Building a Better Tor Experimentation Platform from the Magic of Dynamic ELFs

by

Justin Tracey

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2017

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

Tor is the most popular tool for providing online anonymity. It is used by journalists, activists, and privacy-conscious individuals to provide low-latency private access to the Internet. However, Tor's specific design and implementation is constantly changing to improve the performance and privacy properties it seeks to provide. To test these improvements, some form of experimentation is needed. Running experiments directly on the real Tor network is often not a viable option. The users of Tor are using it presumably because of its privacy protections, and caution must be taken to avoid recording or revealing information from non-consenting parties, particularly when dealing with shortcomings in Tor's privacy protections or using new, untested versions of Tor. Because of the need for reproducible experiments and the aforementioned ethical requirements surrounding Tor experimentation, it is often necessary to use artificially constructed Tor networks.

Several tools are available to construct such networks, such as network emulators like NetMirage, and simulators like Shadow. However, these existing tools do not provide the scalability that would be desirable when running experiments on these networks — with emulators requiring hardware capable of running all hosts in real time simultaneously, and with Shadow (the only maintained network simulator capable of running Tor code) having performance constrained by early design decisions. Since the behavior of a network can change with its size, it is better to use larger networks that more closely resemble the size of the real deployed network. Additionally, the ability to test the functional correctness of a modification to the Tor source code is considerably simpler when there is a means of quickly experimenting on a virtual Tor network to run such tests.

In both of these cases, a higher-performance testing platform is needed. To address this shortcoming, for this thesis we designed and implemented a new model of Tor network simulation, centered around a modified version of the Shadow network simulator, using large numbers of dynamically loaded binaries. This is accomplished by implementing a custom dynamic loader, which we call drow-loader, that allows for dynamically loading more binaries than any other dynamic loader that we are aware of, and with better performance. By using the features of this dynamic loader, we are able to run simulated processes isolated in "namespaces". This allows for reduced lock contention, simpler process modeling, and the ability to migrate simulated processes between worker threads. Using simulated Tor networks ranging from hundreds to tens of thousands of hosts, we then demonstrate the performance improvements our simulation technique provides over the state of the art.

Acknowledgements

I would like to thank my supervisor Ian Goldberg, for his advising and considerable help with this research. I would also like to thank Rob Jansen for his time and assistance with Shadow, as well as comments on this thesis. Finally, I would like to thank my thesis committee members, Urs Hengartner and Tim Brecht, for their valuable comments during the thesis revisions process.

This work benefited from the use of the CrySP RIPPLE Facility at the University of Waterloo.

Table of Contents

List of Tables vii

List of Figures viii

1 Introduction 1

1.1 The Tor Anonymity Network . . . 1

1.2 Tor Experimentation . . . 3

1.3 Contributions . . . 4

2 Related Work 6

2.1 Deployed Networks . . . 6

2.2 Emulators . . . 7

2.3 Network Simulators . . . 8

2.3.1 ns-3 . . . 8

2.3.2 DCE . . . 9

2.3.3 Shadow . . . 9

3 Current State of the Art 10

3.1 Dynamic Linking and Loading . . . 10

3.1.1 Symbol Resolution . . . 12

3.1.2 Thread-Local Storage . . . 14

3.1.3 elf-loader . . . 15
3.2 Shadow . . . 17
3.2.1 Shortcomings . . . 21

4 Improving Tor Simulation **26**
4.1 Architecture . . . 26
4.2 drow-loader . . . 28
4.2.1 Correctness . . . 28
4.2.2 Performance . . . 28
4.2.3 Concurrency . . . 30
4.2.4 Memory Overhead . . . 30
4.2.5 Run-time Symbol Interposition . . . 31
4.2.6 Initialization-Time Configurable Static TLS . . . 33
4.2.7 TLS Migration . . . 35
4.3 Scheduling . . . 36
4.4 Experimental Results . . . 37
4.4.1 Setup . . . 37
4.4.2 drow-loader . . . 39
4.4.3 Shadow . . . 40

5 Future Work **48**

6 Conclusion **50**

Bibliography **51**

List of Tables

4.1 Dynamic loader performance 40

List of Figures

1.1 Tor from a very high level 2

3.1 data structures used in TLS 16
3.2 Logical structure of Shadow 19
3.3 Shadow's state swapping technique 20
3.4 Why Shadow cannot migrate hosts if state swapping is used 23
3.5 Amount of time each worker thread has spent blocked over the course of an experiment 25

4.1 Our new design 27
4.2 Shared memory backing 32
4.3 Amount of time blocked when using work stealing 38
4.4 Thread scalability 42
4.5 Normalized thread scalability 43
4.6 Pace of experiment 44
4.7 Normalized pace of experiment 45
4.8 larger experiment 47

Chapter 1

Introduction

1.1 The Tor Anonymity Network

Pervasive Internet connectivity has been integrated into the daily lives of billions of people. As it exists today, the Internet does not inherently provide anonymity to users' connections. Without some sort of extra care taken, every party that is involved in an Internet connection — be it the destination, the source, or intermediaries on the path of the connection — can identify the IP address of the source and the destination. One could conceive of many situations where this lack of privacy is a cause for concern. For example, someone who suffers from a medical ailment may not wish for the operator of a self-care site to know their identity, nor would a network of dissidents wish to have their identities known to the state.

For years, Tor has been the most popular tool to provide additional privacy to Internet connections. One reason for its popularity is the low latency of Tor. Tor was designed to provide acceptable levels of overhead for typical Internet use cases, such as web browsing, instant messaging, or audio and video streaming. To achieve this low overhead, Tor uses a technique known as *onion routing*. A user runs a Tor client (often referred to as an *onion proxy*) to establish a series of connections encrypted in layers to the volunteers running the Tor software configured in such a way to allow it to be used as an intermediary connection (an *onion router*, or *relay*). After the desired number of "hops" are made (typically three), the final connection to the endpoint is made from the last relay. Using this technique, no point on the route of the connection should be able to simultaneously establish the true source and ultimate destination of the connection. This simple design allows for both the privacy and performance goals Tor seeks to achieve.

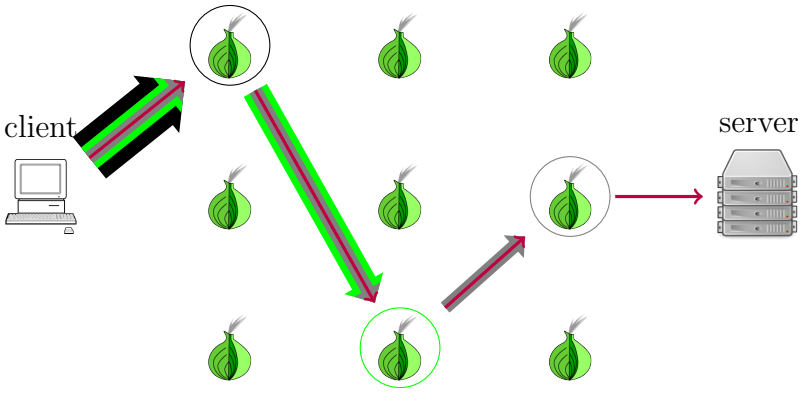

Figure 1.1: The Tor anonymity network functions by routing traffic from a client through several relays to the end destination.

However, it is important to note that Tor does not achieve absolute success in these desired goals. Tor has no formal proof of correctness of its design or implementation that indisputably establishes its privacy goals, or that it has no security vulnerabilities (being networked software connecting to unvetted volunteers, securely handling untrusted input is a crucial aspect of Tor's privacy goals). In fact, Tor and its associated tools (such as Tor Browser, the official web browser for use with a Tor onion proxy) are regularly found to have flaws, [AG16] and the Tor Project will promptly push updates to address them. Similarly, the performance of Tor, while good enough for many use cases and better than many alternatives, is noticeably worse than that of a non-anonymous Internet connection. It is therefore common for research and development on Tor and associated tools to focus on improving performance. Doing so is not only desirable for its own right in usability, but also because the developers of Tor assert that the number of Tor users is correlated to the peak performance of the network [Tor17b; DM09]. Therefore, improving the performance of Tor increases the size of its anonymity set, thereby improving the privacy of those who use Tor.

Since recording began in September 2011, the estimated number of concurrently and directly connected Tor clients has ranged from five hundred thousand to almost six million, with current estimates at around two million [Tor17b]. Tor relays, which have statistics since 2008, have varied in number from 1,000 to 8,000, with approximately 7,000 running as of August 2017.

1.2 Tor Experimentation

In order to demonstrate which changes in a network such as Tor would improve it, or which attacks would subvert its privacy goals or security, one must construct and perform some sort of experiment. The form that this experiment takes depends on the goals of the experimenter.

Naïvely, a researcher may attempt to perform their experiments on the actual network. Beyond the problems of a lack of a controlled environment and the difficulty in creating reproducible results this technique presents, the nature of Tor makes this sort of research antithetical to the goals of the network in many cases. One particularly well known example of this is an incident where data from unpublished research by Carnegie Mellon University was used in an FBI investigation to deanonymize Tor users — resulting in statements from members of the Tor Project and research community that this was unethical behavior by the researchers [Gre15; Fel14]. Because of the sensitive nature of Tor research, the Tor Project has a series of strict guidelines on what constitutes acceptable research methods on Tor [Tor16]. The first of their guidelines is "Use a test Tor network whenever possible." Therefore, in order to protect the users of the Tor network and to act in accordance with the wishes of the Tor project, having a means of testing Tor modifications and attacks on a separate network is necessary.

There exist ways of setting up test networks on real or emulated networks. For example, one could construct a network of virtual machines, each running Tor and configured to run one of the necessary components of the Tor network. Similarly, one could run Chutney [Tor17a], which is a means of running multiple instances of Tor on one machine. However, these techniques do not easily scale natively, as the networks must be set up manually. To facilitate emulation, there exist projects such as Mininet [LHM10], ExperimenTor [Bau+11], SNEAC [Sin14], and NetMirage [Ung17] that assist in setting up larger emulated networks. Additionally, there are real large-scale test networks one could run a Tor network on, such as PlanetLab [Chu+03]. These emulation testbeds, while potentially effective, still have the problems of large overhead, and the associated problems of considerable hardware requirements and difficulty in scaling. Additionally, the reliance on real-time emulation can have effects on the reproducibility of experiments, as they will rely heavily on the actual conditions in which the experiment was run.

The other means of creating a test network is that of simulated networks, using tools such as ns-3 [ns317b; Hen+08]. Because simulators isolate real, wall-clock time from the in-simulation time, simulations are more isolated from their host environments. While simulation allows for experiments that do not depend on actual conditions of the machine

running the experiment, and in some cases can be more scalable, they typically have the drawback of less realism compared to emulated or real test networks, as they do not truly run the software.

To address this shortcoming of simulators, tools were made that allow execution of native code in the simulation, such as DCE [ns317a; Taz+13] and Shadow [JH12]. However, there have been no documented cases of anyone successfully using native Tor source code with DCE [Wil14]. Furthermore, because all other simulators created with Tor as the intended use case are focused on a single aspect of Tor (e.g., TorPS [Joh+13; Joh+17] and COGS [Ela+12]), and because of the advantages of network simulations, Shadow is one of the most commonly used tools for running experiments for Tor research [SGD15].

In order for results on the performance of modifications to Tor code, or on the efficacy of new attacks on Tor, to be valid, they must be tested on networks that closely resemble the Tor network that is used in reality. This means that the network that was used for testing should be of a similar size and topology of the Tor network in use in the real world. Shadow, as a discrete-event network simulator, has a distinct advantage over emulators and real networks in this aspect, as it is capable of running network simulations that far exceed the actual computational and network capacities of the machine used to host the experiment. However, Shadow currently has difficulty maintaining high performance on such large networks. In particular, even on large compute clusters, the performance of Shadow does not improve significantly, or in some cases at all, over typical desktop hardware. This is the case despite the problem of large network simulation at least conceptually being a task well suited for the massive parallelism that some systems available to researchers provide.

1.3 Contributions

In this thesis, we aim to demonstrate the veracity of the following thesis statement:

> We can construct a new platform for simulating Tor networks that allows for greater performance than existing techniques, which makes existing experiments run more quickly, and more realistically sized Tor experiments more feasible.

To demonstrate this, we implemented our technique by modifying the Shadow network simulator. In particular, we have provided:

- A custom replacement to the dynamic loader for Shadow. By replacing this user-space component of the operating system, we were able to revisit earlier design decisions used by Shadow, and we were able to use a new method of representing simulated process execution, which in turn removed performance bottlenecks, reduced memory usage of large experiments, and assured greater simulation realism.

- A run-time scheduler for Shadow host-to-thread assignment. Building on the functionality provided by the new dynamic loader, this scheduler allows the assignment of simulated hosts to threads while the experiment runs, rather than when the simulation initializes, before the respective computational intensity of the provided hosts is known.

- A series of experiments that demonstrate the improved performance of Shadow with these modifications on simulated Tor networks ranging from hundreds to tens of thousands of hosts.

- Public access to our code,[1] so that other researchers may use it to run larger experiments, or moderately sized experiments more quickly.

The rest of this thesis is organized as follows: we give an overview of the existing state of Tor experimentation tools, and network simulation as it applies to Tor, in Chapter 2. Chapter 3 provides the necessary background to understand our contributions, including a design overview of Shadow. In Chapter 4, we detail the methods used in creating our new design, and the reasoning behind them. We also show the results of implementing our design, comparing the performance of Tor simulations using our implementation to the previous state of the art. In Chapter 5, we provide some potential avenues for future research regarding Tor simulations. Finally, we give our concluding remarks in Chapter 6.

[1] Available at https://github.com/shadow/shadow/.

Chapter 2

Related Work

In this chapter, we examine some of the alternatives to Shadow for Tor experiments. Shadow, being more directly related to our work, is detailed in the next chapter, in Section 3.2.

2.1 Deployed Networks

Conceptually, the simplest means of testing networked software such as Tor is to deploy it on an actual network. That is to say, one can simply install Tor clients and relays on several machines that are networked, for example, via LAN or Internet connections, and run them. Perhaps the most immediately obvious problem with this experimental setup is that in order to run an experiment on a network that approaches the size and topology of the real Tor network, it would require access to large numbers of machines, connected all over the planet. Rather than attempt this feat individually, there exist organized testbeds for researchers to share that provide such a network.

One prominent example of such a testbed available for researchers to use is PlanetLab [Chu+03]. Using PlanetLab, researchers are assigned, for a finite time, a "slice" of the entire network to run their experiments on. With 1353 nodes at 717 sites [Pri17] (as of August 2017), PlanetLab allows for access to larger networks than an individual researcher or research group could readily construct on their own. Although this number of nodes could allow for networks within an order of magnitude of the number of deployed Tor relays, it still fails to provide networks that approach the size of the actual Tor network (which would include clients, and for testing, servers to communicate with).

Because of the nature of this style of testbed, the reproducibility of experiments suffers on PlanetLab and similar networks. Experiments that run on PlanetLab suffer from the unpredictability of other experiments concurrently running on the testbed, as well as the variable conditions of the Internet connections that exist between the respective nodes in the network. Furthermore, these conditions are not only variable and unpredictable, but do not permit for intentional changes in network configurations, to model the real Tor network or potential changes in it. For example, if the researcher wished to test the behavior of Tor on a connection between two specific locations that have no sites available on the PlanetLab network, then there is no means to use PlanetLab to do so — that is, there is no way to explicitly specify corresponding latencies, bandwidth, packet loss and jitter to a connection between these locations, and any attempt to mimic them is subject to changes in real network conditions, such as changes in network paths, over which the researcher has no control.

