zpoline: a system call hook mechanism based on binary rewriting

Kenichi Yasukata\textsuperscript{1}, Hajime Tazaki\textsuperscript{1}, Pierre-Louis Aublin\textsuperscript{1}, Kenta Ishiguro\textsuperscript{2}

\textsuperscript{1}IIJ Research Laboratory
\textsuperscript{2}Hosei University
System Call

- System calls are the primary interface for user-space programs to communicate with OS kernels
System Call

• System calls are the primary interface for user-space programs to communicate with OS kernels
System Call

- System calls are the primary interface for user-space programs to communicate with OS kernels.

User-space program

Kernel-space OS subsystem
System Call

- System calls are the primary interface for user-space programs to communicate with OS kernels.
System Call Hook

• System calls are the primary interface for user-space programs to communicate with OS kernels
• A system call hook mechanism intercepts a system call

User-space program

system call

Kernel-space OS subsystem
System Call Hook

- System calls are the primary interface for user-space programs to communicate with OS kernels
- A system call hook mechanism intercepts a system call
System Call Hook

- System calls are the primary interface for user-space programs to communicate with OS kernels
- A system call hook mechanism intercepts a system call, and redirects the execution to a user-defined hook function

![Diagram of system call hook](image-url)
Motivating Use Case

- System call hook mechanisms allow us to transparently apply user-space OS subsystems to existing applications
Motivating Use Case

- System call hook mechanisms allow us to transparently apply user-space OS subsystems to existing applications.
Motivating Use Case

• System call hook mechanisms allow us to transparently apply user-space OS subsystems to existing applications
Motivating Use Case

System call hook mechanisms allow us to transparently apply user-space OS subsystems to existing applications.

User-space network stack

Highly performant

TCP ping-pong performance

Throughput [M reqs/s]

5.3 times faster

Linux TCP stack lwIP on DPDK = user-space network stack
Motivating Use Case

• System call hook mechanisms allow us to transparently apply user-space OS subsystems to existing applications.
Motivating Use Case

• System call hook mechanisms allow us to transparently apply user-space OS subsystems to existing applications.

Normally, adaptation requires changes of a user-space program to apply a specific API of a user-space OS subsystem.

User-space program

\[ \text{system call} \]

Kernel-space OS subsystem

User-space OS subsystem

Highly performant
Motivating Use Case

• System call hook mechanisms allow us to **transparently apply** user-space OS subsystems to existing applications.

Normally, adaptation requires changes of a user-space program to apply a specific API of a user-space OS subsystem.

*We need to change the program*
Motivating Use Case

• System call hook mechanisms allow us to transparently apply user-space OS subsystems to existing applications

If we use a system call hook mechanism, ...

User-space program

\textit{system call}

Kernel-space OS subsystem
Motivating Use Case

• System call hook mechanisms allow us to **transparently apply user-space OS subsystems** to existing applications

If we use a system call hook mechanism, ...

![Diagram](system-call-hook-diagram)

- User-space program
- **User-defined hook function**
- User-space OS subsystem
- Kernel-space OS subsystem

**system call hook**
Motivating Use Case

• System call hook mechanisms allow us to transparently apply user-space OS subsystems to existing applications.

If we use a system call hook mechanism, ...

no modification of the user-space program is necessary

User-space program

User-defined hook function

User-space OS subsystem

Kernel-space OS subsystem

system call hook
Problem

• System call hook mechanisms allow us to \textit{transparently} apply user-space OS subsystems to existing applications.

\textbf{If we use a system call hook mechanism, ...}

\textit{no modification of the user-space program is necessary}

\begin{itemize}
    \item User-space program
    \item User-defined hook function
    \item User-space OS subsystem
    \item Kernel-space OS subsystem
\end{itemize}

\textit{system call hook} \quad \textit{There is no perfect hook mechanism}
System call hook mechanisms allow us to **transparently** apply user-space OS subsystems to existing applications without modification of the user-space program. A user-defined hook function intercepts system calls, allowing the OS subsystem to be invoked. However, there is no perfect hook mechanism.

### Existing Mechanisms
- ptrace
- int3 signaling technique
- Syscall User Dispatch (SUD)
- LD_PRELOAD trick
- Binary rewriting techniques
- ...
Problem

System call hook mechanisms allow us to transparently apply user-space OS subsystems to existing applications. User-space program system call kernel-space OS subsystem

If we use a system call hook mechanism, no modification of the user-space program is necessary.

Existing Mechanisms

- ptrace
- int3 signaling technique
- Syscall User Dispatch (SUD)
- LD_PRELOAD trick
- Binary rewriting techniques
- ...
Problem

Existing Mechanisms
- ptrace
- int3 signaling technique
- Syscall User Dispatch (SUD)
- LD_PRELOAD trick
- Binary rewriting techniques
- ...

Throughput [M reqs/sec]

Compared to
LD_PRELOAD

Relying on kernel facilities

14.7% 17.0%

1.1%
Problem

- ptrace
  - overhead: process scheduling between the tracer and tracee processes

Existing Mechanisms

- ptrace
- int3 signaling
- Syscall User Dispatch (SUD)
- LD_PRELOAD trick
- Binary rewriting techniques

Compared to LD_PRELOAD

Relying on kernel facilities

<table>
<thead>
<tr>
<th>Throughput [M reqs/sec]</th>
<th>lwIP on DPDK: TCP ping-pong</th>
</tr>
</thead>
<tbody>
<tr>
<td>lwIP on DPDK: TCP ping-pong</td>
<td>17.0%</td>
</tr>
<tr>
<td>Compared to LD_PRELOAD</td>
<td>14.7% 17.0%</td>
</tr>
</tbody>
</table>

Kernel-space OS subsystem
Problem

- ptrace
  - overhead: process scheduling between the tracer and tracee processes
- int3 signaling / SUD
  - overhead: context manipulation for a signal() handler (SIGINT/SIGSYS)

