Donky: Domain Keys –
Efficient In-Process Isolation for RISC-V and x86

David Schrammel, Samuel Weiser, Stefan Steinegger, Martin Schwarzl,
Michael Schwarz, Stefan Mangard, Daniel Gruss

IAIK – Graz University of Technology
Motivation

- Modern software incredibly complex
- Often closed-source, 3rd-party libraries with potential unknown vulnerabilities
- Web-Browsers:
  - Handle sensitive information
  - But also run untrusted code
  - Dozens of libraries for media decoding, font shaping, ...
  - Top 2 applications #CVEs: Firefox and Chrome\(^1\)
- Ongoing effort:
  - Rewrite libraries in safe languages
  - Split browser into multiple processes
  - Engineering effort or runtime overhead
- Need efficient sandboxing

\(^1\)https://www.cvedetails.com/top-50-products.php
Software sandboxing

- Kernel-based:
  - Process Isolation: high security, high context-switch cost
  - Kernel-based in-process isolation often require heavy kernel modifications

- Userspace:
  - SFI (e.g., NativeClient)
  - PKU-based (e.g., ERIM)
  - typically fast context-switches but runtime overhead
• Pages tagged with a “protection key"
• Key stored in Page Table Entry
• Intel MPK: 4-bit keys → 16 keys

Virtual address space:

<table>
<thead>
<tr>
<th>Address</th>
<th>PTEs</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xA000</td>
<td><img src="#" alt="PTE1" /></td>
</tr>
<tr>
<td>0xB000</td>
<td><img src="#" alt="PTE2" /></td>
</tr>
<tr>
<td>0xC000</td>
<td><img src="#" alt="PTE3" /></td>
</tr>
<tr>
<td>0xD000</td>
<td></td>
</tr>
</tbody>
</table>

PTEs:

<table>
<thead>
<tr>
<th></th>
<th>ph. addr.</th>
<th>r w x ...</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="#" alt="PTE1" /></td>
<td></td>
<td></td>
</tr>
<tr>
<td><img src="#" alt="PTE2" /></td>
<td></td>
<td></td>
</tr>
<tr>
<td><img src="#" alt="PTE3" /></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

PKRU:

Schrammel et al. — IAIK – Graz University of Technology
• Pages tagged with a “protection key”
• Key stored in Page Table Entry
• Intel MPK: 4-bit keys → 16 keys
• Key-permissions in policy register (e.g., “PKRU”)
• Allows to quickly change memory permissions (from userspace)
PKU-based sandboxing

- How to use MPK for in-process isolation?
  - Only safe call gates modify PKRU
  - No unsafe writes (WRPKRU) to the register exist.
    → Binary scanning/rewriting, $W \oplus X$
  - Limit syscalls that bypass/circumvent PKRU
    → Kernel module, seccomp, ptrace, ...

- PKU-based sandboxing works (e.g., ERIM, Hodor)

- Open questions:
  - Can we sandbox self-modifying code (e.g., JavaScript JIT compiler)?
  - Can we have PKU-based sandboxing without binary scanning?
• Design PKU from the ground up for RISC-V with in-process isolation in mind

• Repurpose *RISC-V Extension for User-Level Interrupts*
• Design PKU from the ground up for RISC-V with in-process isolation in mind
• Repurpose *RISC-V Extension for User-Level Interrupts*

• Modification: Limit register access to the interrupt handler itself.
• Trusted user-space exception handler ("Monitor")
• Monitor intercepts syscalls directly in user-space
PKU for RISC-V

- Design PKU from the ground up for RISC-V with in-process isolation in mind
- Repurpose *RISC-V Extension for User-Level Interrupts*

- Modification: Limit register access to the interrupt handler itself.
- Trusted user-space exception handler (“Monitor”)
- Monitor intercepts syscalls directly in user-space

- RISC-V PTEs allows up to 10-bit keys (1024 domains)
- PKU policy register
  - 4 key-slots with read/write permissions.
  - PKU policy register writable only from monitor
Concept Overview

