Co-opting Linux Processes for High-Performance Network Simulation

Rob Jansen, *U.S. Naval Research Laboratory;* Jim Newsome, *Tor Project;*
Ryan Wails, *Georgetown University, U.S. Naval Research Laboratory*

https://www.usenix.org/conference/atc22/presentation/jansen

This paper is included in the Proceedings of the
2022 USENIX Annual Technical Conference.

July 11–13, 2022 • Carlsbad, CA, USA

978-1-939133-29-8

Open access to the Proceedings of the
2022 USENIX Annual Technical Conference
is sponsored by

Co-opting Linux Processes for High-Performance Network Simulation

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

Jim Newsome
Tor Project
jnewsome@torproject.org

Ryan Wails
Georgetown University,
U.S. Naval Research Laboratory
ryan.wails@nrl.navy.mil

Abstract

Network experimentation tools are vitally important to the process of developing, evaluating, and testing distributed systems. The state-of-the-art simulation tools are either prohibitively inefficient at large scales or are limited by nontrivial architectural challenges, inhibiting their widespread adoption. In this paper, we present the design and implementation of *Phantom*,[1] a novel tool for conducting distributed system experiments. In Phantom, a discrete-event network simulator directly executes unmodified applications as Linux processes and innovatively synthesizes efficient process control, system call interposition, and data transfer methods to co-opt the processes into the simulation environment. Our evaluation demonstrates that Phantom is up to 2.2× faster than Shadow, up to 3.4× faster than NS-3, and up to 43× faster than gRaIL in large P2P benchmarks while offering performance comparable to Shadow in large Tor network simulations.

1 Introduction

Network experimentation tools promote the progression of network science: they aim to *realistically* reproduce the effects of distributed networks at *scale* in a *controlled* environment, enabling the scientific evaluation of performance and security across a range of system characteristics. Experimentation tools are particularly useful for *large-scale* distributed systems that are deployed in the real world, such as the globally expansive domain name system [50], peer-to-peer and content distribution networks [14], decentralized data-storage networks [52], and overlay networks [17]. Due to the sizes of these deployments and the internet's great heterogeneity and rapid change [19], it would be extremely difficult to run scientifically controlled, replicable experiments with them in the real world. Tools that enable realistic, scalable, and controlled experimentation of large-scale distributed systems can help accelerate research, development, and education.

Approved for public release: distribution is unlimited.

Large-scale distributed systems are often characterized by a complex set of algorithms and protocols that run in *application-layer* software. Previous work has found that it is prudent to *directly execute* this software as part of the experimentation process to promote realism [30, 54, 62]. However, there are nontrivial architectural challenges in designing tools that meet the scalability and realism requirements. Emulators such as Mininet [45] do not support large-scale systems because they are vulnerable to time distortion during periods of overload [44]. Simulators such as NS-3 [26] run application *abstractions* in place of real software which can cause unrealistic behavior and lead to invalid results [54].

To meet the large-scale distributed system requirements, the state-of-the-art tools are designed with hybrid architectures wherein a network simulator directly executes application code. However, tools that load and execute applications in *plugin namespaces* (i.e., NS-3-DCE [62] and Shadow [30]) suffer from compatibility and correctness issues and high maintenance costs: applications must be recompiled as plugins, complex code is required to load and run them, and the system calls they make often leak outside of the simulation. On the other hand, tools that run applications as *Linux processes* (i.e., gRaIL [54]) incur considerable inter-process overhead: we have measured at least a 10× performance penalty in running gRaIL due to inefficient process control, system call interposition, and data transfer mechanisms. No existing network simulator simultaneously overcomes the compatibility, correctness, maintenance, and performance challenges found in the state-of-the-art tools.

Introducing Phantom: We present *Phantom*,[1] a novel, multi-process network simulator that: (i) precludes the compatibility, correctness, and maintenance issues that have plagued plugin-based designs; and (ii) overcomes the performance challenges of existing multi-process designs by innovatively synthesizing efficient process control, system call interposition, and data transfer mechanisms. In Phantom, a discrete-event network simulation core directly executes unmodified

[1] We use *Phantom* as a codename in this paper, but our design is merged into the open-source Shadow simulator and synonymous with Shadow v2 [4].

applications as Linux processes, allowing us to take advantage of native Linux process isolation and management. Phantom co-opts the Linux processes into a simulation environment by (i) preloading a shim library (via `LD_PRELOAD`) that is used to establish efficient mechanisms for process control and function interception; (ii) installing a secure computing (i.e., `seccomp`) filter in the processes to guarantee interposition on system calls that are not preloadable; and (iii) using a novel inter-process memory mapper that allows us to directly read and write process memory without incurring inter-process communication (IPC) overhead. Once the processes are co-opted, Phantom efficiently emulates system calls they make and facilitates communication over a simulated network.

Novel Contributions: This paper makes the following novel contributions to the state of the art in network simulation:

- The innovative design of Phantom, which for the first time shows how to minimize inter-process overhead in a hybrid, multi-process network simulator.
- A high-performance implementation of Phantom.
- An extensive evaluation of Phantom through which we find that it is up to 2.2× faster than Shadow, up to 3.4× faster than NS-3, and up to 43× faster than gRaIL in large P2P benchmarks while offering performance comparable to Shadow in large Tor network simulations.
- A verification of Phantom's accuracy in small LAN and WAN networks and in large Tor overlay networks.

Impact: This work has high potential for broad impact across multiple communities for the purposes of research, development, and education. First, researchers building software prototypes can use Phantom to quickly evaluate their new distributed system designs in a large-scale network without needing to worry about complicated deployments that are difficult to manage. Second, Phantom can be built into developers' testing frameworks so that new code can be continuously tested and discovered bugs can be identically reproduced. Third, with facilities to introduce network events (e.g., intermittent delays or failures), Phantom could help teach network and distributed systems courses. The Tor Project has already started using Phantom to develop and test new congestion control protocols before deploying them to the Tor network [57].

Availability: Phantom is merged into the open-source Shadow project as of v2 [4] and our artifacts are publicly available [3].

2 Background and Motivation

We motivate the need for Phantom by identifying the key requirements, existing architectures, and challenges for realistically simulating large-scale distributed systems. (See Appendix A for extended background on related tools.)

2.1 Requirements

Scalability: Recent work finds that it is imperative to run network experiments as close as possible to the deployed scale because reducing the scale can lead to a significant loss of confidence in the experimental results [40]. Although some statistical confidence can be recovered with repeated trials, it can take many more trials at a smaller scale to achieve the same confidence as larger scale simulations [40].

To increase the scale at which we can run network experiments, a correct and valid execution of the simulation workload should not depend on the computational abilities of, or passage of time on, the host machine. Decoupling the simulation from time and computational constraints allows us to scale without introducing artifacts in the results due to over-provisioning and time-distortion [44].

Realism: Distributed systems are often composed of a diverse set of applications that each contain complex logic. We should directly execute these applications in order to guarantee that our experiments identically replicate their logic and obtain the highest application fidelity possible [30, 54, 62].

Deployed system software is often under active development to fix bugs and develop enhancements. We should execute applications the same way they would be executed in deployment; we should not require recompilation or the maintenance of application patches or abstractions. Running unmodified applications enables us to decouple the application logic and programming language from that of the simulation.

Control: Large-scale distributed systems contain many variables, and changing any one of them can have cascading network effects that can lead to unexpected behaviors or results. We should support deterministic execution to obtain scientific control and to guarantee that the results produced by an experiment can be independently and identically replicated.

2.2 Traditional Architectures

Tools implementing strictly traditional architectures are unsuitable for evaluating large-scale distributed systems with logic primarily contained in *application-layer* software.

Simulation: Network simulators such as NS-3 [26] scale independently of the wall-clock time [67] and offer precise experimental control due to deterministic execution [13]. However, simulators traditionally run application *abstractions* in place of real software which can cause unrealistic behavior and lead to invalid results [54]. As a result, traditional simulators do not fulfill the application realism requirement.

Emulation: Network emulators such as Mininet [45] directly execute applications using real kernel network stacks and therefore offer better application realism. However, emulators lack perfect scientific control due to non-determinism [12]. Moreover, emulators are generally unable to scale independently of computational constraints: if the experiment host machine is overloaded, time distortion will exacerbate reproducibility issues [44]. We confirm this claim with an experiment in which we find that as the host machine becomes more loaded with virtual peers, *its packet forwarding capacity is limited* and a decreasing fraction of the sent packets are correctly forwarded (see §5.4 and Figure 14 for details). As a result, traditional emulators are useful only at small scales.

Table 1: Properties of Network Experimentation Architectures

Architecture	Example Tool	Scalability[⋆]	Realism[†]	Control[‡]
Emulation	Mininet [45]	○	●	○
Simulation	NS-3 [26]	●	○	●
Hybrid	This Work	●	●	●

[⋆] Experiments scale independent of time or computational constraints.
[†] Unmodified applications can be directly executed without recompilation.
[‡] Results can be deterministically replicated with the same RNG seed.

2.3 Hybrid Architectures and Challenges

A hybrid architecture is characterized by the ability to directly execute applications to promote realism while still running them in the context of a cohesive network simulation. As a result, a hybrid architecture enjoys the advantages of both emulation and simulation and offers the best opportunity to fulfill the scalability, realism, and control requirements discussed in §2.1 (see Table 1). However, there are numerous challenges with hybrid architectures that we believe have inhibited tools implementing them from achieving widespread adoption. We describe these challenges by the method for executing applications: *plugin namespaces* and *processes*.

Plugin Namespaces: In this approach, the simulator loads each application into a new plugin namespace (e.g., using `dlmopen`) and directly executes the application in the context of that namespace while using function interposition (via `LD_PRELOAD`) to hook the loaded applications into the simulation environment. A plugin design is implemented in both NS-3-DCE [62] and Shadow [30] and has several limitations:

– *Compatibility:* The domain of supported applications is limited to those that are compiled as position-independent libraries (`PIC`) or executables (`PIE`) that export their symbols to the dynamic symbol table (`rdynamic`), are dynamically linked to `libc`, and make all system calls through `libc`. Rebuilding is tedious and impossible if the source code is not available (e.g., closed-source software or malware).
– *Correctness:* Relying solely on preloading is unreliable because only dynamically linked functions (e.g., those in `libc`) can be intercepted using `LD_PRELOAD`; system calls invoked via statically linked code or assembly instructions will leak outside of the simulation and cause errors.
– *Maintainability:* A custom dynamic loader [63] is required to load more than 16 namespaces at once, and a portable threading library [48] is used to support multi-threaded applications (these account for 62k LoC in Shadow; see §4). `libc` functions with nontrivial functionality must be reimplemented in order to intercept the system calls they make.

These challenges have limited Shadow's use to Tor network simulation [40] while work on simulating Bitcoin has been abandoned [48] and work on NS-3-DCE has mostly stalled.

Processes: In this approach, applications are executed as standard Linux processes and hooked into the simulation through the system call interface using standard kernel facilities. This design overcomes many of the limitations of the plugin approach: (i) the simulator can execute any existing application without rebuilding it; (ii) kernel subsystems guarantee reliable process isolation and correct system call interception; and (iii) the maintenance of a custom loader, threading libraries, and reimplemented `libc` functions is no longer required. However, the naïve way of connecting multiple processes in a cohesive simulation as demonstrated in gRaIL [54] requires the kernel's process control (`ptrace`) subsystem and is significantly less performant than the plugin approach: we show in §5.4 that the run time of gRaIL (which extends NS-3) is 13× that of NS-3 alone, and 43× that of Phantom in experiments with fixed P2P messaging workloads. Worse performance in gRaIL's multi-process design can be attributed to:

– *Process control:* The simulator needs to control the execution state of the processes as they progress through simulated time. The `ptrace` process control mechanism (`PTRACE_ATTACH` or `PTRACE_TRACEME`) incurs overhead that is *quadratic* in the total number of attached processes, limiting scalability (see Appendix B.1).
– *System call interposition:* The simulator needs to intercept system calls made in the processes so they can be emulated. The `ptrace` system call mechanism (`PTRACE_SYSCALL`) requires *at least* 4 context switches *for every system call*, contributing substantial overhead relative to a same-process function call (see Appendix B.2).
– *Data transfer:* The simulator needs to access system call arguments referencing process memory (e.g., data buffers). The `ptrace` memory access mechanism (`PTRACE_PEEK` and `PTRACE_POKE`) requires an additional system call and mode transition *for each word of memory*, making it inefficient for large structs and buffers (see Appendix B.3).

Ideally, we want a simulator with the higher performance of the uni-process, plugin-based Shadow design (which does not incur inter-process overhead) *and* the improved compatibility, correctness, and maintainability of the multi-process gRaIL design. However, it was previously unknown if this ideal is attainable due to the multi-process challenges; indeed, we show throughout §5 that even a more efficient use of `ptrace` (see Appendix B) is still less performant than a uni-process design.

3 Design

In this section we describe the novel multi-process Phantom design that eliminates the limitations of the state-of-the-art plugin-based architecture and overcomes the performance challenges of the state-of-the-art process-based simulator.

3.1 Overview

The main component in Phantom is a discrete-event simulator which drives the simulation (see Figure 1). After initialization, the simulator directly executes the real applications of an experiment as Linux processes while using interprocess communication channels (IPC) between the application and simulator processes. Phantom co-opts the applica-

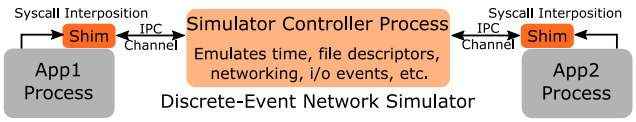


Figure 1: Overview of the Phantom design. Phantom directly executes application processes, intercepting system calls and handling them using a shim and an inter-process communication channel.

tion processes into the simulation by intercepting all system calls they make (e.g., `socket`, `listen`, `connect`, `send`, `recv`, `poll`, etc.) rather than allowing them to be handled by the Linux kernel. Phantom handles intercepted system calls by internally simulating common kernel functionalities that most applications expect to be available, such as networking facilities (e.g., buffers, protocols, and interfaces), event notification facilities (e.g., select, poll, and epoll), and file descriptor facilities (e.g., files, sockets, and pipes). As a result, Phantom emulates a Linux kernel to the applications while connecting them through a virtual, simulated network, and the applications need not be aware that they are running in a simulation.

