Privbox: Faster System Calls Through Sandboxed Privileged Execution

Dmitry Kuznetsov, Adam Morrison
Tel Aviv University
System Calls

- Main interface for requesting operating system services
- Semantically similar to simple function call (i.e. prepare parameters, invoke, receive result)
- Unlike function call, involves many more steps and is much slower!
  - E.g. hardware: privilege level change
- Spectre/Meltdown mitigations (e.g. PTI) make things even worse
System Call Overhead

- Particularly bad for system call heavy workloads
  - Recall: almost all I/O operations eventually translate to a system call
  - System call heavy = I/O heavy

- Back-of-the-envelope: Redis
  - 200k requests / second (single threaded, w/o pipelining)
  - At least 2 system calls per request (recv + send)
  - ~900 cycles per system call
  - Over 13% of a core running at 2.6GHz
Existing Approaches

- **Batching** (*preadv/...*): perform less round-trips to kernel by doing several operations each entry:
  - ✗ Possible only for specific operations
Existing Approaches

- **Batching** (*preadv/…*): perform less round-trips to kernel by doing several operations each entry:
  - × Possible only for specific operations
- **Entry-less mechanisms** (*FlexSC, io_uring*): request system calls through memory interface:
  - × Requires kernel-side polling
  - × Makes system calls asynchronous
Existing Approaches

- **Batching** (*preadv/...*): perform less round-trips to kernel by doing several operations each entry:
  - ✗ Possible only for specific operations

- **Entry-less mechanisms** (*FlexSC, io_uring*): request system calls through memory interface:
  - ✗ Requires kernel-side polling
  - ✗ Makes system calls asynchronous

- **Kernel bypass** (*DPDK, SPDK*): map whole device into process memory:
  - ✗ No high-level abstractions from kernel (files, sockets)
  - ✗ Not possible to share the devices between processes
Existing Approaches

- Batching (*preadv/...*): perform less round-trips to kernel by doing several operations each entry:
  - ✗ Possible only for specific operations

- Entry-less mechanisms (**FlexSC, io_uring**): request system calls through memory interface:
  - ✗ Requires kernel-side polling
  - ✗ Makes system calls asynchronous

- Kernel bypass (**DPDK, SPDK**): map whole device into process memory:
  - ✗ No high-level abstractions from kernel (files, sockets)
  - ✗ Not possible to share the devices between processes

➤ All of the above require software re-architecture!
Privbox

- New mechanism for system call intensive workloads that allows system calls with less overhead
- Privbox achieves this by allowing user programs to load and execute system call heavy code under a new semi-privileged (almost kernel-like) but sandboxed execution mode
Privbox

- New mechanism for system call intensive workloads that allows system calls with less overhead
- Privbox achieves this by allowing user programs to load and execute system call heavy code under a new semi-privileged (almost kernel-like) but sandboxed execution mode.

Advantages:

✓ 2.2x less system call overhead
✓ System call retain familiar and synchronous semantics
✓ Does not require software re-architecture or major source code changes
  ➢ Example: Memcached:
    ✓ Ported to use Privbox in under one hour and 70 LOC
Privbox Mechanism

Regular execution

User process
- main:
  - do_IO()
- do_IO:
  - loop:
    - read()

Kernel
- sys_read
- syscall entry

System call (slow)
Function call (fast)
Privbox Mechanism

Regular execution vs Execution with Privbox

- **Function call (fast)**
- **System call (slow)**

*code inside Privbox is running in privileged CPU mode, but instrumented and sandboxed for security*
Semi-Privileged Execution Mode (SPEM)

- New execution mode for user processes
  - Based on Kernel-mode Linux
- Used during Privboxed code invocation

Details:

- Runs under privileged CPU mode (e.g. ring 0)
- Allows system calls through system call gate function
- Identical to regular processes from all other perspectives
  - Same permission checks, scheduling, etc

<table>
<thead>
<tr>
<th></th>
<th>Regular</th>
<th>SPEM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Subject to permissions checks</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Preemptible</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Can block</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>...</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>
System Call Gate Function

- Function in kernel memory
- Similar to syscall instruction handler
  - ✔ But with less steps
- Same semantics
- Reach kernel code through function calls
  - ✔ No need to change privilege level

**Diagram:**
- User process
- Privbox
- System call gate function
- Kernel
- Function call (fast)
- System call (slow)
Flow