2.2 Emulators

Network emulators are tools that allow for networked software to be run on a virtual network. Unlike real networks, emulated networks can be significantly larger than the number of physical machines they run on.

The architecture of network emulators used for Tor experimentation, such as ModelNet, SNEAC, and NetMirage, are very similar (mainly because the design of SNEAC was based on that of ModelNet, and NetMirage is based on SNEAC). Some number of "edge node" machines, each of which run multiple instances of the Tor software, web servers, etc., are connected through a single "core". The core is configured to simulate the behavior of the network between the instances of each emulated host (bandwidth, latency, packet loss, etc.).

In contrast to network simulators, emulators run the software that comprises the network, with the measured timing information being the actual time of the experiment. For this reason, the limiting constraint of an emulated network is, at best, the amount of resources available for the emulated software to run without impacting the performance of other instances. Furthermore, emulators are directly impacted by real-world conditions. It is unlikely that the results from an emulated experiment would be perfectly reproducible across distinct emulation setups, as factors such as the specific CPU qualities, storage medium, or even ambient temperature could impact the behavior of the emulated network. This variability reduces the precision of results, making it less certain that measured effects

are the result of changes in controlled variables (as it could instead be as a result of noise in local conditions).

It is not uncommon for emulated networks to be seen as being more realistic than simulated networks, as they are truly running a network, and not a simplified model [SGD15]. More specifically, because they rely on the native network stack of the underlying operating system instead of a model of it (as well as all other parts of the OS), it is assumed there will be fewer discrepancies between the experimental setup and actual execution of the software. In practice, however, this is not always the case, and it is not uncommon for emulators to rely on features of the underlying OS that are either not designed to be used in emulation, or contain bugs [BFP15]. Therefore, despite their reputation, emulators are not always more realistic than a simulated network.

2.3 Network Simulators

2.3.1 ns-3

Commonly used in research from the network community are network simulators. Unlike network emulators, network simulators are typically designed to model the idealized behavior of the network, rather than run it directly. One of the most popular of the available network simulation tools is ns-3. Indeed, there are some papers that have used ns-3 to simulate Tor, so as to test modifications to the design of the network [TS11; TS13; TS16]. However, because of the complexity of Tor's implementation, and the number of implementation details that are not documented outside of the C code it is written in, the accuracy of these types of experiments, as they relate to the real Tor network, has been called into question [JH12; Ged16].

ns-3, as well as ns-1 and ns-2 before it, are discrete-event network simulators. This means that time is broken into discrete intervals, with "events" coordinated between them such that causality is preserved. Because of this discrete nature, the time of the events of the simulation can be isolated from the time of the events that they truly occur in. For instance, one could conceivably suspend the experiment for an arbitrary amount of time, then resume it, or change the available resources on the machine the experiment is running on, without affecting the results of the experiment. Similarly, the results of an experiment can be made fully deterministic, by ensuring any randomness used in the application and simulated conditions are seeded for such determinism. If this is done, the conditions in which one starts or runs an experiment will not have an effect on the behavior of the experiment, making reproducibility of results far simpler than real or emulated networks.

Because it is more relevant for the purposes of this thesis, details of the functionality of an example discrete-event network simulator can be found in the relevant section on Shadow (Section 3.2).

2.3.2 DCE

One means of bridging the gap between network simulators and emulators are simulators that execute the code of the software that is being simulated, instead of using simplified models of their behavior. The result is still a simulation and not an emulation, because simulated time is still distinct from actual wall clock time, and because some set of system interactions (such as network calls) are mediated through the simulator. DCE ("Direct Code Execution") is a module for ns-3 designed to allow ns-3 to do exactly this [ns317a; Taz+13]. It can run some simple, but commonly used network applications, such as ping and iperf. However, despite some interest, DCE has not been made to function with the Tor source code [Wil14]. Because little progress has been made on this front, precisely how much work on Tor or DCE would be required to make the two compatible is unknown. At the very least, either the many instances of `clock_gettime()` would have to be removed from Tor, or proper support for this system call would have to be added to DCE. Furthermore, there has yet to be conclusive results for the scalability of DCE. Indeed, our implementation largely originates from modifications to the "elf-loader" dynamic loader that was originally a higher-performance, non-default option provided by DCE, which we found to be insufficient to feasibly be used for large simulations in its original state. The original papers that demonstrate DCE and elf-loader provide no results for experiments with more than 65 nodes in their networks [Taz+13; Lac10] — far short of the thousands of relays in the real Tor network.

2.3.3 Shadow

Shadow is also a discrete-event network simulator that allows for running real applications [JH12]. While it is generally less popular than ns-3 itself, it is the most commonly used network simulator for Tor experiments [SGD15]. Because it is the current state of the art, and because it is what we ultimately derive our implementation from, we go into greater detail on Shadow's design and use in Section 3.2.

Chapter 3

Current State of the Art

This chapter provides some background necessary to understanding our design for network simulation. We first examine the state of the art with regards to dynamic linking and loading — a central technique of our design. We then detail the design of Shadow — the most commonly used network simulator for Tor, and the code base our implementation is built on.

3.1 Dynamic Linking and Loading

The primary contributions to improving Shadow's performance are obtained via a custom dynamic loader. Before describing the advantages of a dynamic loader specialized to network simulation, we will describe the functionality a dynamic loader provides. While many of the concepts described here are generalizable, we will focus on the specific case of GNU/Linux running ELF executables. That is to say, a system using the Linux kernel, with GNU components in the user space; the most important such component is glibc — the GNU implementation of the C standard library.

As most programmers likely know, nearly all programs use pre-written code called libraries, so as to avoid re-implementing the same functionality in every program (e.g., printing text to the screen, performing cryptographic operations, etc.). The functionalities that libraries provide are commonly thought of as being included into the program when it is being built, or rather "linked", via one of two methods: static linking, or dynamic linking.

Static linking can be thought of as simply copying and pasting the library code into the code of the program that makes use of the library. While simple, it is also redundant, and more difficult to maintain across projects that use the same library. It is also not particularly relevant to the rest of this thesis, aside from contrasting it to dynamic linking.

In *dynamic linking*, libraries are themselves compiled into machine-readable binaries. These binaries are called "shared libraries" generally, but in the case of ELF, they are referred to as *shared objects*, and typically have a `.so` file extension. (Standalone executables and shared objects are collectively referred to as *objects*.) When building an executable, the compiler and linker will add the names of these shared objects to the ELF file, so that they may be used at the time of execution.

When running a modern operating system, it is rare that an executable is invoked directly. Instead, a program called the *dynamic linker* is run. At start-up time, the dynamic linker loads the desired executable into memory, as well as all of the dependencies of the program listed in the ELF file. Each of these dependencies can be loaded at any memory address, so long as there is enough contiguous space at that address to store the entire uncompressed ELF file (to allow for this position independence, the locations of many values in the executable are stored as offsets from the beginning of the file or from the instruction pointer, so all segments must be contiguous). Loading dependencies via this technique has several advantages. For example, suppose a shared object were updated, because a bug in a function was fixed in the corresponding library. Since the new version of the library would be loaded at the next execution of the program, all programs that use that shared object would also have that bug fixed, without rebuilding each of those programs individually. Additionally, only one instance of the library needs to be stored, regardless of the number of programs that use it.

Static and dynamic linking are the common techniques of building an executable linked to specific libraries. However, the above two methods are not the only means of attaining library functionality in a program. Another means of executing library code is *dynamic loading*. In dynamic loading, the desired library is loaded into memory during execution. In the case of C code on Unix-like systems, the code may provide the path of the shared object as an argument to the `dlopen()` function call, from the `dl` library. When using dynamic loading, the program makes use of functionality defined in another program called the *dynamic loader*. The dynamic loader will perform some sort of dynamic loading operation, such as loading a shared object into memory and returning a handle to it, as is the case for `dlopen()`, or resolving a symbol with a specific name and returning the pointer to that symbol, as is the case for `dlsym()` (see Section 3.1.1 for details on what is meant by "resolving").

Because this functionality is fundamentally an extension of the services the dynamic linker provides, in practice, the dynamic linker and dynamic loader are the same program. Whether one uses the term "dynamic linker" or "dynamic loader" may therefore depend on the context in which one is using it, but the terms are often used interchangeably. The GNU dynamic loader is implemented as part of the glibc code base, and typically referred to as "`ld.so`", though the name of the actual executable on a system may vary.

3.1.1 Symbol Resolution

One of the primary roles of a dynamic linker/loader is to resolve symbols. A *symbol* is a unit of information used by the linker — as well as the dynamic linker, and anything else that makes use of linker-relevant information. A symbol could, for example, represent a function, a global variable, or a file section. Some symbol types are documented in open standards, while others are implementation specific.

As part of the ELF specification, certain sections of an ELF file list all of the available symbols in what is appropriately called a *symbol table*. When the dynamic linker needs to resolve a symbol, it will begin to search through the symbol table of each shared object in the requester's scope. In the simplest cases of dynamic linking, this scope will be all of the shared objects that the requester is linked against. When the dynamic linker finds the requested symbol, it adjusts a data structure known as the Global Offset Table (GOT) of the requester, so that all future lookups of that symbol through this table will directly point to the correct address of the symbol.

When and why this lookup occurs varies based on factors such as compilation options, run-time flags, and use cases. For example, suppose an executable has been compiled with the "lazy loading" option enabled, and made use of the `time()` function by linking against glibc. Upon the first invocation of `time()` from the executable, it will use the GOT to get a function pointer, which will point to the dynamic loader's symbol lookup function. The dynamic loader will then iterate through each of the shared objects the requesting object was linked against, in the order the linker specified, until it finds one that defines a symbol with the name `time`, which modulo any additional complications, will be in glibc.[1] It will then modify the requester's GOT so that the relevant entry no longer points to the dynamic loader, but the address in memory at which glibc's copy of `time()` was loaded.

[1] For the sake of simplicity, we are ignoring additional checks made and implementation details, such as version numbers, how the GOT functions, and how the dynamic linker checks for a symbol in an object. See "How To Write Shared Libraries" [Dre11] for more detail on the specifics of glibc's symbol resolution procedure.

It then calls this requested function. On the other hand, if lazy loading was not enabled, this lookup would have occurred when the requesting object was first loaded.

One common way to affect how symbol resolution occurs is through the `LD_PRELOAD` environment variable. If the `LD_PRELOAD` environment variable is set to a valid shared object (or a list of shared objects), the dynamic linker will add it to the front of the scope for all symbol lookups. This means that if the `LD_PRELOAD` shared object defines a symbol with the same name (and, optionally, version) as a symbol in a shared object that was linked against, the symbol in the `LD_PRELOAD` shared object will be used instead of the definition in the linked library.

For example, suppose the previous example had `LD_PRELOAD` set to a shared object with a symbol corresponding to something to the effect of the following function:

```
time_t time() {
  return 7;
}
```

Then every call to the `time()` function would return not the current time, as typically expected, but instead the number 7. This effect is called *interposition* — the function definition in the `LD_PRELOAD` library interposes the function definition in glibc.

One more modification to scope resolution that must be explained is `dlmopen()`, as opposed to `dlopen()`. Under normal conditions, when loading a library into memory (whether this is due to dynamic linking, dynamic loading, or even static linking), there is no reason for the library to be loaded more than once. A global variable should only ever be defined once, a library function should only have one definition, etc.. As such, attempts to link or load a shared object multiple times will usually result in linking to the same library or returning the same handle as the first load of that library (respectively). For example, code that both dynamically links against glibc, and loads glibc using `dlopen()`, will only have one instance of glibc in the program's memory, including only one instance of glibc's internal state.

However, there are cases where this is not the desired behavior — particularly with dynamic loading, where it could be the intended behavior for the shared object to be loaded multiple times, with isolated state for each. To achieve this, some dynamic loaders (including glibc's) provide the non-POSIX extension of `dlmopen()`. `dlmopen()` behaves largely the same as `dlopen()`, with the caveat that it takes an additional argument of a *namespace*. A namespace is a symbol resolution scope that is isolated from the default

scope in which the program exists.[2] By loading a shared object in different namespaces, the state of one instance of the shared object in memory will be distinct from the state of another instance of the same shared object that was loaded in a separate namespace. This will prove useful for our purposes of network simulation, in that it can be used to create isolated instances of a shared object to represent distinct simulated processes.

3.1.2 Thread-Local Storage

One particular symbol type that behaves differently from all others is *Thread-Local Storage*, or TLS. TLS are values that are specific to a particular thread; that is, one thread's instance of a TLS value is independent of another thread's. There are many methods of implementing TLS, such as the POSIX thread key-value stores, but for this thesis, the relevant technique is that of TLS variable definitions.

This method of TLS was designed for the C and C++ languages, but it may be used by any language that compiles to a compatible ABI[3]. Originally, it was a non-standard extension to the C/C++ languages, utilized, for example, via the `__thread` keyword in GNU's extensions to C/C++. It has since been added to the C11 and C++11 official standards, via the `_Thread_local` and `thread_local` keywords in C [ISO11a] and C++ [ISO11b], respectively. As an explicit example, a GNU C program that defines a global variable as "`__thread int x;`" will have a global variable `x`, whose associated memory (and therefore value) is specific to each POSIX thread.

Unlike other symbols in shared objects, which can be loaded at an arbitrary location in memory (so long as they are in their respective, contiguously allocated file section), the dynamic loader must keep track of TLS allocations in a more standardized fashion [Dre13]. This is due, in part, to the fact that these symbols must be stored in such a way that is ABI compatible with the POSIX thread implementation. By ensuring that the linker, the dynamic linker, and the POSIX thread implementation all use compatible representations of TLS symbols, features associated with TLS (such as their allocation/deallocation during the construction/destruction of a thread, respectively) can be safely implemented.

A shared object may use one of two types of TLS models (both of which are available on each POSIX thread): dynamic TLS or static TLS [Dre13]. *Dynamic TLS* is the default

[2]There are some exceptions to this isolation, such as the dynamic linker itself, but they are minor enough to be ignored for our purposes.

[3]An ABI (Application Binary Interface) is the set of constraints on compiled code to interoperate with other compiled code, such as the exact data alignment and layout of parameters to correctly call a function on them.

form of TLS, and is stored internally as a dynamically allocated vector of pointers that grows and shrinks as objects are loaded and unloaded. *Static TLS* is TLS that has been marked as incompatible with dynamic loading. Because it is not to be loaded during execution, its size is known at the time of program initialization (before execution, while the dynamic linker loads dynamically linked libraries into memory), and it is allocated as part of the POSIX thread data structure. Since the only advantage it provides is one less layer of indirection (and therefore faster access to the data), and it is not the default behavior, static TLS is rarely used by shared objects. The specifics of the memory layout and data structures used can be seen in Figure 3.1.