3.2 Components

3.2.1 Simulation Controller Process

Phantom is a parallel, conservative-time, discrete-event network simulator that emulates a Linux kernel to the applications it executes. Simulations are driven by a single controller process which has two primary functions that occur successively during an *initialization* phase and an *execution* phase.
Initialization Phase: During initialization, the controller reads and processes configuration inputs. The inputs specify a number of virtual hosts that should be simulated, a network graph model that should be used to model network characteristics such as routing, latency, and packet loss between the virtual hosts, and the file paths and arguments needed to directly execute the applications on the virtual hosts. The controller initializes internal simulation state accordingly.
Execution Phase: Simulation work is organized into *events* that each occur at a discrete simulation time. Each event is assigned to a virtual host and stored in a host-specific *event queue*: a min-heap that sorts events by their simulation time.

The controller manages the global simulation clock and synchronizes simulation time by using time barriers to establish discrete *execution rounds*: time intervals during which events may be safely executed in parallel. The time barrier in a round is set such that no event that is executed for any host in that round will enqueue a new event for any other host in the same round. This conservative-time algorithm guarantees that simulation time always advances on each host, even when concurrently executing distinct hosts' events. When the next event time in every host's event queue exceeds the time barrier for the current round, the controller updates the global clock and advances the execution round.

3.2.2 Parallel Worker Threads

Phantom concurrently executes the events in each execution round using *worker threads* (workers) that are managed with high level abstractions we call *logical processors* (LPs). Phantom allows a configurable number of LPs and controls the state of an independently configurable number of workers such that only a number of workers equal to the number of LPs are concurrently active.[2]

The following algorithm employs a work stealing [10, 65] strategy to schedule the worker threads, ensuring that each LP will always be running a worker thread as long as one with remaining work exists. When an execution round begins, one worker thread starts *running* for each LP while the remaining workers remain *waiting*. While running, a worker dequeues and executes all events that occur within the current round (as set by the controller) for all hosts assigned to it. When a worker completes all outstanding events for the current round, it: (i) starts running another waiting worker that has yet to run in this round (if any exist); and (ii) starts waiting to be run again during the following round. An execution round ends when all workers have entered the waiting state.

3.2.3 Direct Application Execution

During initialization, each virtual host is configured to directly execute some number of applications. Phantom internally creates virtual process and thread data structures to store the state needed to manage the execution of the applications (e.g., file descriptor tables and standard input/output handles).
Managed Processes and Threads: Phantom directly executes specified application binaries and allows for configuration of the command-line arguments and the start time within the simulation. Each application is launched by a Phantom worker with a `vfork`+`execvpe` sequence.

The application execution procedure results in the creation of one or more Linux processes and threads that are *managed* by their parent Phantom worker. Each worker (i) uses our preload shim library to co-opt their managed processes into the simulation, and (ii) uses our inter-process communication mechanisms to modulate the running state of the managed processes such that only one of a worker and its managed processes are running at any time (thus maintaining that only one task per LP is concurrently active).
Preload Shim: In order to assist with controlling the managed processes and threads, we create a custom shared library, subsequently referred to as "the shim", which is loaded into each managed process's address space using the `LD_PRELOAD` environment variable. We use the shim to: (i) execute initialization code in the shim's constructor functions and establish an inter-process communication channel (see §3.2.6); and (ii) intercept functions defined in libraries that are dynamically linked to the applications (e.g., `libc`; see §3.2.4).

[2] Limiting the number of LPs to be at most the number of available CPU cores avoids performance degradation caused by CPU oversubscription.

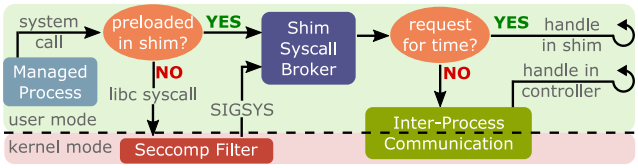


Figure 2: Control flow when intercepting system calls in Phantom.

3.2.4 System Call Interposition

Phantom co-opts processes into the simulation by intercepting functions at the system call interface using two interception strategies: *preloading* and `seccomp` (see Figure 2).

Primary Strategy: Preloading: Recall from §3.2.3 that Phantom preloads a shared library shim into each process it executes using the `LD_PRELOAD` environment variable. Because the shim is preloaded, the dynamic loader loads the shim before all other shared objects linked to the managed process and the shim is the first library searched when attempting to dynamically resolve symbols. This feature allows us to selectively override functions in other shared libraries by supplying identically named functions with alternative implementations inside the shim. Preloading is efficient, as it changes only the address of the instruction that is next executed when a dynamically-linked function is invoked. Therefore, we use preloading as our primary interception strategy.

Notice that preloading works by intercepting *shared library functions*, not *system calls*. While preloading can interpose dynamically linked calls to `libc` system call wrapper functions made from outside of `libc`, it cannot interpose the statically linked calls made from *inside* of `libc` (e.g., internal calls from `printf` to `write`).[3] If using preloading alone, we would need to reimplement `printf` and any other `libc` functionality we wanted to support and not just the system call wrappers—an untenable engineering burden. Preloading alone would also fail to intercept system calls made without using `libc` at all, e.g., those made by directly using a `syscall` instruction.

Secondary Strategy: `seccomp`: Phantom intercepts system calls that are not handled by the preloading strategy using the kernel's `seccomp` (secure computing) facility. The `seccomp` facility enables a process to set a filter on the system calls that are made by the process and to associate an action with the filter. We install a `seccomp` filter that traps all system calls except for: (i) `sigreturn`; and (ii) system calls originating from Phantom's own preloaded shim. We install a `SIGSYS` signal handler for system calls trapped by the `seccomp` filter; whenever a system call matching the filter is invoked, the kernel traps it and instead calls our signal handler function.

We use `seccomp` as our *secondary* interception strategy because, although it can intercept all system calls, it is less efficient than preloading; it requires: (i) a mode transition from the process to the kernel when the system call is invoked; (ii) execution of the `seccomp` filter; and (iii) a mode transition back to the process to invoke the shim callback function. Because most system calls are preloadable, we infrequently incur the additional overhead from `seccomp` in practice.

[3] APIs that invoke vDSO functions (e.g., `time`) rather than make system calls can either be preloaded or we can dynamically rewrite the vDSO to guarantee that it makes interposable system calls [55].

3.2.5 Emulating System Calls

Both system call interception strategies from §3.2.4 result in a `syscall` handler function being executed in the shim, i.e., within the managed process. System calls can be emulated either directly in the shim or in the controller (see Figure 2).

In the Shim: Frequently made system calls that can be emulated using little state from the controller can be serviced directly in the shim without incurring additional overhead related to IPC. For example, the shim directly handles the `time`, `gettimeofday`, and `clock_gettime` system calls by arranging for the controller to share and maintain the current simulation time in a shared memory control block that is accessible to the shim as described in §3.2.6.

In the Controller: The remaining system calls are serviced in the simulator controller process. The system call number and arguments are sent to the controller using the IPC control channel as described in §3.2.6. The controller handles the system calls internally using lightweight implementations that effectively form a simulated kernel that completely replaces the functionality normally provided by the Linux kernel. The simulated kernel (re)implements (i.e., simulates) important system functionality, including: the passage of time; input and output operations on file, socket, pipe, timer, and event descriptors; packet transmissions with respect to transport layer protocols such as TCP and UDP; and aspects of computer networking including routing, queuing, and bandwidth limits. (See Appendix D for additional details.) Importantly, this approach enables us to establish a private, simulated network environment that is completely isolated from the real network, but is internally interoperable and entirely controllable.

Determinism: Phantom uses a pseudorandom generator that is seeded with a configurable seed as its single source of randomness throughout the simulation. Care is taken to ensure that all random bytes that are needed during the simulation are initiated from this source, including during the emulation of system calls such as `getrandom` and when emulating reads from files like `/dev/*random`. This approach allows Phantom to produce deterministic simulations, improving scientific control over the experimentation process and enabling experimental results to be replicated.

3.2.6 Managed Process-to-Controller Communication

We use control channels to exchange *fixed*-size messages with each managed process (e.g., system call arguments), and a memory manager to exchange *dynamic* amounts of data (e.g., a buffer passed to a `send` system call; see Figure 3).

Control Channel: Phantom establishes a control channel with the shim of each managed process by allocating an initial block of shared memory and sharing the handle to this

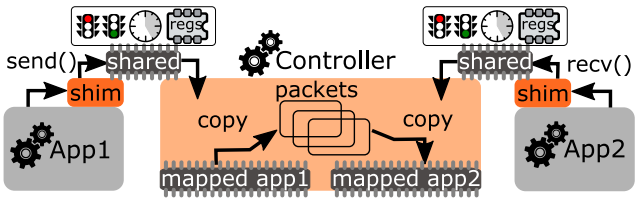


Figure 3: Phantom uses shared memory as a control channel, modulating control using semaphores. The app1 shim intercepts `send`, writes its arguments into the shared syscall registers, and then uses semaphores to pass control. The controller reads the registers, uses the memory manager to directly copy the send buffer into simulated packets, and schedules an event so the packets arrive at the receiver following network semantics. The controller writes the retval register and passes control back so app1 continues running. An analogous process occurs when app2 calls `recv` (or any other system call).

memory during process startup using an environment variable. This *control block* uses a fixed data structure layout that includes semaphores and messaging state (e.g., system call arguments). The semaphores provide a safe and efficient way for a message sender to signal that a new message is available and for a message receiver to wait for a new message; the controller uses this functionality to modulate the execution state of the process (see §3.2.7). We use shared memory and semaphores because we found this combination to perform better than alternative approaches (see Appendix C).

Memory Manager: We designed an inter-process memory access manager to enable the controller to directly and efficiently read and write the memory of each managed process without extraneous data copies or control messages. The memory manager tracks the memory mappings that are active across various regions of a process's memory, which are analogous to the mappings found in the `/proc/<pid>/maps` file. Upon initialization, the memory manager creates a sparse memory file for each process, where a virtual address in the process corresponds to the same offset in the file. The memory manager initially *remaps* the process's stack and heap memory regions into this file. As the process runs, the memory manager brokers all read, write, or other mapping requests that involve managed process memory in order to: (i) also map requests for anonymous private regions (such as those made when serving large allocation requests) into the shared file; (ii) maintain a consistent view of the process's address space; and (iii) simplify system call handling by translating memory pointers to shared memory pointers as needed. Whenever the memory manager receives an access request for an address that is not mapped into the shared file, it utilizes the kernel's `process_vm_readv` and `process_vm_writev` facilities to directly transfer data between the controller and the managed process's address space without copying it into kernel space.

3.2.7 Managed Process/Thread Scheduling

We use the IPC control channel from §3.2.6 to control the execution state of each managed process. When a process first loads, it immediately waits on the channel semaphore to receive a message from Phantom before starting. When a Phantom worker runs (following the algorithm in §3.2.2), the worker initially sends a start message to the process it manages and waits to receive a message back from the process. The process then runs until it invokes a system call that is interposed as described in §3.2.4, sends a system call request message back through the control channel to the waiting Phantom worker, and waits to receive the system call result message from Phantom.

There are two possible scheduling outcomes when a Phantom worker handles a system call requested by a managed process. For system calls that can be handled immediately (non-blocking calls, or blocking calls for which a result is ready), the Phantom worker returns the result over the control channel and the scheduling cycle continues. For system calls that cannot be handled immediately (blocking system calls whose result is not ready), the Phantom worker must wait for some condition to become true (e.g., a packet to arrive or a timeout to occur). Such conditions are internally registered, and then the worker leaves the managed process in an idle state while it continues executing simulation events (and advancing simulation time). When the condition later becomes true (e.g., a timeout occurred), the worker executes an event that causes it to check the system call state and return the timeout result to the process over the control channel. The process continues executing and the scheduling cycle continues.

The effect of this scheduling process is that each Phantom worker only allows a single thread of execution across all processes it manages; each of the remaining managed processes/threads will always be idle, waiting for a result message from the worker for the previously requested system call. Using this scheduling process, Phantom has precise control over the execution state of all managed processes and guarantees nonconcurrent access of managed processes' memory through the memory manager from §3.2.6.

3.2.8 Linux CPU Scheduling

Phantom is designed to work with the Linux CPU affinity (i.e., CPU pinning) scheduling feature. CPU affinity is a scheduling attribute associated with running Linux processes. A process's CPU affinity can be adjusted to restrict the process to run only on a specified subset of CPUs (e.g., a single CPU). CPU pinning can improve performance by reducing the frequency of cache misses, CPU migrations, and context switches. In particular, Linux semaphores shared between two same-core processes incur fewer context switches than when shared between cross-core processes (see Appendix C). Recall that Phantom will run either a worker thread or one of its managed processes, but never both at the same time. This design choice enables us to naturally pin each worker and all of its managed processes to the same core in order to capitalize on the CPU pinning performance benefits.

4 Implementation

We implement Phantom using the plugin-based Shadow as a basis because: (i) we will show in §5.4 that Shadow outperforms other simulators; and (ii) it will be fairer to compare the plugin- and process-based architectures using tools that share the same foundation. See Appendix A.3 for Shadow details.

Transforming Shadow: We forked Shadow v1.14.0 and identified the components that are no longer necessary for Phantom. Of the 94,259 lines of code (LoC) in Shadow v1.14.0,[4] we removed 47,959 LoC (50.9%) containing a custom version of the GNU portable threads library that was used to simulate application threading [48], 14,498 LoC (15.9%) containing a custom loader that dynamically loads plugins using `dlmopen` [63], and 6,559 LoC (7.0%) that implemented the interface between Shadow and the `libc` functions it preloads. We also found that 6,315 LoC (6.7%) implemented tests and 2,123 LoC (2.3%) implemented tools, leaving just 16,805 LoC (17.8%) implementing core simulator functionality that Phantom integrates (see Appendix D for more details).

