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

- 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

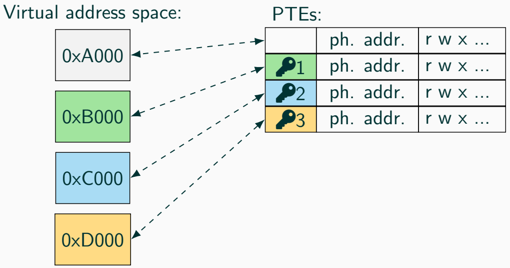
[1] https://www.cvedetails.com/top-50-products.php

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

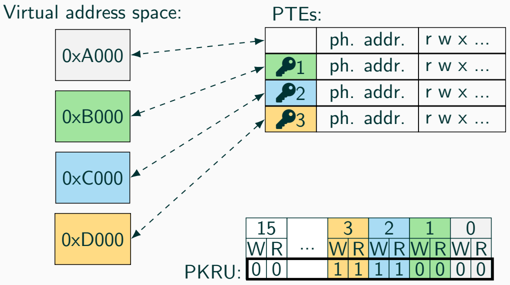
PKU: Memory Protection Keys for Userspace

- Pages tagged with a “protection key”
- Key stored in Page Table Entry
- Intel MPK: 4-bit keys $\rightarrow$ 16 keys



PKU: Memory Protection Keys for Userspace

- 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⊕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*

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

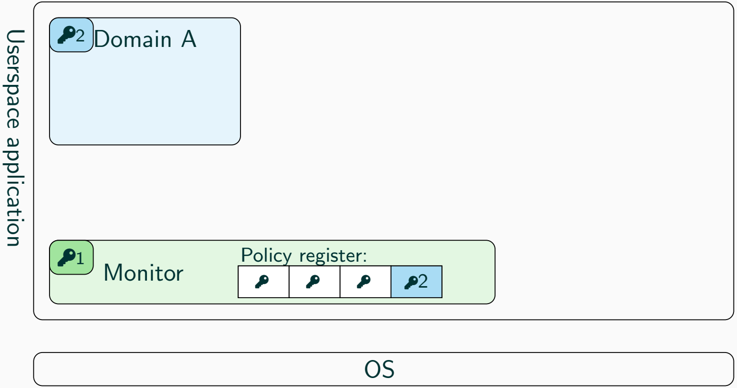
PKU for RISC-V

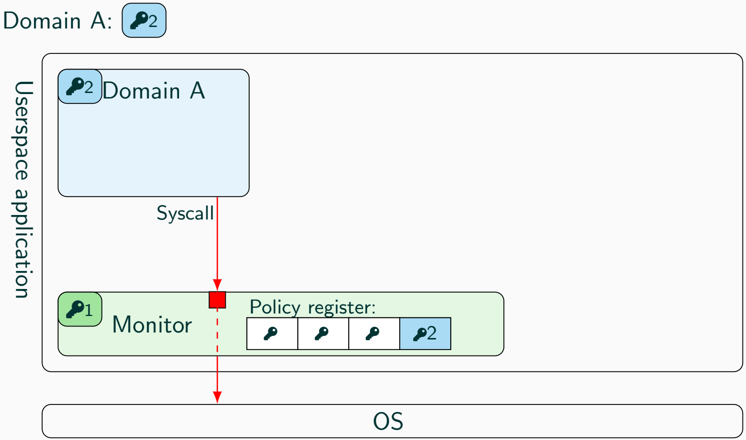
- Design PKU from the ground up for RISC-V
  with in-process isolation in mind
- Repurpose *RISC-V Extension for User-Level Interrupts*