3.1.3 elf-loader

As mentioned previously, the dynamic loader of GNU/Linux (`ld.so`) is implemented as part of the glibc code base. To be clear, despite glibc being an implementation of the C standard library, and `ld.so` being a part of glibc, `ld.so` is a dynamic linker/loader, and is therefore a standalone executable (as well as a shared object). In fact, it can be executed from the command line; though as the output of doing so says, this is rarely desired. Instead, the `PT_INTERP` field of an ELF executable contains the path of the dynamic linker.[4] The operating system's program loader reads this field, loads the file it points to, and executes it [TIS95]. As such, the dynamic loader used can be set at compile time (or, for that matter, modified in an existing binary, if the replacement path string is no longer than the original). Doing so, however, is very unusual, as any replacement dynamic loader would require ABI compatibility with the libc implementation (i.e., glibc), and if debugger support is required, the debugger (e.g., gdb, which is written under the assumption that the glibc dynamic linker, and therefore the glibc ABI, is used). The ABI cannot be changed, as doing so would require recompiling glibc, and then relinking all programs and libraries that make use of it[5] (a practically infeasible task for most users, as the C standard library is linked to most applications and libraries). It would then stand to reason that for a custom dynamic loader to be valuable, the existing dynamic loader would have to be missing functionality that is very useful (otherwise we would make do without it), but not generalizable enough to be implemented upstream (or else we would simply make the necessary modifications and submit them to the glibc maintainers).

[4] Strictly speaking, this field points to the "Program Interpreter", which is the program that ensures the executable is loaded at the correct virtual address. In practice, this is the same program as the dynamic linker (and therefore, the dynamic loader as well).

[5] While some changes can be made to a library without requiring relinking dynamically linked programs, ABI changes do require relinking, as they change how the symbols are used.

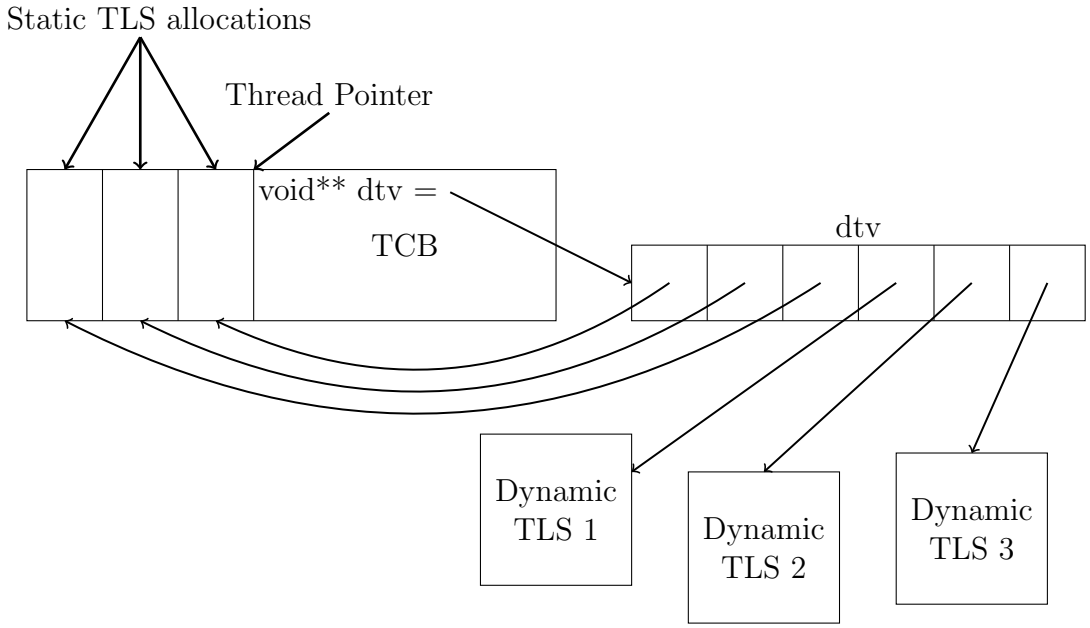

Figure 3.1: The data structures used in TLS for x86-64 POSIX threads [Dre13]. The thread pointer points to the start of a standard data structure called the TCB (Thread Control Block). The first field of this structure points to the dtv (Dynamic Thread Vector). The dtv consists of pointers to each object's actual TLS allocation. In the case of Static TLS, these allocations must be in the contiguous memory immediately before the thread pointer (or equivalently, before the TCB), so that they may be accessed without using the dtv (as offsets from the thread pointer register, %fs). In the case of Dynamic TLS, these allocations can instead be at arbitrary memory locations, with access done as an instance of symbol resolution, through the dynamic linker. While the dtv can be changed and reallocated, the thread pointer should not be modified, and therefore all Static TLS is allocated at thread initialization.

One such limitation of the glibc dynamic loader is that its implementation of `dlmopen` is limited to 16 concurrent namespaces. This number is somewhat arbitrary, in that it is simply the size of a statically allocated array that stores all namespaces. But while the number could conceivably be increased in the code, glibc is written assuming a statically allocated array of namespaces, and does not have any mechanism of dynamically allocating this array at run or initialization time. Furthermore, significantly increasing this number would add a non-trivial amount of overhead to all programs dynamically linked to glibc — a cost the developers are unlikely to be willing to make just so a specific use case of network simulation can make use of it. Therefore, for the foreseeable future, namespaces in glibc will remain some constant number, set at the compile time of glibc (again, recompiling then relinking for our own purposes being an overly burdensome task on the user).

The small bound on the number of calls to `dlmopen` was the primary motivation for the creation of a custom dynamic loader, called "elf-loader", as part of the DCE project [Taz+13]. The developers of elf-loader found that modifying glibc to allow for more namespaces dynamically would be prohibitively difficult, and therefore opted to construct an entirely new (though again, glibc ABI compatible) dynamic linker.

elf-loader is an optional alternative to DCE's default mechanism of swapping global state (similar to the method Shadow used, see below in Section 3.2). The developers of DCE found using elf-loader in DCE improved memory usage and greatly improved performance over state swapping by simplifying simulated process representations [Taz+13; Lac10], but it is not used by default in DCE because of unspecified "issues" [ns313].

3.2 Shadow

Shadow is a discrete-event network simulator, but with the original design goal of running Tor code. Accordingly, Shadow is more directly comparable to ns-3 using DCE than to ns-3 itself. Indeed, Shadow conceptually resembles ns-3 with DCE in its design in many ways (though it is originally forked from the DVN network simulator [Foo+10]). However, unlike DCE, Shadow, since its inception, has been able to run Tor (as one would expect). Shadow is at least 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 does not 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 seen some use in Bitcoin simulations [MJ15], Tor remains the most popular use case of Shadow. Conversely, Shadow is currently one of the most popular means of performing Tor experiments [SGD15].

Here, we will provide a brief overview of the design of Shadow, as it existed before our modifications.

A *host* or *virtual node* is an entity on the simulated network. While all of Shadow runs in a single process from the perspective of the OS, hosts can run multiple *simulated processes* (that represent the Linux processes being simulated), each of which may use multiple logical threads (run on a single system thread using a modified version of the GNU Pthreads library), as shown in Figure 3.2. The types of hosts, which 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, these hosts are distributed evenly across some user-specified number of system threads, called "worker threads".

To execute a simulated process on a host, Shadow dynamically loads the process executable, known as a *plugin*, as a shared object into memory. 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*). Upon switching execution from one process to another, the data segment of the previously executing process is copied elsewhere in memory and exchanged with the newly executing process — a procedure we call *state swapping* (see Figure 3.3). Using this technique, multiple instantiations of a process corresponding to a particular plugin may be simulated, despite the dynamic loader only allocating memory for the plugin once.

An optimization Shadow implements for the sake of parallel processing is how these plugins are dynamically loaded. As described above, Shadow relies on state swapping for running multiple simulated processes from the same shared object. While state swapping does achieve this goal, it does not allow for doing so concurrently, as only one state may be in the allocated memory for the shared object at any one time (as a result of using `dlopen`, see Section 3.1.1). To circumvent this problem, Shadow copies the shared object file representing the plugin in the file system, once for each worker thread. Because it is now multiple distinct shared objects, the dynamic loader can now load it each time with its own state, without relying on namespaces. Thus, each worker thread has its own copy of the plugin, and they can concurrently run different simulated processes.

As a discrete-event network simulator, Shadow's execution centers around the processing of *events*. An event is the fundamental unit of simulation in the simulator. For example, an experiment may have events corresponding to sending or receiving a network packet, the beginning or the end of an experiment, or the starting or shutdown of a host. To construct these events from the execution of plugin code, Shadow relies on the `LD_PRELOAD` feature of the dynamic loader. By setting the `LD_PRELOAD` environment variable to a shared object

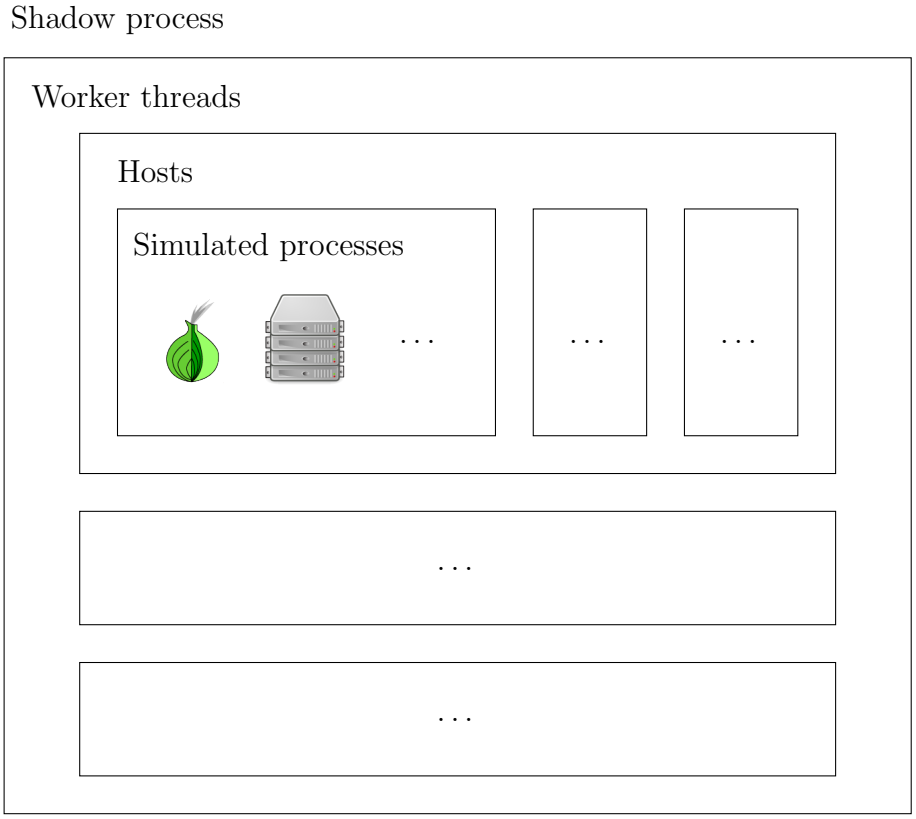

Figure 3.2: The logical structure of Shadow. There is one Shadow process, with some number of worker threads, each with some set of hosts assigned to it, each of those having some number of simulated processes to run (for example, Tor, a web server, etc.), which correspond to plugins. Not shown are logical threads, which would be a sub-division of each simulated process.

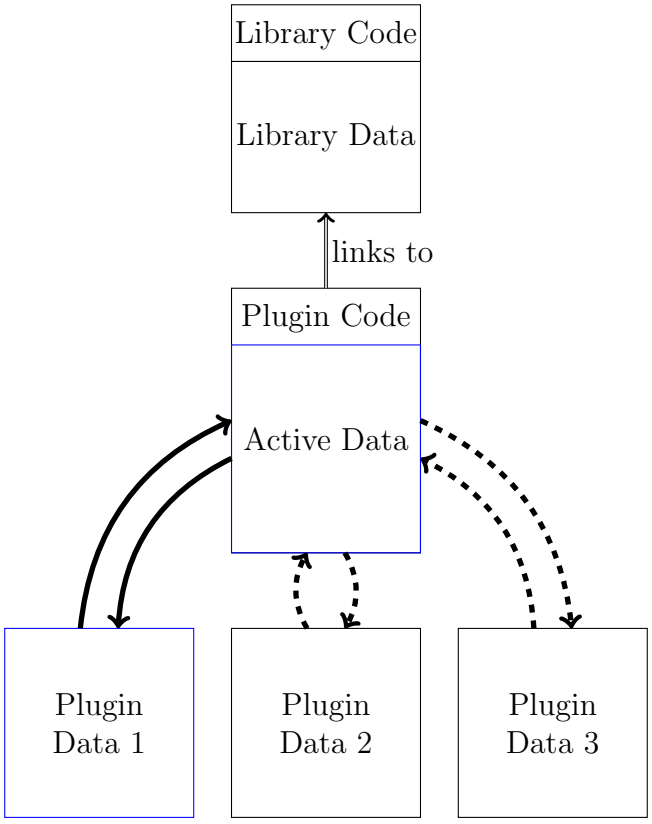

Figure 3.3: Shadow's state swapping technique. Each instance of a plugin (e.g., Tor) has its data segment's previous state copied into the data segment of the corresponding shared object when executing, then copied back out when it is done for this round. All plugins link to a single instance of each library.

that defines functions with the same names as the functions that correspond to desired events (as well as any functions that would cause indeterminacy), Shadow interposes the functions that were linked against and calls Shadow's versions instead. This interposed version of the function will then have whatever behavior is needed for the simulation to proceed. For example, when a plugin attempts to call the "send" function (which is itself a C POSIX library wrapper for the "send" system call), it will instead call Shadow's definition, which creates an event corresponding to this action.

Conceptually, we can then think of all of these events being placed in a global queue, ordered by the simulation time at which the event is supposed to occur. Each event is popped off the queue as it is processed by its respective host, causing the current simulation time to progress with the head of the queue. But to allow for parallel processing, the queue is broken into discrete *rounds*, which consist of a start and ending simulation time. The amount of simulation time in a round is set at the start of the experiment to the minimum time it takes for one host to affect another (via network latency set in the configuration file). Every host on every worker thread can then process all events that occur before the end of the round, without fear of violating causality, from some set of queues. When a worker thread has completed processing all events in the current round, it waits on a barrier for all other threads to complete their events for the same round (henceforth referred to as the *round barrier*, or simply *the barrier*). How these queues are organized is set by the "scheduler policy" option.[6] In the default setting, this consists of one queue of events per host, in an attempt to reduce contention for locks on the queues. Each host, in the order they were assigned to the thread, then executes all of its available events in this round, and the algorithm repeats until completion.

3.2.1 Shortcomings

While the design of Shadow has been good enough to become one of the more popular means of simulating Tor networks, it is not without its shortcomings. Most important for large experiments is the lack of scalability in Shadow's multithreading. While using Shadow for Tor research, we found that a small experiment we were using finished more quickly on an eight-core desktop machine with eight worker threads, than on a machine in the CrySP RIPPLE Facility [Wat13] with 160 cores, irrespective of the number of threads, because of a marginally higher CPU clock speed on the desktop machine. (More rigorous experiments

[6]Scheduler policies were added to the development branch of Shadow our implementation is forked from, prior to said fork. The release version of Shadow will have them available at the same time our changes are released upstream.

on the scaling of Shadow across thread counts with a more detailed experimental setup can be found in Section 4.4.3.)