Implementing Phantom: We implemented Phantom's design from §3 on top of our stripped down version of Shadow. Our full Phantom implementation supports 164 system calls and contains 56,742 LoC: tests account for about 15,653 LoC (27.6%), tools account for 1,956 LoC (3.4%), and the remaining 39,133 LoC (69.0%) implements core functionality.

5 Evaluation

We evaluate Phantom by running micro- and macrobenchmarks, by verifying its simulation accuracy, and by comparing it to related tools. (See Appendix E for additional details.)

In our benchmarks, we compare three distinct state-of-the-art simulator architectures (see Appendix A): (i) multi-process, `seccomp` (Phantom); (ii) multi-process, `ptrace` (gRaIL); and (iii) uni-process, plugin namespaces (Shadow). For fairness, we compare all three architectures running on top of an identical simulator framework and network stack (i.e., Shadow's), thus ensuring that we can isolate performance differences and attribute them *exclusively* to the change in architecture and not to, e.g., differently inefficient code running in independent code-bases.[5]

All experiments use CPU pinning and our primary interception strategy (preloading) unless otherwise noted. All simulations were repeated ten times with unique seeds; we present the results as the mean across the ten trials with 99% CIs.

[4] LoC are counted with the `scc` tool: https://github.com/boyter/scc

[5] Because gRaIL was originally implemented on top of NS-3, we ported the design to Shadow by implementing `ptrace` as an optional alternative to the `seccomp` secondary interposition strategy. We found and mitigated many sources of `ptrace` overhead (see Appendix B) and our implementation should be considered an optimized, near-best-case version of gRaIL.

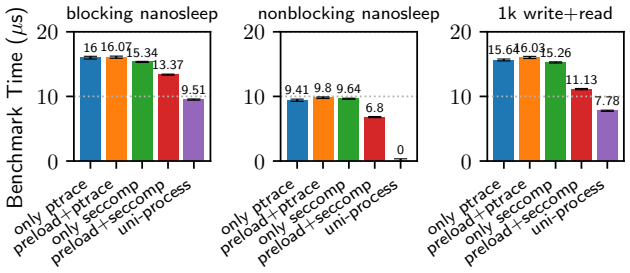


Figure 4: Time to execute blocking, nonblocking, and io-based system calls using several interception methods, compared to Shadow's uni-process preload-based design.

5.1 Performance: Microbenchmarks

Setup: We anticipate that one of the major sources of overhead in a multi-process design is due to inter-process communication and context switching, which in Phantom occurs whenever a system call is executed. There are three main types of system calls: (i) blocking calls that require the simulator to update state (e.g., advance time) before returning; (ii) nonblocking calls that can return immediately; and (iii) input/output (io) calls that involve reading or writing a dynamically sized buffer. We benchmark these operations using a small program that either invokes the `nanosleep` system call (with a timeout of 1 for blocking or 0 for nonblocking), or invokes a `write` and then a `read` operation on a `pipe`. The program loops repeatedly for 10k iterations and measures the time required to complete each benchmark after timing 10k iterations of a no-op as a baseline. We report the difference between the mean time to execute each of the three benchmarks and the mean time to execute the no-op baseline.

Results: We ran the benchmarks in our multi-process architecture using several alternative interception methods, and in Shadow's uni-process preload-based architecture. Figure 4 shows similar trends across all three benchmarks. First, we notice that using preloading and `ptrace` together is slightly slower than using `ptrace` alone; this is because the shim intercepts the system call and then (since it does not have a handler) it invokes the `system` function to pass control to `ptrace`, which adds a few instructions relative to the standard use of `ptrace`. Second, `seccomp` with preloading is significantly faster than `seccomp` alone (and both `ptrace` modes), because preloading allows us to intercept system calls without incurring the overhead of a mode transition and the execution of the `seccomp` filter. Third, the uni-process design is the fastest of all methods tested; while the blocking and io-based system calls incur some overhead due to switching portable threads (using `setjmp` and `longjmp`), a non-blocking system call is effectively a function call.

Figure 5 shows the results from running our io-based benchmark while setting the buffer size to 1k, 4k, 16k, and 64k bytes. Phantom with the relatively simple approach of using `process_vm_readv` and `process_vm_writev` is the

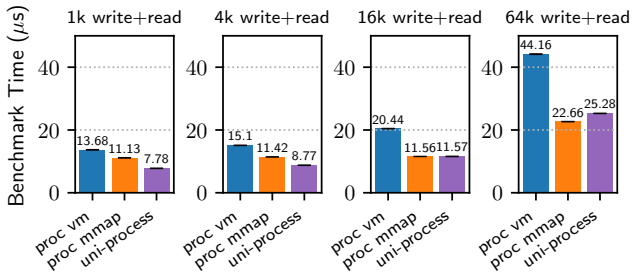


Figure 5: Time to execute io-based system calls using Phantom's `process_vm_read` and `process_vm_write` fallback facilities compared to its primary inter-process memory mapping design.

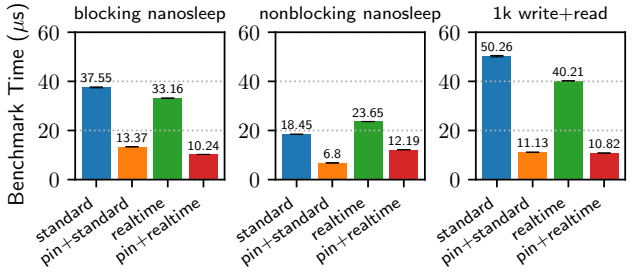


Figure 6: Time to execute our microbenchmarks in Phantom when using the Linux CPU pinning and realtime scheduling features.

slowest. This is partly from the context switch overhead between Phantom and the managed thread for each `read` and `write` call, and the kernel/user mode transition overhead of making the `process_vm` system calls. While these overheads are not dependent on the buffer size, and are amortized for larger buffer sizes, the `process_vm` system calls have significant per-page overhead for validating permissions, pinning each page in memory before doing the copy, and then unpinning them. Hence, the `process_vm` approach gets significantly *worse* than the alternatives as the buffer size increases.

While Phantom's `mmap`-based approach (§3.2.6) still has the fixed overhead of context switches between Phantom and the managed thread for each `read` and `write`, it uses an interval-map of mapped *regions* (not *pages*) to validate and translate each pointer. Since this cost is fixed, rather than per-page as with the `process_vm` calls, its overhead relative to the uni-process approach is amortized for larger buffers and becomes less than the uni-process overhead for 64 KB buffers.

Figure 6 shows the time to execute our benchmarks using the Linux CPU pinning feature described in §3.2.8 in addition to the `sched_fifo` Linux realtime scheduler. We observe that CPU pinning significantly improves Phantom's performance while mixed results are obtained when using realtime scheduling. Run time under realtime scheduling decreases by 48–73% when adding CPU pinning, indicating that the primary benefit is from pinning. CPU pinning improves performance particularly well in Phantom due to our design in which a worker modulates its running state and that of each of its managed processes such that no two of these run concurrently. Therefore, workers and their managed processes will effectively share the same CPU core, improving caching and limiting cross-core migration. We also tested Phantom using `ptrace` and Shadow's uni-process design and found that pinning provides comparable or better performance than other modes (see Appendix E.1 for more details).

5.2 Performance: Macrobenchmarks

While microbenchmarks enable us to test the effects of system call operations in isolation, macrobenchmarks provide us with a more wholistic understanding of performance while simulating a larger distributed network.

5.2.1 Setup

To run our macrobenchmarks, we write a simple peer-to-peer (P2P) messaging application whose behavior is inspired by the parallel hold (PHOLD) model commonly used to benchmark discrete-event simulators [20].[6] Our P2P application uses standard UDP sockets for network communication and works as follows. Each peer first creates and sends some number m of messages at startup using `sendto` and then uses `poll` to wait for incoming messages to arrive. Whenever a message is received with `recvfrom`: (i) a number c of AES encryptions and c AES decryptions are performed to produce computational load; and (ii) a new message with a 1k payload is created and sent to produce network load. Whenever a peer sends a message, it makes a weighted choice of the destination peer where peers' weights are drawn from a configurable probability distribution W; we use W to create unbalanced workloads across peers.

To benchmark performance we create distributed networks with p peers, each running our P2P application on a distinct virtual host. The network latency between each pair of hosts is set to 50 ms and each host's bandwidth is unrestricted. All peers start at the same time and run for ten simulated seconds, resulting in 200 communication rounds. Unless otherwise mentioned, our experiments use defaults of p=1k peers, m=100 messages, c=0 AES (encrypt, decrypt) sequences, and W is the exponential function e^{-3x} for $x \in [0,1]$ (to produce unbalanced peer workloads).

5.2.2 Results

Interception Strategy and LP Count: We run experiments that vary the interception strategy and LP count to investigate their effects on performance. Our results in Figure 7 show that the uni-process Shadow simulation completes faster than the multi-process Phantom simulations when using 14 or fewer LPs (consistent with our microbenchmark results). However, when the number of LPs exceeds 14, the `seccomp` interception strategy (with or without preloading) performs better than both the `ptrace` strategy and the uni-process design. We observe diminishing returns and performance regressions

[6] We modify the PHOLD model because it is shown to lead to well-balanced workloads that are not representative of real-world networks [11].

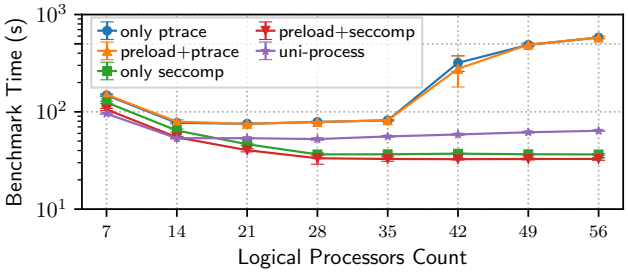


Figure 7: The time to complete our P2P benchmark across a varying set of interception strategies and number of logical processors (i.e. concurrently active worker threads). The y-axis is in log scale.

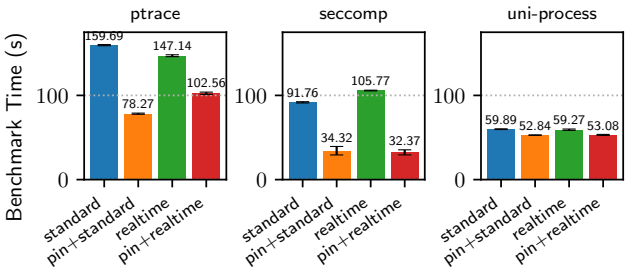


Figure 8: Time to complete our P2P benchmark when using the Linux CPU pinning and realtime scheduling features.

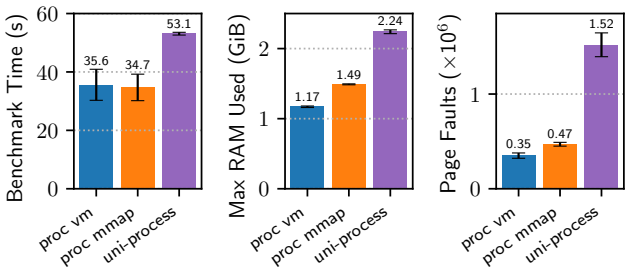


Figure 9: Performance of our P2P benchmark using Phantom's `process_vm_read` and `process_vm_write` fallback facilities compared to its primary inter-process memory mapping design.

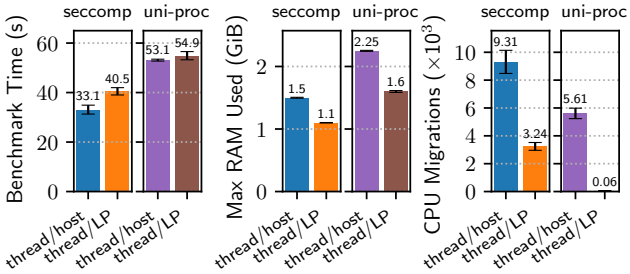


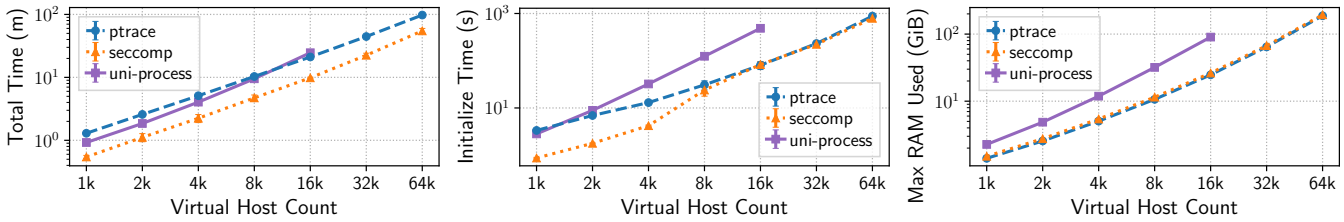
Figure 10: Performance of our P2P benchmark when Phantom (`seccomp`) and Shadow (uni-proc) are configured to use one worker thread per virtual host or one worker per logical processor (LP).

with `ptrace` and the uni-process design when using more than 14 LPs (the number of cores available on each of the two CPUs), while `seccomp` is able to make effective use of additional LPs. Finally, using a large number of LPs causes more than a 7× slowdown in `ptrace` (from 79s with 14 LPs to 581s with 56 LPs) which we speculate is due to inefficient kernel facilities. We observed similar trends in the initialization time—the time for Phantom to launch all processes or Shadow to load all namespaces: Phantom with `seccomp` is more than 3× as fast (0.84s) as both `ptrace` (3.5s) and the uni-process design (2.6s) at initializing processes when using 28 LPs (see Appendix E.2 for more details).

We conclude from our results that using a combination of the preloading and `seccomp` interception strategies leads to the best performance in Phantom: preloading with `seccomp` improves performance over `seccomp` alone because it reduces the overhead from mode transitions and executions of the `seccomp` filter. As in the microbenchmarks, preloading has little effect when used with `ptrace` as expected. Finally, using a number of LPs equal to the number of CPU cores (i.e., half of the available hyper-threads) produces reasonable performance for both Phantom and Shadow. Hence, we use 28 LPs and enable preloading in the remaining experiments.