Compared to LD_PRELOAD

Relying on kernel facilities

Throughput [M reqs/sec]

lwIP on DPDK: TCP ping-pong

- ptrace: 1.1%
- int3 signaling: 14.7%
- SUD: 17.0%
- LD_PRELOAD: 17.0%
Problem

• ptrace
  • overhead: process scheduling between the tracer and tracee processes

• int3 signaling / SUD
  • overhead: context manipulation for a signal() handler (SIGINT/SIGSYS)

• LD_PRELOAD just replaces function calls, therefore, it is fast

Compared to LD_PRELOAD

Relying on kernel facilities

<table>
<thead>
<tr>
<th>Mechanism</th>
<th>Throughput [M reqs/sec]</th>
</tr>
</thead>
<tbody>
<tr>
<td>ptrace</td>
<td>14.7%</td>
</tr>
<tr>
<td>int3 signaling</td>
<td>17.0%</td>
</tr>
<tr>
<td>SUD</td>
<td>1.1%</td>
</tr>
<tr>
<td>LD_PRELOAD</td>
<td>1.1%</td>
</tr>
</tbody>
</table>
Problem

- System call hook mechanisms allow us to transparently apply user-space OS subsystems to existing applications.

User-space program system call

Kernel-space OS subsystem system call hook

User-defined hook function

No modification of the user-space program is necessary.

If we use a system call hook mechanism, there is no perfect hook mechanism:

- ptrace
- int3 signaling technique
- Syscall User Dispatch (SUD)
- LD_PRELOAD trick
- Binary rewriting techniques

...
Problem

- System call hook mechanisms allow us to transparently apply user-space OS subsystems to existing applications.

User-space program system call

Kernel-space OS subsystem

There is no perfect hook mechanism:

- ptrace
- int3 signaling technique
- Syscall User Dispatch (SUD)
- LD_PRELOAD trick
- Binary rewriting techniques

Existing Mechanisms

Throughput [M reqs/sec]

lwIP on DPDK: TCP ping-pong

17.0%

Compared to LD_PRELOAD
Relying on kernel facilities:

- ptrace: overhead: process scheduling between the tracer and tracee processes
- int3 signaling / SUD: overhead: context manipulation for a signal() handler (SIGINT/SIGSYS)
- LD_PRELOAD just replaces function calls, therefore, it is fast

```
app_function(...) {
    ...
    write(...) 
    ...
}
```

User-defined library call

User-defined library call

Function call replacement

LD_PRELOAD
Problem

- System call hook mechanisms allow us to transparently apply user-space OS subsystems to existing applications.

User-space program system call

Kernel-space OS subsystem

No modification of the user-space program is necessary.

If we use a system call hook mechanism, we have the following existing mechanisms:

- ptrace
- int3 signaling technique
- Syscall User Dispatch (SUD)
- LD_PRELOAD trick
- Binary rewriting techniques

There is no perfect hook mechanism.

Compared to LD_PRELOAD:
- Relying on kernel facilities:
  - ptrace: overhead due to process scheduling between the tracer and tracee processes
  - int3 signaling / SUD: overhead due to context manipulation for a signal() handler (SIGINT/SIGSYS)
- LD_PRELOAD just replaces function calls, therefore, it is fast.

```c
app_function(...)
{
    ...
    special_write(...)
    ...
}
```

User-defined write() library call

LD_PRELOAD

lwIP on DPDK: TCP ping-pong

Throughput [M reqs/sec]

Compared to LD_PRELOAD:

17.0% 14.7% 17.0%
Problem

• System call hook mechanisms allow us to transparently apply user-space OS subsystems to existing applications.

User-space program system call

Kernel-space OS subsystem

There is no perfect hook mechanism

• ptrace
• int3 signaling technique
• Syscall User Dispatch (SUD)
• LD_PRELOAD trick
• Binary rewriting techniques

Existing Mechanisms

<table>
<thead>
<tr>
<th>Throughput [M reqs/sec]</th>
<th>lwIP on DPDK : TCP ping-pong</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Compared to LD_PRELOAD</td>
</tr>
<tr>
<td></td>
<td>Relying on kernel facilities</td>
</tr>
<tr>
<td></td>
<td>ptrace</td>
</tr>
<tr>
<td></td>
<td>int3 signaling / SUD</td>
</tr>
<tr>
<td></td>
<td>LD_PRELOAD just replaces function calls, therefore, it is fast</td>
</tr>
</tbody>
</table>

```c
app_function(...)
{
    ...
    special_write(...)
    ...
}

special_write(...)
{
    asm volatile ( trigger write syscall )
}
```

LD_PRELOAD
Problem

- System call hook mechanisms allow us to transparently apply user-space OS subsystems to existing applications.

User-space program system call

Kernel-space OS subsystem

User-defined hook function

No modification of the user-space program is necessary.

If we use a system call hook mechanism, we can...
Problem

- ptrace
  - overhead: process scheduling between the tracer and tracee processes
- int3 signaling / SUD
  - overhead: context manipulation for a signal() handler (SIGINT/SIGSYS)
- LD_PRELOAD just replaces function calls, therefore, it is fast

If we use a system call hook mechanism, there is no perfect hook mechanism...