Domain A: 

Userspace application

Monitor

Policy register:

OS

Vault:

Malicious code

/Others

Schrammel et al. — IAIK – Graz University of Technology
Schrammel et al. — IAIK – Graz University of Technology
Concept Overview

Domain A: Vault: Malicious:

Vault: Malicious code

Userspace application

Policy register:

Syscall

OS
Domain A: \( P_2 \)

Vault: \( P_{2R}, P_3, P_5 \)

Malicious: \( P_4, P_5 \)

Userspace application

Domain A

Vault

Malicious code

Shared

Policy register:

\( P_1, P_5, P_4 \)

Syscall

OS
Domain Transitions: dcalls

```c
wrapper
{
    dcall();
    ...
}
```

- Load monitor stack
- Security checks
- Stack create
- Policy update
- Stack switch
- Jump to domain

Access policy:

P1 P1 P2

sp →

Domain A

- dcall();
- ...

Monitor

Domain B

- dcall()
  - }

π2

π1

π3
Domain Transitions: dcalls

```
sp → __________________________
     |                              |
     |                              |
     |                              |
     |                              |
     |                              |
     +-----------------------------+

Domain A

dcall();
...

wrapper

Monitor

Domain B

dcall()
{
...
}

Access policy:

1  2  3  4

 pyl pyl pyl pyl

• load monitor stack
• security checks
• stack create
• policy update
• stack switch
• jmp to domain

sp → saved regs
sp → sp
sp → sp
sp → sp
sp → saved regs

Schrammel et al. — IAIK – Graz University of Technology
Domain Transitions: dcalls

Domain A

Domain B

Monitor

Access policy:

```c
wrapper

void dcall()
{
    ...
}
```

- load monitor stack
- security checks
- stack create
- policy update
- stack switch
- jmp to domain

(sp → saved regs)

(ὑ1, ὑ2, ὑ3, ὑ4)

Schrammel et al. — IAIK – Graz University of Technology
Domain Transitions: dcalls

Domain A

```
dcall();
...
```

Monitor

```
wrapper
dcall()
{
...
}
```

Domain B

```
dcall()
{
...
}
```

Access policy:

- \( \rho_1 \)
- \( \rho_2 \)
- \( \rho_3 \)
- \( \rho_2 \)

sp \rightarrow \text{saved regs}

- load monitor stack
- security checks
- stack create
- policy update
- stack switch
- jmp to domain
Domain Transitions: dcalls

```
wrapper
dcall()
{
  ...
}
```

- load monitor stack
- security checks
- stack create
- policy update
- stack switch
- jmp to domain

Access policy:

[Access permissions are shown with icons representing different access levels.]

Schrammel et al. — IAIK – Graz University of Technology
Domain Transitions: dcalls

Monitor
- load monitor stack
- security checks

Access policy:

Domain A
- dcall();
- ...
- wrapper

Domain B
- dcall()
- {
- ...
- }

Saved regs

sp → saved regs

Domain A

Domain B

Monitor
Domain Transitions: dcalls

**Domain A**
- `dcall();`
- `...`

**Monitor**
- `sp → saved regs`
- Load monitor stack
- Security checks
- Stack create
- Policy update
- Stack switch
- Jmp to domain

**Domain B**
- `dcall() {
  ...
  }

**Access policy:**

- Policy 1
- Policy 2
- Policy 3
- Policy 4

Schrammel et al. — IAIK – Graz University of Technology
Domain Transitions: dcalls

- Domain A
  - dcall();
  - \( \text{wrapper} \)

- Monitor
  - load monitor stack
  - security checks
  - stack create
  - policy update

- Domain B
  - dcall()

Access policy:

\[ p_2 \rightarrow p_1 \rightarrow p_3 \]
Domain Transitions: dcalls

Monitor
- load monitor stack
- security checks
- stack create
- policy update
- stack switch

Access policy:

Domain A
- dcall();
- ...

Domain B
- dcall()
- }

saved regs

sp → saved regs

p₂

Domain A

Monitor

p₁

p₃

Domain B

wrapper
Domain Transitions: dcalls