Linux CPU Scheduling: Figure 8 shows the results of our investigation into the effects of Linux CPU scheduling on the performance of our macrobenchmark. As in our microbenchmarks, we find that CPU pinning has a positive effect on performance: Phantom with `seccomp` completes the benchmark 2.5× faster with CPU pinning than with standard scheduling. Pinning has a similar but slightly smaller relative effect on `ptrace`, and a positive but minor effect on the uni-process design. We observe in our measurements that pinning reduces the number of CPU migrations that occur during the simulation from 5.8M to 8.6k for Phantom with `seccomp`, from 3.6M to 8.1k for Phantom with `ptrace`, and from 180k to 5.7k for Shadow's uni-process design. Realtime scheduling again shows mixed results, but always performs better than standard scheduling when combined with pinning. We conclude that pinning provides a consistently positive effect on performance, and enable it in the remaining experiments.

Inter-Process Memory Manager: Figure 9 shows the performance of Phantom's inter-process memory mapping design across three metrics. First, we find that Phantom completes the benchmark in comparable time when: (i) using the primary `mmap`-based approach; and (ii) being restricted to the fallback approach of using `process_vm_read` and `process_vm_write`. Recall that our P2P macrobenchmark sends messages with 1k payloads, and in our microbenchmark we found that the performance of the memory mapping approach improves relative to the fallback mechanism as the payload size increases. Second, we observe that the memory mapping design uses slightly more RAM and causes slightly more page faults because it requires additional state to track the memory mappings. Phantom always finishes the benchmark sooner than Shadow's uni-process design (<70%) while using less RAM (<65%) and causing fewer page faults (<35%). (We find that the same general trends hold when running Phantom with `ptrace`; see Appendix E.2 for details.)



(a) Time to Complete Benchmark **(b)** Time to Launch Processes or Load Namespaces **(c)** Memory Used during Benchmark

Figure 11: Performance of our P2P benchmark when scaling the number of hosts from 1k to 64k hosts (i.e., processes in Phantom or namespaces in Shadow). Both axes are plotted in log scale on all subplots. Running 32k or more hosts in Shadow exceeded the machine's RAM (256 GiB).

Worker Thread Scheduling: Figure 10 shows the performance of our work-stealing worker thread scheduling design (described in §3.2.2) in which we run one worker thread per virtual host, compared to a work-stealing algorithm from Shadow that runs one worker thread per logical processor (LP). The benchmark completes more quickly when using one worker thread per host (i.e., 1000 workers in our benchmark) than when using one worker per LP (i.e., 28 workers in this experiment), for both Phantom and Shadow. We observe that by using additional threads, we increase the maximum RAM used and the number of CPU migrations that occur while running the benchmark. (We again find that the same general trends hold when running Phantom with `ptrace`; see Appendix E.2 for details.) We conclude that using more threads should be done in consideration of available RAM.

Peer Workload: We conducted an investigation into the effects of varying peer workloads by varying the number of messages m, the number of AES (encrypt, decrypt) sequences c, and the peer weight distribution W. As expected, increasing m and c resulted in a roughly linear increase in benchmark times in both Phantom and Shadow, while the workload distributions we tested had minor effect on performance. We present more details in Appendix E.2 due to space constraints.

Distributed Network Scale: We investigate the performance of our P2P benchmark while scaling the network size from 1k to 64k hosts. Our results in Figure 11 show that Phantom with `seccomp` outperforms `ptrace` and Shadow's uni-process design in terms of benchmark time, while initialization time (the time to launch all processes in Phantom or load all namespaces in Shadow) and memory usage both scale more efficiently in Phantom than in Shadow. (Running 32k or more hosts in Shadow exceeded the machine's RAM (256 GiB).)

Figure 11a shows that the benchmark time exhibits growth that is nearly linear in the number of hosts (`ptrace`: $r = 0.999$, `seccomp`: $r = 0.995$, uni-process: $r = 0.994$, where $r = 1$ indicates perfect correlation) with 91ms per host for `ptrace`, 51ms per host for `seccomp`, and 96ms per host for uni-process. Accordingly, uni-process completes the 16k benchmark in 25m compared to 21m for `ptrace` and 10m for `seccomp` despite running the 1k benchmark in 55s compared to 78s for `ptrace` and 33s for `seccomp`. Phantom with `seccomp` completed the benchmark fastest for all tested network sizes.

Figure 11b shows how the initialization time changes as the host count increases. Here, clear separation between Phantom's design (which launches multiple processes) and Shadow's uni-process design (which loads multiple namespaces) can be observed through visual inspection. Despite a slight shift in growth between 4k and 16k hosts for `seccomp`, we find that Phantom is more efficient and scalable than Shadow at initializing virtual hosts' processes. For example, at 16k hosts Shadow completed initialization in about 8m while Phantom completed it in less than 1.5m.

Figure 11c shows that Phantom uses significantly less RAM to complete the benchmark than Shadow. At 1k hosts Shadow uses 2.3 GiB but Phantom with `seccomp` only uses 1.5 GiB (∼65%), while at 16k hosts Shadow uses 90 GiB but Phantom with `seccomp` only uses 26 GiB (∼29%). Phantom's relatively lower memory usage allows us to scale the number of hosts in the P2P benchmark to 4× the size of the largest network in which a benchmark was successful in Shadow.

Conclusions: We draw two primary conclusions from our benchmarks. First, Phantom outperforms the state-of-the-art uni-process Shadow design by effectively mitigating the multi-process performance challenges identified in §2.3. Second, Phantom consistently outperforms a simulator built around our implementation of `ptrace` (which we argue in Appendix B outperforms gRaIL's use of `ptrace`).

5.3 Accuracy: Verification

In this section, we verify that Phantom can accurately simulate basic network characteristics as well as more complex Tor overlay networks [17]. Recall that, as described in §4, Phantom integrates the network stack from Shadow; we do not claim the design or implementation of this network stack as a contribution of this paper. Shadow's network has already been extensively validated in previous work [30, 32, 37, 40]. Therefore, our primary focus is to verify that Phantom does not *reduce* the accuracy of the simulated network relative to Shadow; we consider our verification successful if Phantom and Shadow produce similar simulated network results.

Basic Network Verification: We evaluate the extent to which Phantom can accurately simulate basic network characteristics that are typical of LAN and WAN networks. Our evaluation considers two nodes that communicate over a single link.

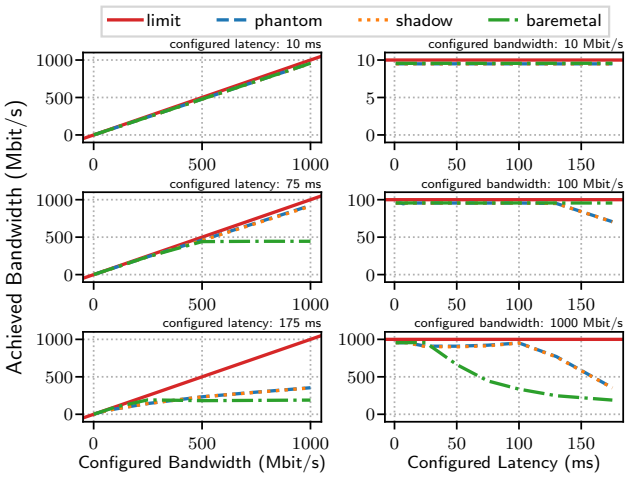


Figure 12: Our basic network verification experiments show that Phantom and Shadow produce identical `iperf` results across a range of configured bandwidths and latencies and that Phantom achieves comparable or higher link utilization than our baremetal setup.

We configure the link with a latency and bandwidth capacity, and then use `iperf` and a UDP ping application to measure the available network bandwidth and latency, respectively, between the nodes. We conducted the experiment across a range of latency and bandwidth settings in Phantom, Shadow, and using two baremetal machines in our lab connected by a 10 Gbit/s physical link (where we used `netem` to emulate the configured latency and bandwidth). Our results in Figure 12 show that: (i) Phantom and Shadow produce indistinguishable results across all tested bandwidths and latencies; and (ii) Phantom generally achieves comparable or higher link utilization than the `netem`-based baremetal setup. (See Appendix E.3.1 for more details, including a description of our latency verification which shows a maximum error of 3%.)

Tor Network Verification: We evaluate the extent to which Phantom can accurately simulate more complex Tor networks using the state-of-the-art Tor modeling tools and methods [40]. We configure a Tor network using a total of 12,232 Linux processes to generate a total of 74 Gbit/s of network traffic, which is equivalent to the expected combined traffic of about 238k users and represents a scale of about 30% of the public Tor network (more explanation is provided in Appendix E.3.2).

We run 10 Tor simulations for 60 simulated minutes each in both Phantom and Shadow and find that: (i) both tools require 27 real hours to run each simulation; and (ii) Shadow uses at most 1116 GiB of RAM while Phantom uses at most 1032 GiB (92.5% relative to Shadow). We measure no significant difference in the simulated network performance across 6 metrics including circuit build time, circuit round trip time, circuit goodput, and Tor network transfer times for 50 KiB, 1 MiB, and 5 MiB files: Figure 13 shows that the performance distributions from Phantom and Shadow are within CI bounds. We conclude that Phantom does not reduce the accuracy relative to Shadow in conducting network experiments.

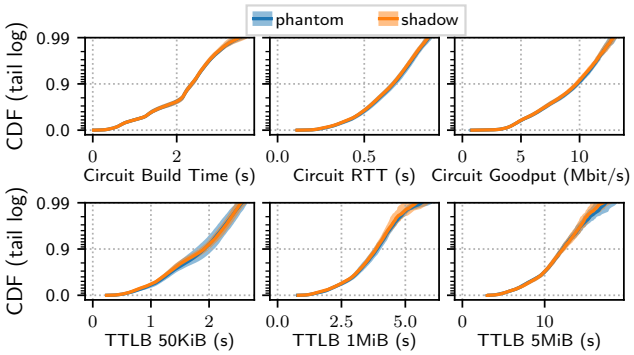


Figure 13: Our Tor network verification experiments show that Phantom and Shadow produce nearly identical Tor performance results (within CI bounds) since they share a network stack.

See Appendix E.3.2 for additional analyses and performance comparisons across a total of 6 network scale factors.

5.4 Comparison to Related Tools

In this section, we compare Phantom to popular tools implementing emulation, simulation, and hybrid architectures.

Mininet: Mininet is a network emulator that creates (i) a network of virtual hosts that run applications as Linux processes; (ii) virtual network interfaces within the Linux kernel; and (iii) virtual switches, controllers, and links that are managed by Mininet [45]. (See Appendix A.1 for more details.)

Mininet's network emulation architecture offers poor control and scalability because the host kernel is responsible for handling packet events, and the packet routing process is unpredictable and sensitive to load. If the host machine becomes overloaded, Mininet will experience time distortion that will degrade experiment realism and control.

We demonstrate Mininet's limitations by running a peer-to-peer benchmark (see §5.2.1) while scaling the number of peers in the experiment. Because each peer introduces a constant number of packets into the experiment, the expected number of packets and work to perform in the experiment grows linearly with the number of hosts. However, we find that load and network congestion on the host machine affects the outcome of the experiment. Figure 14 shows the average number of packets received per second by the virtual hosts (averaged over 10 trial runs). As the host machine becomes more loaded with virtual peers, *its packet forwarding capacity is limited*, and fewer packets than expected are forwarded. In contrast, Figure 14 shows that Phantom produces the expected packet throughput in simulated time (Phantom may run faster or slower than real time as necessary to achieve correctness).

NS-3 and gRaIL: NS-3 is a popular network simulator that simulates all aspects of networking *and* all application logic [26], while gRaIL extends NS-3 by enabling simulated nodes to directly execute applications as standard Linux processes managed by the kernel's `ptrace` facility [54]. (See Appendix A.2 for more details.)

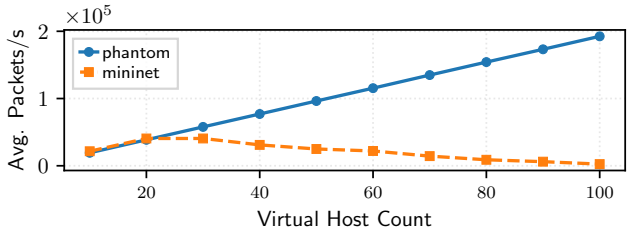


Figure 14: Average packet forwarding rate of a fixed P2P messaging workload in Phantom and Mininet as the number of hosts is varied.

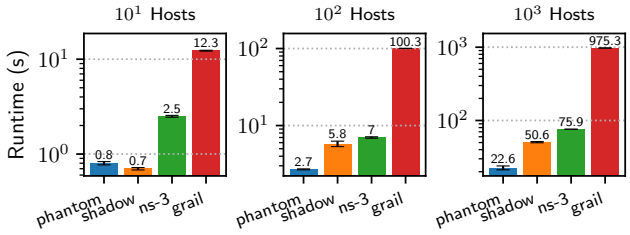


Figure 15: Runtime of a fixed P2P messaging workload in Phantom, Shadow, NS-3, and gRaIL as the number of hosts is varied.

We evaluate the performance cost of running our P2P benchmark from §5.2.1, replicating its logic so that it could also run as an NS-3 application. We configured the benchmark with m=100 messages, c=0 AES operations, and W=uniform distribution. We vary the number of hosts while configuring 50 ms pairwise latencies and 1 Gbit/s network bandwidths, and run experiments using multiple simulation tools.

Figure 15 shows the real time (mean of 10 trial runs) required to complete 10 seconds of network simulation parallelized across all cores on a blade server (see Appendix E.1) as the number of hosts is varied.[7] In our 1k host experiment, we find that (i) Phantom is 2.2× and 3.4× faster than Shadow and NS-3, respectively; and (ii) gRaIL's inefficient multi-process design is about 13× slower than NS-3 alone, and 43× slower than Phantom, demonstrating that Phantom effectively eliminates IPC overhead as a performance bottleneck and overcomes the multi-process challenges from §2.3.

Shadow: Shadow [30] implements a hybrid, uni-process architecture in which applications are directly executed in plugin namespaces and preloading is used (via `LD_PRELOAD`) to intercept `libc` function calls and hook the applications into the simulation. (See Appendix A.3 for more details.)