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

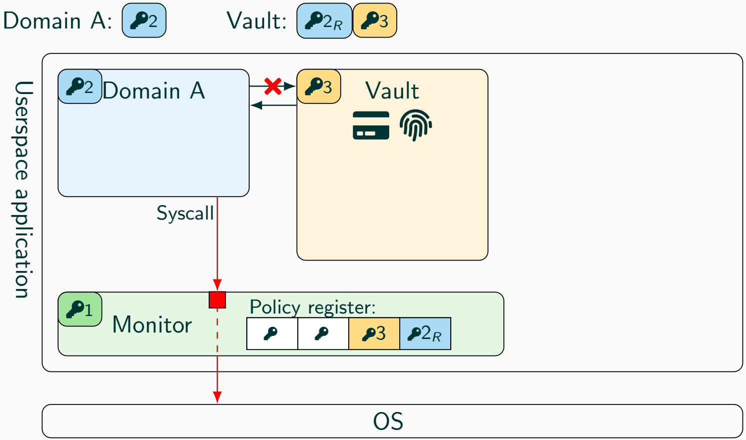
- RISC-V PTEs allows up to 10-bit keys (1024 domains)
- PKU policy register
  - 4 key-slots with read/write permissions.
  - PKU policy register writable only from monitor

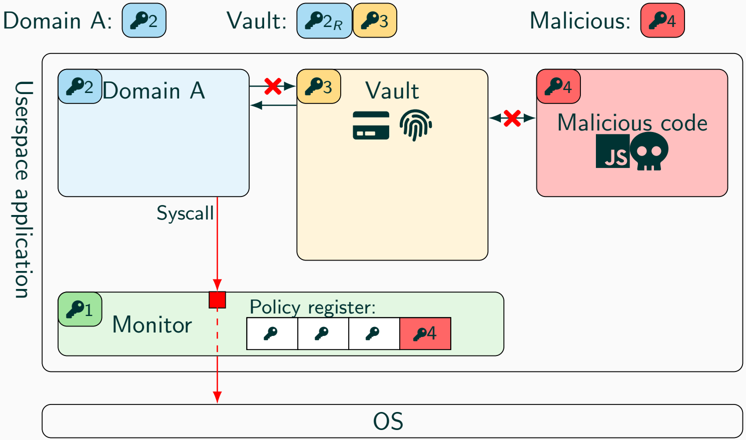
Domain A: 2


Userspace application
2
Domain A
1
Monitor
Policy register:
2
OS

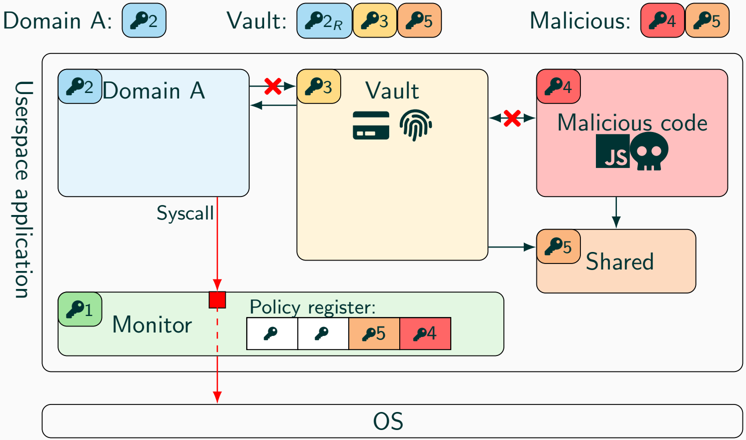


Domain A: 2
Userspace application
2 Domain A
Syscall
1 Monitor
Policy register:
2
OS

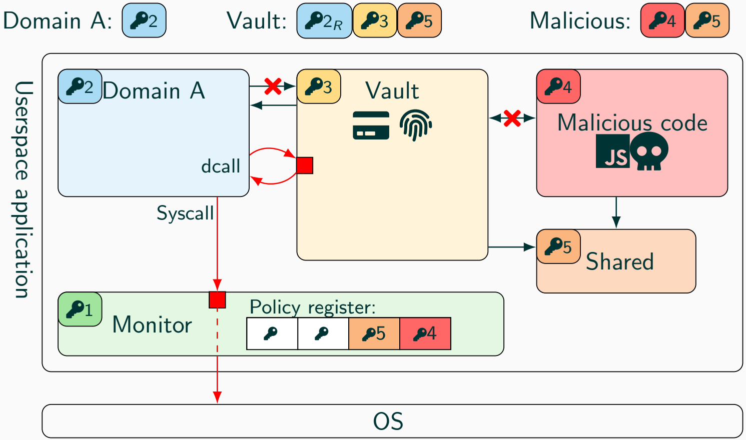

Domain A: 2
Vault: 2_R 3
Userspace application
2 Domain A
3 Vault
Syscall
1 Monitor
Policy register:
3 2_R
OS

