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

USENIX Annual Technical Conference
Carlsbad, CA, USA
July 11th, 2022
Main Takeaways

Designed a new, hybrid network simulator/emulator

- co-opts Linux processes into a discrete-event network simulation that emulates kernel functionality
- enables large-scale, distributed system experiments

- Merged into the open-source Shadow project and synonymous with Shadow v2
- Artifacts: https://netsim-atc2022.github.io

- **2.3x** faster than Shadow v1
- **3.4x** faster than NS-3
- **43x** faster than gRaIL [ToN’19]
Outline

motivation
design
evaluation
Important properties of test networks:
- **Controllable**: isolate important factors
- **Replicable**: identically replicate experiments (determinism)

Requirements for large distributed systems:
- **Accurate**: directly execute system software (not an abstraction)
- **Scalable**: decouple from time, computational constraints of host
Problems with Traditional Approaches

- **Simulation (e.g., ns-3)**
  - Not realistic: runs abstractions instead of real applications
  - Hard to maintain and can lead to invalid results
- **Emulation (e.g., mininet)**
  - Not controllable: results will not be identical
  - Not scalable: CPU overload → time distortion

```
Mininet > sudo mn
```
Problems with Traditional Approaches

- Simulation (e.g., ns-3)
  - Not realistic: runs abstractions instead of real applications
  - Hard to maintain and can lead to invalid results
- Emulation (e.g., mininet)
  - Not controllable: results will not be identical
  - Not scalable: CPU overload → time distortion

![Graph showing packet count vs. virtual host count](graph.png)

- Forwarding capacity limited: fewer packets than expected are forwarded
- CPU overload with >20 hosts

Mininet > sudo mn
Hybrid Architectures and Challenges

- **Hybrid architecture**
  - Network simulation, but directly execute application code
  - Enjoys advantages of both simulation and emulation
  - Best opportunity to scale to large-scale distributed systems

<table>
<thead>
<tr>
<th>Architecture</th>
<th>Example Tool</th>
<th>Scalability</th>
<th>Realism</th>
<th>Control</th>
</tr>
</thead>
<tbody>
<tr>
<td>Emulation</td>
<td>Mininet</td>
<td>✗</td>
<td>✔</td>
<td>✗</td>
</tr>
<tr>
<td>Simulation</td>
<td>NS-3</td>
<td>✔</td>
<td>✗</td>
<td>✔</td>
</tr>
<tr>
<td>Hybrid</td>
<td>This work</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
</tbody>
</table>
Executing application code via plugin (link-map) namespaces

- appid = dlmopen(app.so)
- func = dlsym(appid, “main”)
- func()

NS-3-DCE, Shadow

Limitations

- Compatibility (must build PIC/PIE)
- Correctness (intercept libcalls only)
- Maintainability (custom ld, threading)
Executing application code via plugin (link-map) namespaces

- `appid = dlmopen(app.so)`
- `func = dlsym(appid, "main")`
- `func()`

NS-3-DCE, Shadow

Limitations

- Compatibility (must build PIC/PIE)
- Correctness (intercept libcalls only)
- Maintainability (custom ld, threading)

### Table 2: Application Properties Supported in Hybrid Simulators

<table>
<thead>
<tr>
<th>Application Property</th>
<th>Shadow</th>
<th>Phantom</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multiple threads (e.g., support for pthreads)</td>
<td>✔️</td>
<td>✔️</td>
</tr>
<tr>
<td>Multiple processes (e.g., support for fork)</td>
<td>🟢</td>
<td>🟢</td>
</tr>
<tr>
<td>Not position-independent (i.e., PIC or PIE)</td>
<td>🟢</td>
<td>🟢</td>
</tr>
<tr>
<td>Not dynamically linked to libc</td>
<td>🟢</td>
<td>🟢</td>
</tr>
<tr>
<td>Symbols not exported to dynamic symbol table</td>
<td>🟢</td>
<td>🟢</td>
</tr>
<tr>
<td>System calls made in statically linked code</td>
<td>🟢</td>
<td>🟢</td>
</tr>
<tr>
<td>System calls made in assembly (i.e., avoiding libc)</td>
<td>🟢</td>
<td>🟢</td>
</tr>
<tr>
<td>100% statically linked (e.g., some go programs)</td>
<td>🟢</td>
<td>🟢</td>
</tr>
</tbody>
</table>

- 🟢 Does not work in tool or architecture
- 🟢 Works in tool & architecture
- 🟢 Not implemented in tool (as of writing) but supported by architecture
Executing code via Linux processes

- `fork()` + `exec()` $\rightarrow$ `ptrace()`

**gRaIL [ToN’19]**

Limitations: `ptrace` is slow!