**Existing Mechanisms**

- ptrace
- int3 signaling
- Syscall User Dispatch (SUD)
- LD_PRELOAD trick
- Binary rewriting techniques

**Throughput [M reqs/sec]**

- lwIP on DPDK: TCP ping-pong
  - Compared to LD_PRELOAD
    - Relying on kernel facilities
      - ptrace
        - overhead: process scheduling between the tracer and tracee processes
      - int3 signaling / SUD
        - overhead: context manipulation for a signal() handler (SIGINT/SIGSYS)
      - LD_PRELOAD just replaces function calls, therefore, it is fast
System call hook mechanisms allow us to transparently apply user-space OS subsystems to existing applications. If we use a system call hook mechanism, no modification of the user-space program is necessary. There is no perfect hook mechanism. Here are some existing mechanisms:

- ptrace
- int3 signaling technique
- Syscall User Dispatch (SUD)
- LD_PRELOAD trick
- Binary rewriting techniques

Compared to relying on kernel facilities:

- ptrace: overhead due to process scheduling between the tracer and tracee processes
- int3 signaling / SUD: overhead due to context manipulation for a signal() handler (SIGINT/SIGSYS)
- LD_PRELOAD just replaces function calls, therefore, it is fast.

```
app_function(...) {
    ...
    special_write(...) 
    ...
}
```

Hook is not applied because names are different:

```
special_write(...) {
    asm volatile (
        trigger write syscall
    )
}
```

User-defined library call (because names are different)

```
write() {
    ...
    special_write(...) 
    ...
}
```

```
asm volatile ( 
    trigger write syscall
)
```

LD PRELOAD
Problem

- System call hook mechanisms allow us to transparently apply user-space OS subsystems to existing applications.

User-space program system call

Kernel-space OS subsystem

User-defined hook function

No modification of the user-space program is necessary.

If we use a system call hook mechanism, ...

There is no perfect hook mechanism:

- ptrace
- int3 signaling technique
- Syscall User Dispatch (SUD)
- LD_PRELOAD trick
- Binary rewriting techniques

...
Problem

System call hook mechanisms allow us to **transparently** apply user-space OS subsystems to existing applications

**Hook mechanism, ...**

If we use a system call hook mechanism, ...

**Existing Mechanisms**

- ptrace
- int3 signaling technique
- Syscall User Dispatch (SUD)
- LD_PRELOAD trick
- Binary rewriting techniques
- ...

There is no perfect hook mechanism

Kernel-space OS subsystem

User-defined hook function

User-space OS subsystem

system call hook
Problem

• System call hook mechanisms allow us to **transparently** apply **user-space OS subsystems** to existing applications.

**User-space program**

*system call hook*

**Kernel-space OS subsystem**

If we use a system call hook mechanism, ...

There is no perfect hook mechanism

**Existing Mechanisms**

- ptrace
- int3 signaling technique
- Syscall User Dispatch (SUD)
- LD_PRELOAD trick
- Binary rewriting techniques
- …

**High performance penalty**
Problem

System call hook mechanisms allow us to **transparently** apply user-space OS subsystems to existing applications. If we use a system call hook mechanism, ... no modification of the user-space program is necessary.

There is no perfect hook mechanism:

- ptrace
- int3 signaling technique
- Syscall User Dispatch (SUD)
- ELF LOAD trick
- Binary rewriting techniques
- ...

**High performance penalty**

**Sometimes fail to hook**

**system call hook**

There is no perfect hook mechanism

Kernel-space OS subsystem

User-space OS subsystem

User-defined hook function
Applicability of user-space OS subsystems has been limited regardless of their benefits.

- System call hook mechanisms allow us to **transparently** apply user-space OS subsystems to existing applications.
- No modification of the user-space program is necessary.
- If we use a system call hook mechanism, ...

**Existing Mechanisms**
- High performance penalty
- Sometimes fail to hook

**System Call Hook**

- User-defined hook function
- User-space OS subsystem

**Kernel-space OS subsystem**
Contribution

• **zpoline**: a system call hook mechanism for x86-64 CPUs
  • based on binary rewriting
  • free from the drawbacks of the previous mechanisms

• This work addresses a challenge that is specific to binary rewriting approaches
Binary Rewriting Approach

• On x86-64 CPUs, syscall and sysenter instructions trigger a system call
  • syscall: 0x0f 0x05, sysenter: 0x0f 0x34
Binary Rewriting Approach

• On x86-64 CPUs, syscall and syenter instructions trigger a system call
  • syscall: 0x0f 0x05, syenter: 0x0f 0x34
Binary Rewriting Approach

• On x86-64 CPUs, syscall and sysenter instructions trigger a system call
  • syscall: 0x0f 0x05, sysenter: 0x0f 0x34

• What we wish to achieve
Binary Rewriting Approach

• On x86-64 CPUs, syscall and sysenter instructions trigger a system call
  • syscall: 0x0f 0x05, sysenter: 0x0f 0x34

• What we wish to achieve
Binary Rewriting Approach

- On x86-64 CPUs, syscall and sysenter instructions trigger a system call
  - syscall: 0x0f 0x05, sysenter: 0x0f 0x34

- What we wish to achieve
  - replace syscall/sysenter instruction with something

![Virtual Memory Diagram](image)
Binary Rewriting Approach

• On x86-64 CPUs, syscall and sysenter instructions trigger a system call
  • syscall: 0x0f 0x05, sysenter: 0x0f 0x34

• What we wish to achieve
  • replace syscall/sysenter instruction with something
Binary Rewriting Approach

- On x86-64 CPUs, syscall and sysenter instructions trigger a system call
  - syscall: 0x0f 0x05, sysenter: 0x0f 0x34

- What we wish to achieve
  - replace syscall/sysenter instruction with something
  - to jump to a user-defined hook function
Binary Rewriting Approach

• On x86-64 CPUs, syscall and sysenter instructions trigger a system call
  • syscall: 0x0f 0x05, sysenter: 0x0f 0x34

• What we wish to achieve
  • replace syscall/sysenter instruction with something
  • to jump to a user-defined hook function

• Question: what should we put here?
Challenge

- On x86-64 CPUs, syscall and sysenter instructions trigger a system call
  - syscall: 0x0f 0x05, sysenter: 0x0f 0x34
Challenge

• On x86-64 CPUs, syscall and sysenter instructions trigger a system call
  • syscall: 0x0f 0x05, sysenter: 0x0f 0x34

