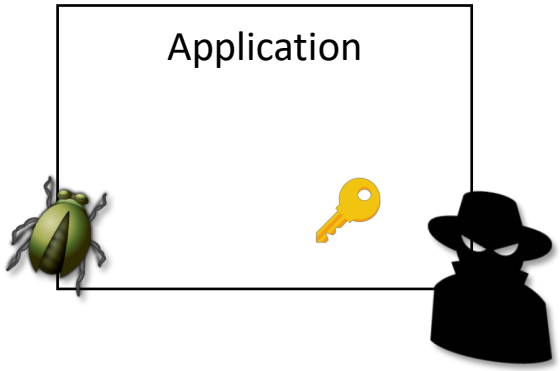
# ERIM: Secure, Efficient in-process Isolation with Memory Protection Keys

**Anjo Vahldiek-Oberwagner**, Eslam Elnikety, Nuno O. Duarte,
Michael Sammler, Peter Druschel, Deepak Garg

MAX PLANCK INSTITUTE
**FOR SOFTWARE SYSTEMS**

# Applications in the **Absence of Isolation**

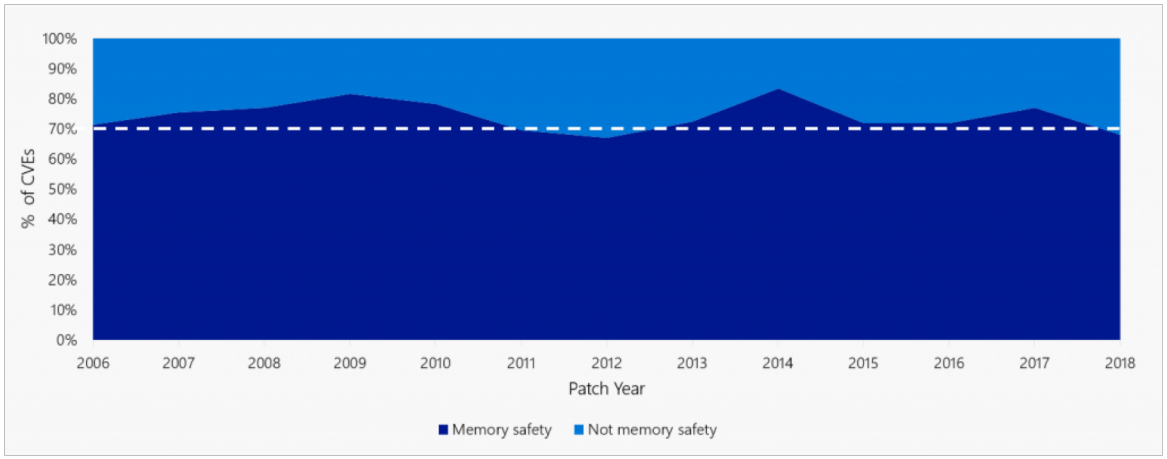
- All state accessible at **all times** to
  - Bugs
  - Security vulnerabilities



# Applications in the **Absence of Isolation**

**Heartbleed Bug**

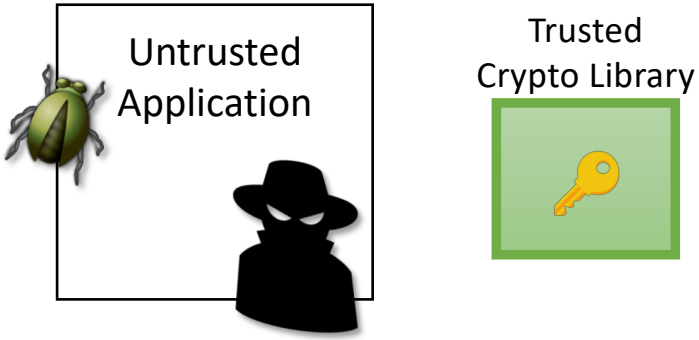
**~70% of CVE assigned by Microsoft are memory safety issues.**