There are several reasons for this, but they largely originate from Shadow's method of loading 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 amount of overhead over directly running the application.

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.

Yet another drawback of this 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, as previously stated, 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 after the state was migrated to another thread's copy of the plugin (Figure 3.4). 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 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 3.5, which compares the "round" Shadow is in (which has a direct correlation to simulation time) to the amount of wall clock time each worker thread has spent idle (as well as the elapsed time of the experiment over all). We see that the fraction of time each

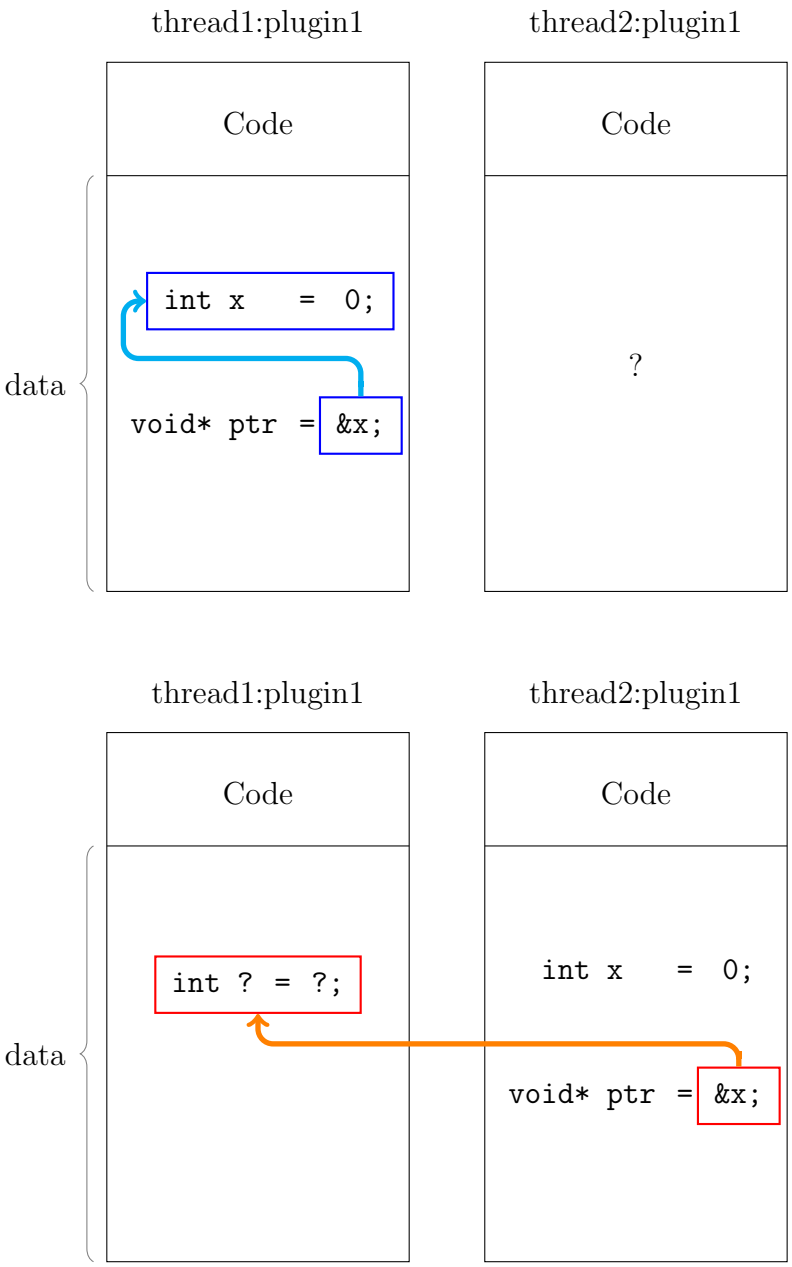

Figure 3.4: Why Shadow cannot migrate hosts if state swapping is used. Top: The state of a plugin active on thread 1 before attempted migration. Bottom: The state of the plugin after attempting to migrate to thread 2, creating a stale pointer to some other instance of the plugin's state on the original thread.

thread spends idle remain in their relative positions throughout the experiment.

Finally, because of the hoisting technique described earlier, Shadow in its original state can only be compiled with the LLVM/Clang C compiler. 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, but we believe are due to the hoisting technique.

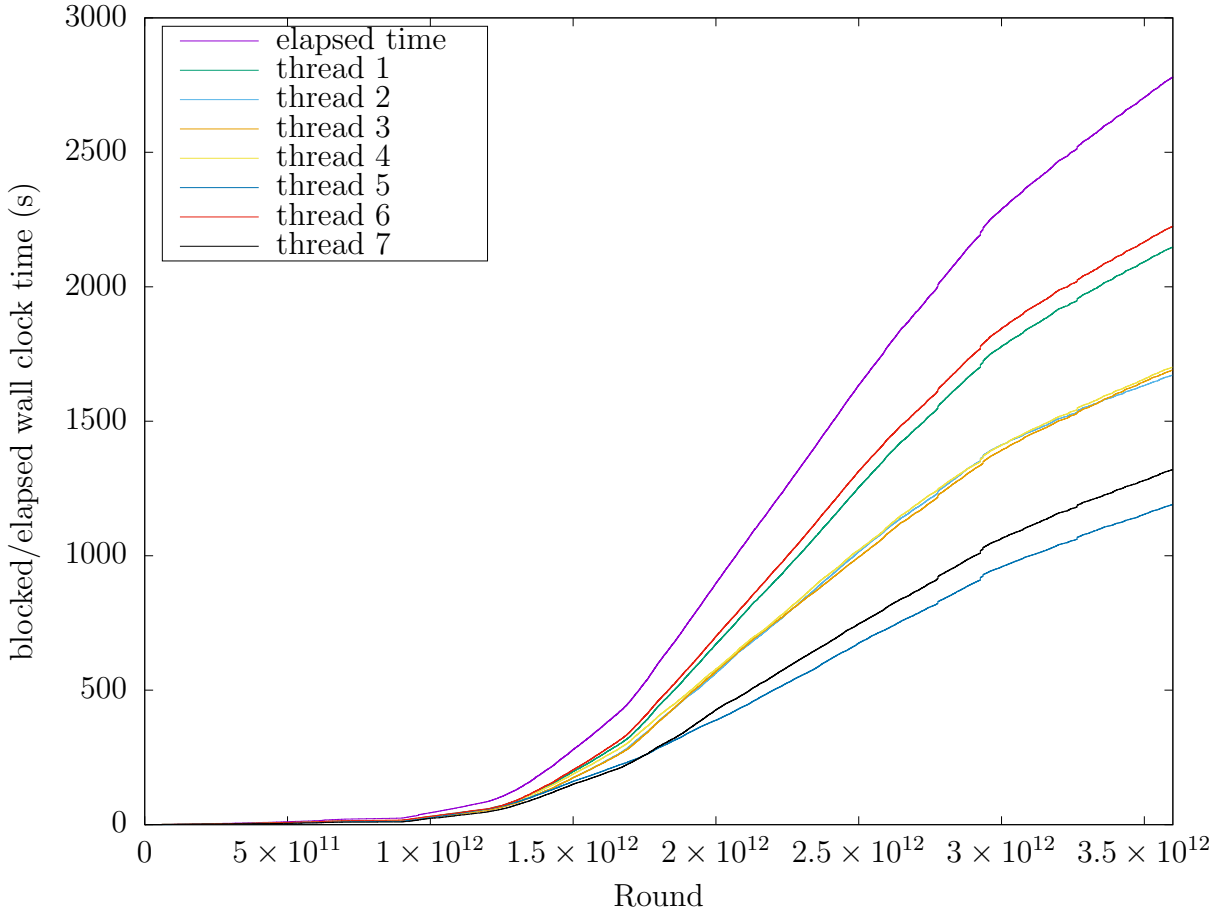

Figure 3.5: Amount of time each worker thread has spent blocked over the course of an experiment, as well as the elapsed time of the experiment. The graph is not normalized in order to show the absolute cost of the blocking — the right side, where the lines spread, is visibly the more time-consuming portion of the experiment. Some of the worker threads are blocked 80% of the time throughout the experiment. Simulated Tor network with 268 hosts, simulating 1 hour of operation, on 7 worker threads.

Chapter 4

Improving Tor Simulation

To address the shortcomings of existing Tor experimentation platforms, we designed a new simulation architecture and implemented this design by modifying an existing simulation framework. In this chapter, we explain this design and how it was implemented. First, we detail the general architecture of our implementation, which is forked from Shadow. Then, we describe drow-loader — our fork of elf-loader, which adds a series of performance and feature improvements, and has been integrated into our modified version of Shadow. Finally, we detail how we used the new functionality drow-loader provides to add a scheduling mechanism to our version of Shadow, allowing for more even distribution of work through the migration of hosts between worker threads.

4.1 Architecture

In this section, we describe the overall design of our implementation. In particular, we draw distinctions between it and the original Shadow design it is derived from, so as to better understand its advantages. A visual representation of these differences can be seen in Figure 3.3 and Figure 4.1 for reference as we describe the distinctions.

Like the original Shadow, we rely on the dynamic loading of shared objects to run plugins that represent the simulated programs. Unlike the original Shadow design, however, we make use of drow-loader (see Section 4.2 for details) to open the shared object with `dlmopen`, 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,

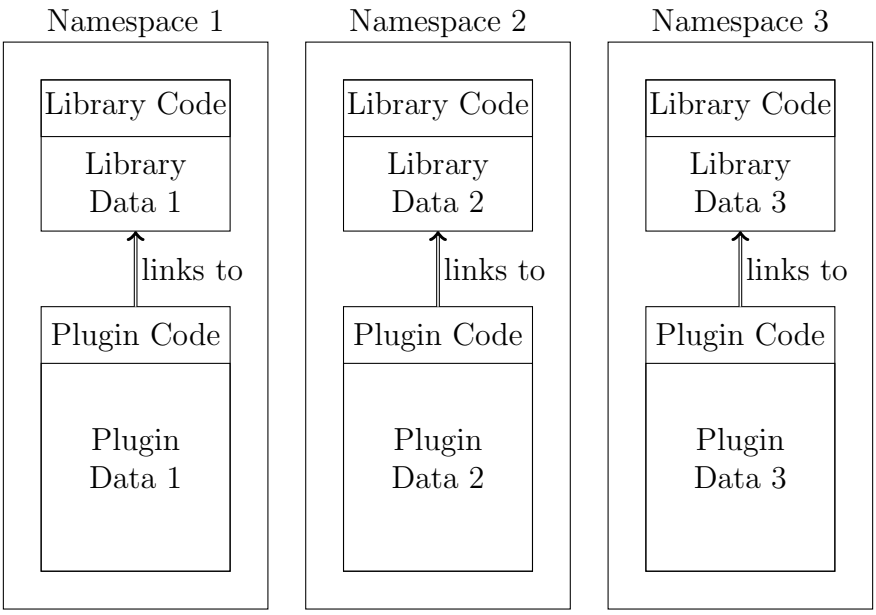

Figure 4.1: Our new design. Each instance of the plugin has its own dedicated namespace, which includes its code, data, and linked libraries.

including code and data, ready for execution at any time. (At first this may seem to increase the required memory overhead, but in practice it does not, as described below in Section 4.2.4.) 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 (notably, GCC).

In addition to the advantage of each plugin having its own namespace, all dependencies of the plugin are also loaded into the respective namespace. This contrasts with Shadow's original design, where there can only be one instance of any library in memory for the entire simulation, irrespective to the number of simulated processes making use of it. Aside from greater assurances of correct behavior of applications by isolating each dependency's state, this has the direct benefit of removing the global lock that was added to the Tor plugin on use of the OpenSSL library, meaning that Tor code making use of OpenSSL can be run in parallel.

4.2 drow-loader

drow-loader[1] is a custom dynamic loader intended for use with network simulation. As elf-loader is the only dynamic loader written to support large numbers of `dlmopen` calls, and it sought to maintain glibc ABI compatibility, as we do, it was a natural choice as a starting point for drow-loader. However, it was found to be insufficient in many regards for our uses, hence our creation of the fork we call drow-loader. As of this writing (September 2017), we have added approximately 7,500 lines of C and removed 4,500 (elf-loader originally having approximately 13,000 lines of C). This section is divided according to the goals we had for drow-loader, and how we addressed them.

4.2.1 Correctness

For a dynamic loader to be useful, be it in network simulation or any other application, it must behave as expected by the compiled objects. While elf-loader's authors claim that it "could replace the system dynamic loader seamlessly" [Lac10], we found that there were flaws in how it matched the behavior both of the ELF standards, and of glibc's ABI. Accordingly, some of the primary changes to elf-loader were to its loading and symbol resolution behavior. To ensure the correct execution of given programs, we spent a considerable amount of time identifying the causes of specific `ld.so` behavior and adjusting drow-loader so that it matched. Some examples include symbol version checking, scope ordering, glibc stack management compatibility, support for existing flags and symbol types, and several undocumented glibc behaviors.

4.2.2 Performance

A crucial goal in the development of our modifications to elf-loader is that all dynamic loader operations used by the network simulator (dynamic loading, symbol resolution, etc.) should be sublinear time in both the number of namespaces and the total number of objects. This is due to the intended use of Shadow loading each process, for thousands of hosts, into its own namespace, which in turn has its own copy of each necessary object (for example, Tor, OpenSSL, etc.) in memory. Any linear operation in the dynamic loader would therefore turn into a quadratic operation in execution, as said linear operation must be called a linear number of times.

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

It is worth noting that the necessity of sublinear time does not extend to the number of objects within a single namespace, or unique objects — that is, drow-loader does not need to run in sublinear time in the number of objects that do not map from the same object file. Such use cases are not relevant to Shadow, as no existing programs make use of a large number of namespaces or unique objects. If these programs did exist, they would be unable to run with anything resembling reasonable performance, or at all, on the glibc dynamic loader, since the glibc loader does not support large namespace counts, and some operations within a namespace are inherently linear (e.g., symbol lookup requires a search through the objects in scope) [Dre11].

To accommodate a large number of `dlmopen()` calls, elf-loader relies on linked lists as its primary data structure. While linked lists are simple to grow, many operations on linked lists are not fast enough for our purposes. Lookup of a particular handle, for example, is an operation frequently used by a dynamic loader that is not well suited for linked lists. Additionally, elf-loader has many algorithms in its code where it was clearly not intended for use with large numbers of namespaces or objects, with many operations having linear or even quadratic run-time in the number of namespaces or loaded objects.

Unfortunately, there are some instances where the existing data structures are required to maintain ABI compatibility with glibc and gdb. Most notably, the internal data structure glibc uses to keep track of objects associated with a namespace is a form of linked list structure, referred to as a *linkmap*,[2] which is required for ABI compatibility.

To address these problems, drow-loader uses different data structures that either replace or work in conjunction with these linked lists. For example, a red-black tree is used to map a given memory address to the instance of an object it is associated with, instead of iterating over all objects until one with a memory mapping that contains that address is found. Similarly, a hash table is used to map handles to their linkmap entries, so that a lookup in the linkmap is now a constant-time operation, instead of linear. Each of these data structures is implemented within drow-loader's code. This is because the dynamic loader cannot rely on dynamically linked libraries such as glibc to function. Therefore, the dynamic loader requires custom implementations of basic functionality (such as memory allocation and system calls), which in turn prevents static linking to many common library implementations.