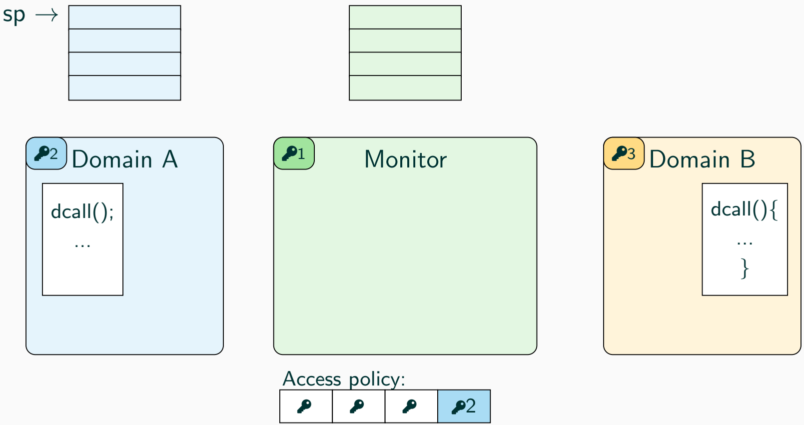

Domain A: 2
Vault: 2R 3
Malicious: 4
Userspace application
2 Domain A
3 Vault
4 Malicious code
JS
Syscall
1 Monitor
Policy register:
4
OS

Concept Overview

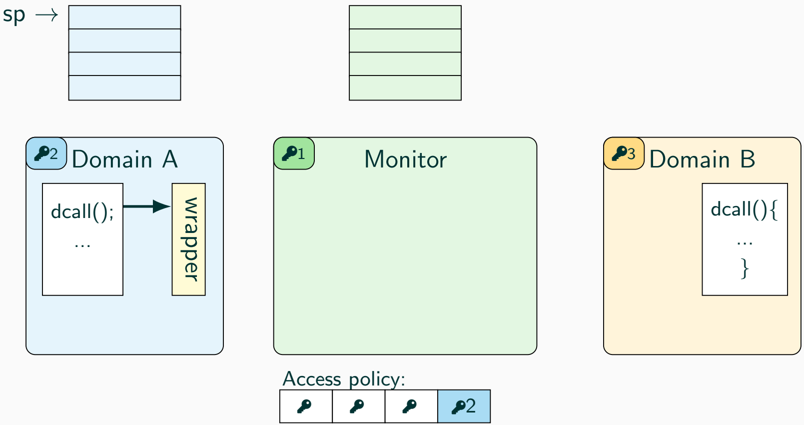


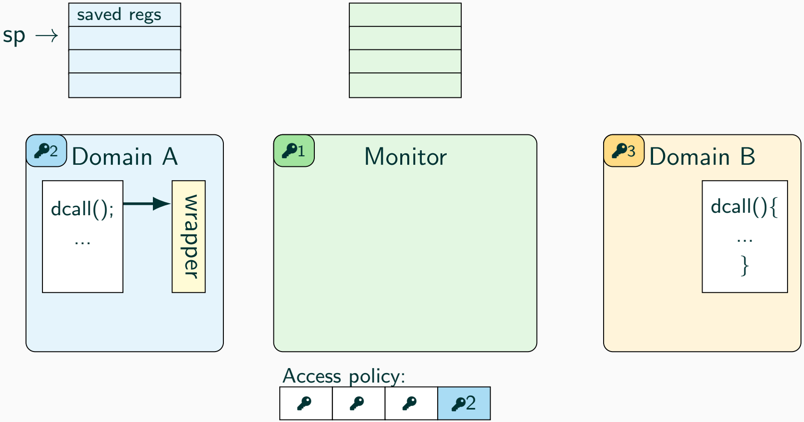
Concept Overview




sp →
Domain A
2
dcall();
...
Monitor
1
Domain B
3
dcall(){
...
}
Access policy:
2