Although Figure 15 shows that it performs well, Shadow's plugin architecture is limited in its compatibility, correctness, and maintainability: as shown in Table 2, applications running in Shadow must be compiled as position-independent, must be dynamically linked to `libc`, and must not make system calls via statically linked or assembly code (or else they will not be interceptable). Because we cannot guarantee that

[7] Runtimes are normalized by the amount of work performed (i.e., packets delivered) by each simulator; this resulted in runtime adjustments of $< 5\%$ for all but gRaIL, which delivered only 56% of the expected packets and thus had its runtime adjusted by a factor of 1.8.

Table 2: Application Properties Supported in Hybrid Simulators

Application Property	Shadow	Phantom
Multiple threads (e.g., support for `pthreads`)	●	●
Multiple processes (e.g., support for `fork`)	◐	◐
Not position-independent (i.e., `PIC` or `PIE`)	○	●
Not dynamically linked to `libc`	○	●
Symbols not exported to dynamic symbol table	○	●
System calls made in statically linked code	○	●
System calls made in assembly (i.e., avoiding `libc`)	○	●
100% statically linked (e.g., some go programs)	○	◐

○ Does not work in tool or architecture ● Works in tool & architecture
◐ Not implemented in tool (as of writing) but supported by architecture

`libc` functions will not internally issue multiple unique system calls, Shadow's design requires reimplementing *both* the kernel system calls *and* the `libc` functions that invoke them.

Phantom overcomes Shadow's limitations by running applications as standard Linux processes, allowing us to take advantage of the kernel's high-performance process isolation features. Moreover, Phantom uses `seccomp` to guarantee that system calls can be intercepted no matter how the application initiates them (see Table 2), enabling us to reduce the emulation scope to the system call interface and support a much larger set of applications; Phantom's supported application set is primarily limited by the system calls and protocols implemented in its simulated kernel, which can be extended over time (our design could also be incorporated into other simulators, such as NS-3). Phantom enjoys these advantages while also meeting or exceeding Shadow's performance (as we have shown throughout §5).

6 Conclusion

We have designed, implemented, and thoroughly evaluated Phantom, a novel, high-performance network simulator for large-scale distributed systems. Phantom's multi-process design eliminates the compatibility, correctness, and maintainability limitations that we believe have inhibited the widespread adoption of existing plugin-based simulators. With our innovative synthesis of efficient process control, system call interposition, and data transfer mechanisms, Phantom also overcomes the inter-process performance challenges of the state-of-the-art multi-process simulator. Through our extensive evaluation, we have demonstrated that Phantom achieves better performance and is more scalable than alternative simulators across a variety of important benchmarks.

Acknowledgments: We thank our shepherd and the anonymous reviewers for their valuable feedback. We thank Steven Engler for discussions about design and support during development. This work has been partially supported by the Office of Naval Research (ONR), the Defense Advanced Research Projects Agency (DARPA), and the National Science Foundation (NSF) under award CNS-1925497.

References

[1] The Tor Metrics Portal. https://metrics.torproject.org, April 2021.

[2] Performance Experiments. https://gitlab.torproject.org/legacy/trac/-/wikis/org/roadmaps/CoreTor/PerformanceExperiments, September 2021.

[3] Artifact for "Co-opting Linux Processes for High-Performance Network Simulation". https://netsim-atc2022.github.io, May 2022.

[4] Shadow: real applications, simulated networks. https://shadow.github.io, May 2022.

[5] M. AlSabah and I. Goldberg. PCTCP: Per-circuit TCP-over-IPsec Transport for Anonymous Communication Overlay Networks. In *ACM Conference on Computer and Communications Security (CCS)*, 2013.

[6] M. AlSabah and I. Goldberg. Performance and Security Improvements for Tor: A Survey. *ACM Computing Surveys (CSUR)*, 49(2):32, 2016.

[7] M. AlSabah, K. Bauer, I. Goldberg, D. Grunwald, D. McCoy, S. Savage, and G. M. Voelker. DefenestraTor: Throwing Out Windows in Tor. In *Privacy Enhancing Technologies Symposium (PETS)*, 2011.

[8] M. AlSabah, K. Bauer, T. Elahi, and I. Goldberg. The Path Less Travelled: Overcoming Tor's Bottlenecks with Traffic Splitting. In *Privacy Enhancing Technologies Symposium (PETS)*, 2013.

[9] A. Barton and M. Wright. DeNASA: Destination-Naive AS-Awareness in Anonymous Communications. *Proceedings on Privacy Enhancing Technologies (PoPETs)*, 2016(4):356–372, 2016.

[10] R. D. Blumofe and C. E. Leiserson. Scheduling multithreaded computations by work stealing. *J. ACM*, 46(5): 720–748, Sept. 1999.

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

[12] R. Chertov, S. Fahmy, and N. B. Shroff. Fidelity of network simulation and emulation: A case study of tcp-targeted denial of service attacks. *ACM Transactions on Modeling and Computer Simulation*, 19(1), Jan. 2009.

[13] W.-F. Chiang, G. Gopalakrishnan, Z. Rakamaric, D. H. Ahn, and G. L. Lee. Determinism and reproducibility in large-scale hpc systems. In *Workshop on Determinism and Correctness in Parallel Programming*, 2013.

[14] B. Cohen. Incentives build robustness in bittorrent. In *Workshop on Economics of Peer-to-Peer systems*, 2003.

[15] B. Conrad and F. Shirazi. Analyzing the Effectiveness of DoS Attacks on Tor. In *Conference on Security of Information and Networks*, 2014.

[16] S. Dahal, J. Lee, J. Kang, and S. Shin. Analysis on End-to-End Node Selection Probability in Tor Network. In *International Conference on Information Networking (ICOIN)*, 2015.

[17] R. Dingledine, N. Mathewson, and P. Syverson. Tor: The Second-Generation Onion Router. In *USENIX Security Symposium (USENIX-Sec)*, 2004.

[18] T.-N. Dinh, F. Rochet, O. Pereira, and D. S. Wallach. Scaling Up Anonymous Communication with Efficient Nanopayment Channels. *Proceedings on Privacy Enhancing Technologies (PoPETs)*, 2020(3):175–203, 2020.

[19] S. Floyd and V. Paxson. Difficulties in simulating the internet. *IEEE/ACM Transactions on Networking*, 9(4), 2001.

[20] R. M. Fujimoto. Performance of time warp under synthetic workloads. In *SCS Multiconference on Distributed Simulation*, 1990.

[21] J. Geddes, R. Jansen, and N. Hopper. How Low Can You Go: Balancing Performance with Anonymity in Tor. In *Privacy Enhancing Technologies Symposium (PETS)*, 2013.

[22] J. Geddes, R. Jansen, and N. Hopper. IMUX: Managing Tor Connections from Two to Infinity, and Beyond. In *ACM Workshop on Privacy in the Electronic Society (WPES)*, 2014.

[23] J. Geddes, M. Schliep, and N. Hopper. ABRA CADABRA: Magically Increasing Network Utilization in Tor by Avoiding Bottlenecks. In *ACM Workshop on Privacy in the Electronic Society (WPES)*, 2016.

[24] D. Gopal and N. Heninger. Torchestra: Reducing Interactive Traffic Delays over Tor. In *ACM Workshop on Privacy in the Electronic Society (WPES)*, 2012.

[25] H. Hanley, Y. Sun, S. Wagh, and P. Mittal. DPSelect: A Differential Privacy Based Guard Relay Selection Algorithm for Tor. *Proceedings on Privacy Enhancing Technologies (PoPETs)*, 2019(2):166–186, 2019.

[26] T. R. Henderson, S. Roy, S. Floyd, and G. F. Riley. ns-3 project goals. In *Workshop on NS-2: the IP network simulator*, 2006. See also https://www.nsnam.org.

[27] N. Hopper. Challenges in protecting Tor hidden services from botnet abuse. In *Financial Cryptography and Data Security (FC)*, 2014.

[28] M. Imani, A. Barton, and M. Wright. Guard Sets in Tor using AS Relationships. *Proceedings on Privacy Enhancing Technologies (PoPETs)*, 2018(1):145–165, 2018.

[29] M. Imani, M. Amirabadi, and M. Wright. Modified Relay Selection and Circuit Selection for Faster Tor. *IET Communications*, 13(17):2723–2734, 2019.

[30] R. Jansen and N. Hopper. Shadow: Running Tor in a Box for Accurate and Efficient Experimentation. In *Network and Distributed System Security Symposium (NDSS)*, 2012. See also https://shadow.github.io.

[31] R. Jansen, N. Hopper, and Y. Kim. Recruiting New Tor Relays with BRAIDS. In *ACM Conference on Computer and Communications Security (CCS)*, 2010.

[32] R. Jansen, K. Bauer, N. Hopper, and R. Dingledine. Methodically Modeling the Tor Network. In *USENIX Workshop on Cyber Security Experimentation and Test (CSET)*, 2012.

[33] R. Jansen, P. F. Syverson, and N. Hopper. Throttling Tor Bandwidth Parasites. In *USENIX Security Symposium (USENIX-Sec)*, 2012.

[34] R. Jansen, A. Johnson, and P. Syverson. LIRA: Lightweight Incentivized Routing for Anonymity. In *Network and Distributed System Security Symposium (NDSS)*, 2013.

[35] R. Jansen, J. Geddes, C. Wacek, M. Sherr, and P. Syverson. Never Been KIST: Tor's Congestion Management Blossoms with Kernel-Informed Socket Transport. In *USENIX Security Symposium (USENIX-Sec)*, 2014.

[36] R. Jansen, F. Tschorsch, A. Johnson, and B. Scheuermann. The Sniper Attack: Anonymously Deanonymizing and Disabling the Tor Network. In *Network and Distributed System Security Symposium (NDSS)*, 2014.

[37] R. Jansen, M. Traudt, J. Geddes, C. Wacek, M. Sherr, and P. Syverson. KIST: Kernel-Informed Socket Transport for Tor. *ACM Transactions on Privacy and Security (TOPS)*, 22(1):3:1–3:37, December 2018.

[38] R. Jansen, M. Traudt, and N. Hopper. Privacy-Preserving Dynamic Learning of Tor Network Traffic. In *ACM Conference on Computer and Communications Security (CCS)*, 2018. See also https://tmodel-ccs2018.github.io.

[39] R. Jansen, T. Vaidya, and M. Sherr. Point Break: A Study of Bandwidth Denial-of-Service Attacks against Tor. In *USENIX Security Symposium (USENIX-Sec)*, 2019.

[40] R. Jansen, J. Tracey, and I. Goldberg. Once is never enough: Foundations for sound statistical inference in Tor network experimentation. In *USENIX Security Symposium (USENIX-Sec)*, 2021. See also https://neverenough-sec2021.github.io.

[41] A. Johnson, R. Jansen, N. Hopper, A. Segal, and P. Syverson. PeerFlow: Secure Load Balancing in Tor. *Proceedings on Privacy Enhancing Technologies (PoPETs)*, 2017(2):74–94, 2017.

[42] A. Johnson, R. Jansen, A. D. Jaggard, J. Feigenbaum, and P. Syverson. Avoiding The Man on the Wire: Improving Tor's Security with Trust-Aware Path Selection. In *Network and Distributed System Security Symposium (NDSS)*, 2017.

[43] K. Kiran, S. S. Chalke, M. Usman, P. D. Shenoy, and K. Venugopal. Anonymity and Performance Analysis of Stream Isolation in Tor Network. In *International Conference on Computing, Communication and Networking Technologies (ICCCNT)*, 2019.

[44] J. Lamps, V. Babu, D. M. Nicol, V. Adam, and R. Kumar. Temporal integration of emulation and network simulators on linux multiprocessors. *ACM Transactions on Modeling and Computer Simulation*, 28(1), Jan. 2018.

[45] B. Lantz, B. Heller, and N. McKeown. A network in a laptop: rapid prototyping for software-defined networks. In *Workshop on Hot Topics in Networks (HotNets)*, 2010. See also http://mininet.org.

[46] D. Lin, M. Sherr, and B. T. Loo. Scalable and Anonymous Group Communication with MTor. *Proceedings on Privacy Enhancing Technologies (PoPETs)*, 2016(2): 22–39, 2016.

[47] Z. Liu, Y. Liu, P. Winter, P. Mittal, and Y.-C. Hu. TorPolice: Towards Enforcing Service-Defined Access Policies for Anonymous Communication in the Tor Network. In *International Conference on Network Protocols*, 2017.

[48] A. Miller and R. Jansen. Shadow-Bitcoin: Scalable Simulation via Direct Execution of Multi-threaded Applications. In *USENIX Workshop on Cyber Security Experimentation and Test (CSET)*, 2015. See also https://github.com/shadow/shadow-plugin-bitcoin.

[49] A. Mitseva, M. Aleksandrova, T. Engel, and A. Panchenko. Security and Performance Implications of BGP Rerouting-Resistant Guard Selection Algorithms for Tor. In *IFIP International Conference on ICT Systems Security and Privacy Protection*, 2020.

[50] P. Mockapetris and K. J. Dunlap. Development of the domain name system. In *Communications Architectures and Protocols*, SIGCOMM '88, page 123–133, 1988.

[51] W. B. Moore, C. Wacek, and M. Sherr. Exploring the Potential Benefits of Expanded Rate Limiting in Tor: Slow and Steady Wins the Race with Tortoise. In *Annual Computer Security Applications Conference (ACSAC)*, 2011.

[52] S. Nakamoto. Bitcoin: A peer-to-peer electronic cash system. 2008.

[53] R. Naumann, S. Dietzel, and B. Scheuermann. Towards more realistic network simulations: Leveraging the system-call barrier. In *Ad Hoc Networks*, pages 180–191. Springer, 2017.

[54] R. Naumann, S. Dietzel, and B. Scheuermann. Push the barrier: Discrete event protocol emulation. *IEEE/ACM Transactions on Networking*, 27(2):635–648, 2019.

[55] O. S. Navarro Leija, K. Shiptoski, R. G. Scott, B. Wang, N. Renner, R. R. Newton, and J. Devietti. Reproducible containers. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2020.