With the modifications made, drow-loader achieves its goal of sublinear time dynamic loading, and can be used to load hundreds of thousands of instances of a minimal shared

[2]This is the origin of the "lm" in many of the namespace operations exposed to userspace, e.g., the "`Lmid_t`" data type, or the "`LM_ID_NEWLM`" flag used in `dlmopen()`.

object on standard desktop hardware in minutes.[3] For measured results, see Section 4.4.2.

4.2.3 Concurrency

Another goal which, while less important than sub-linear dynamic loader operations, is important to the desired performance of Shadow, is to maximize the concurrency of these operations. If the dynamic loader operations are done under a single global lock, as is the case for the glibc dynamic loader and elf-loader, then it is likely that many of the concurrency improvements we would make to Shadow would be negated in practice. One operation where this would clearly be the case are dynamic loading, where the initial creation of hosts would contend for the lock. Another would be symbol resolution, where the first use of any dynamically linked symbol (e.g., the first call of a function) would be mutually exclusive with any other first use of any symbol.

To make drow-loader more parallelizable, we first created a reader-writer lock implementation. By doing so, this allowed for the data structures used to be read by multiple threads, exclusive of any writes to those data structures. Then, we reduced the scope of the global lock into locks on the data structures themselves, such that each is always in a consistent state, without locking the other data structures involved that are not currently being modified. Currently, there are twelve locks with global scope, all of which are only used briefly or in rarely used functionality. If it is later found that these locks are significantly impacting concurrency, some can be eliminated with the use of lock-free data structures.

4.2.4 Memory Overhead

In order to feasibly run Shadow with drow-loader, it is necessary to minimize memory overhead. To reduce the memory usage, drow-loader makes use of shared memory mappings. Because the shared objects are loaded into memory via mmap system calls, instead of copied into allocated memory, they can make use of Linux's Copy On Write, or COW, functionality. Using COW, multiple instances of memory from the same mapping will not

[3] A "minimal shared object" here refers to a shared object that has no symbols aside from a main function, linked against glibc. Larger and more complicated objects will take more time. Part of the dynamic loading process is calling the initialization functions of the loaded objects. As these functions can have arbitrarily defined behavior, they are not part of the analysis given here. While this is not a realistic use of the dynamic loader, it isolates the performance evaluation to the dynamic loader and not the shared object.

be given their own physical memory until one instance is written to, at which point the page is copied and the new copy is used as the target of the write. All of this happens transparently to the application, as this is done in kernel allocated pages while the application only sees its virtual memory. By doing this, parts of the files that are never written to in memory are shared among each other, reducing the duplicated physical memory needed to represent the file. One such section of the file where this would typically be used is in the `.text` section, where the executable, read-only code is stored. However, this is not the case in elf-loader or drow-loader. Because many of the shared objects used are linked against glibc's dynamic loader, elf-loader and drow-loader must modify this section to replace any references to the original loader, so that they instead reference the new loader. As a write operation, this causes COW to copy out the memory into separate pages, despite the resulting memory contents being the same for all instances (that is, all of these pages undergo identical modifications, but the corresponding memory does not merge back together). To circumvent this, drow-loader explicitly loads all read-only segments into shared memory mappings, by opening a file descriptor in `/dev/shm/`, which on Linux is mapped to memory, and not storage. By doing so, the modified segments are still shared, even after modification. The resulting memory layout, from the perspective of the kernel, can be seen in Figure 4.2.

4.2.5 Run-time Symbol Interposition

A problem with the `LD_PRELOAD` technique of function interposition that Shadow relies on which was not mentioned previously is that it is global for the entire program. While this is typically acceptable, there are situations where a more localized approach would be desirable.

For example, suppose one wanted to run a simulated process that relied on `LD_PRELOAD` under normal circumstances on some fraction of hosts (an example of one such instance would be Torsocks, an application wrapper that forces a non-Tor application to use Tor — since Torsocks is used by some Tor users, it might be useful in an experiment to simulate clients that make use of it). The environment variable would either not include the additional shared object, or include it for all processes on all hosts. While one could construct a new shared object that conditionally forwards interposed requests to the original scope or the simulated `LD_PRELOAD` based on the simulated process the call originates from, this is a non-trivial amount of work that would have to be done for each type of process that relies on `LD_PRELOAD`.

More generally, any time it is desirable to modify the behavior of a specific plugin's calls to other shared objects (instead of every plugin's calls to other shared objects), a more

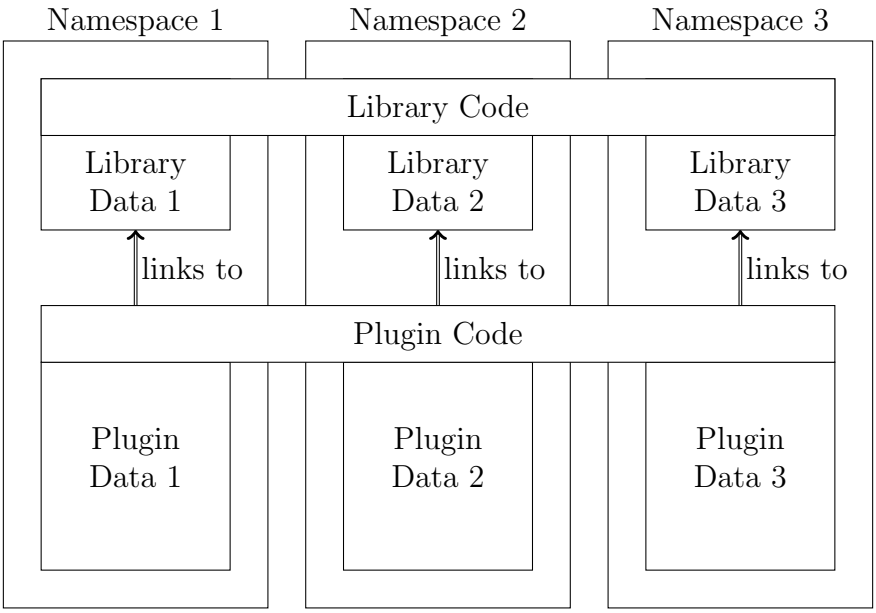

Figure 4.2: The memory layout found in Figure 4.1, from the perspective of the kernel. Each read-only segment, and any pages that have not been modified, that are mapped from the same file are backed by the same physical memory (in this case exemplified by the code of each shared object).

specifically focused interposition is warranted. An example of this in practice is Tor's use of cryptographic operations in OpenSSL, which are interposed to improve performance. These interpositions should be able to be constructed via configuration for that simulated process or host, not globally.

To achieve this goal, we add two new flags to `dlmopen()`. The first, `RTLD_PRELOAD`, causes the newly loaded shared object to be loaded into the front of all scopes, as though it were loaded from the `LD_PRELOAD` environment variable. This flag is almost identical in behavior to the previous use of `LD_PRELOAD`, but has the advantage that it does not rely on the user environment, making it more self-contained. It also allows for the paths to be specified later in the run of the experiment, after the configuration file has been parsed, without restarting Shadow (although as described in Section 4.2.6, Shadow is still restarted for other reasons). The second new flag drow-loader supports in `dlmopen()` is `RTLD_INTERPOSE`, which behaves much as `RTLD_PRELOAD`, but limits the scopes in which the new shared object is added to a specific namespace. This allows for the configuration file to specify shared objects that interpose a plugin's symbol resolutions, while still being local to a particular simulated process's namespace.

4.2.6 Initialization-Time Configurable Static TLS

One problem encountered with utilizing large numbers of `dlmopen()` calls involves the details of how memory is allocated for static TLS. As previously mentioned, TLS comes in either dynamic or static models, chosen by the object as a flag in the ELF headers. Also previously mentioned was that static TLS is used by a small minority of shared objects, which is fortunate, as it is intended to explicitly forbid dynamic loading of the object. Unfortunately, one of the few shared objects that makes use of static TLS (for performance reasons) is glibc, which is linked with almost all shared objects and executables on a GNU/Linux system. Since dynamically loading a shared object also dynamically loads all dependencies of an object (including glibc), this means strictly speaking, almost nothing is entirely safe to dynamically load. In practice, dynamic loading works because attempting to dynamically load glibc will typically cause the dynamic loader to find and return the copy of glibc that was presumably already loaded during dynamic linking. However, when a new namespace is created, this is not the case, as a newly initialized namespace will only have a set of base objects in it, and not glibc. Therefore, when a new copy of glibc (or any other static TLS object) is loaded into a new namespace, it must store its TLS in some unused memory allocated for static TLS. Unfortunately, the memory for static TLS cannot be (re)allocated during execution, as it is part of the POSIX thread data structure, which

cannot safely be moved once a thread has started executing.[4] The glibc dynamic loader and elf-loader address this problem by allocating a small amount of extra space to the static TLS memory allocation when performing dynamic linking, at program initialization, so that some amount of objects with static TLS can be dynamically loaded anyway. Once this space is used, additional attempts to load objects with static TLS will fail, causing the program to crash.

In general, the entire design of static TLS is built around it not being dynamically loaded, and the performance improvements this provides. Indeed, according to the Intel ABI specification for TLS on Itanium, the entire purpose of the static TLS flag is "... so that the dynamic loader can easily reject attempts to load such a file dynamically." [Int01]

To make the dynamic loader accept attempts to load such files dynamically, drow-loader relies on the fact that an executing program can have more knowledge of static TLS requirements than the dynamic linker. It does this by providing two new features. The first is support for an additional environment variable, `LD_STATIC_TLS_EXTRA`, that, if set, determines the size of the extra static TLS space it allocates at the time of dynamic linking. This means that a user can set the size of the static TLS allocation without recompiling the dynamic loader — a significant improvement over the GNU dynamic loader and elf-loader. The second feature is an additional request type to the `dlinfo()` function. `dlinfo()` is a non-standard dynamic loader function provided by many dynamic loaders, including the GNU dynamic loader, that is intended to provide requested information from the dynamic loader (e.g., the namespace an object was loaded in, or the path an object was loaded from). We add a request type that returns the amount of currently used static TLS. For a program such as Shadow to allocate a variable amount of TLS, it executes in two stages. First, the program calculates the amount of static TLS it will need during the course of execution. In the case of Shadow, it does this by reading the amount of static TLS used before and after loading one instance of a host, then multiplying the amount of additional static TLS used by the number of hosts of that type it will run, for each type of host (e.g., a Tor relay, a Tor client, a web server, etc.). Once it has calculated the total TLS needed, the program re-executes itself, this time with `LD_STATIC_TLS_EXTRA` set to the sum of the previously calculated products — that is, the program calls `exec` on the program with the original arguments, plus the additional environment variable set, so that the dynamic linker can appropriately allocate the correct amount of static TLS memory. By doing this, the program can dynamically load as many plugins as needed in separate namespaces,

[4]Specifically, code is compiled using the x86-64 POSIX thread ABI specified by Intel. Only the kernel can modify (or for that matter, read the raw value of) the thread pointer, and while a system call does exist to change it (arch_prctl()), it is not intended for frequent use (some kernels have it disabled entirely). To quote the man page, "Programs that [set the thread pointer] directly are very likely to crash." [Kle17]

despite their use of static TLS from glibc.

While this technique could be considered a bit of a hack, it has proven to be effective. It is also worth mentioning that Shadow already relied on re-executing itself, to set the `LD_PRELOAD` environment variable to the location of its interposing shared object. However, this reason is no longer necessary, due to the new `RTLD_PRELOAD` feature drow-loader provides (as described in Section 4.2.5).

4.2.7 TLS Migration

One of the primary reasons for using drow-loader with Shadow is to allow for migrating hosts from one thread to another. However, if this happens, then the TLS for the shared objects associated with that host's processes will no longer be on the correct thread. Therefore, we added a new function to the dynamic loader API for drow-loader, which takes two thread pointers as arguments, and exchanges their TLS.

In the case of dynamic TLS, we simply swap the pointers to the TLS allocations, which are stored in the dynamic thread vector. Because the thread data being swapped in this case is used by the application code to get the address of the TLS and is not the TLS itself, this means that the swapping is transparent to the application code (so long as the executing threads are swapped as well).

In the case of static TLS, which is stored as part of the thread pointer data structure, we must swap the TLS directly. Unfortunately, this has two drawbacks. One is that swapping data is more expensive than swapping pointers. Thankfully, in addition to static TLS being rare, in practice it is less than a kilobyte in size for each instance of glibc (e.g., on Ubuntu 16.04 x86-64, it is 120 bytes). The second problem is that because the data is being swapped directly, any pointers in any shared object that point to that static TLS will point to the wrong thread's static TLS after swapping. This is, in essence, the same problem that Shadow's state swapping technique has with host migration (Figure 3.4), which we were trying to avoid. Unfortunately, there is no generalizable solution to this problem. Instead, we identify any instances where this is problematic, and use interposition to fix it. However, these instances are rare. For one, unlike in the case of host migration while using the state swapping technique, where any pointers to anywhere in the entire data segment will become invalid, here, only pointers to the (very small) static TLS allocation are problematic. Furthermore, static TLS values are typically only used internally by glibc, which uses them directly as global variables and not via pointers. For now, the only case that addressing this problem has been necessary is in adjusting code that stores the address of `errno`. In any case, static TLS swapping is unlikely to be problematic, given that prior

to drow-loader, Shadow functioned with TLS being global to all states for a shared object on a particular thread (as TLS is not part of the state that was migrated when swapping hosts). This implies that in practice, static TLS is not particularly stateful.

4.3 Scheduling

One of the primary motivations for implementing the new dynamic loader functionality was to allow for a better scheduling mechanism than the scheduler policies Shadow previously allowed (see Section 3.2). With the additional features provided by drow-loader, we were able to implement a means of migrating hosts from one thread to another during the course of execution. As a result, we were able to better distribute work among worker threads while the experiment runs using a new scheduler option we implemented.

The new scheduling algorithm is a form of work stealing [VR88; BL99]. 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 all worker threads have no hosts that have 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 4.3 and Figure 3.5, no thread spends significantly more time than any other being blocked. Because hosts are migrated only when a worker thread is idle after executing its currently assigned hosts, we avoid much of the cost of TLS swapping 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. Contrast this with a scheduler that attempts to redistribute hosts at intervals based on estimates of the amount of work each host performs (as is typical in many classic operating system scheduling algorithms). Such an algorithm would likely assign a host that is highly active at infrequent intervals a large estimate of anticipated work, despite the fact that most rounds it would be doing very little work. The

work stealing algorithm we use avoids this problem, since no estimates are made, and hosts are stolen from other threads on an as-needed basis, and not assigned in anticipation.

In the unlikely event that a user encounters a case where our scheduling algorithm decreases performance, one of the original scheduling algorithms may be selected instead via command line arguments. This could be the case when all hosts are identical and evenly divisible by the number of worker threads, though even then, the difference would likely be negligible.

4.4 Experimental Results

In this section, we give some experimental results from our implementation of our new Tor experimentation platform. We start by giving some comparative microbenchmarks for elf-loader and drow-loader to demonstrate performance and memory improvements. Then, we compare the performance of Shadow in its original design (state swapping), our new design making use of drow-loader with `dlmopen()`, and finally with our new scheduler.