1. Developer marks code intended for Privbox

   Developer marks system call intensive code paths
Flow

1. Developer marks code intended for Privbox
2. Developer compiles code with a custom compiler that introduces instrumentation

- Instrumentation adds overhead
- Therefore, annotating the whole program can be suboptimal
Flow

1. Developer marks code intended for Privbox
2. Developer compiles code with a custom compiler that introduces instrumentation
3. Program loads instrumented code into a Privbox environment

- Custom compiler adds instrumentation
- Instrumentation adds overhead
- Therefore, annotating the whole program can be suboptimal
Flow

1. Developer marks code intended for Privbox
2. Developer compiles code with a custom compiler that introduces instrumentation
3. Program loads instrumented code into a Privbox environment
4. Program can invoke loaded code through a special system call that transfers control to invoked code under Semi-Privileged Execution mode

Developer marks system call intensive code paths
Custom compiler adds instrumentation

Instrumentation adds overhead
Therefore, annotating the whole program can be suboptimal

Privbox Verifier + Loader
Kernel
Semi privileged execution mode
Privbox code region
Custom page table
Porting Programs to Privbox

Standard application

```c
do_IO(...) {
    for (...) { ... }
    return result;
}

main(...) {
    ...
    do_IO(...);
    ...
}
```

Application with Privbox

```c
do_IO(...) {
    for (...) { ... }
    return result;
}

main(...) {
    ...
    do_IO(...);
    ...
}
```
Porting Programs to Privbox

1. Developer marks system call intensive code

```c
#include <sys/privbox.h>

PRIVBOX_MARKER

do_IO(...) {
    for (...) { ...}
    return result;
}

main(...) {
    ...
    do_IO(...);
    ...
}
```

Standard application

```c
#include <sys/privbox.h>

PRIVBOX_MARKER // 1.
do_IO(...) {
    for (...) { ...}
    return result;
} 

main(...) {
    ...
    do_IO(...);
    ...
}
```

Application with Privbox
Porting Programs to Privbox

1. Developer marks system call intensive code
2. Program loads code into a Privbox
Porting Programs to Privbox

1. Developer marks system call intensive code
2. Program loads code into a Privbox
3. Program invokes code inside Privbox

```
#include <sys/privbox.h>

PRIVBOX_MARKER // 1.
do_IO(...) {
    for (...) { ...}
    return result;
}

main(...) {
    privbox_load(do_IO); // 2.
    privbox_invoke(do_IO); // 3.
    ...
}
```
Porting Programs to Privbox

1. Developer marks system call intensive code
2. Program loads code into a Privbox
3. Program invokes code inside Privbox

✓ Minimal code changes
✓ Well suited for I/O threaded workloads

#include <sys/privbox.h>
PRIVBOX_MARKER // 1.
do_IO(...) {
    for (...) { ...}
    return result;
}
main(...) {
    ...
    do_IO(...);
    ...
}

Application with Privbox

Instrumentation

Compute threads
Comm. over memory

IO threads
Privboxed code
System calls
Safety Requirements

Problem:

- Privbox executes code with kernel-like privileges (e.g. ring 0)
- Malicious user code can gain complete control of the machine
Safety Requirements

Problem:

- Privbox executes code with kernel-like privileges (e.g. ring 0)
- Malicious user code can gain complete control of the machine

High-level safety objective: no new access through Privbox
Safety Requirements

Problem:

- Privbox executes code with kernel-like privileges (e.g. ring 0)
- Malicious user code can gain complete control of the machine

High-level safety objective: no new access through Privbox

Sandbox imposes following properties on loaded code:

1. No privileged instructions
2. No kernel memory accesses
3. No branching to unverified code
Compilation and Verification

- Safety of Privbox relies on verification of loaded code
- Inspired by Native Client work
- **Privbox Compiler:**
  - Transforms potentially unsafe instructions into equivalent but verifiably safe instruction sequences
Compilation and Verification

- Safety of Privbox relies on verification of loaded code
- Inspired by Native Client work
- **Privbox Compiler:**
  - Transforms potentially unsafe instructions into equivalent but verifiably safe instruction sequences
- **Privbox Verifier:**
  - Triggered each time code is loaded into Privbox
  - Disassemble loaded code
  - Reject if code violates safety requirements
Verification

- **Challenge:**
  - Variable length instructions hamper the ability to disassemble code
Verification