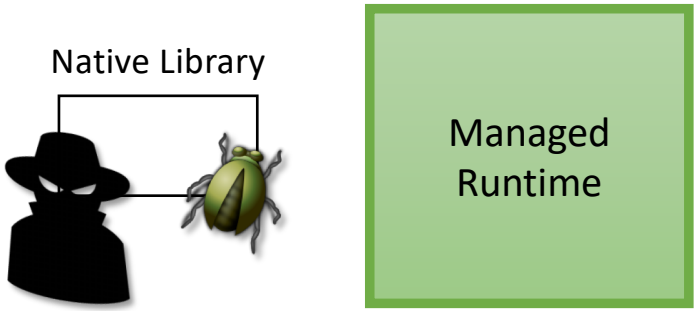

Microsoft Security Response Center: “A proactive approach to more secure code”, 2019

# Example In-Process Isolation Use Cases

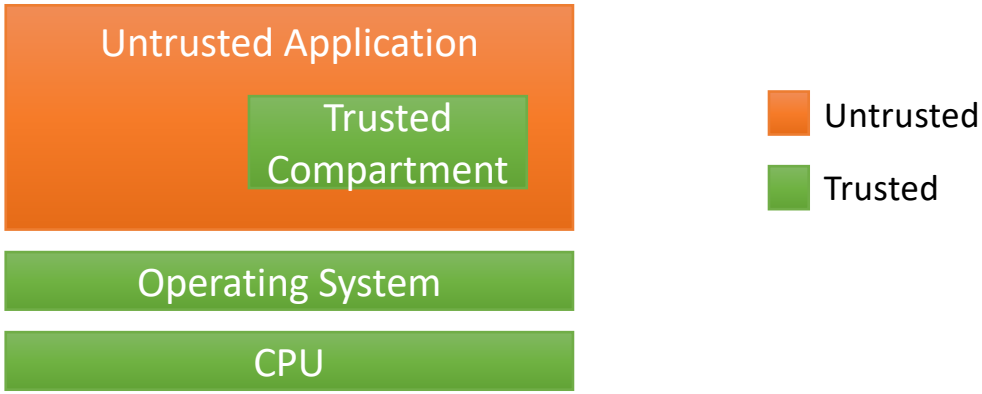
## Cryptographic Secrets




## Managed runtimes from native libraries



# User-space Threat Model




Attacker's Capabilities include, but not limited to

- Control-flow hijacks
- Memory corruption (i.e., out-of-bounds accesses)

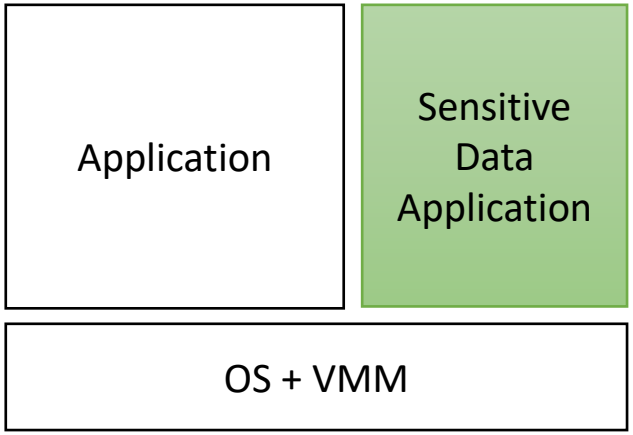
Out of scope:

- Side-channel, row hammer or microarchitectural attacks

# State of In-Application Isolation Techniques

| | Execution overhead | | Switch overhead |
|---|---|---|---|
| | Untrusted | Trusted | |
| **OS/VMM -based[2]** | **Low** | **Low** | **Medium** |
| Lang. & RT[3] | Medium – High | None | None |
| ERIM | Low | None | Low |

**OS/VMM Technique**



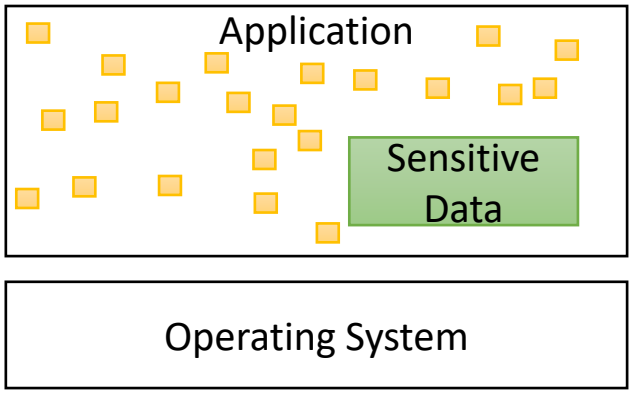

[1] LwC, SMVs, Shreds, Wedge, Nexen, Dune, SeCage, TrustVisor

[2] SFI

# State of In-Application Isolation Techniques

| | Execution overhead | | Switch overhead |
|---|---|---|---|
| | Untrusted | Trusted | |
| OS/VMM-based[2] | Low | Low | Medium |
| **Lang. & RT[3]** | **Medium – High** | **None** | **None** |
| ERIM | Low | None | Low |