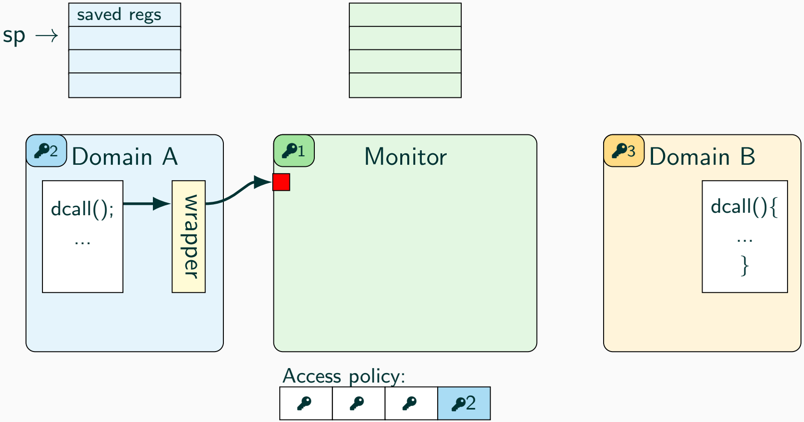
Domain Transitions: dcalls

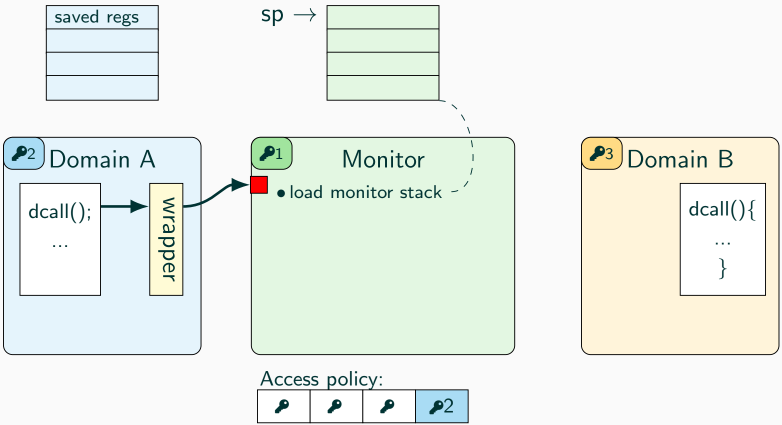




sp →
saved regs
2 Domain A
dcall();
...
wrapper
1 Monitor
3 Domain B
dcall(){
...
}
Access policy:
2