```c
void dcall()
{
    // Load monitor stack
    // Security checks
    // Stack create
    // Policy update
    // Stack switch
    // Jump to domain

    // Access policy:
    // P P P P
}
```
Domain Transitions: dcalls

Domain A

```
dcall();
...
```

Monitor

- load monitor stack
- security checks
- stack create
- policy update
- stack switch
- jmp to domain

Access policy:

Domain B

```
dcall()
{
...
}
```

Diagram showing the transition from Domain A to Domain B through a monitor with steps for saving registers and access policy.
Domain Transitions: dcall

- Load monitor stack
- Security checks
- Stack create
- Policy update
- Stack switch
- Jump to domain

Access policy:
- P
- P
- P
- P
- P

Schrammel et al. — IAIK – Graz University of Technology
Domain Transitions: dcall

- dcall();
- ...
- wrapper

Monitor
- load monitor stack
- security checks
- stack create
- policy update
- stack switch
- jmp to domain

Access policy:
- $\mathcal{P}_1$
- $\mathcal{P}_2$
- $\mathcal{P}_3$

Domain A
- dcall;

Domain B
- dcall();
- ...
Domain Transitions: dcalls

```cpp
wrapper
dcall();
...
```

```cpp
...}
```

- load monitor stack
- security checks
- stack create
- policy update
- stack switch
- jmp to domain

Access policy:

- \( \varphi_1 \)
- \( \varphi_2 \)
- \( \varphi_3 \)
- \( \varphi_4 \)
Domain Transitions: dcalls

- dcall();
- ...

Monitor:
- load monitor stack
- security checks
- stack create
- policy update
- stack switch
- jmp to domain

Access policy:
\[ \mathcal{P}_1 \]

Domain A

\[ \mathcal{P}_2 \]

Domain B

\[ \mathcal{P}_3 \]
• Hardware-Software co-design

• Small hardware extension for RISC-V
  • Based on RISC-V \textit{N extension} – “Standard Extension for User-Level Interrupts”
  • Implemented on RISC-V CPU Ariane/CVA6\textsuperscript{2}

• Software library
  • API for managing domains/keys/transitions
  • Wraps standard library functions (memory management, threads, signals)
  • Compatible with Intel MPK

\textsuperscript{2}https://github.com/openhwgroup/cva6
- Evaluated on a RISC-V CPU and CPUs with Intel MPK
- Domain transition overhead
  - 0.2–1.2x the time of a simple syscall
  - 16–116x faster than process context switches (process-based isolation)
Evaluation

- Evaluated on a RISC-V CPU and CPUs with Intel MPK
- Domain transition overhead
  - 0.2–1.2x the time of a simple syscall
  - 16–116x faster than process context switches (process-based isolation)
- SPEC CPU 2017: ≈0.1% overhead
- Mbed TLS
  - 1KiB block: 0–15% overhead (across all cryptographic functions)
  - Poly1305, 16 bytes:
    - Donky: 3–4.7x slower
    - Process isolation: 42–118x slower
- Isolate Google’s JavaScript engine “V8”: 0–2% overhead
Summary

- Efficient and secure in-process isolation
- Domain switches and syscall filtering entirely in userspace
- Zero overhead within a domain & small switching overhead
- No binary scanning, $W\oplus X$, or CFI
- Support self-modifying code (JIT compiler)
- Configurable trust relationships
- Up to 1024 domains/sandboxes

- Open source software and hardware implementation\(^3\)

\(^3\)https://github.com/IAIK/Donky
Donky: Domain Keys – Efficient In-Process Isolation for RISC-V and x86

David Schrammel, Samuel Weiser, Stefan Steinegger, Martin Schwarzl, Michael Schwarz, Stefan Mangard, Daniel Gruss

IAIK – Graz University of Technology