- Process control: overhead **quadratic** in total number of processes
- Syscall interception: 4 context switches **for every syscall**
- Data transfer: extra syscall + mode change **for every word of memory**

![Runtime (s) vs. Hosts](chart)

**gRaIL (on ns-3)** is **13x slower** than ns-3 alone
Can we design a tool with the **performance benefits** of a uni-process plugin-based architecture **AND** the improved **modularity** and **isolation** of a multi-process architecture?
Outline

motivation
design
evaluation
Design Overview

- Discrete-event packet-level network simulator
- Directly executes apps as standard Linux processes
- Intercepts all system calls made by apps and emulates them

Simulates system call behavior and networking
- File descriptors (files, sockets, pipes)
- Event notification (poll, epoll, select)
- Networking (buffers, protocols, ifaces)
- DNS and routing (latency, bandwidth)
Parallel Worker Threads

Goal: efficiently parallelize simulation workload

Sim Controller Process

1. parallel workers
2. direct execution
3. syscall interposition
4. syscall emulation
5. IPC
6. process control
7. CPU affinity
Goal: efficiently parallelize simulation workload

Sim Controller Process

Virtual hosts and processes

Physical processors
Goal: efficiently parallelize simulation workload

- Parallel workers
- Direct execution
- Syscall interposition
- Syscall emulation
- IPC
- Process control
- CPU affinity

Sim Controller Process

Virtual hosts and processes

How many worker threads should we run?

Physical processors
Goal: efficiently parallelize simulation workload

Sim Controller Process

How many worker threads should we run?

1. parallel workers
2. direct execution
3. syscall interposition
4. syscall emulation
5. IPC
6. process control
7. CPU affinity

Benchmark Time (s)

Max RAM Used (GiB)
Goal: efficiently parallelize simulation workload

Sim Controller Process

Virtual hosts and processes

One thread per host!

Physical processors

1. parallel workers
2. direct execution
3. syscall interposition
4. syscall emulation
5. IPC
6. process control
7. CPU affinity
Parallel Worker Threads

Goal: efficiently parallelize simulation workload
Goal: efficiently parallelize simulation workload

Thread scheduling:

1. Work stealing
2. Each LP starts a thread
   1. Runs all assigned events in current round (1 ms)
   2. Set thread to waiting
   3. Starts next waiting thread (if any)
3. When all threads waiting
   - Advance round clock
   - Repeat
Direct Execution

1. parallel workers
2. **direct execution**
3. syscall interposition
4. syscall emulation
5. IPC
6. process control
7. CPU affinity

Sim Controller Process

Linux Kernel Syscall API

Linux Kernel, Devices, Network
1. parallel workers
2. direct execution
3. syscall interposition
4. syscall emulation
5. IPC
6. process control
7. CPU affinity

App Process

Injected shim.so library

Sim Controller Process

1. LD_PRELOAD=shim.so
2. IPC="ipc.shm"
3. vfork() + execvpe()
Direct Execution

1. parallel workers
2. **direct execution**
3. syscall interposition
4. syscall emulation
5. IPC
6. process control
7. CPU affinity

---

**App Process**

- Injected shim.so library

**Sim Controller Process**

- Attach to "ipc.shm" for IPC

**IPC**

---

**Linux Kernel SySCALL API**

**Linux Kernel, Devices, Network**
Direct Execution

- parallel workers
- direct execution
- syscall interposition
- syscall emulation
- IPC
- process control
- CPU affinity

App is running and can communicate with controller

App Process

Injected shim.so library

Sim Controller Process

IPC

Linux Kernel SySCALL API

Linux Kernel, Devices, Network
Co-opting Linux Processes for High-Performance Network Simulation

1. parallel workers
2. direct execution
3. syscall interposition
4. syscall emulation
5. IPC
6. process control
7. CPU affinity

Syscall Interposition

- **Application Code**
- **Lib 0 (e.g. libc)**
- **Library 1**
- **...**
- **Library N**
- **Injected shim.so library**
- **Controller Code**
- **Controller Libraries**

Linked libraries:

- **Linux Kernel Syscall API**
- **Linux Kernel, Devices, Network**

Injected shim.so library
1. Intercept libcalls (LD_PRELOAD)

