter [8]. Previous work has surveyed a more complete list of Tor experimentation tools [17].

Since these tools were developed, they have been used in a wide range of research analyzing Tor. Some of that work includes new path selection algorithms [23], changes to traffic scheduling [9] and load balancing [1, 13], and an analysis of denial-of-service attacks [12]. We refer the reader to previous work for a more complete survey of Tor research [2], much of which has experimented with Tor in some way.

The design of ExperimenTor and Shadow have changed over the years. SNEAC improves upon the performance of ExperimenTor by using Linux Containers

and the kernel's network emulation module `netem` [18], while NetMirage[3] improves upon SNEAC with more efficient tooling and improvements to the network representation. Shadow was enhanced to support important TCP features [9] as well as running processes that use multiple threads (a requirement for running Bitcoin) [14]. Our work adds performance enhancements to Shadow to further improve its utility.

Ns-3[2] is a related network simulator that has previously been used for Tor research in its normal mode of operation (i.e., abstract simulation) [21], but it focuses more on network protocol accuracy than scalability. Its *direct code execution* (DCE) mode of operation [19] is more closely comparable to Shadow; however, it has not been shown to directly execute Tor (as Shadow does) and we found significant performance issues in the loader used in DCE mode that we believe makes experiments larger than a few hundred hosts impractical.

8 Conclusion

In this work, we identified performance issues in the Shadow simulator and designed new logging, dynamic loading, and scheduling algorithms to overcome these issues. We have empirically demonstrated the performance benefits of each of our algorithms as they were cumulatively added to Shadow using parallel discrete-event simulation benchmarks and a more pertinent case study of the Tor network. In addition to showing significant performance benefits for Shadow experiments, we also eliminate correctness errors that were previously caused by plugins sharing a single global process namespace.

Acknowledgments

This work has been partially supported by the Office of Naval Research, by the National Science Foundation under grant number CNS-1527401, and by NSERC under grant number STPGP-463324. The views expressed in this work are strictly those of the authors and do not necessarily reflect the official policy or position of any employer or funding agency. This work benefited from the use of the CrySP RIPPLE Facility at the University of Waterloo.

References

[1] ALSABAH, M., BAUER, K., ELAHI, T., AND GOLDBERG, I. The path less travelled: Overcoming Tor's bottlenecks with traffic splitting. In *Privacy Enhancing Technologies Symposium (PETS)* (2013).

[2] ALSABAH, M., AND GOLDBERG, I. Performance and security improvements for Tor: A survey. *ACM Computing Surveys (CSUR) 49*, 2 (2016), 32.

[3] BAUER, K., SHERR, M., MCCOY, D., AND GRUNWALD, D. ExperimenTor: A Testbed for Safe and Realistic Tor Experimentation. In *USENIX Workshop on Cyber Security Experimentation and Test (CSET)* (2011).

[4] BLUMOFE, R. D., AND LEISERSON, C. E. Scheduling multithreaded computations by work stealing. *J. ACM 46*, 5 (Sept. 1999), 720–748.

[5] BONNET, V. Benchmarking parallel discrete event simulations. Master's thesis, Utrecht University, 2017.

[6] DINGLEDINE, R., MATHEWSON, N., AND SYVERSON, P. Tor: The second-generation onion router. In *USENIX Security Symposium (USENIX)* (2004).

[7] FUJIMOTO, R. M. Performance of time warp under synthetic workloads. In *SCS Multiconference on Distributed Simulation* (1990).

[8] JANSEN, R., BAUER, K., HOPPER, N., AND DINGLEDINE, R. Methodically modeling the Tor network. In *USENIX Workshop on Cyber Security Experimentation and Test (CSET)* (2012).

[9] 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 Symposium (USENIX)* (2014).

[10] JANSEN, R., AND HOPPER, N. Shadow: Running Tor in a box for accurate and efficient experimentation. In *Network and Distributed System Security Symposium (NDSS)* (2012). See also: https://shadow.github.io.

[11] JANSEN, R., HOPPER, N., AND KIM, Y. Recruiting new Tor relays with BRAIDS. In *ACM Conference on Computer and Communications Security (CCS)* (2010).

[12] JANSEN, R., TSCHORSCH, F., JOHNSON, A., AND SCHEUERMANN, B. The Sniper Attack: Anonymously Deanonymizing and Disabling the Tor Network. In *Network and Distributed System Security Symposium (NDSS)* (2014).

[13] JOHNSON, A., JANSEN, R., HOPPER, N., SEGAL, A., AND SYVERSON, P. PeerFlow: Secure load balancing in Tor. *Proceedings on Privacy Enhancing Technologies 2017*, 2 (2017), 74–94.

[14] MILLER, A., AND JANSEN, R. Shadow-Bitcoin: Scalable simulation via direct execution of multi-threaded applications. In *USENIX Workshop on Cyber Security Experimentation and Test (CSET)* (2015).

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

[16] NGAN, T.-W. J., DINGLEDINE, R., AND WALLACH, D. S. Building Incentives into Tor. In *Financial Cryptography and Data Security (FC)* (2010).

[17] SHIRAZI, F., GOEHRING, M., AND DIAZ, C. Tor experimentation tools. In *International Workshop on Privacy Engineering (IWPE)* (2015).

[18] SINGH, S. Large-scale emulation of anonymous communication networks. Master's thesis, University of Waterloo, 2014.

[19] TAZAKI, H., UARBANI, F., MANCINI, E., LACAGE, M., CAMARA, D., TURLETTI, T., AND DABBOUS, W. Direct code execution: Revisiting library os architecture for reproducible network experiments. In *Proceedings of the Ninth ACM Conference on Emerging Networking Experiments and Technologies* (2013).

[20] TRACEY, J. Building a Better Tor Experimentation Platform from the Magic of Dynamic ELFs. Master's thesis, University of Waterloo, 2017. https://hdl.handle.net/10012/12602.

[21] TSCHORSCH, F., AND SCHEUERMANN, B. Mind the gap: Towards a backpressure-based transport protocol for the Tor network. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)* (2016).

[22] VANDEVOORDE, M. T., AND ROBERTS, E. S. Workcrews: An abstraction for controlling parallelism. *International Journal of Parallel Programming 17*, 4 (1988), 347–366.

[23] WACEK, C., TAN, H., BAUER, K., AND SHERR, M. An empirical evaluation of relay selection in Tor. In *Network and Distributed System Security Symposium (NDSS)* (2013).