● **Challenge:**
  ○ Variable length instructions hamper the ability to disassemble code

● **Code chunk:**
  ○ A fixed in size and aligned in memory group of instructions
Verification

- **Challenge:**
  - Variable length instructions hamper the ability to disassemble code

- **Code chunk:**
  - A fixed in size and aligned in memory group of instructions

- **Solution:**
  - Pack code into *code chunks*
  - Restrict branching to chunk-aligned addresses
Privileged Instructions

- Trivial:
  - Check during disassembly
  - Reject if present

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>No priv. instructions</td>
<td>✓</td>
</tr>
<tr>
<td>No kernel access</td>
<td>N/A</td>
</tr>
<tr>
<td>No branching outside sandbox</td>
<td>N/A</td>
</tr>
</tbody>
</table>
Load/Store Instructions

- Load/store instructions have memory operands
- Effective address of memory operand may be known only at run time
Load/Store Instructions

- Load/store instructions have memory operands
- Effective address of memory operand may be known only at run time
- **Safety requirement:**
  - No kernel memory access

<table>
<thead>
<tr>
<th>Requirement</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>No priv. instructions</td>
<td>N/A</td>
</tr>
<tr>
<td>No kernel access</td>
<td>✓</td>
</tr>
<tr>
<td>No branching outside sandbox</td>
<td>N/A</td>
</tr>
</tbody>
</table>
Load/Store Instructions

- Load/store instructions have memory operands
- Effective address of memory operand may be known only at run time
- **Safety requirement:**
  - No kernel memory access
- **Sanitation:**
  - Mask most significant bit of memory operand
  - \( \text{addr} \rightarrow \text{addr} \& \sim (1 << 63) \)
  - … no longer a kernel address

<table>
<thead>
<tr>
<th>No priv. instructions</th>
<th>N/A</th>
</tr>
</thead>
<tbody>
<tr>
<td>No kernel access</td>
<td>✓</td>
</tr>
<tr>
<td>No branching</td>
<td>N/A</td>
</tr>
<tr>
<td>outside sandbox</td>
<td></td>
</tr>
</tbody>
</table>

47-bit address space

47-bit address space

47-bit address space

47-bit address space

64-bit address space

64-bit address space

Memory load:

\[
\text{%dest} = \text{mov } \text{disp}(\%base, \text{scale}, \%index) \\
\text{%tmp1} = \text{lea } \text{disp}(\%base, \text{scale}, \%index) \\
\text{%tmp2} = \text{btr } \$63, \%tmp1 \\
\text{%dest} = \text{mov } (\%tmp2)
\]
Branching Instructions

- Indirect branches (and returns) branch to addresses stored in registers or memory
- Effective address might be known only at run time
Branching Instructions

- Indirect branches (and returns) branch to addresses stored in registers or memory
- Effective address might be known only at run time
- **Safety requirement:**
  - No kernel memory access
  - Branch only to chunk beginnings

<table>
<thead>
<tr>
<th>No priv. instructions</th>
<th>N/A</th>
</tr>
</thead>
<tbody>
<tr>
<td>No kernel access</td>
<td>✓</td>
</tr>
<tr>
<td>No branching outside sandbox</td>
<td>✓</td>
</tr>
</tbody>
</table>
Branching Instructions

- Indirect branches (and returns) branch to addresses stored in registers or memory
- Effective address might be known only at run time
- **Safety requirement:**
  - No kernel memory access
  - Branch only to chunk beginnings
- **Sanitation:**
  - Mask MSB and clear lowest bits
  - \( \text{addr} \Rightarrow \text{addr} \& \sim(1<<63) \& \sim31 \)
  - ... non-kernel address and chunk-aligned.

### Indirect function call:
```
call disp(%base, scale, %index)
```

\( \%\text{tmp1} = \text{lea} \; \text{disp}(%\text{base}, \text{scale}, \%\text{index}) \)
\( \%\text{tmp2} = \text{btr} \; \$63, \%\text{tmp1} \)
\( \%\text{tmp3} = \text{and} \; \sim\$31, \%\text{tmp2} \)
```
call *%\text{tmp3}
```
Branching Instructions