4.4.1 Setup

The experiments were run on different hardware setups, generally using the most readily accessible machine capable of running the desired experiment. We used a total of three hardware configurations — one from a desktop computer, and two that use the RIPPLE Facility [Wat13].

- **Desktop:** AMD X8 FX-8370E (8 cores @ 3.3 GHz) machine with 16 GB of RAM, running Ubuntu 16.04. Used primarily for drow-loader microbenchmarks, where overall experiment time and resource usage is small.
- **Ticks:** Six completely independent machines, each with 8 Intel Xeon E7-8870 CPUs (10 cores + hyperthreading @ 2.40 GHz), 1 TB of RAM, running Ubuntu 14.04. Used for the majority of Shadow experiments, where it was desirable to test the scalability across the number of threads. The six machines were used concurrently to more quickly run experiments in parallel, and did not utilize any sort of distributed computation.
- **Tock:** A single machine with specifications identical to the Ticks, but with 2 TB of RAM instead of 1 TB. Used for the larger Shadow experiments, which use more than 1 TB of RAM.

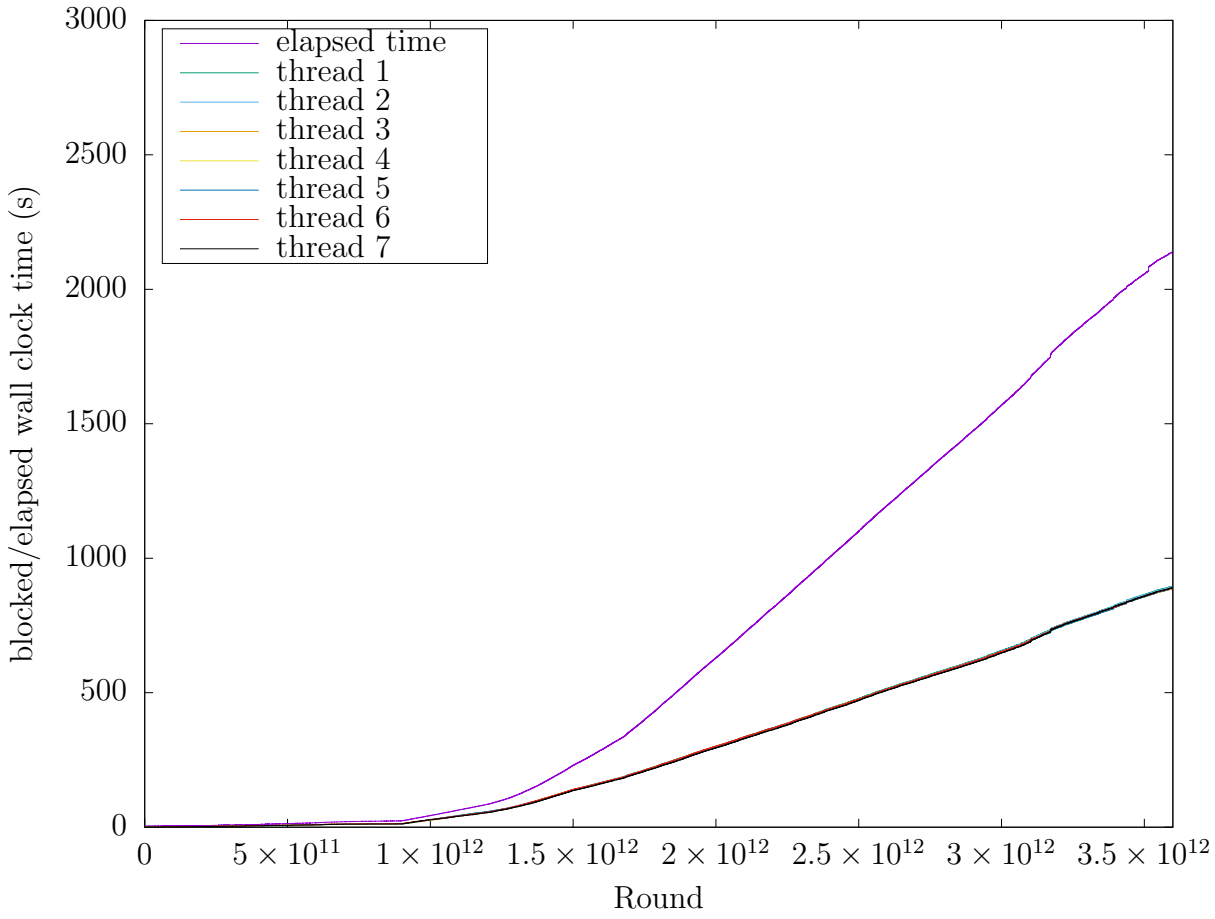

Figure 4.3: Amount of time each worker thread has spent blocked over the course of an experiment when using work stealing. Simulated Tor network with 268 hosts, simulating 1 hour of operation, on 7 worker threads. Individual threads cannot be readily seen because their blocked times are nearly identical. Contrast with Figure 3.5.

4.4.2 drow-loader

To demonstrate the performance improvements of drow-loader over elf-loader, we ran a series of simple experiments. We first built a minimal shared object, consisting only of an empty main function, linked against glibc. We then wrote a small program in C that creates a specified number of new namespaces, and dynamically loads the minimal shared object into each one. Using this program, we can compare the running time for a given number of object loads, as well as memory usage. The results of these experiments can be seen in Table 4.1.

It is worth noting that, since we rely on Linux's Copy-On-Write mechanism (COW) and shared pages for reduced memory usage, which are transparent to the application itself, we cannot rely on the reported Resident Set Size (RSS) for measuring memory usage. Instead, we record the amount of memory measured as used/available for the entire system, as reported by the kernel via `/proc/meminfo`, immediately before the experiment terminates and immediately after it terminates. While this is less precise in that it contains noise from the other programs running on the system (for example, daemons), it is a significantly more accurate measure of real-world memory usage. Indeed, the reported RSS can and will report that the process is using (almost arbitrarily) more memory than the machine even physically has available to it, if enough objects originating from the same physical memory backing are mapped in the process. We were unable to find any alternative means of obtaining an accurate measure of a single process' memory usage that takes into account these deduplication effects, likely because these shared mappings can be shared across multiple processes as well as in the same process itself, making distinguishing which specific process is "using" the physical memory difficult or impossible in practice.

The missing entries for 50,000 and 100,000 shared object loads in elf-loader are due to a bug in the original elf-loader code involving the interaction between the glibc ABI for stack management and static TLS. This bug caused consistent crashes when used with sufficiently many loads, even when modifying the size of the static TLS allocation and recompiling appropriately.

As can be seen in the table, elf-loader exhibits superlinear growth in both runtime and memory usage. Given that each host will be running multiple simulated processes and at least one preload library, this growth is clearly at such a rate to be untenable for larger experiments, even if it were the case that it functioned properly. On the other hand, drow-loader shows linear growth for memory usage, and while higher load counts begin to diverge from linear in execution time, they are still well within acceptable ranges for large experiments (and several orders of magnitude less than what we would expect elf-loader to perform at).

Table 4.1: A comparison of the average amount of time and memory elf-loader and drow-loader take to create a specified number namespaces and dynamically load a minimal shared object into each one. Single-threaded, run with 10 iterations on the Desktop configuration. 95% confidence intervals provided.

loads	elf-loader memory (MB)	drow-loader memory (MB)	elf-loader time (s)	drow-loader time (s)
1000	68 ±2	63.2±0.9	4.9± 0.1	0.275±0.007
2000	130 ±2	128.2±0.6	43 ± 2	0.57 ±0.04
5000	477 ±2	324 ±2	780 ± 20	1.43 ±0.05
10000	1368 ±1	649 ±1	8300 ±300	3.09 ±0.04
50000	N/A	3242 ±1	N/A	29.4 ±0.4
100000	N/A	6484 ±7	N/A	95 ±1

4.4.3 Shadow

To measure the performance impact of drow-loader on Shadow, we opted to measure actual workloads across our modifications. In particular, all results are from simulated Tor networks of one of two configurations. For all experiments, the first 30 minutes of simulation time is designated for bootstrapping the network. This bootstrapping consists of some configuration-specified time to initialize the Tor protocol, which is largely unsuitable for experimental use. (Specifically, it provides a spacing out of the Tor protocols for communicating with the directory authorities, so that they are not inundated with packets in a single round.) The types of operations the hosts are doing during this time are entirely different from the normal portion of the experiment, and are not generally intended to be an accurate reflection of any reality, but instead to simply to get the network to a more realistic state. Once this 30-minute period has passed, the actual experimental data starts being generated.

For the first of these experiments, a smaller configuration was used, set up such that one hour of simulation time takes approximately one hour of wall clock time in the original Shadow when run with eight worker threads. The smaller configuration 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. The smaller

configuration was run with four different versions of Shadow: Shadow before any of our modifications; Shadow using drow-loader; Shadow using drow-loader with compiler optimizations enabled; and Shadow using drow-loader with compiler optimizations, and our new scheduling algorithm. Each of these were run with a number of worker threads varying from 1 to 18, once on each tick machine (for a total of six runs per thread per Shadow version). The results of these experiments can be seen in Figures 4.4, 4.5, and 4.6.

In Figure 4.4, we plot the number of worker threads vs. the average total runtime of the experiment, with standard deviations. We can see that the performance of Shadow with drow-loader is slightly worse than what it was originally. While we cannot say for certain why this is the case, one possibility is the use of a different compiler and default optimizations.[5] However, once we enable compiler optimizations, overall performance is improved by approximately 30%–40%, for all worker thread counts. After enabling work stealing, the performance improved by up to 20% over the compiler-optimized version for smaller worker thread counts, for an overall performance improvement of 40% over the original Shadow code in most cases, as seen in Figure 4.5.

In Figure 4.6, we take a closer look at the differences in the progress of an experiment over time. Each line in the figure is a single run of the associated Shadow implementation (original, drow-loader, drow-loader with compiler optimizations, drow-loader with compiler optimizations and work stealing), from one of the 18-thread runs on one of the tick machines. On the horizontal axis is wall clock time, and on the vertical is simulation time, so that the performance at a particular moment in time could be thought of as the slope of the line. Here, we see that the original Shadow implementation and the Shadow implementation with just drow-loader have similar bootstrap times. It is after bootstrapping completes that a slight difference in slope makes the original Shadow code faster. Similarly, when enabling compiler optimizations, and then work stealing, with the drow-loader version of Shadow, it is the performance increase observed after initialization that has the largest overall performance impact (see Figure 4.7). One reason this may be the case is that there is simply more time spent outside of the initialization period, so this is where improvements are most noticeable.

Finally, we tested Shadow with a significantly larger experiment setup. The larger test consists of 42 simulated minutes[6] of 56,898 hosts: 5,100 web servers, 1,998 Tor relays,

[5] As stated in Section 3.2.1, prior to our modifications, Shadow could only use the default optimizations, and only be used with LLVM. Currently, drow-loader makes use of some handwritten assembly from elf-loader that does not assemble with LLVM (though this could likely be fixed, we were unable to do so).

[6] The experiment was originally set up to be 45 minutes of simulated time, but an unfortunately timed power failure caused the truncation of one of the experiments. Since there is no particular activity at the end of a simulation that we are interested in measuring, and because of the length of time the experiments

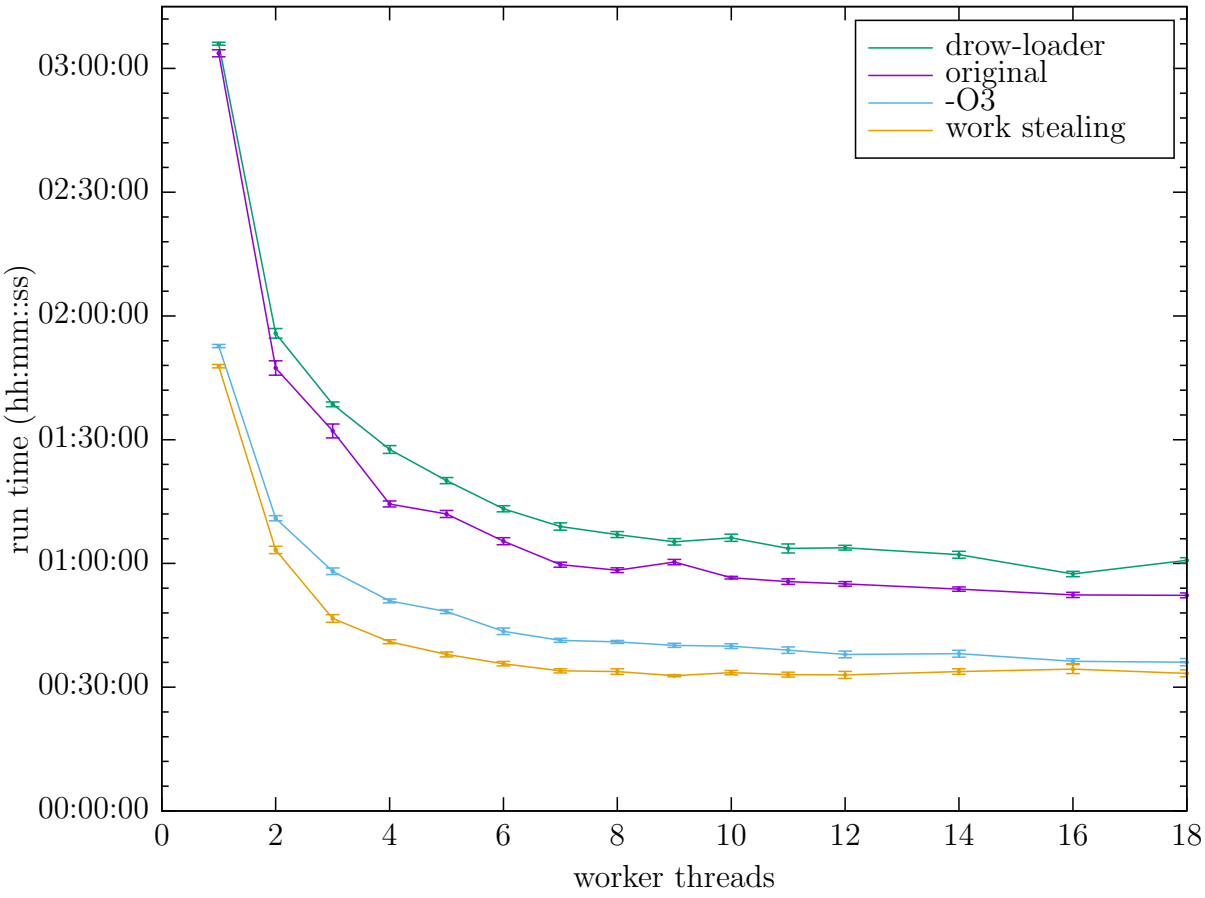

Figure 4.4: The average amount of wall clock time it takes to simulate one hour of the smaller experiment, with 268 hosts on a varying number of threads, run once on each tick machine (for a total of 6 runs). Lower is better. Standard deviations shown as error bars.