• syscall and sysenter are 2-byte instructions
Challenge

- On x86-64 CPUs, syscall and sysenter instructions trigger a system call
  - syscall: \texttt{0x0f 0x05}, sysenter: \texttt{0x0f 0x34}

- syscall and sysenter are \textbf{2-byte} instructions

- Specification for a jump destination address needs more than 2 bytes
Challenge

• On x86-64 CPUs, syscall and sysenter instructions trigger a system call
  • syscall: 0x0f 0x05, sysenter: 0x0f 0x34

• syscall and sysenter are 2-byte instructions

• Specification for a jump destination address needs more than 2 bytes
Challenge

• On x86-64 CPUs, syscall and sysenter instructions trigger a system call
  • syscall: 0x0f 0x05, sysenter: 0x0f 0x34

• syscall and sysenter are 2-byte instructions

• Specification for a jump destination address needs more than 2 bytes

If we put ADDR, subsequent instructions are overwritten
Challenge

• On x86-64 CPUs, syscall and sysenter instructions trigger a system call
  • syscall: 0x0f 0x05, sysenter: 0x0f 0x34

• syscall and sysenter are 2-byte instructions

• Specification for a jump destination address needs more than 2 bytes
  If we put ADDR, subsequent instructions are overwritten
Challenge

- On x86-64 CPUs, syscall and sysenter instructions trigger a system call
  - syscall: 0x0f 0x05, sysenter: 0x0f 0x34
- syscall and sysenter are 2-byte instructions
- Specification for a jump destination address needs more than 2 bytes

If we put ADDR, subsequent instructions are overwritten; jump to the overwritten part leads to unexpected behaviors.
Challenge

- On x86-64 CPUs, syscall and sysenter instructions trigger a system call
  - syscall: 0x0f 0x05, sysenter: 0x0f 0x34
- syscall and sysenter are 2-byte instructions
- Specification for a jump destination address needs more than 2 bytes

Because of this issue, previous binary rewriting techniques
- could not ensure exhaustive hooking
- or, overwrite neighbour instructions

If we put ADDR, subsequent instructions are overwritten.
Jump to the overwritten part leads to unexpected behaviors.
On x86-64 CPUs, syscall and sysenter instructions trigger a system call.

- syscall: 0x0f 0x05
- sysenter: 0x0f 0x34

 syscall and sysenter are 2-byte instructions.

Specification for a jump destination address needs more than 2 bytes.

Because of this issue, previous binary rewriting techniques:

- could not ensure exhaustive hooking;
- or, overwrite neighbour instructions.

Because ADDR is bigger than 2 bytes, if we put ADDR, subsequent instructions are overwritten.

Jump to the overwritten part leads to unexpected behaviors.
Calling Convention

• How to invoke a system call
Calling Convention

• How to invoke a system call
  • A user-space program sets a system call number, predefined by the kernel, to the \texttt{rax} register
    • e.g., 0: read(), 1: write(), 2: open(), ...

Virtual Memory

\begin{verbatim}
0x0000
0x0001
0x0002
... 
0x0f
0x05
...
\end{verbatim}
Calling Convention

- How to invoke a system call
  - A user-space program sets a system call number, predefined by the kernel, to the rax register
    - e.g., 0: read(), 1: write(), 2: open(), ...
  - The user-space program executes syscall/sysenter
Calling Convention

- How to invoke a system call
  - A user-space program sets a system call number, predefined by the kernel, to the **rax register**
    - e.g., 0: read(), 1: write(), 2: open(), ...
  - The user-space program executes syscall/sysenter
    ---- the context is switched to the kernel ----
Calling Convention

• How to invoke a system call
  • A user-space program sets a system call number, predefined by the kernel, to the **rax register**
    • e.g., 0: read(), 1: write(), 2: open(), ...
  • The user-space program executes syscall/sysenter
    ---- the context is switched to the kernel ----
  • Kernel executes a system call specified through the system call number set to the **rax register**
Calling Convention

• How to invoke a system call
  • A user-space program sets a system call number, predefined by the kernel, to the **rax register**
    • e.g., 0: read(), 1: write(), 2: open(), ...
  • The user-space program executes syscall/sysenter
    ---- the context is switched to the kernel ----
  • Kernel executes a system call specified through the system call number set to the **rax register**
    • if the rax register has 0, the kernel executes **read()**
    • if the rax register has 1, the kernel executes **write()**
    • if the rax register has 2, the kernel executes **open()**
Calling Convention

• How to invoke a system call
  • A user-space program sets a system call number, predefined by the kernel, to the **rax register**
    • e.g., 0: read(), 1: write(), 2: open(), ...
  • The user-space program executes syscall/sysenter
    ---- the context is switched to the kernel ----
  • Kernel executes a system call specified through the system call number set to the **rax register**

**Point: Calling Convention**

When syscall/sysenter is executed, the rax register always has a system call number,
Calling Convention

• How to invoke a system call
  • A user-space program sets a system call number, predefined by the kernel, to the **rax register**
    • e.g., 0: read(), 1: write(), 2: open(), ...
  • The user-space program executes syscall/sysenter
    ---- the context is switched to the kernel ----
  • Kernel executes a system call specified through the system call number set to the **rax register**

**Point: Calling Convention**

When syscall/sysenter is executed, the rax register always has a system call number, which is 0 ~ around 500 (defined in the kernel)
zpoline

• zpoline replaces syscall/sysenter with callq *%rax

Virtual Memory

0x0000
0x0001
0x0002
... 

... 

user-defined hook function

... 

set syscall num to rax register

syscall 0x0f
0x05
... 