**Language and Runtime Techniques**



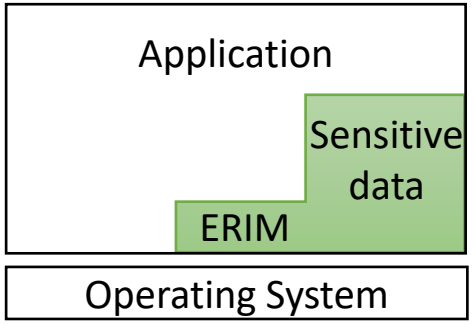

[1] LwC, SMVs, Shreds, Wedge, Nexen, Dune, SeCage, TrustVisor

[2] SFI

# State of In-Application Isolation Techniques

| | Execution overhead | | Switch overhead |
|---|---|---|---|
| | Untrusted | Trusted | |
| OS/VMM-based[2] | Low | Low | Medium |
| Lang. & RT[3] | Medium – High | None | None |
| **ERIM** | **Low** | **None** | **Low** |

**ERIM**

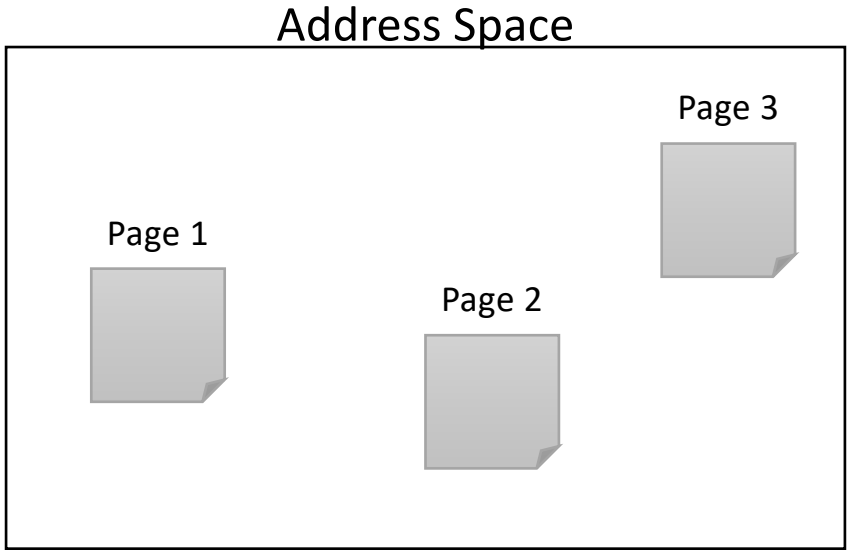



[1] LwC, SMVs, Shreds, Wedge, Nexen, Dune, SeCage, TrustVisor

[2] SFI, Native Client, Memsentry-MPX

# Memory Protection Keys (MPK)

- Available in Skylake server CPUs