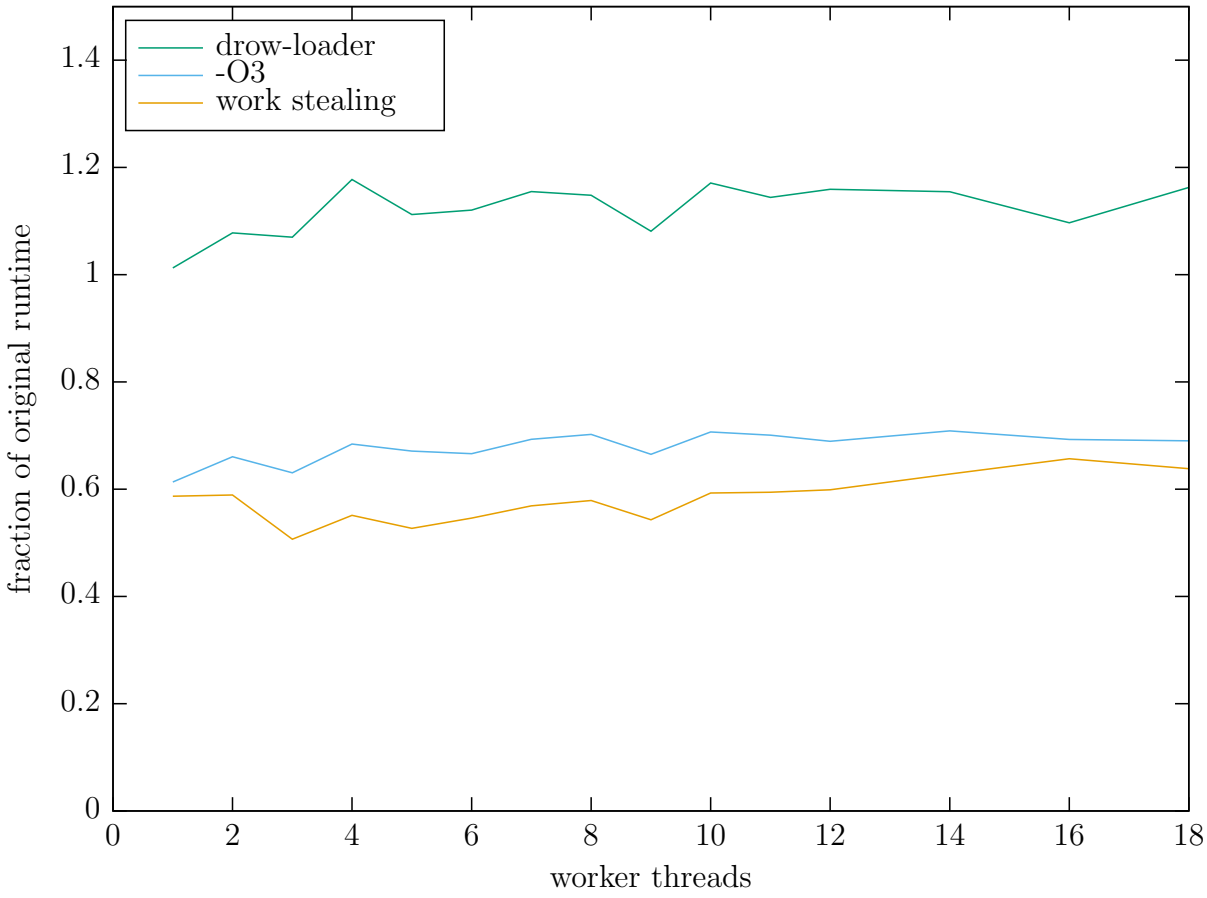

Figure 4.5: Performance normalized to the original performance of Shadow — that is, the value of each line in Figure 4.4 divided by the value on the "original" line. Again, lower is better.

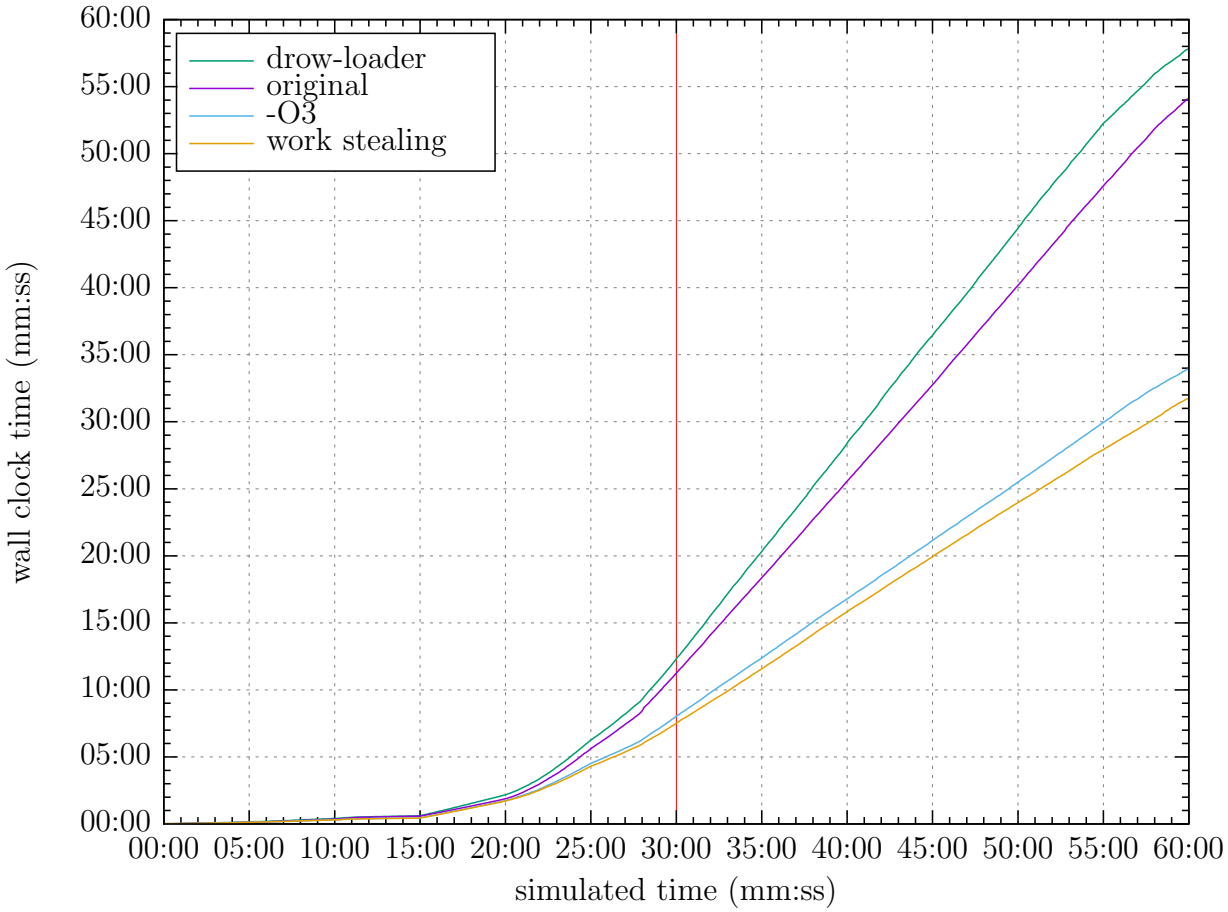

Figure 4.6: The amount of wall clock time to reach a specific simulation time in the smaller experiment, with 268 hosts on 18 worker threads. Lower is better. The simulation time where bootstrapping finishes is shown as the solid red line at the 30-minute mark.

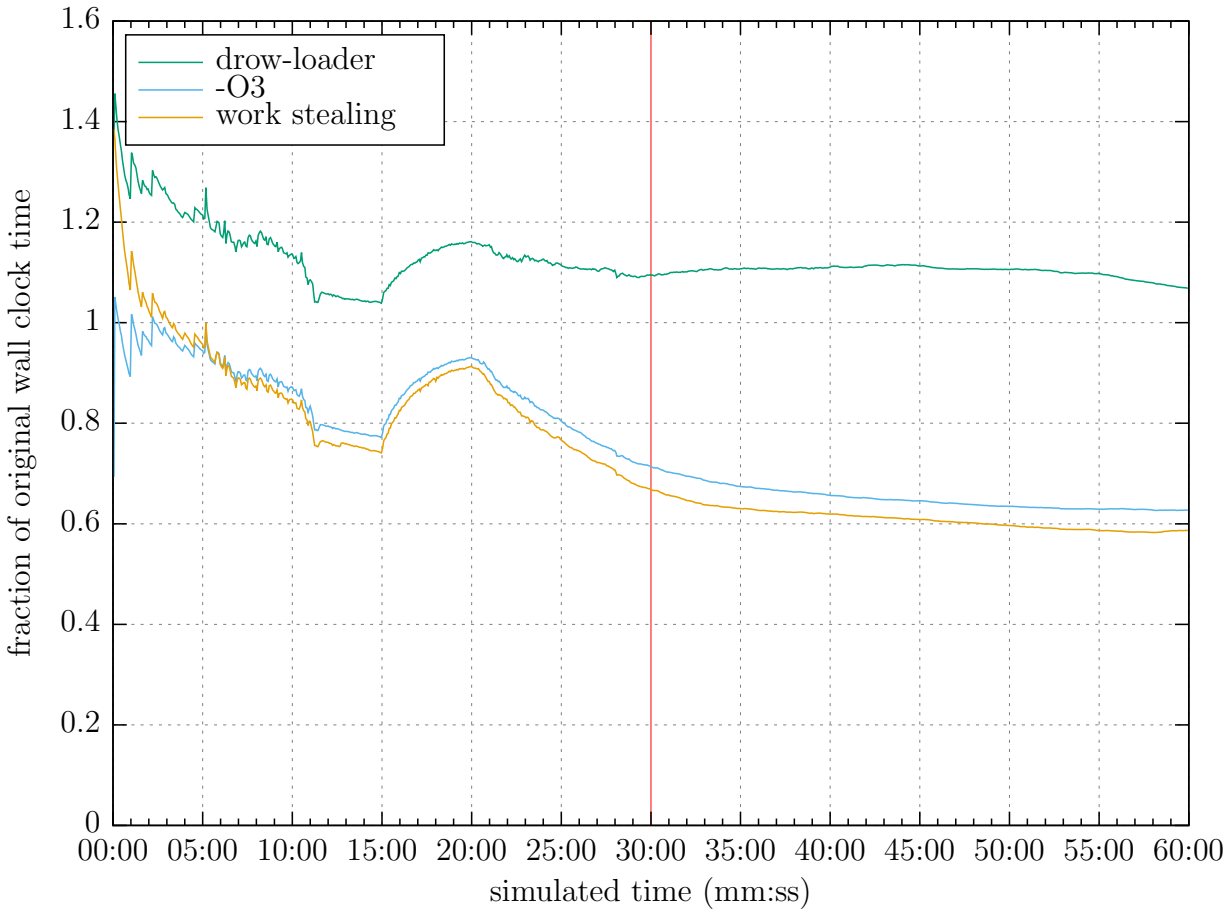

Figure 4.7: The amount of wall clock time to reach a specific simulation time, normalized to original Shadow — that is, the value of each line in Figure 4.6 divided by the value on the "original" line. Lower is again better. The simulation time where bootstrapping finishes is shown as the solid red line at the 30-minute mark. As would be expected, the left side of the chart has more noise, as there is less wall clock time spent on it.

and 49,800 clients. Again, the first 30 minutes of the experiment are configured to be used for bootstrapping. As the experiments take on the order of 5 days to complete, and because there is only one machine in RIPPLE that is capable of running them, we opted to measure the differences in three Shadow implementations (original, drow-loader with compiler optimizations, drow-loader with compiler optimizations and work stealing) with 64 worker threads one time each. The results of these experiments can be seen in Figure 4.8.

While the impact of drow-loader on Shadow is still noticeable for an experiment of this size, the impact appears to be reduced. This may be due to a greater amount of time being spent in the running of the hosts themselves, rather than in the Shadow code whose design was improved. Unlike the smaller experiment, the most significant improvements appear in the initialization portion of the experiment, where drow-loader's greater concurrency in loading and bootstrapping Tor is beneficial.

The performance impact of work stealing was more noticeable, with a decrease in overall run time of approximately 5%. Furthermore, there was a considerable difference in the amount of time threads were blocked on the round barrier (and thus doing no work). When using drow-loader without work stealing, the time spent blocked on the barrier ranged from 20%–63% across threads, with an average blocked time of 45%. With work stealing, the barrier block time ranged from 18.6%–18.7% across all 64 worker threads.

One potential explanation for the lack of a more significant speedup is the reality of Amdahl's law, which states that the speedup of a workload is determined by the fraction of the workload where the speedup applies and the size of that speed up [Amd67]. While reducing blocked time does not translate directly into a value of "increased parallelization" (since it is typically the case that not all threads block at once), if we estimate the reduction of blocked time as an increase in the parallel portion of an experiment by an average of 26% (approximately the improvement we saw from work stealing), we can use Amdahl's law to show with 64 worker threads, the expected speedup would be

$$\frac{1}{(1-.26)+\frac{.26}{64}} = 1.34$$

which, while non-negligible, is less significant than the improvements in the smaller experiment. Again, the smaller experiment had a more significant impact from the compiler optimizations than the larger experiment (due to Amdahl's law as applied to the fraction of the experiment in which the optimizations were applied).

take to complete, we simply look at the first 42 simulated minutes of all experiments.

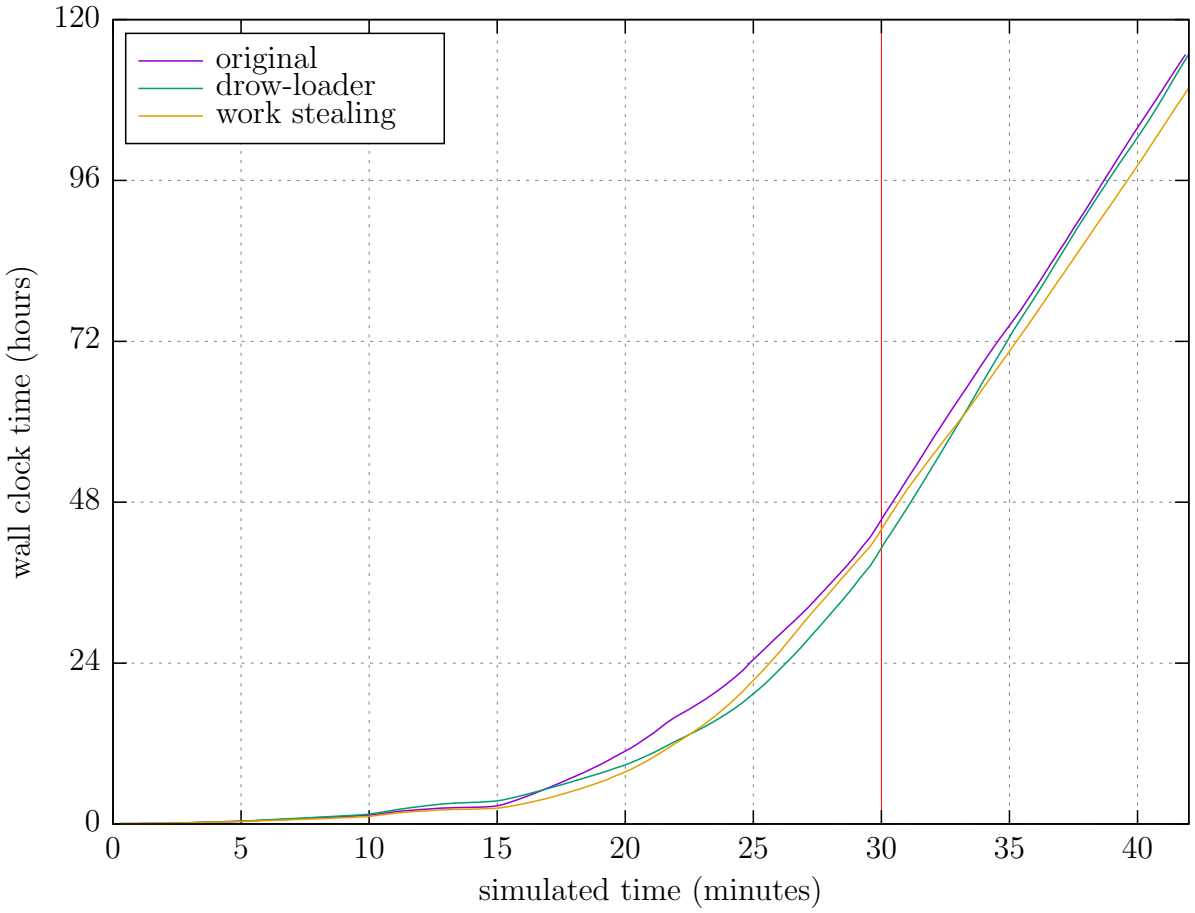

Figure 4.8: The amount of wall clock time to reach a specific simulation time in the larger experiment, with 56,898 hosts on 64 worker threads. Again, lower is better. Simulation time where bootstrapping finishes shown as the solid red line at the 30 minute mark.