Point: Calling Convention

When syscall/sysenter is executed, the rax register always has a system call number, which is 0 ~ around 500 (defined in the kernel)
zpoline

- zpoline replaces syscall/sysenter with callq *%rax
- callq *%rax is a 2-byte instruction (0xff 0xd0)

Point: Calling Convention

When syscall/sysenter is executed, the rax register always has a system call number, which is 0 ~ around 500 (defined in the kernel)
zpoline replaces syscall/sysenter with callq *%rax
- callq *%rax is a 2-byte instruction (0xff 0xd0)
  - Neighbour instructions are not overwritten

Point: Calling Convention
When syscall/sysenter is executed, the rax register always has a system call number, which is 0 ~ around 500 (defined in the kernel)
zpoline

- zpoline replaces syscall/sysenter with callq *%rax
  - callq *%rax is a 2-byte instruction (0xff 0xd0)
    - Neighbour instructions are not overwritten
  - callq *%rax is an instruction to jump to the address stored in the rax register

**Point: Calling Convention**

When syscall/sysenter is executed, the rax register always has a system call number, which is 0 ~ around 500 (defined in the kernel)
zpoline

- zpoline replaces syscall/sysenter with callq *%rax
  - callq *%rax is a **2-byte** instruction (0xff 0xd0)
    - Neighbour instructions are not overwritten
  - callq *%rax is an instruction to jump to the address stored in the rax register

**Point: Calling Convention**

When syscall/sysenter is executed, the rax register always has a system call number, which is 0 ~ around 500 (defined in the kernel)
zpoline

- zpoline replaces syscall/sysenter with callq *%rax
  - callq *%rax is a 2-byte instruction (0xff 0xd0)
    - Neighbour instructions are not overwritten
  - callq *%rax is an instruction to jump to the address stored in the rax register

After the binary rewriting

Point: Calling Convention

When syscall/sysenter is executed, the rax register always has a system call number, which is 0 ~ around 500 (defined in the kernel)
zpoline

- zpoline replaces syscall/sysenter with callq *%rax
  - callq *%rax is a 2-byte instruction (0xff 0xd0)
    - Neighbour instructions are not overwritten
  - callq *%rax is an instruction to jump to the address stored in the rax register

**After the binary rewriting**

**Point: Calling Convention**

When syscall/sysenter callq *%rax is executed, the rax register always has a system call number, which is 0 ~ around 500 (defined in the kernel)
• zpoline replaces syscall/sysenter with callq *%rax
  • callq *%rax is a 2-byte instruction (0xff 0xd0)
    • Neighbour instructions are not overwritten
  • callq *%rax is an instruction to jump to the address stored in the rax register

After the binary rewriting

Point: Calling Convention

When syscall/sysenter callq *%rax is executed, the rax register always has a system call number, which is 0 ~ around 500 (defined in the kernel)
zpoline

- zpoline replaces syscall/sysenter with callq *%rax
  - callq *%rax is a 2-byte instruction (0xff 0xd0)
    - Neighbour instructions are not overwritten
  - callq *%rax is an instruction to jump to the address stored in the rax register

After the binary rewriting

Point: Calling Convention

When syscall/sysenter callq *%rax is executed, the rax register always has a system call number, which is 0 ~ around 500 (defined in the kernel)
**zpoline**

- zpoline replaces syscall/sysenter with callq *%rax
  - callq *%rax is a **2-byte** instruction (0xff 0xd0)
  - Neighbour instructions are not overwritten
  - callq *%rax is an instruction to jump to the address stored in the rax register

---

**After the binary rewriting**

**Point: Calling Convention**

When syscall/sysenter `callq *%rax` is executed, the rax register always has a system call number, which is 0 ~ around 500 (defined in the kernel)
zpoline

- zpoline replaces syscall/sysenter with callq *%rax
  - callq *%rax is a 2-byte instruction (0xff 0xd0)
    - Neighbour instructions are not overwritten
  - callq *%rax is an instruction to jump to the address stored in the rax register
  - replaced callq *%rax jumps to address 0~around 500

After the binary rewriting

Point: Calling Convention

When syscall/sysenter callq *%rax is executed, the rax register always has a system call number, which is 0 ~ around 500 (defined in the kernel)
zpoline replaces syscall/sysenter with callq *%rax
- callq *%rax is a 2-byte instruction (0xff 0xd0)
  - Neighbour instructions are not overwritten
- callq *%rax is an instruction to jump to the address stored in the rax register
- replaced callq *%rax jumps to address 0~around 500

Point: Calling Convention

When syscall/sysenter callq *%rax is executed, the rax register always has a system call number, which is 0 ~ around 500 (defined in the kernel)
zpoline replaces syscall/sysenter with callq *%rax

• callq *%rax is a 2-byte instruction (0xff 0xd0)
  • Neighbour instructions are not overwritten
• callq *%rax is an instruction to jump to the address stored in the rax register
• replaced callq *%rax jumps to address 0~around 500

address range, potentially replaced “callq *%rax” jumps to (N is the max syscall number)
zpoline replaces syscall/sysenter with callq *%rax

- callq *%rax is a **2-byte** instruction (0xff 0xd0)
  - Neighbour instructions are not overwritten
- callq *%rax is an instruction to jump to the address stored in the rax register
- replaced callq *%rax jumps to address 0~around 500

How to redirect to the user-defined hook function?
zpoline replaces syscall/sysenter with callq *%rax

- callq *%rax is a 2-byte instruction (0xff 0xd0)
  - Neighbour instructions are not overwritten
- callq *%rax is an instruction to jump to the address stored in the rax register
- replaced callq *%rax jumps to address 0~around 500

How to redirect to the user-defined hook function?