- Tag memory pages with PKEY

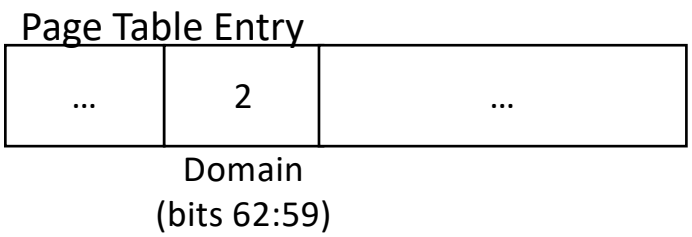


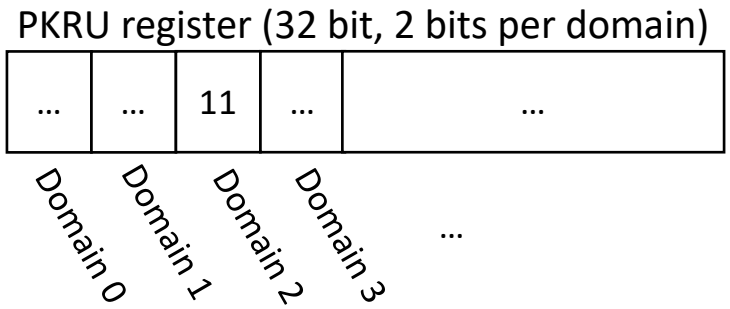

Page Table Entry (PTE)

| | | | | |
|---|---|---|---|---|
| … | **PKEY 0** | … | Page 1 | … |

# Intel Memory Protection Keys (MPK)

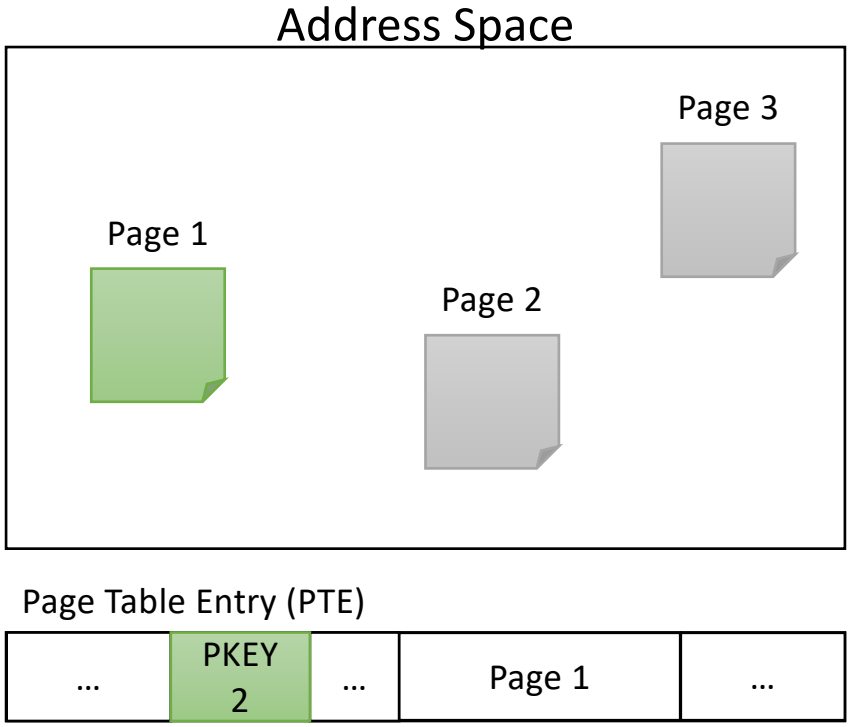
- Tag memory pages with a memory domains (bits 62:59 in page table)
- Permission register (PKRU) enables R/W to a domain
- Update accessible permissions from userspace
  - Fast switching, without context/PT switch
- By itself, protects against **bugs only**



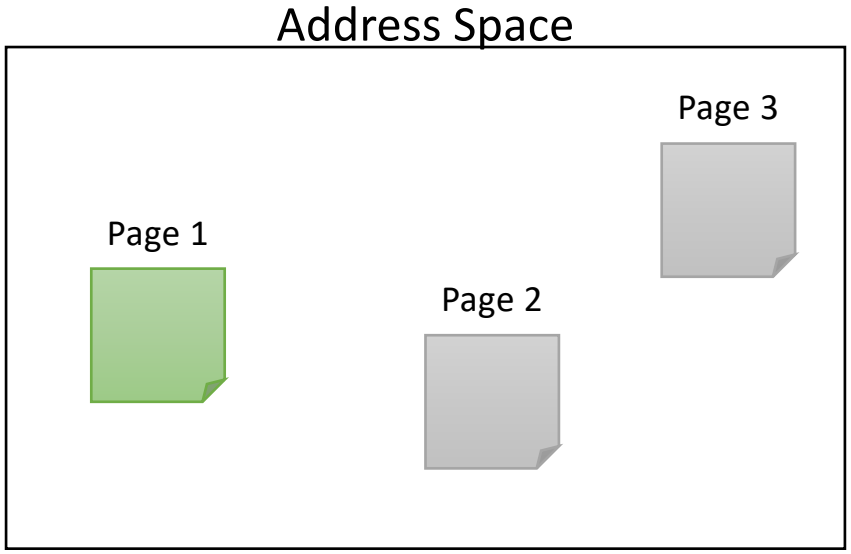
# Intel Memory Protection Keys (MPK)