- Application Code
- Lib 0
- Library 1 (e.g. libc)
- ... Library N

Injected shim.so library

Linux Kernel SySCALL API

- Linux Kernel, Devices, Network

- Controller Code
- Controller Libraries

Syscall Interposition

1. parallel workers
2. direct execution
3. syscall interposition
4. syscall emulation
5. IPC
6. process control
7. CPU affinity
1. Intercept libcalls (LD_PRELOAD)
2. Intercept syscalls (seccomp)
3. syscall interposition
4. syscall emulation
5. IPC
6. process control
7. CPU affinity

Syscall Interposition

Application Code
Lib 0
Library 1 (e.g. libc)
...
Library N

Injected shim.so library

Linux Kernel Syscall API

Linux Kernel, Devices, Network

Controller Code
Controller Libraries
Co-opting Linux Processes for High-Performance Network Simulation

1. parallel workers
2. direct execution
3. syscall interposition
4. syscall emulation
5. IPC
6. process control
7. CPU affinity

Syscall Interposition

1. Intercept libcalls (LD_PRELOAD)
2. Intercept syscalls (seccomp)

Application Code
Lib 0
Library 1 (e.g. libc)
...
Library N

Injected shim.so library

Controller Code
Controller Libraries
Linux Kernel Syscall API
Linux Kernel, Devices, Network
Symscall Interposition

1. Intercept libcalls (LD_PRELOAD)
2. Intercept syscalls (seccomp)

<table>
<thead>
<tr>
<th>Benchmark Time (μs)</th>
<th>blocking nanosleep</th>
<th>nonblocking nanosleep</th>
<th>1k write+read</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>ony seccomp</td>
<td>seccomp</td>
<td>uni-process</td>
</tr>
<tr>
<td>pread</td>
<td>15.34</td>
<td>9.51</td>
<td>15.26</td>
</tr>
<tr>
<td>preload+seccomp</td>
<td>13.37</td>
<td>6.8</td>
<td>11.13</td>
</tr>
<tr>
<td>uni-process</td>
<td>9.1</td>
<td>0</td>
<td>7.78</td>
</tr>
</tbody>
</table>

1. parallel workers
2. direct execution
3. syscall interposition
4. syscall emulation
5. IPC
6. process control
7. CPU affinity
Syscall Emulation

1. parallel workers
2. direct execution
3. syscall interposition
4. syscall emulation
5. IPC
6. process control
7. CPU affinity

- File descriptors (files, sockets, pipes)
- Event notification (poll, epoll, select)
- Networking (buffers, protocols, ifaces)
- DNS and routing (latency, bandwidth)
1. parallel workers
2. direct execution
3. syscall interposition
4. syscall emulation
5. IPC
6. process control
7. CPU affinity

Syscall Emulation

- Emulate in Shim
  - faster
  - for hotpath syscalls (time)

Application Code
Lib 0
Injected shim.so library

Library 1 (e.g. libc)
...
Library N

Controller Code
Controller Libraries
Linux Kernel Syscall API
Linux Kernel, Devices, Network

- File descriptors (files, sockets, pipes)
- Event notification (poll, epoll, select)
- Networking (buffers, protocols, ifaces)
- DNS and routing (latency, bandwidth)
1. parallel workers
2. direct execution
3. syscall interposition
4. syscall emulation
5. IPC
6. process control
7. CPU affinity

Syscall Emulation

- File descriptors (files, sockets, pipes)
- Event notification (poll, epoll, select)
- Networking (buffers, protocols, ifaces)
- DNS and routing (latency, bandwidth)
Inter-Process Communication

1. parallel workers
2. direct execution
3. syscall interposition
4. syscall emulation
5. IPC
6. process control
7. CPU affinity

App Process

Injected shim.so library

Sim Controller Process

IPC
1. parallel workers
2. direct execution
3. syscall interposition
4. syscall emulation
5. IPC
6. process control
7. CPU affinity

Inter-Process Communication

App Process

Injected shim.so library

shared memory

Sim Controller Process

U.S. Naval Research Laboratory
Inter-Process Communication

App Process

Injected shim.so library

shared memory

Sim Controller Process

Shared memory + semaphores is the fastest IPC method for two processes running on the same core.