- zpoline instantiates trampoline code at address 0
zpoline replaces syscall/sysenter with callq *%rax
- callq *%rax is a 2-byte instruction (0xff 0xd0)
  - Neighbour instructions are not overwritten
- callq *%rax is an instruction to jump to the address stored in the rax register
- replaced callq *%rax jumps to address 0~around 500

How to redirect to the user-defined hook function?
- zpoline instantiates trampoline code at address 0
  - fills address range 0 to $N$ with nop (0x90)
zpoline replaces syscall/sysenter with callq *%rax
- callq *%rax is a 2-byte instruction (0xff 0xd0)
  - Neighbour instructions are not overwritten
- callq *%rax is an instruction to jump to the address stored in the rax register
- replaced callq *%rax jumps to address 0~around 500

How to redirect to the user-defined hook function?
- zpoline instantiates trampoline code at address 0
  - fills address range 0 to $N$ with nop (0x90)
zpoline replaces syscall/sysenter with callq *%rax

- callq *%rax is a 2-byte instruction (0xff 0xd0)
  - Neighbour instructions are not overwritten
- callq *%rax is an instruction to jump to the address stored in the rax register
- replaced callq *%rax jumps to address 0~around 500

How to redirect to the user-defined hook function?

- zpoline instantiates trampoline code at address 0
  - fills address range 0 to $N$ with nop (0x90)
    - puts code to jump to the hook function next to the last nop
zpoline replaces syscall/sysenter with callq *%rax

- callq *%rax is a \textbf{2-byte} instruction (0xff 0xd0)
  - Neighbour instructions are not overwritten
- callq *%rax is an instruction to jump to the address stored in the rax register
- replaced callq *%rax jumps to address 0~around 500

How to redirect to the user-defined hook function?

- zpoline instantiates trampoline code at address 0
  - fills address range 0 to $N$ with nop (0x90)
  - puts code to jump to the hook function next to the last nop

We could reach the user-defined hook function!
zpoline replaces syscall/sysenter with callq *%rax
- callq *%rax is a **2-byte** instruction (0xff 0xd0)
  - Neighbour instructions are not overwritten
- callq *%rax is an instruction to jump to the address stored in the rax register
- replaced callq *%rax jumps to address 0~around 500

How to redirect to the user-defined hook function?
- zpoline instantiates trampoline code at address 0
  - fills address range 0 to \( N \) with nop (0x90)
    - puts code to jump to the hook function next to the last nop

We could reach the user-defined hook function!
NULL Access Termination

- A buggy program may access NULL (address 0)
NULL Access Termination

- A buggy program may access NULL (address 0)
NULL Access Termination

• A buggy program may access NULL (address 0)
NULL Access Termination

- A buggy program may access NULL (address 0)
  - In principle, NULL access has to be terminated
NULL Access Termination

- A buggy program may access NULL (address 0)
  - In principle, NULL access has to be terminated
  - Normally, a page fault happens because no physical memory is mapped to virtual address 0
NULL Access Termination

- A buggy program may access NULL (address 0)
  - In principle, NULL access has to be terminated
  - Normally, a page fault happens because no physical memory is mapped to virtual address 0
  - zpoline uses virtual address 0, therefore, the page fault does not happen
NULL Access Termination

• A buggy program may access NULL (address 0)
  • In principle, NULL access has to be terminated
  • Normally, a page fault happens because no physical memory is mapped to virtual address 0
  • zpoline uses virtual address 0, therefore, the page fault does not happen

• The buggy program continues to run
NULL Access Termination

- A buggy program may access NULL (address 0)
  - In principle, NULL access has to be terminated
  - Normally, a page fault happens because no physical memory is mapped to virtual address 0
  - zpoline uses virtual address 0, therefore, the page fault does not happen
  - The buggy program continues to run

How can we detect and terminate a buggy NULL access?
NULL Access Termination

- Memory access: read / write / execute
NULL Access Termination

• Memory access: read / write / execute
• Solution
  • read/write: configure the trampoline code as XOM
NULL Access Termination

- Memory access: read / write / execute
- Solution
  - read/write: configure the trampoline code as XOM
    - read/write access to the trampoline code causes a fault
      - This can be done by mprotect() system call
NULL Access Termination

- Memory access: read / write / execute
- Solution
  - read/write: configure the trampoline code as XOM
    - read/write access to the trampoline code causes a fault
      - This can be done by mprotect() system call
  - execute: check the caller address
NULL Access Termination

• Memory access: read / write / execute
• Solution
  • read/write: configure the trampoline code as XOM
    • read/write access to the trampoline code causes a fault
      • This can be done by mprotect() system call
  • execute: check the caller address
    1. during the binary rewriting phase, we collect the addresses of replaced syscall/sysenter

Virtual Memory

<table>
<thead>
<tr>
<th>0x0000</th>
<th>0x0001</th>
<th>0x0002</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

user-defined hook function

set syscall num to rax register

callq *%rax

Bug
Access NULL
NULL Access Termination

- Memory access: read / write / execute
- Solution
  - read/write: configure the trampoline code as XOM
    - read/write access to the trampoline code causes a fault
      - This can be done by mprotect() system call
  - execute: check the caller address
    1. during the binary rewriting phase, we collect the addresses of replaced syscall/sysenter

During binary rewriting phase ...
NULL Access Termination

- Memory access: read / write / execute
- Solution
  - read/write: configure the trampoline code as XOM
    - read/write access to the trampoline code causes a fault
      - This can be done by mprotect() system call
  - execute: check the caller address
    1. during the binary rewriting phase, we collect the addresses of replaced syscall/sysenter

During binary rewriting phase ...

List of replaced addresses: [...]
NULL Access Termination

- Memory access: read / write / execute
- Solution
  - read/write: configure the trampoline code as XOM
    - read/write access to the trampoline code causes a fault
      - This can be done by mprotect() system call
  - execute: check the caller address
    1. during the binary rewriting phase, we collect the addresses of replaced syscall/sysenter

During binary rewriting phase ...

List of replaced addresses: [...]

![Virtual Memory Diagram]
NULL Access Termination

• Memory access: read / write / execute
• Solution
  • read/write: configure the trampoline code as XOM
    • read/write access to the trampoline code causes a fault
      • This can be done by mprotect() system call
  • execute: check the caller address
    1. during the binary rewriting phase, we collect the addresses of replaced syscall/sysenter

During binary rewriting phase ... 