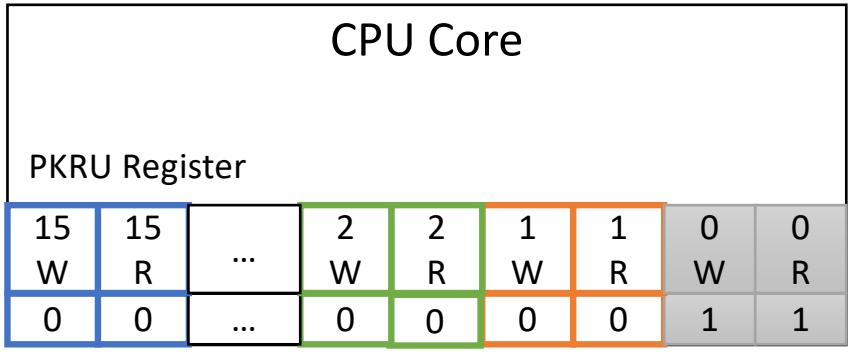
- Available in Skylake server CPUs
- Tag memory pages with PKEY



# Intel Memory Protection Keys (MPK)

- Available in Skylake server CPUs
- Tag memory pages with PKEY
- Permission Register (PKRU)



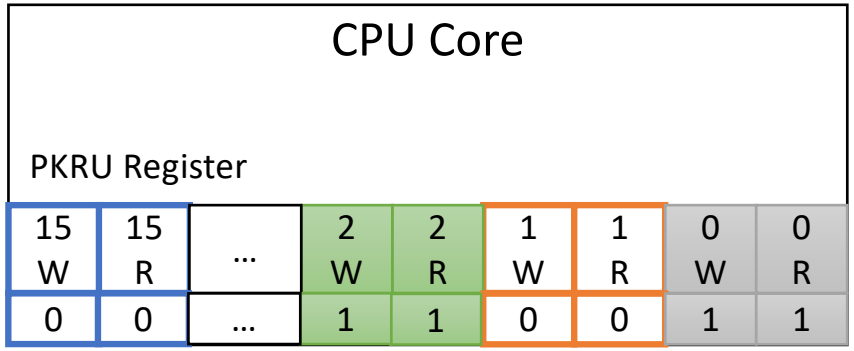



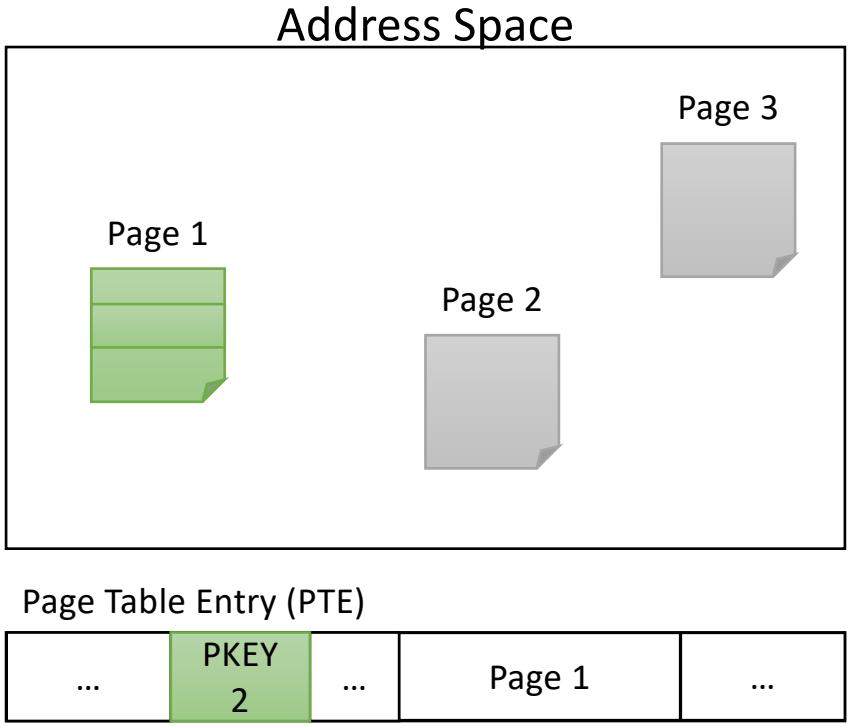

Page Table Entry (PTE)