- Indirect branches (and returns) branch to addresses stored in registers or memory
- Effective address might be known only at run time
- **Safety requirement:**
  - No kernel memory access
  - Branch only to chunk beginnings
- **Sanitation:**
  - Mask MSB and clear lowest bits
  - $addr \Rightarrow addr \& \sim (1\ll 63) \& \sim 31$
  - ... non-kernel address and chunk-aligned.
  - **✗ Can still branch to aligned user address!**

<table>
<thead>
<tr>
<th>No priv. instructions</th>
<th>N/A</th>
</tr>
</thead>
<tbody>
<tr>
<td>No kernel access</td>
<td>✓</td>
</tr>
<tr>
<td>No branching outside sandbox</td>
<td>✓</td>
</tr>
</tbody>
</table>

Indirect function call:
```
call disp(%base, scale, %index)
```

```
%tmp1 = lea disp(%base, scale, %index)
%tmp2 = btr $63, %tmp1
%tmp3 = and ~$31, %tmp2
call *%tmp3
```
Memory Layout

1. **Privboxed code** region inside user memory
   - Immutable by user-space

   - Kernel memory
     - Syscall gate function
     - Non-canonical addresses
     - Privboxed code
     - User memory (non-executable)
Memory Layout

1. **Privboxed code** region inside user memory
   - Immutable by user-space
2. **User memory** mirroring regular memory layout
   - ✔ Non-executable: completes branching instrumentation
     - Recall: instrumented branches can only target *non-kernel, 32-byte aligned* addresses
Memory Layout

1. **Privboxed code** region inside user memory
   - Immutable by user-space
2. **User memory** mirroring regular memory layout
   - ✓ Non-executable: completes branching instrumentation
     - Recall: instrumented branches can only target _non-kernel, 32-byte aligned_ addresses
3. **Kernel memory** is mapped and accessible
   - ○ Enables direct branching to syscall gate function!
   - ○ Undesired kernel accesses blocked by instrumentation
Semi-Privileged Access Prevention (SPAP)

- **Observation**: Majority of overhead comes from load/store instrumentation
- **Solution**: we propose a new, SMAP/SMEP-like, hardware extension
  - Mechanism:
    - Generate faults on supervisor page (kernel memory) access
    - ...when executing from non-supervisor pages under privileged mode (SPEM)
  - Minimal expected overhead (very similar to SMAP/SMEP)
  - Details in paper
Semi-Privileged Access Prevention (SPAP)

- **Observation:** Majority of overhead comes from load/store instrumentation
- **Solution:** we propose a new, SMAP/SMEP-like, hardware extension
  - **Mechanism:**
    - Generate faults on supervisor page (kernel memory) access
    - ...when executing from non-supervisor pages under privileged mode (SPEM)
  - **Outcome:**
    - ✓ Load/store instrumentation no longer required
    - ✓ Branching instrumentation need only to take care of alignment
Evaluation: Entry Overheads

Benchmark: measurement of system call entry/exit overhead (on x86)

Results:

✓ Privbox is **2.2x** faster than regular system call on system with PTI
Evaluation: I/O Threaded Workloads

Benchmark: server with I/O isolated to dedicated threads

Results:

✓ Up to 72% speedup for scenarios where I/O is the bottleneck (on kernels with PTI)
Evaluation: Real-world Workloads

redis

- **Benchmark**: redis-bench / memtier_benchmark
- **Results**:
  - ✓ Up to 7.6% speedup on hardware that requires PTI
  - ✓ Up to 11% speedup if hardware supported SPAP

memcached

- **Benchmark**: memtier_benchmark
- **Results**:
  - ✓ Up to 5.5% speedup on hardware that requires PTI
  - ✓ Up to 8.4% speedup if hardware supported SPAP

Note: Lower bounds, whole processes instrumented
Conclusion

✓ **Privbox**: faster system calls with familiar semantics
✓ No need to re-architect software
✓ 2.2x times faster system call entry/exit
✓ Up to 72% speedup for IO-threaded workloads
✓ Lower bound of 7% speedup for workloads like Redis/Memcached
✓ Github: [https://github.com/privbox](https://github.com/privbox)
Privbox vs eBPF

Privbox:

- Safety guarantees:
  - Memory accesses
- Scope:
  - Full programs
- Execution model:
  - Runs like regular process
  - Uses system call as needed

eBPF:

- Safety guarantees:
  - Memory safety
  - Termination
- Scope:
  - Callback functions, small programs
- Execution model:
  - Invoked by kernel on events
  - Can invoke only specific helpers