List of replaced addresses : [A, ...]
NULL Access Termination

- Memory access: read / write / execute
- Solution
  - read/write: configure the trampoline code as XOM
    - read/write access to the trampoline code causes a fault
      - This can be done by mprotect() system call
  - execute: check the caller address
    1. during the binary rewriting phase, we collect the addresses of replaced syscall/sysenter
    2. at runtime, in the hook function, we check if the caller is one of the replaced addresses

List of replaced addresses: \([A, ...]\)
NULL Access Termination

At runtime ...
- Memory access: read / write / execute
- Solution
  - read/write: configure the trampoline code as XOM
    - read/write access to the trampoline code causes a fault
      - This can be done by mprotect() system call
  - execute: check the caller address
    1. during the binary rewriting phase, we collect the addresses of replaced syscall/sysenter
    2. at runtime, in the hook function, we check if the caller is one of the replaced addresses

List of replaced addresses: \([A, \ldots]\)
NULL Access Termination

At runtime ...

• Memory access: read / write / execute

• Solution
  • read/write: configure the trampoline code as XOM
  • read/write access to the trampoline code causes a fault
    • This can be done by mprotect() system call
  • execute: check the caller address
    1. during the binary rewriting phase, we collect the addresses of replaced syscall/sysenter
    2. at runtime, in the hook function, we check if the caller is one of the replaced addresses

List of replaced addresses : \([A, ...)\)

The caller address is \(A\)
\(A\) is in the list, so this is a valid access
NULL Access Termination

At runtime ...

• Memory access: read / write / execute

• Solution
  • read/write: configure the trampoline code as XOM
    • read/write access to the trampoline code causes a fault
      • This can be done by mprotect() system call
  • execute: check the caller address
    1. during the binary rewriting phase, we collect the addresses of replaced syscall/sysenter
    2. at runtime, in the hook function, we check if the caller is one of the replaced addresses

List of replaced addresses: \([A, \ldots]\)
NULL Access Termination

At runtime ...

- Memory access: read / write / execute
- Solution
  - read/write: configure the trampoline code as XOM
    - read/write access to the trampoline code causes a fault
      - This can be done by mprotect() system call
  - execute: check the caller address
    1. during the binary rewriting phase, we collect the addresses of replaced syscall/sysenter
    2. at runtime, in the hook function, we check if the caller is one of the replaced addresses

List of replaced addresses: \([A, \ldots]\)
NULL Access Termination

At runtime ...

- Memory access: read / write / execute
- Solution
  - read/write: configure the trampoline code as XOM
  - read/write access to the trampoline code causes a fault
    - This can be done by mprotect() system call
  - execute: check the caller address
    1. during the binary rewriting phase, we collect the addresses of replaced syscall/sysenter
    2. at runtime, in the hook function, we check if the caller is one of the replaced addresses

List of replaced addresses: \([A, \ldots]\)

The caller address is \(B\)

\(B\) is NOT in the list, so this is an invalid access
NULL Access Termination

At runtime...

- Memory access: read / write / execute
- Solution
  - read/write: configure the trampoline code as XOM
  - read/write access to the trampoline code causes a fault
  - This can be done by `mprotect()` system call
  - execute: check the caller address
    1. during the binary rewriting phase, we collect the addresses of replaced `syscall/sysenter`
    2. at runtime, in the hook function, we check if the caller is one of the replaced addresses

List of replaced addresses: [A, ...]

The caller address is B, B is NOT in the list, so this is an invalid access stop the program
NULL Access Termination

At runtime ...  
- Memory access:
  - read / write / execute
- Solution:
  - read/write: configure the trampoline code as XOM
  - read/write access to the trampoline code causes a fault
    - This can be done by `mprotect()` system call
  - execute: check the caller address
    1. during the binary rewriting phase, we collect the addresses of replaced syscall/sysenter
    2. at runtime, in the hook function, we check if the caller is one of the replaced addresses
      - Current prototype uses bitmap to implement this check

List of replaced addresses: \([A, ...]\)
System Call Hook Overhead

• Time to hook `getpid()` and return a dummy value

<table>
<thead>
<tr>
<th>Mechanism</th>
<th>Time [ns]</th>
</tr>
</thead>
<tbody>
<tr>
<td>ptrace</td>
<td>31201</td>
</tr>
<tr>
<td>int3 signaling</td>
<td>1342</td>
</tr>
<tr>
<td>SUD</td>
<td>1156</td>
</tr>
<tr>
<td>zpoline</td>
<td>41</td>
</tr>
<tr>
<td>LD_PRELOAD</td>
<td>6</td>
</tr>
</tbody>
</table>
System Call Hook Overhead

- Time to hook getpid() and return a dummy value

<table>
<thead>
<tr>
<th>Mechanism</th>
<th>Time [ns]</th>
<th>Improvement</th>
</tr>
</thead>
<tbody>
<tr>
<td>ptrace</td>
<td>31201</td>
<td></td>
</tr>
<tr>
<td>int3 signaling</td>
<td>1342</td>
<td>32.7x</td>
</tr>
<tr>
<td>SUD</td>
<td>1156</td>
<td>28.1x</td>
</tr>
<tr>
<td>zpoline</td>
<td>41</td>
<td></td>
</tr>
<tr>
<td>LD_PRELOAD</td>
<td>6</td>
<td></td>
</tr>
</tbody>
</table>
System Call Hook Overhead

- Time to hook `getpid()` and return a dummy value

<table>
<thead>
<tr>
<th>Mechanism</th>
<th>Time [ns]</th>
</tr>
</thead>
<tbody>
<tr>
<td>ptrace</td>
<td>31201</td>
</tr>
<tr>
<td>int3 signaling</td>
<td>1342</td>
</tr>
<tr>
<td>SUD</td>
<td>1156</td>
</tr>
<tr>
<td>zpoline</td>
<td>NULL exec check: 1 ns out of 41</td>
</tr>
<tr>
<td>LD_PRELOAD</td>
<td>6</td>
</tr>
</tbody>
</table>
System Call Hook Overhead