Chapter 5

Future Work

While our new process simulation technique comes with its own observable benefits of greater simplicity and performance, the primary contribution comes from the ability to provide further improvements to Tor simulation using Shadow. There are many ways in which Shadow's design and implementation can be improved to allow for performance that more easily accommodates larger and more realistic networks.

For one, the ability to migrate hosts across worker threads allows for a new line of research in scheduling algorithms for Shadow. While we have already implemented a new scheduler that provides a noticeable performance improvement over Shadow's execution, there is no reason to believe that this algorithm is optimal. Potential avenues for further research along these lines include reducing lock contention, improved data structures for managing events, and applying the existing body of research and state of the art on scheduling algorithms to Shadow. For example, it is possible that every thread attempting to query every other thread's host queue every round is overly thorough, and less time could be spent attempting to steal work. As another example, it is possible that there would be a means of reducing the remaining 18.6% of barrier waiting by sorting the hosts or events to be run by their expected execution times. Since this blocking is uniform across threads now, a reduction in this number should translate directly into lower execution times. Without a more thorough investigation, we do not know exactly how advantageous such modifications would be.

That said, there is also the possibility that the most fruitful performance improvements will come from aspects outside the scheduler. For example, it may be possible to improve performance by using hosts whose effective behaviors are the same as multiple hosts (so that, for example, one Tor client instance could simulate several), resulting in fewer hosts

needed overall. To do so, one could utilize our newly provided functionality of function interposition specific to a particular host. Using this feature, Shadow could create events specific to the host, allowing for, as an example, different simulated behaviors for Tor clients and Tor relays.

An area of research that requires more examination is the possibility of single Shadow experiments run across multiple machines. While it is unlikely that doing so would see a significant performance improvement, at the very least until a greater performance improvement can be seen from greater worker thread counts on a single machine, there is still merit to pursuing this feature because of the greater availability of multiple machines than single machines with sufficient memory for large experiments. It stands to reason then that adding such functionality would allow for more research groups to experiment on large-scale Tor simulations. The greater encapsulation of hosts provided by our modifications to Shadow is likely to prove useful in implementing this functionality.

Finally, it is perhaps worth investigating why when DCE integrated elf-loader, it had a significantly larger performance impact than drow-loader did with Shadow. [Taz+13; Lac10] While it is likely that the cause is the much smaller networks and presumably much simpler hosts being run in the DCE experiments, it is also possible that DCE and/or ns-3 are making use of techniques that would prove useful to integrate into Shadow.

Chapter 6

Conclusion

This thesis aimed to allow for faster simulation of existing Tor experiments, and to allow for larger Tor experiments to be run, by constructing a new experimentation platform to do so. In pursuit of that aim, we have presented a new design for Shadow's execution of simulated processes and implemented said design. We did so using drow-loader, our custom dynamic loader, which is catered for network simulation with its unique ability to dynamically load hundreds of thousands of ELF files into isolated namespaces. We have also implemented a new scheduler for Shadow, to demonstrate the usefulness of this new design. We evaluated our implementation, and have shown considerable performance improvements, with overall run times reduced by up to 49% in some configurations.

One way this could prove beneficial is to Tor development. As can be seen in some of the experiments performed, it is not uncommon for Shadow to finish a simulation in less wall clock time than the total simulation time. By providing faster turnaround times for more moderately sized experiments that can be used to test the correctness of modifications to the Tor source code, Tor developers could take a more iterative approach to their development practices.

We have also provided examples of several lines of future research that could continue to build upon the work described here. By pursuing these lines of research, we can hope to achieve the ultimate goal of simulating Tor networks that rival, or even exceed, the size of the true Tor network.

Bibliography

[AG16] Mashael AlSabah and Ian Goldberg. "Performance and Security Improvements for Tor: A Survey". In: *ACM Computing Surveys (CSUR)* 49.2 (2016), p. 32.

[Amd67] Gene M Amdahl. "Validity of the single processor approach to achieving large scale computing capabilities". In: *Proceedings of the April 18-20, 1967, spring joint computer conference*. ACM. 1967, pp. 483–485.

[Bau+11] Kevin Bauer, Micah Sherr, Damon McCoy, and Dirk Grunwald. "ExperimenTor: A Testbed for Safe and Realistic Tor Experimentation". In: *Proceedings of the USENIX Workshop on Cyber Security Experimentation and Test (CSET 2011)*. USENIX. Aug. 2011.

[BFP15] Joseph D. Beshay, Andrea Francini, and Ravi Prakash. "On the Fidelity of Single-Machine Network Emulation in Linux". In: *International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems - MASCOTS*. IEEE. Oct. 2015.

[BL99] Robert D. Blumofe and Charles E. Leiserson. "Scheduling Multithreaded Computations by Work Stealing". In: *J. ACM* 46.5 (Sept. 1999), pp. 720–748. ISSN: 0004-5411. DOI: 10.1145/324133.324234. URL: http://doi.acm.org/10.1145/324133.324234.

[Chu+03] Brent Chun, David Culler, Timothy Roscoe, Andy Bavier, Larry Peterson, Mike Wawrzoniak, and Mic Bowman. "PlanetLab: An Overlay Testbed for Broad-Coverage Services". In: *ACM SIGCOMM Computer Communication Review*. Vol. 33. 3. ACM. Aug. 2003, pp. 3–12.

[DM09] Roger Dingledine and Steven J Murdoch. "Performance Improvements on Tor or, Why Tor is slow and what we're going to do about it". In: (2009). URL: http://www.torproject.org/press/presskit/2009-03-11-performance.pdf.

[Dre11] Ulrich Drepper. *How To Write Shared Libraries*. Version 4.1.2. 2011. URL: https://www.akkadia.org/drepper/dsohowto.pdf (visited on 08/2017).

[Dre13] Ulrich Drepper. *ELF Handling For Thread-Local Storage*. Version 0.21. 2013. URL: https://www.akkadia.org/drepper/tls.pdf (visited on 08/2017).

[Ela+12] Tariq Elahi, Kevin Bauer, Mashael AlSabah, Roger Dingledine, and Ian Goldberg. "Changing of the Guards: A Framework for Understanding and Improving Entry Guard Selection in Tor". In: *Proceedings of the 2012 ACM Workshop on Privacy in the Electronic Society*. WPES '12. Raleigh, North Carolina, USA: ACM, 2012, pp. 43–54. ISBN: 978-1-4503-1663-7. DOI: 10.1145/2381966.2381973. URL: http://doi.acm.org/10.1145/2381966.2381973.

[Fel14] Ed Felten. *Why were CERT researchers attacking Tor?* July 2014. URL: https://freedom-to-tinker.com/2014/07/31/why-were-cert-researchers-attacking-tor/ (visited on 09/2017).

[Foo+10] Denis Foo Kune, Tyson Malchow, James Tyra, Nick Hopper, and Yongdae Kim. *The Distributed Virtual Network for High Fidelity Large Scale Peer to Peer Network Simulation*. University of Minnesota, 2010. URL: https://www.cs.umn.edu/research/technical_reports/view/10-029 (visited on 08/2017).

[Ged16] John D Geddes. "Privacy and Performance Trade-offs in Anonymous Communication Networks". PhD thesis. University of Minnesota, 2016.

[Gre15] Andy Greenberg. "Tor Says Feds Paid Carnegie Mellon $1M to Help Unmask Users". In: *Wired* (Nov. 2015). URL: https://www.wired.com/2015/11/tor-says-feds-paid-carnegie-mellon-1m-to-help-unmask-users/ (visited on 09/2017).

[Hen+08] Thomas R Henderson, Mathieu Lacage, George F Riley, Craig Dowell, and Joseph Kopena. "Network simulations with the ns-3 simulator". In: *SIGCOMM demonstration* 14 (2008).

[Int01] Intel Corporation. *Intel Itanium Processor-specific Application Binary Interface (ABI)*. Tech. rep. May 2001. URL: https://refspecs.linuxfoundation.org/elf/IA64-SysV-psABI.pdf (visited on 08/2017).

[ISO11a] ISO. *Information technology – Programming languages – C*. Standard ISO/IEC 9899:2011. Geneva, CH: International Organization for Standardization, Dec. 2011.

[ISO11b] ISO. *Information technology – Programming languages – C++*. Standard ISO/IEC 14882:2011. Geneva, CH: International Organization for Standardization, Sept. 2011.

[JH12] Rob Jansen and Nicholas Hopper. "Shadow: Running Tor in a Box for Accurate and Efficient Experimentation". In: *Proceedings of the Network and Distributed System Security Symposium — NDSS'12*. Internet Society. Feb. 2012.

[Joh+13] Aaron Johnson, Chris Wacek, Rob Jansen, Micah Sherr, and Paul Syverson. "Users Get Routed: Traffic Correlation on Tor by Realistic Adversaries". In: *Proceedings of the 20th ACM Conference on Computer and Communications Security (CCS 2013)*. ACM, 2013.

[Joh+17] Aaron Johnson, Chris Wacek, Rob Jansen, Micah Sherr, and Paul Syverson. *TorPS*. 2017. URL: https://torps.github.io/ (visited on 08/2017).

[Kle17] Andi Kleen. *arch_prctl(2) Linux User's Manual*. 4.13. Sept. 2017.

[Lac10] Mathieu Lacage. "Experimentation Tools for Networking Research". PhD thesis. l'Université de Nice-Sophia Antipolis, Nov. 2010.

[LHM10] Bob Lantz, Brandon Heller, and Nick McKeown. "A Network in a Laptop: Rapid Prototyping for Software-Defined Networks". In: *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*. ACM. 2010.

[MJ15] Andrew Miller and Rob Jansen. "Shadow-Bitcoin: Scalable Simulation via Direct Execution of Multi-threaded Applications". In: CSET'15 (2015). URL: http://dl.acm.org/citation.cfm?id=2831120.2831127.

[ns313] ns-3 Project. *Using Alternative, Fast Loader*. 2013. URL: https://www.nsnam.org/docs/dce/manual/html/dce-user-elfloader.html (visited on 08/2017).

[ns317a] ns-3 Project. *DCE*. 2017. URL: https://www.nsnam.org/overview/projects/direct-code-execution/ (visited on 08/2017).

[ns317b] ns-3 Project. *ns-3*. 2017. URL: https://www.nsnam.org/ (visited on 08/2017).

[Pri17] The Trustees of Princeton University. *PlanetLab*. 2017. URL: https://www.planet-lab.org/ (visited on 08/2017).

[SGD15] Fatemeh Shirazi, Matthias Goehring, and Claudia Diaz. "Tor Experimentation Tools". In: *International Workshop on Privacy Engineering*. IWPE '15. Washington, DC, USA: IEEE Computer Society, 2015, pp. 206–213. ISBN: 978-1-4799-9933-0. DOI: 10.1109/SPW.2015.20. URL: http://dx.doi.org/10.1109/SPW.2015.20.

[Sin14] Sukhbir Singh. "Large-Scale Emulation of Anonymous Communication Networks". MMath thesis. University of Waterloo, 2014. URL: http://hdl.handle.net/10012/8642 (visited on 08/2017).

[Taz+13] Hajime Tazaki, Frédéric Uarbani, Emilio Mancini, Mathieu Lacage, Daniel Camara, Thierry Turletti, and Walid Dabbous. "Direct Code Execution: Revisiting Library OS Architecture for Reproducible Network Experiments". In: *Proceedings of the Ninth ACM Conference on Emerging Networking Experiments and Technologies*. CoNEXT '13. Santa Barbara, California, USA: ACM, 2013, pp. 217–228. ISBN: 978-1-4503-2101-3. DOI: 10.1145/2535372.2535374. URL: http://doi.acm.org/10.1145/2535372.2535374.

[TIS95] TIS Committee. *Tool Interface Standard (TIS) Executable and Linking Format (ELF) Specification*. Tech. rep. May 1995. URL: http://refspecs.linuxbase.org/elf/elf.pdf (visited on 08/2017).

[Tor16] Tor Project. *Tor Research Safety Board*. 2016. URL: https://research.torproject.org/safetyboard.html (visited on 08/2017).

[Tor17a] Tor Project. *Chutney*. 2017. URL: https://gitweb.torproject.org/chutney.git (visited on 08/2017).

[Tor17b] Tor Project. *Tor Metrics Portal*. 2017. URL: https://metrics.torproject.org/ (visited on 08/2017).

[TS11] Florian Tschorsch and Björn Scheuermann. "Tor is unfair — And what to do about it". In: *Local Computer Networks (LCN), 2011 IEEE 36th Conference on*. IEEE. 2011, pp. 432–440.

[TS13] Florian Tschorsch and Björn Scheuermann. "How (not) to build a transport layer for anonymity overlays". In: *ACM SIGMETRICS Performance Evaluation Review* 40.4 (2013), pp. 101–106.

[TS16] Florian Tschorsch and Björn Scheuermann. "Mind the Gap: Towards a Backpressure-Based Transport Protocol for the Tor Network." In: *NSDI*. 2016, pp. 597–610.

[Ung17] Nik Unger. *NetMirage*. 2017. URL: https://crysp.uwaterloo.ca/software/netmirage/ (visited on 08/2017).

[VR88] Mark T Vandevoorde and Eric S Roberts. "WorkCrews: An abstraction for controlling parallelism". In: *International Journal of Parallel Programming* 17.4 (1988), pp. 347–366.

[Wat13] University of Waterloo. *CrySP RIPPLE Facility*. 2013. URL: https://ripple.uwaterloo.ca/ (visited on 07/2017).

[Wil14] Tim Wilson-Brown. *Get tor working with ns-3*. 2014. URL: https://trac.torproject.org/projects/tor/ticket/13978 (visited on 08/2017).