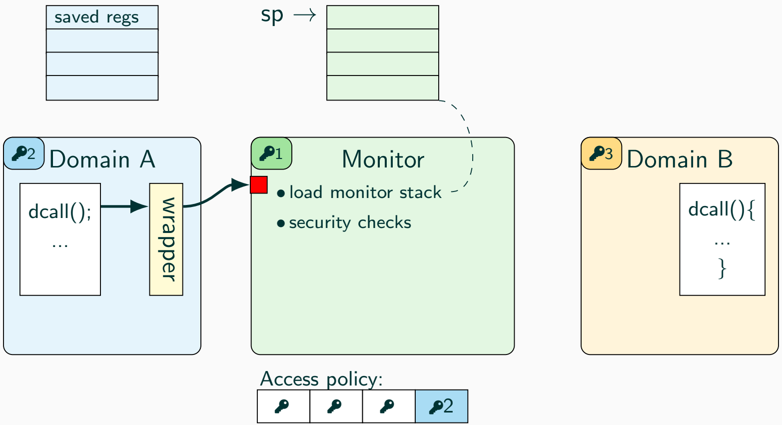
Domain Transitions: dcalls

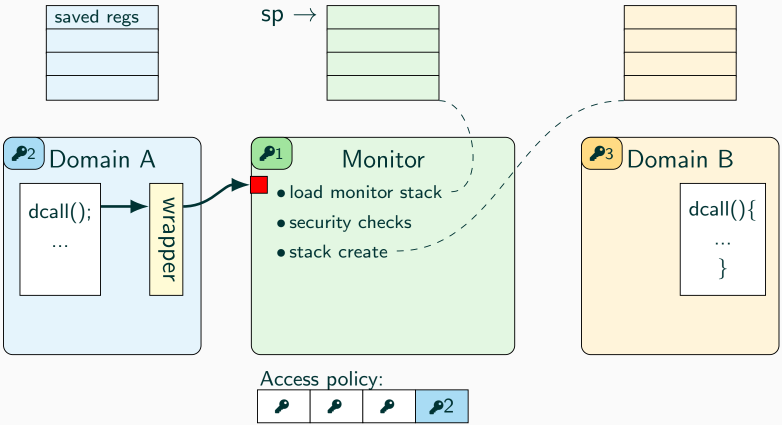




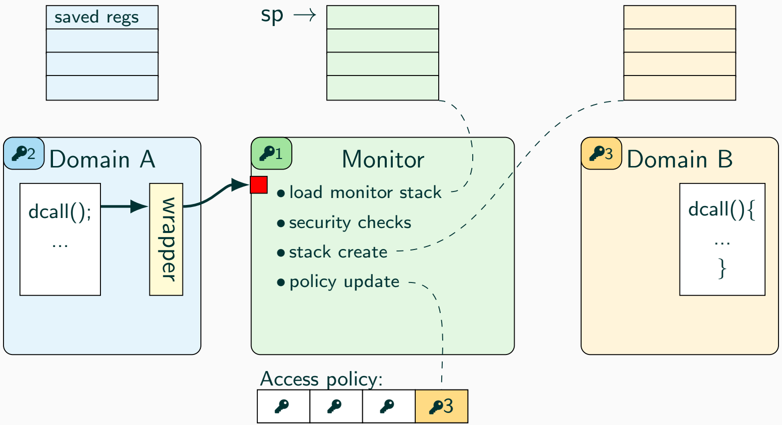
saved regs
sp →
2 Domain A
dcall();
...
wrapper
1 Monitor
• load monitor stack
3 Domain B
dcall(){
...
}
Access policy:
2