[56] J. Newsome (sporksmith). do_wait: make PIDTYPE_PID case O(1) instead of O(n). https://github.com/torvalds/linux/commit/5449162ac001a926ad8884882b071601df5edb44, May 2021.

[57] M. Perry. Shadow Experiments for Congestion Control. https://gitlab.torproject.org/tpo/core/tor/-/issues/40404, December 2021.

[58] F. Rochet and O. Pereira. Waterfilling: Balancing the Tor network with maximum diversity. *Proceedings on Privacy Enhancing Technologies (PoPETs)*, 2017(2):4–22, 2017.

[59] F. Rochet and O. Pereira. Dropping on the Edge: Flexibility and Traffic Confirmation in Onion Routing Protocols. *Proceedings on Privacy Enhancing Technologies (PoPETs)*, 2018(2):27–46, 2018.

[60] F. Rochet, R. Wails, A. Johnson, P. Mittal, and O. Pereira. CLAPS: Client-Location-Aware Path Selection in Tor. In *ACM Conference on Computer and Communications Security (CCS)*, 2020.

[61] F. Shirazi, C. Diaz, and J. Wright. Towards Measuring Resilience in Anonymous Communication Networks. In *ACM Workshop on Privacy in the Electronic Society (WPES)*, 2015.

[62] H. Tazaki, F. Uarbani, E. Mancini, M. Lacage, D. Camara, T. Turletti, and W. Dabbous. Direct code execution: Revisiting library os architecture for reproducible network experiments. In *ACM conference on Emerging networking experiments and technologies*, 2013.

[63] J. Tracey, R. Jansen, and I. Goldberg. High Performance Tor Experimentation from the Magic of Dynamic ELFs. In *USENIX Workshop on Cyber Security Experimentation and Test (CSET)*, 2018.

[64] F. Tschorsch and B. Scheuermann. Mind the Gap: Towards a Backpressure-Based Transport Protocol for the Tor Network. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2016.

[65] M. T. Vandevoorde and E. S. Roberts. Workcrews: An abstraction for controlling parallelism. *International Journal of Parallel Programming*, 17(4):347–366, 1988.

[66] C. Wacek, H. Tan, K. Bauer, and M. Sherr. An Empirical Evaluation of Relay Selection in Tor. In *Network and Distributed System Security Symposium (NDSS)*, 2013.

[67] G. Yan et al. Simulation of large scale networks using ssf. In *Winter Simulation Conference*, 2003.

[68] L. Yang and F. Li. mTor: A Multipath Tor Routing Beyond Bandwidth Throttling. In *2015 IEEE Conference on Communications and Network Security (CNS)*, 2015.

[69] L. Yang and F. Li. Enhancing Traffic Analysis Resistance for Tor Hidden Services with Multipath Routing. In *International Conference on Security and Privacy in Communication Systems*, 2015.

Appendices

A Background Details for Related Tools

In this appendix we provide extended background on existing tools for network experimentation. We consider popular tools from the architecture categories listed in §2.2 and §2.3: Mininet [45] (emulation), NS-3 [26] (simulation), gRaIL [54] (hybrid, multi-process, `ptrace` controller), and Shadow (hybrid, uni-process, plugin namespaces) [30].

A.1 Mininet

Mininet is a popular network emulator that implements a common design approach for network experimentation tools. Mininet creates a network of virtual hosts that run applications as standard Linux processes, virtual switches that support OpenFlow for custom routing and software-defined networking, and virtual controllers and links. Mininet executes application binaries and routes network packets through virtual network interfaces created within the Linux kernel [45]. The packets are routed by virtual switching and routing appliances managed by Mininet. Network attributes, such as link bandwidth and latency, are emulated using Linux's traffic control (`tc`) facilities.

Mininet's design is very flexible: most any application can be run directly without the need of explicitly programmed network simulation routines, making it easy to spin up a new network and quickly start testing software. However, Mininet offers poor control and scalability because the host kernel is responsible for handling packet events, which is unpredictable and sensitive to load as we show in §5.4.

A.2 NS-3 and gRaIL

NS-3 is one of the most widely-used network simulation tools used by network researchers. NS-3 simulations are composed of virtual nodes running application and routing software that are implemented entirely within NS-3's application logic (written in C++). NS-3 experiments offer a high degree of control and reproducibility, because all aspects of the networking—from generating a packet within an application to physically transmitting the packet's bits over physical media—are simulated. However, a serious drawback of NS-3's design is that real applications cannot be run within the simulation. For example, to run `ping` between two NS-3 nodes, a ping application simulator must be hand-crafted within NS-3 application code (as opposed to directing the nodes to execute a `ping` binary). Although this requirement may be acceptable to simulate simple applications, generating realistic simulations of complex protocols (e.g., Tor [17]) is difficult due to engineering complexities (e.g., the Tor codebase contains tens of thousands of lines of code).

Two NS-3 modules have been developed that do allow for real application execution within NS-3 simulations: (i) direct-code execution (DCE) mode [62], and (ii) the discrete event protocol emulation vessel (gRaIL) [54]. In the more recent gRaIL approach, NS-3 nodes are configured to run real application binaries (e.g., `/usr/sbin/ping`) which are forked and executed as genuine Linux processes. The progress of these processes are managed by Linux's process tracing facility `ptrace`, which allows the NS-3 process to intercept system calls made by the applications and translate them into NS-3 simulation events. Configuring NS-3 with gRaIL improves simulation realism, but has a very high performance cost as we show in §5.4.

A.3 Shadow

Shadow is a hybrid, uni-process network experimentation tool that incorporates aspects of both simulation and emulation [30]. At its core, Shadow is a conservative-time discrete-event network simulator that simulates network protocols (e.g., TCP and UDP), threading (using GNU portable threads [48]), and other kernel operations. Shadow dynamically loads applications into their own namespaces (using `dlmopen` and a custom loader [63]) and directly executes application code in the simulator process. Shadow hooks the applications into the simulation using function interposition, but emulates a Linux environment so that application code functions as if it was running in Linux.

Shadow represents the state-of-the-art hybrid network simulator tool for directly executing applications in large-scale distributed system simulations. A primary reason is that Shadow is designed to be high-performance: it runs as a single process (with multiple threads) to avoid inter-process overhead and unnecessary data copies. This has led Shadow to become the standard tool for simulating the Tor anonymity network [40]. However, Shadow has not had widespread use outside of the niche Tor application; Shadow has been shown to simulate Bitcoin networks [48], but that work has since been abandoned due to compatibility, correctness, and maintainability issues as we describe in §5.4.

B Interposing System Calls with `ptrace`

As described in §4, we implemented a system call interposition strategy based on `ptrace` to better understand the performance limits of a simulator designed around `ptrace`. Our `ptrace` implementation provides an alternative to Phantom's `seccomp` secondary interposition strategy (see §3.2.4).

gRaIL [54] is designed solely around the use of `ptrace` to control processes, system calls, and data transfer. Unfortunately, during our `ptrace` implementation and evaluation, we learned that the way that gRaIL uses `ptrace` (which is a standard and intuitive way to use `ptrace`) results in several scalability and performance problems that significantly reduce the performance of a hybrid network simulator. Since we wanted to understand the performance limits of `ptrace`, we developed enhancements to make `ptrace` more efficient and work around its bottlenecks.

In this appendix, we describe what we learned about making a ptrace-based system more performant and scalable. We argue that our improvements make our application of ptrace in a simulator significantly more performant and scalable than gRaIL's. Moreover, our implementation inside of Phantom enables us to more fairly evaluate and compare *the best version of gRaIL that it could be* rather than its inefficient prototype.

B.1 Scaling `waitpid` to Many Tracees

In initial evaluations, we were surprised to find that when adding n hosts to a PHOLD [20] simulation, in which the total number of messages passed scales linearly with the number of hosts, the simulation time grew *quadratically* with n instead of linearly. This turned out to be because the waitpid syscall, which is used by a ptrace tracer to wait for the next ptrace stop from a given tracee, performed a linear scan of child tasks, making its performance O(n) in the number of processes in the simulation; i.e. adding a process with some fixed amount of work to the simulation not only added that amount of work, but also made the management of every other process in the simulation slower.

Since this behavior of waitpid is potentially surprising, and could hurt performance for other large-scale uses of ptrace, we implemented a kernel patch making it O(1) instead of O(n). That patch was accepted, and first included in Linux kernel version v5.13-rc1 [56].

Since we wanted good performance in today's Linux distributions without needing to install a custom kernel, we also implemented a workaround in Phantom's ptrace code. Initially we worked around this with a dedicated "fork proxy" thread to initially fork each managed process. This way the (at the time) one-per-CPU "worker thread" weren't parents of the managed processes. However, *waitpid* also performed an O(n) linear scan of *tracees*. This meant that when switching from running one managed thread to another, the worker thread needed to ptrace-detach from the blocked thread (sending it a SIGSTOP to prevent it from running), and ptrace-reattach to the next thread to run. This workaround added substantial overhead, but was an overall performance improvement for simulations involving more than around 1000 managed threads per worker thread.

We were later able to remove the fork-proxy workaround when we moved to the logical-processor-based scheduler described in §3.2.2. Since each worker thread only manages the processes and tasks of a single simulated host, waitpid does not become more expensive as hosts are added.

B.2 Reducing Per-syscall `ptrace` Stops

Many ptrace-based systems, including gRaIL [53, 54] use the PTRACE_SYSCALL command to execute the tracee until its next syscall. When the tracee makes a syscall, the tracee is put into a syscall-enter-stop. The tracer can then fetch the memory registers of traced program to examine the syscall arguments using a PTRACE_GETREGS command (and memory referenced by those parameters as per Appendix B.3). In the case where the tracer desires to *emulate* the syscall, as is usually the case in Phantom, the tracer can:

1. issue a PTRACE_SETREGS command to change the syscall-number being requested to an invalid one;
2. issue another PTRACE_SYSCALL command to allow the tracee to execute the syscall, which will result in the issuing of an ENOSYS signal that puts the tracee into a syscall-exit-stop;
3. overwrite the error result to emulate the original syscall using PTRACE_SETREGS etc.; and
4. allow the tracee to continue running again with another PTRACE_SYSCALL command.

Using this approach, there are a minimum of 4 context-switches per syscall (assuming the tracee and tracer are executing on the same CPU):

1. tracee to tracer at the syscall-enter-stop;
2. tracer to tracee to execute the (no-op) syscall;
3. tracee to tracer at the syscall-exit-stop; and
4. tracer to tracee to resume the tracee's execution.

The ptrace syscall has an alternative command for when syscalls are to be *emulated* instead of just monitored: the PTRACE_SYSEMU command. As with PTRACE_SYSCALL, the tracee enters ptrace-enter-stop when first encountering a syscall. If the tracer continues again using PTRACE_SYSEMU, there is no syscall-exit-stop, saving 2 context-switches.

The primary downside of using PTRACE_SYSEMU is that if we *really do want* the managed process to execute the original syscall (perhaps with modified arguments), we can no longer just resume the original syscall, because the kernel does not execute the syscall when using PTRACE_SYSEMU. In Phantom we instead first get out of the ptrace-enter-stop, with a PTRACE_SINGLESTEP command, overwrite the instruction pointer to "rewind" it to point to the syscall instruction again, and then PTRACE_SINGLESTEP again to actually execute it. This adds an extra ptrace-stop relative to the case where we would use PTRACE_SYSCALL, but this tradeoff is worthwhile when *most* syscalls are being emulated (as is the case in Phantom).

B.3 Efficiently Accessing Tracee Memory

The mechanism for accessing tracee memory via ptrace itself is PTRACE_PEEK and PTRACE_POKE. This is the mechanism used by many ptrace-based systems, including gRaIL [53, 54] and DetTrace [55]. Unfortunately, this mechanism requires a separate syscall and accompanying mode transition to access *each word* of memory, making it inefficient for large structs and buffers.

We could reduce the number of syscalls required for large memory accesses by instead reading and writing the /proc/[pid]/mem pseudo-file. After opening the file (which requires already being ptrace-attached), accessing a contiguous buffer can be done with a single pread or pwrite syscall

and multiple buffers can be accessed at the same time with `preadv` and `pwritev`.

However, we instead make use of the `process_vm_readv` and `process_vm_writev` syscalls, which are analagous to `preadv` and `pwritev` but are specialized for accessing the memory of another process. They are a bit simpler to use, since they take the `pid` of the target process instead of needing to open and maintain a file descriptor. They also do not require the caller to be `ptrace`-attached to the target process—a feature that Phantom utilizes in order to use the same code for accessing managed process memory no matter if we are using the `ptrace`- or `seccomp`-based syscall interception strategies.

As discussed in §3.2.6, in Phantom we make most memory accesses even more efficient by remapping some of the tracee's memory regions into a shared memory file, which is also mapped into Phantom. As we show in Figure 5, this further increases cross-process data transfer performance.

B.4 Enabling Work Stealing

In Shadow, there is roughly one worker thread per-CPU, and in each round of the simulation, each worker thread processes a queue of simulated hosts. When a worker thread's queue is empty, it *steals* hosts from another worker's queue.

Unfortunately, when using `ptrace`, only the thread that originally `ptrace`-attached a traced thread is permitted to issue `ptrace` commands; other threads in the process are not. This means that one worker thread cannot steal a simulated host from another worker thread and control its managed processes while the original worker thread is still attached.

We briefly pursued patching the kernel to lift this restriction, but it would involve a fair bit of complexity, and there is understandable hesitancy from kernel developers to add further complexity to the already quite complex `ptrace` subsystem in order to support a somewhat niche use-case.

We initially solved this problem by detaching `ptrace` from each managed thread in a host when done processing that host's events for the round, and re-attaching each thread the first time we need to control it each round. This process had substantial overhead, which was incurred even if the host and its threads were executed by the same worker the next round.

Phantom's logical-processor-based scheduler design described in §3.2.2 addresses this problem. In Phantom's scheduling architecture, worker threads are stolen by logical processors, but hosts never move between worker threads. Therefore a worker thread can stay attached to its managed threads for the entire simulation.