Latency (μs)

0.0

2.5

5.0

Same Core

Atomic Flag

Message Queue

Semaphore

Unix Domain Socket

4.4

5.1

4.1

6.3

Same Core

Shared memory + semaphores is the fastest IPC method for two processes running on the same core.
1. parallel workers
2. direct execution
3. syscall interposition
4. syscall emulation
5. IPC
6. process control
7. CPU affinity

Inter-Process Communication

App Process

Injected shim.so library

shared memory

Sim Controller Process

1. Fixed-size control block

- semaphores
- syscall registers
- simulation time
Inter-Process Communication

1. parallel workers
2. direct execution
3. syscall interposition
4. syscall emulation
5. IPC
6. process control
7. CPU affinity

App Process
- Injected shim.so library
- Shared memory

Sim Controller Process

1. Fixed-size control block
2. Remap app memory to directly read/write data buffers

- Semaphore
- Syscall registers
- Simulation time
- App stack
- App heap
Inter-Process Communication

1. parallel workers
2. direct execution
3. syscall interposition
4. syscall emulation
5. IPC
6. process control
7. CPU affinity

---

**App Process**
- Injected shim.so library
- Shared memory

**Sim Controller Process**

### Benchmark Time (μs)

<table>
<thead>
<tr>
<th>Benchmark Time (μs)</th>
<th>proc vm</th>
<th>proc mmap</th>
<th>uni-process</th>
</tr>
</thead>
<tbody>
<tr>
<td>1k write+read</td>
<td>13.68</td>
<td>11.13</td>
<td>7.78</td>
</tr>
<tr>
<td>4k write+read</td>
<td>15.1</td>
<td>11.42</td>
<td>8.77</td>
</tr>
<tr>
<td>16k write+read</td>
<td>20.44</td>
<td>11.56</td>
<td>11.57</td>
</tr>
<tr>
<td>64k write+read</td>
<td>44.16</td>
<td>22.66</td>
<td>25.28</td>
</tr>
</tbody>
</table>

2. remap app memory to directly read/write data buffers
1. parallel workers
2. direct execution
3. syscall interposition
4. syscall emulation
5. IPC
6. process control
7. CPU affinity

- Process Control
- App Process
- Sim Controller Process
- Injected shim.so library
- Shared memory
- Semaphores
- Syscall registers
- Simulation time
- App stack
- App heap
Process Control

1. parallel workers
2. direct execution
3. syscall interposition
4. syscall emulation
5. IPC
6. process control
7. CPU affinity

App Process

Injected shim.so library

Sim Controller Process

1. Write syscall registers
2. Signal controller semaphore
3. Wait on app semaphore

semaphores

syscall registers

simulation time

app stack

app heap
Process Control

1. parallel workers
2. direct execution
3. syscall interposition
4. syscall emulation
5. IPC
6. process control
7. CPU affinity

App Process

Injected shim.so library

Shared memory

Sim Controller Process

1. Write syscall registers
2. Signal controller semaphore
3. Wait on app semaphore

Semaphore
Syscall registers
Simulation time

1. Read syscall registers
2. Return if nonblocking else leave app idle and return when ready
3. Signal app semaphore
4. Wait on controller semaphore

App stack
App heap
Process Control

App Process

Injected shim.so library

shared memory

Sim Controller Process

Properties:
- Controller worker thread and its app process run synchronously
- Ensures nonconcurrent access to app stack and heap memory

1. Write syscall registers
2. Signal controller semaphore
3. Wait on app semaphore

4. Read syscall registers
5. Return if nonblocking else leave app idle and return when ready
6. Signal app semaphore
7. Wait on controller semaphore
Process Control

1. parallel workers
2. direct execution
3. syscall interposition
4. syscall emulation
5. IPC
6. process control
7. CPU affinity

Use CPU pinning to pin each worker thread and all of its managed processes to the same core.

Benchmark Time (μs)

- Blocking nanosleep
- Nonblocking nanosleep
- 1k write+read

App Process
- Injected shim.so library

Sim Controller Process
- shared memory

Use CPU pinning to pin each worker thread and all of its managed processes to the same core.
Outline

motivation
design
evaluation
Evaluation: Large P2P Benchmarks

Faster and more scalable than the uni-process, plugin architecture!

Uses significantly less memory than the uni-process, plugin architecture!

- Faster and more scalable than the uni-process, plugin architecture!
- Uses significantly less memory than the uni-process, plugin architecture!
Evaluation: Large Tor Networks

Performance comparable to the state of the art for large Tor networks

Uses significantly less memory than the uni-process, plugin architecture!

![Graphs showing relative run time and RAM usage for different Tor network model scales.](image)
Main Takeaways

Designed a new, hybrid network simulator/emulator
- co-opts Linux processes into a discrete-event network simulation that emulates kernel functionality
- enables large-scale, distributed system experiments

- Merged into the open-source Shadow project and synonymous with Shadow v2
- Artifacts: https://netsim-atc2022.github.io

• 2.3x faster than Shadow v1
• 3.4x faster than NS-3
• 43x faster than gRaIL [ToN’19]