Domain Transitions: dcalls

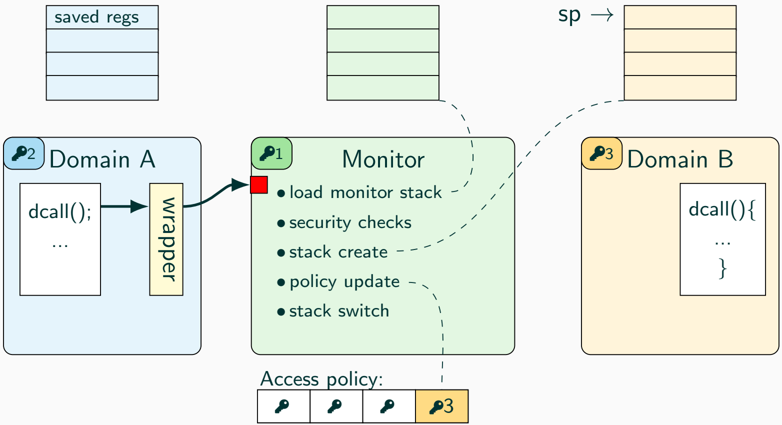




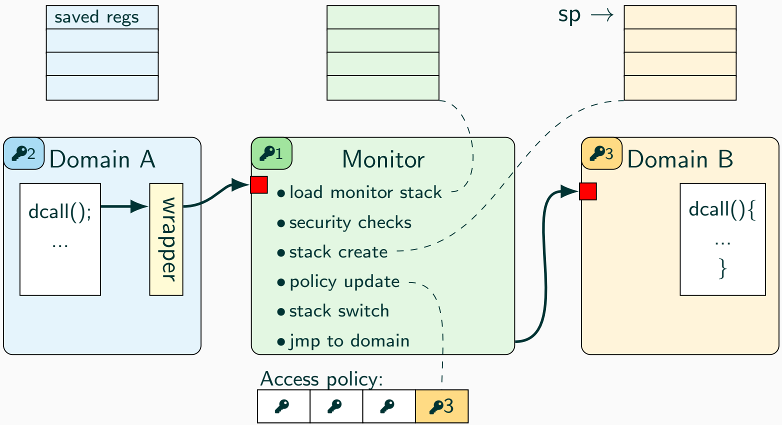
saved regs
sp →
Domain A
dcall();
...
wrapper
Monitor
• load monitor stack
• security checks
• stack create
Domain B
dcall(){
...
}
2
1
3
Access policy:
2


saved regs
sp →
Domain A
dcall();
...
wrapper
Monitor
load monitor stack
security checks
stack create
policy update
Domain B
dcall(){
...
}
Access policy:
2
1
3
3