- Time to hook getpid() and return a dummy value

<table>
<thead>
<tr>
<th>Mechanism</th>
<th>Time [ns]</th>
</tr>
</thead>
<tbody>
<tr>
<td>ptrace</td>
<td>31201</td>
</tr>
<tr>
<td>int3 signaling</td>
<td>1342</td>
</tr>
<tr>
<td>SUD</td>
<td>1156</td>
</tr>
<tr>
<td>zpoline</td>
<td>41</td>
</tr>
<tr>
<td>LD_PRELOAD</td>
<td>6</td>
</tr>
</tbody>
</table>

Virtual Memory

- User-defined hook function
- Set syscall num to rax register
- Callq *%rax
- +35ns overhead
System Call Hook Overhead

- Time to hook getpid() and return a dummy value

<table>
<thead>
<tr>
<th>Mechanism</th>
<th>Time [ns]</th>
</tr>
</thead>
<tbody>
<tr>
<td>ptrace</td>
<td>31201</td>
</tr>
<tr>
<td>int3 signaling</td>
<td>1342</td>
</tr>
<tr>
<td>SUD</td>
<td>1156</td>
</tr>
<tr>
<td>zpoline</td>
<td>41</td>
</tr>
<tr>
<td>LD_PRELOAD</td>
<td>6</td>
</tr>
</tbody>
</table>
Application Performance

- We **transparency**ly apply lwIP + DPDK to an application using different system call hook mechanisms.
Application Performance

• We **transparently** apply lwIP + DPDK to an application using different system call hook mechanisms

Simple HTTP server

lwIP + DPDK

100 Gbps
Application Performance

• We **transparently** apply lwIP + DPDK to an application using different system call hook mechanisms

Simple HTTP server

```
syscall hook

lwIP + DPDK
```

ptrace, int3, SUD, zpoline, LD_PRELOAD

100 Gbps
Application Performance

- We **transparently** apply lwIP + DPDK to an application using different system call hook mechanisms.

**Simple HTTP server**

- syscall hook
- lwIP + DPDK

**ptrace, int3, SUD, zpoline, LD_PRELOAD**

**wrk: benchmark client**

- fetch 64B content

100 Gbps
Application Performance

- We **transparently** apply lwIP + DPDK to an application using different system call hook mechanisms.

Simple HTTP server

```
syscall hook
```

LwIP + DPDK

```
ptrace, int3, SUD, zpoline, LD_PRELOAD
```

wrk: benchmark client
fetch 64B content

100 Gbps

Throughput [M reqs/sec]

Linux
Application Performance

- We **transparently** apply lwIP + DPDK to an application using different system call hook mechanisms.

Simple HTTP server

**syscall hook**

lwIP + DPDK

ptrace, int3, SUD, zpoline, LD_PRELOAD

wrk: benchmark client
fetch 64B content

100 Gbps

Compared to LD_PRELOAD

- ptrace: 1.1%
- int3 signaling: 14.7%
- SUD: 17.0%
- zpoline: 87.3%
- LD_PRELOAD: 1.1%
Application Performance

• We **transparently** apply lwIP + DPDK to an application using different system call hook mechanisms

Redis

syscall hook

lwIP + DPDK

Redis

pttrace, int3, SUD, zpoline, LD_PRELOAD

redis-benchmark

GET 100%

100 Gbps
Application Performance

- We **transparently** apply lwIP + DPDK to an application using different system call hook mechanisms.

Redis

- syscall hook

lwIP + DPDK

redis-benchmark

GET 100%

Throughput [K reqs/sec]

- ptrace, int3, SUD, zpoline, LD_PRELOAD

100 Gbps

Linux
Application Performance

- We **transparently** apply lwIP + DPDK to an application using different system call hook mechanisms.

---

**Redis**

```plaintext
syscall hook
```

**lwIP + DPDK**

**Redis-benchmark GET 100%**

---

**Throughput [K reqs/sec]**

```
<table>
<thead>
<tr>
<th>Method</th>
<th>Throughput</th>
</tr>
</thead>
<tbody>
<tr>
<td>ptrace, int3, SUD, zpoline, LD_PRELOAD</td>
<td>94.8%</td>
</tr>
<tr>
<td>Linux</td>
<td>15.0%</td>
</tr>
<tr>
<td>SUD</td>
<td>15.0%</td>
</tr>
<tr>
<td>zpoline</td>
<td>17.7%</td>
</tr>
<tr>
<td>LD_PRELOAD</td>
<td>1.2%</td>
</tr>
</tbody>
</table>
```

---

**100 Gbps**
Summary

• zpoline: a system call hook mechanism for x86-64 CPUs
  • based on binary rewriting
    • replaces syscall/sysenter with callq *%rax
    • instantiates the trampoline code at virtual address 0 (zero)
  • free from the drawbacks of the previous mechanisms
  • keeps the performance benefit of user-space OS subsystems

• Source code: [https://github.com/yasukata/zpoline](https://github.com/yasukata/zpoline)
  • since October 2021
Speeding up the Trampoline Code

• Inspired from USENIX ATC’23 reviewers who suggested to employ a one-byte short jump instruction for speeding up
  • Put it on the addresses corresponding to obsolete system calls

• Optimization: repeat 0xeb 0x6a 0x90 instead of nops
  • Hook overhead reduction from 41 ns to 10 ns

Syscall number:          3 x n + 0              3 x n + 1              3 x n + 2
                      jmp 0x6a                      jmp 0x6a                      jmp 0x6a
                      nop                      push 0x90                      nop
                      jmp 0x6a                      jmp 0x6a                      jmp 0x6a
                      nop                      nop                      nop

We pop 0x90 in the hook function