B.5 Avoiding `ptrace` Stops on Some Syscalls

As shown in Figure 4, intercepting a syscall via `LD_PRELOAD` and servicing it via IPC is significantly faster than intercepting and servicing it via `ptrace`. Unfortunately it is difficult to create a hybrid approach that uses `LD_PRELOAD` but falls back to `ptrace`, because `ptrace` stops for *every* syscall. Even if we avoid a `ptrace`-stop by intercepting a syscall via `LD_PRELOAD`, any syscall we make to communicate with Phantom will still generate a `ptrace`-stop, negating the performance benefit.

We prototyped a solution that worked around this problem by never making a syscall to perform IPC; after sending a message in the shared memory segment, we would do a busy-wait to receive the response instead of making a `futex` syscall. This approach actually worked well for simulations that otherwise had idle cores to spare—the Phantom worker thread could service the syscall on one CPU while the managed thread spun in its loop on another. However, since this effectively halved the maximum concurrency, we ultimately discarded this approach.

A better solution to this problem is to use a `seccomp` filter to have some syscalls generate a `SECCOMP_RET_TRACE` event, and then use `ptrace` to catch those instead of stopping on every syscall [55].

In Phantom we similarly leveraged `seccomp`, but configured `seccomp` to trap to a signal handler (`SECCOMP_TRAP`) in the managed thread. The signal handler uses IPC if needed to communicate with Phantom, doing away with `ptrace` altogether. We have not performed a direct performance comparison between this `SECCOMP_TRAP` approach and the `SECCOMP_RET_TRACE` approach, but expect it to have similar or better performance: while `SECCOMP_RET_TRACE` skips the transfer from control in the kernel to the managed thread before context-switching to the tracing thread, using memory-based IPC from the `SECCOMP_TRAP` handler lets us transfer the register values more cheaply than `PTRACE_GETREGS` and `PTRACE_SETREGS`. More importantly, we can service some syscalls from the `SECCOMP_TRAP` handler without having to context-switch to Phantom at all, as described in §3.2.5.

C Context Switching Performance

In Phantom's process-oriented architecture, control and messages must be exchanged between Phantom's worker processes and managed application processes via interprocess communication (IPC). Linux and the POSIX standard offer many facilities for exchanging data across process boundaries and synchronizing processes. In this appendix, we examine the cost associated with a process context switch when facilitated by a given synchronization method. The control flow being measured is as follows, for a communicating parent process and child process pair:

1. The child waits for control from the parent;
2. The parent signals to the child that it should run, and waits for the child;
3. The child gains control, immediately signals back to the parent that it should run, and goes into the wait state; and
4. The parent wakes up and regains control.

In other words, this benchmark measures the latency required to perform a context-switch round-trip from a parent process to a child process and then back to the parent.

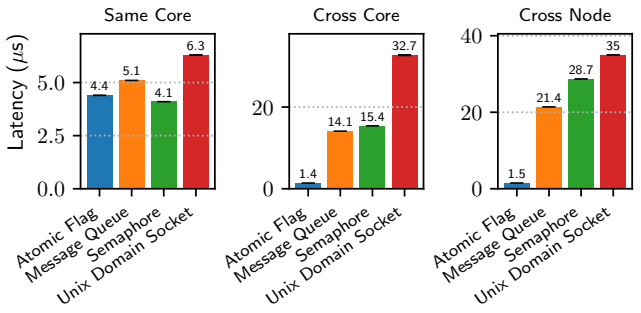


Figure 16: Time required to perform a process context-switch round trip under various synchronization mechanisms and process CPU affinity values.

Four synchronization mechanisms are included in the benchmark:

1. An `_Atomic _Bool` flag placed in shared memory. Waiting is performed by repeatedly checking if the value is `true` (i.e., spinning), and signaling is performed by setting the flags value to `true`.
2. A POSIX semaphore place in shared memory. Waiting is performed with `sem_wait` and signaling is performed with `sem_post`.
3. A Unix domain socket with ends shared by the parent and child process. Waiting is performed with `recv` called on the socket's file descriptor and signaling is performed with `send` called on the descriptor.
4. A POSIX message queue shared by the processes. Waiting is performed by calling `mq_recv` and signaling is performed with `mq_send`.

In addition to the mechanism, in this benchmark we also vary the CPU affinity of the parent and child processes. The cost of context switching varies depending upon which processors are executing the processes. The benchmark tests three cases: (i) when the parent and child are pinned to the same CPU core; (ii) when the parent and child are pinned to separate cores on the same CPU; and (iii) when the parent and child are pinned to separate cores on separate NUMA nodes and CPUs.

We ran these benchmarks on a machine with two 14-core Intel Xeon E5-2697 CPUs clocked at 2.60 GHz (the same machine used in our §5 experiments). The machine was running CentOS 7 and Linux kernel version 5.11.6-1. Figure 16 shows the average context-switch round-trip time taken over 100k repeated trials. (99% confidence interval ranges are drawn at the top of each bar, but in every case the interval size is almost zero.) These results show that POSIX message queues and semaphores slightly outperform Unix domain sockets, and that semaphores are the most efficient synchronization method when the processes are pinned to the same core (which is the case in Phantom). Cross-core and cross-node context switching is significantly more expensive than same-core, with the exception of the atomic boolean flag; when using this synchronization mechanism, having two CPUs that can spin in parallel minimizes latency. However, spinning can waste CPU cycles and is not economical when the CPU is under load. Hence, we find that POSIX semaphores are the most performant mechanism to synchronize control between Phantom and its managed, child processes.

D Simulated System and Network Facilities

In this appendix, we describe at a high level the system and network facilities that Phantom simulates. These simulated components are mostly borrowed from Shadow (see §4), but significant attention was required to integrate these with Phantom's new system call interposition, memory manager, and process scheduling interfaces.

Time: As a simulator, Phantom has complete control over simulated time. Attempts by managed processes to obtain the current time are handled by returning the simulated time (relative to a recent epoch) instead. (Because retrieving time is a hot-path function, time-related system calls are handled in the shim as described in §3.2.5.)

Input/Output: Phantom simulates file descriptors and tracks them using a lookup table for each managed process. This ensures a consistent mapping of file descriptor numbers to the internal objects needed to operate on them. Files are simulated internally by using real OS files and translating between the simulated and real file descriptor numbers. Other descriptors can be completely simulated internally, including sockets, pipes, timers, and events. Event notification facilities (e.g., `select`, `poll`, and `epoll`) can also be simulated by tracking the state of each simulated descriptor and triggering callback events when those states change in a way that requires action. Both blocking and non-blocking operations are supported as described in §3.2.7.

Transport: Phantom is a packet-level simulator that implements simulated versions of protocols such as TCP and UDP. Although packet-level semantics are simulated with respect to the associated socket protocols, packet payloads (application data) sent by the managed processes are copied only once into internal buffers. Transferring this data between virtual hosts across the simulated network amounts to transferring the memory address of the original data location, minimizing overhead when transferring simulated packets.

Network: Phantom simulates DNS using a simple name to virtual IP address mapping. Routing is simplified to running shortest path over a configurable network graph to compute end-to-end latency and packet loss. In addition to these network characteristics, packets sent over the simulated network will also be subject to: (i) virtual host bandwidth limits which are simulated using token buckets; and (ii) network queuing semantics which are simulated using an implementation of the CoDel (controlled delay) network scheduling algorithm.

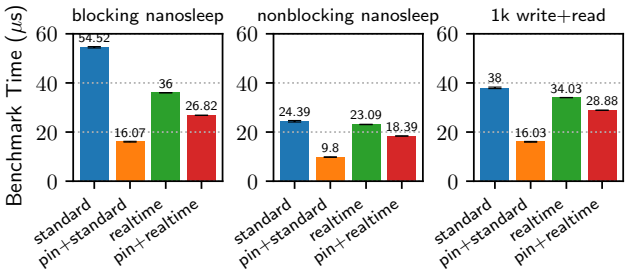


Figure 17: Time to execute our microbenchmarks from §5.1 when Phantom is configured to run with the `ptrace` interception strategy and the Linux CPU pinning and realtime scheduling features.

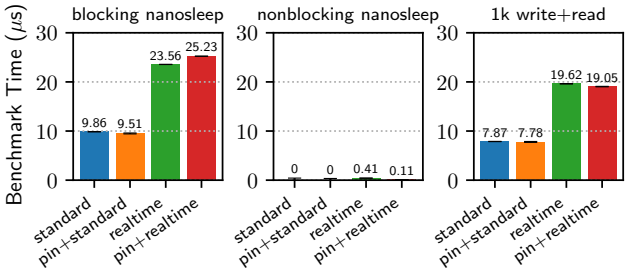


Figure 18: Time to execute our microbenchmarks from §5.1 in the uni-process, preload-based Shadow architecture when using the Linux CPU pinning and realtime scheduling features.

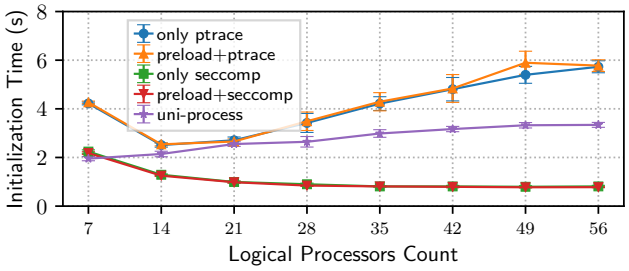


Figure 19: Initialization time is the time for Phantom to launch all managed processes (or for uni-process Shadow to load all namespaces) and run them until the first blocking system call.

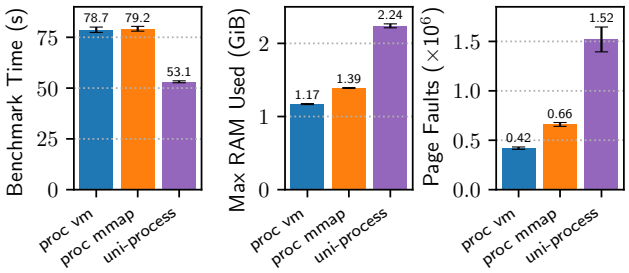


Figure 20: Performance of our P2P benchmark with Phantom using `ptrace` interception with `process_vm_read` and `process_vm_write` fallback facilities compared to its primary inter-process memory mapping design.

E Extended Evaluation

In this appendix, we include some extended evaluation details and results that we were unable to include in the main body of the paper (in §5) due to space constraints.

E.1 Extended Microbenchmarks

We conducted the evaluation here and in §5.1 using a blade server cluster in which each blade contained identical hardware: 256 GiB of RAM and 2×14 core Intel Xeon E5-2697v3 CPUs (56 total hyper-threads) running at 2.6 GHz. Each blade machine was running CentOS 7 and Linux kernel version 5.11.6-1. We configured our experiments to run in docker containers to ensure that we were running identical software stacks across the blade machines.

Figure 17 shows the effect of Linux CPU scheduling features when running Phantom with a `ptrace` interception strategy. We find that CPU pinning alone performs the best and that adding realtime scheduling along with CPU pinning has an adverse effect. Running with realtime scheduling alone only slightly improves performance over using the standard Linux scheduling mechanisms.

Figure 18 shows the effect of Linux CPU scheduling features on a uni-process design. The choice of Linux CPU scheduling feature has little effect on the nonblocking `nanosleep` benchmark, since it is effectively a function call and already incredibly efficient. Interestingly, realtime scheduling reduces performance for both the blocking and io-based benchmarks, and adding pinning and realtime scheduling together does not mitigate these effects.

E.2 Extended Macrobenchmarks

We conducted the evaluation here and in §5.2 using the same blade server cluster that we used for the microbenchmarks (see Appendix E.1).

Interception Strategies: Figure 19 shows the initialization time for Phantom with various interception strategies compared to the uni-process Shadow design. As expected, preloading has an insignificant effect on launch/load times. When using `seccomp`, Phantom completes the initialization process the faster than the uni-process design. Initialization takes the longest when using `ptrace`, and scales the worst as the number of LPs increases.

Memory Manager: Figure 20 shows that, when configured to use the `ptrace` interception strategy, the inter-process memory mapping design performs comparably to `process_vm_read` and `process_vm_write` fallback facilities despite using slightly more RAM and causing slightly more page faults. Phantom's memory manager uses less RAM than Shadow's uni-process design in all cases.

Thread Scheduler: Figure 21 shows the performance of the work-stealing thread schedulers when running Phantom with the `ptrace` interception strategy compared to the performance when using the schedulers in the uni-process Shadow design. Consistent with our results from §5.2, we find that

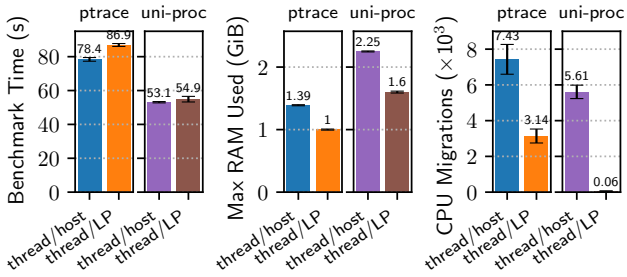


Figure 21: Performance of our P2P benchmark when our worker thread scheduler uses one thread per virtual host or one thread per logical processor (LP).

using one worker thread per virtual host decreases the time to complete the P2P benchmark relative to using one worker thread per LP despite using more RAM and causing more CPU migrations.

Peer Workload: We investigated the performance effect of varying the workloads in our P2P benchmark. In §5.2.1 we described the configurable parameters in our benchmark: each peer sends m messages, when receiving a message a peer performs c AES (encrypt, decrypt) sequences before sending another message, and message destinations are selected according to a peer weighting function W.