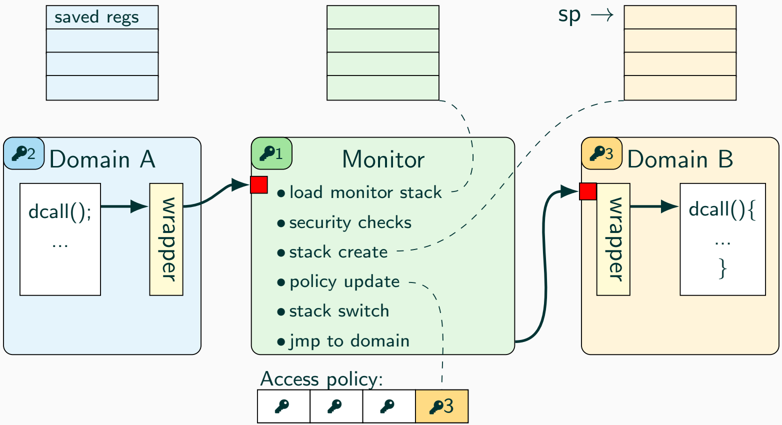
Domain Transitions: dcalls

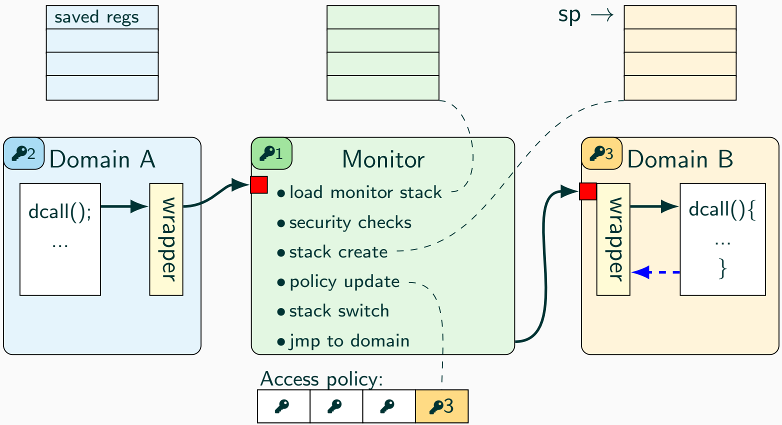


Domain Transitions: dcalls

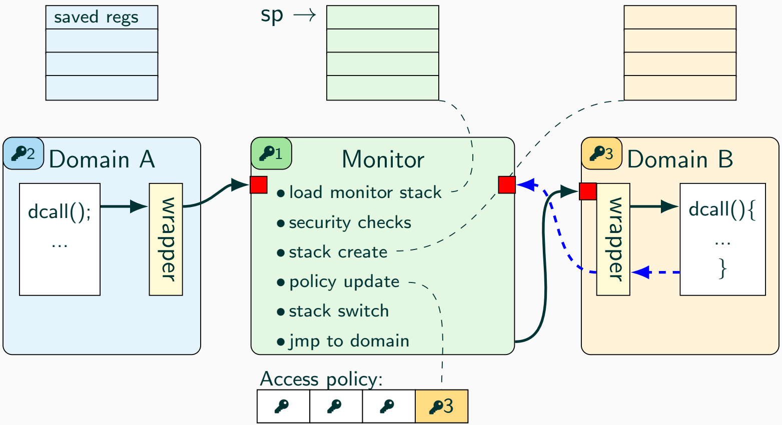


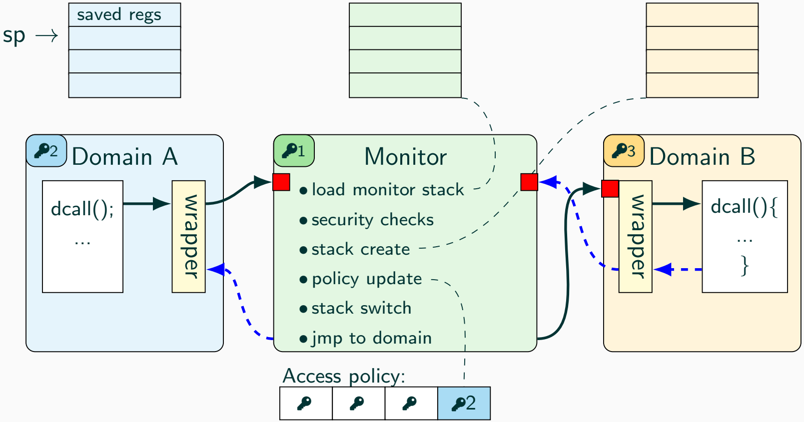
Domain Transitions: dcalls

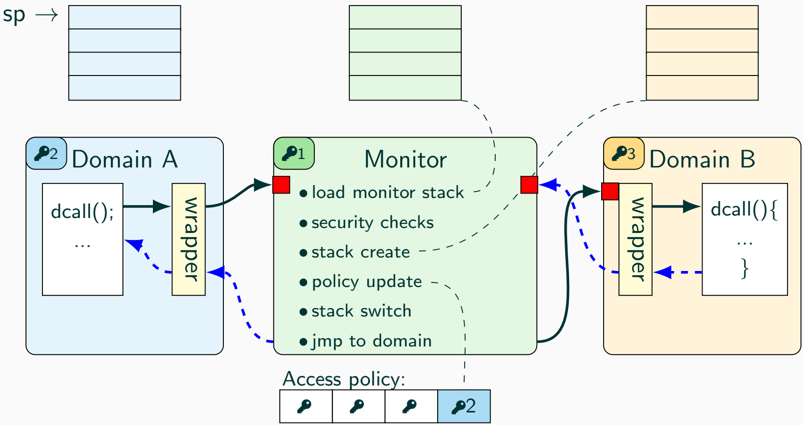



saved regs
sp →
2 Domain A
dcall();
...
wrapper
1 Monitor
load monitor stack
security checks
stack create
policy update
stack switch
jmp to domain
3 Domain B
wrapper
dcall(){
...
}
Access policy:
3

Domain Transitions: dcalls




sp →
saved regs
2
Domain A
dcall();
...
wrapper
1
Monitor
• load monitor stack
• security checks
• stack create
• policy update
• stack switch
• jmp to domain
3
Domain B
wrapper
dcall(){
...
}
Access policy:
2



sp →
2
Domain A
dcall();
...
wrapper
1
Monitor
load monitor stack
security checks
stack create
policy update
stack switch
jmp to domain
3
Domain B
wrapper
dcall(){
...
}
Access policy:
2

Implementation

- Hardware-Software co-design
- Small hardware extension for RISC-V
  - Based on RISC-V *N extension* – "Standard Extension for User-Level Interrupts"
  - Implemented on RISC-V CPU Ariane/CVA6[2]
- Software library
  - API for managing domains/keys/transitions
  - Wraps standard library functions (memory management, threads, signals)
  - Compatible with Intel MPK

[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 implemenation[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