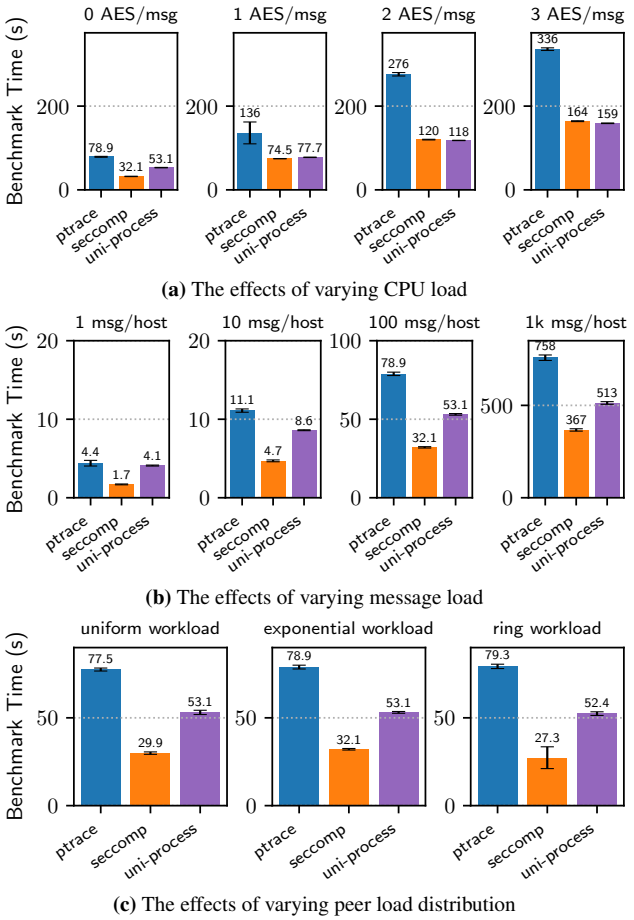
Figure 22a shows the results when varying c in the range $[0,3]$. In Phantom with `seccomp` interception and in Shadow's uni-process design, we observe a roughly linear increase in the benchmark time when increasing the number of AES operations that must be performed upon receipt of every message as expected. Interestingly, the linear constant appears to be slightly larger for `seccomp` than for uni-process, and much larger for `ptrace`. We speculate that the observed performance may be due to caching differences.

Figure 22b shows the results when varying m in the set $\{1,10,100,1k\}$. After subtracting the baseline initialization time (represented roughly by the 1 msg/host case), we observe a linear increase in the benchmark time as we increase the message load; this is the expected result since increasing the message load also increases the amount of work the simulator must perform.

Figure 22c shows the results when varying the workload distribution. In addition to the exponential function defined in §5.2.1 as e^{-3x} for $x \in [0,1]$, we also consider a "uniform" distribution function where each peer has an equal probability of being selected as the destination for any message, and a "ring" distribution function where each peer simply selects its closest neighbor as the destination of its outgoing messages. We can see from Figure 22c that these alternative workload distribution functions have minor effect on performance in our simulations.

E.3 Extended Network Verification

In this appendix, we present extended details and results from our evaluation of Phantom's ability to accurately represent network characteristics from §5.3.



(a) The effects of varying CPU load

(b) The effects of varying message load

(c) The effects of varying peer load distribution

Figure 22: Performance of our P2P benchmark in Phantom using the `seccomp` and `ptrace` interception strategies and in Shadow's uni-process design while varying the benchmark CPU and network load parameters.

E.3.1 Extended Basic Network Verification

We verify that Phantom accurately models networked application behavior when those applications are run in different networks with a variety of bandwidth and latency properties. We use two test applications, iPerf and a UDP echo application, to measure Phantom's simulated bandwidth and latency characteristics. We compare the measurements collected from Phantom to a set of baremetal measurements we collected by running the same applications on two physically-networked servers with emulated bandwidth and latency properties. Additionally, we collected iPerf bandwidth measurements using Shadow so that we could verify that Phantom does not simulate network activity differently than Shadow does. Below we describe the experimental setup and the the results from our two experiments.

Setup: To collect simulated bandwidth measurements, we ran a series of iPerf(v2) simulations with Phantom and Shadow. In each simulation, a single client and server communicate

using iPerf's single-threaded TCP benchmark. The client sends traffic for 10 s, and the server runs until all the data from the client is received. Bandwidth is recorded from the server, which is receiving the traffic, in three-second intervals. Each simulation uses a network configured with a different bandwidth and one-way latency. For these experiments, we consider bandwidth values between 1 Mbit/s and 1 Gbit/s, and one-way latency values between 1 ms and 175 ms. We determined that 175 ms was a realistic upper-bound on latency in wide-area networks by examining RIPE Atlas ping measurements from Jan. 11, 2021: we found that the maximum latency reported by the probes for all built-in measurements was 173.5 ms after removing outliers.

To collect simulated latency measurements, a custom UDP echo application is ran between a client and a server with Phantom. The echo application simply measures the time required for the client to echo a UDP packet sent by the server, which estimates the round-trip time between the server and the client. We consider the same range of latency values in these experiments as in the bandwidth experiments.

To collect emulated baremetal measurements for comparison, we ran the same applications on two physically-networked servers. The machines had 2×Intel Xeon E5-2697 v3 CPUs, 256 GiB of RAM, and ran Debian 11 with Linux Kernel v5.10.0-8. They were both connected with NetXtreme II BCM57810 10 Gigabit Ethernet NICs through a 10 Gbit/s switch. We used Linux's `netem` facilities to configure each machine's NIC with a specified bandwidth rate limit and packet latency. Additionally, we set the machine's TCP stack to use the Reno congestion control algorithm, which is also implemented in Shadow and Phantom.

iPerf Bandwidth Measurements: Figure 12 in §5.3 compares iPerf-reported bandwidth from the baremetal measurements, Phantom, and Shadow. In the left three plots, latency is held constant (at either 10 ms, 75 ms, or 175 ms) and iPerf-reported bandwidth is plotted versus the experimentally-configured bandwidth limit. In the right three plots, bandwidth is held constant (at either 10 Mbit/s, 100 Mbit/s, or 1 Gbit/s) and iPerf-reported bandwidth is plotted versus the experimentally-configured packet one-way latency. These plots show the bandwidth reported by iPerf during the 3-second interval closest to the half-way point of the transfer, which estimates the sustained maximum bandwidth achieved between the client and the server. We find that Phantom does not change modeling accuracy relative to Shadow, and that Phantom-reported performance matches baremetal-reported performance in most network conditions. With more extreme bandwidth and latency values, Phantom is able to achieve higher link-utilization than baremetal. These differences may be accounted for by different parameterizations of Phantom's TCP stack and Linux's (e.g., different initial window sizes). Extending and tuning Phantom's TCP stack to more closely approximate the behavior of Linux's networking facilities is a promising direction for future work.

Table 3: The Number of Virtual Hosts, Processes, and the Amount of Traffic in each Simulated Tor Network of the Given Scale

Network Scale	5%	10%	15%	20%	25%	30%
Clients	436	871	1307	1742	2178	2614
Relays	349	694	1039	1385	1732	2076
Servers	40	79	119	158	198	238
Total Virtual Hosts	825	1644	2465	3285	4108	4928
Tor	785	1565	2346	3127	3910	4690
OnionTrace	785	1565	2346	3127	3910	4690
TGen	476	950	1426	1900	2376	2852
Total Processes	2046	4080	6118	8154	10196	12232
Simulated Gbit/s⋆	12	24	37	49	62	74
Equivalent Tor Users	39.6k	79.2k	119k	158k	198k	238k

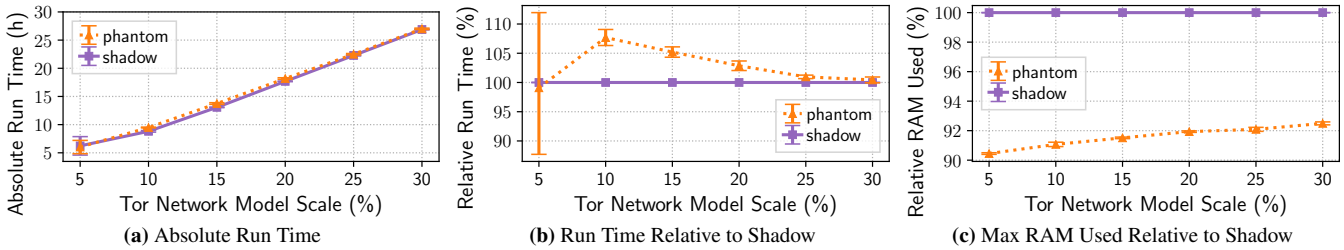
⋆ Mean across 20 total simulations for each network scale.

Latency Measurements: Both the Phantom and baremetal measured RTT latency values matched nearly identically with the value specified in the experiment for all configured latencies. The largest percent-error between the simulated and emulated results (taking the emulated measurements to be ground-truth) was 3%, which occurred at the lowest configured one-way latency (1 ms): the Phantom-measured RTT was 2 ms, whereas the baremetal measured RTT was 2.07 ms. The difference can be accounted for by software processing time, which Phantom does not simulate.

E.3.2 Extended Tor Network Verification

We consider large-scale Tor network simulation as a practical use case for Phantom. By supporting Tor, we believe Phantom will have broader impact particularly among researchers in the privacy-enhancing technologies community since they commonly use simulation [5, 7–9, 15, 16, 18, 21–25, 27–29, 31, 33–37, 39, 41–43, 46, 47, 49, 51, 58–61, 64, 66, 68, 69] to explore Tor performance and security research problems [6]. Moreover, the Tor Project has recently adopted the use of simulation in their own network planning and performance analyses [2], and Phantom has already been used to guide these efforts [57].

The Tor anonymity network [17] contains over 6k relay nodes [1] and about 800k users that are simultaneously active, i.e., running a Tor client and generating network traffic [38]. Constructing a simulated Tor network that is representative of the real Tor network involves a significant modeling and configuration effort [40]. Important factors that must be considered include the number of virtual hosts running Tor clients and relays, traffic generator clients and servers, the network latency between these hosts, the bandwidth available to these hosts, and the configured behavior of the clients, relays, and traffic generators. Fortunately, recent foundational work on Tor network experimentation has contributed methods and tools to guide our experimentation process [40]. We use these tools directly to create Tor network configs and run experiments in both Phantom and Shadow (Tor network modeling is outside the scope of this paper).



(a) Absolute Run Time **(b)** Run Time Relative to Shadow **(c)** Max RAM Used Relative to Shadow

Figure 23: The time and memory required to complete each Tor network simulation in Phantom (using `seccomp` interception) and in Shadow's uni-process design as the network model scale increases. (b) and (c) show performance relative to Shadow's baseline.

Setup: We generate 10 unique Tor network configs for each of 6 network scale factors using Tor network state from 2021-01; Table 3 shows the total number of virtual hosts and Linux processes used at each scale. Each client host runs three processes: (i) a TGen traffic generator process that generates traffic according to Markov models created by measuring the real Tor network [38]; (ii) a Tor process in client mode that forwards the TGen traffic into our private Tor network; and (iii) an OnionTrace process that connects to Tor to gather statistics and log information. Each relay host runs a Tor process in relay mode that forwards traffic in our private Tor network, and an OnionTrace process that gathers statistics and logs. Each server host simply runs a TGen server process that coordinates with TGen clients to generate traffic. Table 3 also shows the total volume of traffic being simulated at each scale, and the equivalent expected number of Tor users that it would take to generate that traffic in the public Tor network.

We run each of the resulting 60 Tor networks using Tor v0.4.5.9 in Phantom (using `seccomp`) and in Shadow (the state-of-the-art Tor network simulator). We conducted the evaluation using a blade server cluster in which each blade contained identical hardware: 1.25 TiB of RAM and 4×8 core Intel Xeon E5-4627v2 CPUs (without hyper-threading support) running at 3.30 GHz. For all experiments, we enable CPU pinning, disable realtime scheduling, and use 32 LPs (one LP per core) following our results from §5. We present the results as the mean across the ten networks at each scale with 95% confidence intervals (CIs).

Verification Results: We analyze the performance characteristics in each simulation, e.g., the simulated time to transfer data through the simulated Tor network. Recall from §5.3 that our primary goal is to validate that Phantom does not *reduce* the accuracy of the simulated network stack that it integrates and that we consider our validation successful if Phantom and Shadow produce similar simulated network results.

Figure 13 in §5.3 shows the simulated Tor performance results from using both Phantom and Shadow to each simulate the ten 30% scale Tor networks. Shown are several Tor performance metrics, including: circuit build times; circuit round trip times (time from data request to first byte of response); circuit goodput (transfer rate for range [0.5 MiB, 1 MiB] over 1 MiB and 5 MiB transfers); and client download times for transfers of 50 KiB, 1 MiB, and 5 MiB. The shaded areas represent 95% confidence intervals that were computed following recently published methods [40]. We do not find a significant difference in performance measured in the simulated Tor network across several metrics when comparing Phantom to Shadow. Again, this is the desired and expected result since both Shadow and Phantom share a network stack implementation. We conclude that Phantom maintains the same level of simulator accuracy that Shadow provided; i.e., Phantom's multi-process design does not degrade the level of accuracy that can be provided by a simulated network.

Performance Results: Figure 23 shows the simulators' performance when running the Tor network simulations.

Figure 23a shows that the absolute time to complete a 60 simulated minute experiment is roughly linear in the network scale, where Phantom completed the experiment in the 10%, 20%, and 30% networks in 9, 18, and 27 hours, respectively. Phantom's run time is remarkably similar Shadow's, even though Phantom incurs IPC overhead while Shadow does not; this demonstrates the efficiency of Phantom's design even when simulating somewhat complex distributed systems.

We plot simulation run time relative to Shadow's baseline in Figure 23b. Although there is large uncertainty in small network scales (consistent with prior work [40]), we observe that Phantom is competitive to Shadow's uni-process performance at smaller scales and comparable at larger scales: at 30% scale, Shadow's performance falls within Phantom's 95% CI.

Figure 23c shows the max RAM used by Phantom to simulate each network scale relative to Shadow's baseline. We observe that Phantom is more memory-efficient than Shadow: while Shadow used 178, 357, 540, 727, 919, and 1116 GiB at network scales of 5%, 10%, 15%, 20%, 25%, and 30%, respectively, Phantom used at least 90.4% (at 5% scale) and at most 92.5% (at 30% scale) of RAM relative to Shadow.

We conclude that Phantom effectively overcomes the multi-process performance challenges identified in §2.3: Phantom's multi-process design offers comparable performance to Shadow's uni-process design while being more memory efficient. In addition to its performance, Phantom offers significant improvements over Shadow because its multi-process design precludes Shadow's compatibility, correctness, and maintainability limitations that we identified in §2.3.