| … | PKEY 2 | … | Page 1 | … |
|---|---|---|---|---|

# Intel Memory Protection Keys (MPK)

- Available in Skylake server CPUs
- Tag memory pages with PKEY
- Permission Register (PKRU)
- Userspace instruction to update PKRU
  - Fast switch between 11 – 260 cycles/switch



# Intel Memory Protection Keys (MPK)

- Available in Skylake server CPUs
- Tag memory pages with PKEY
- Permission Register (PKRU)
- Userspace instruction to upd
  - Fast switch at 50 cycles/switch

**By itself,**
**MPK does not protect**
**against malicious attacks.**

Address Space

Page 3

CPU Core

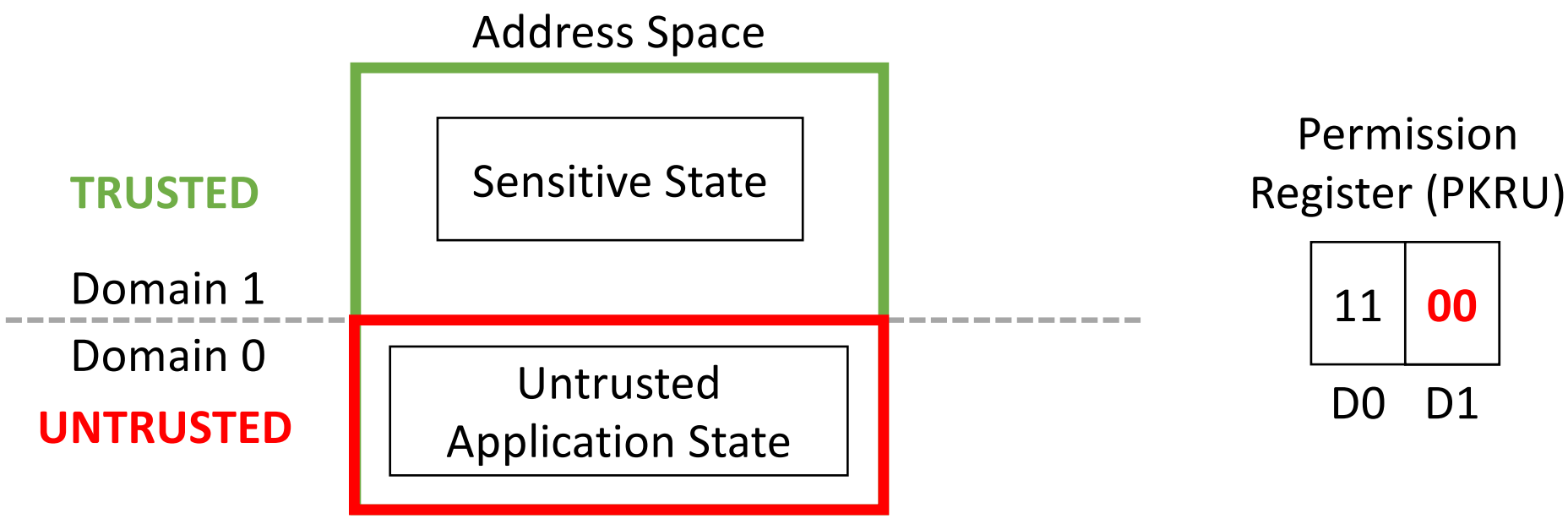
PKRU Register

| 15 W | 15 R | … | 2 W | 2 R | 1 W | 1 R | 0 W | 0 R |
|---|---|---|---|---|---|---|---|---|
| 1 | 1 | … | 1 | 1 | 1 | 1 | 1 | 1 |

Page Table Entry (PTE)

| … | PKEY 2 | … | Page 1 | … |
|---|---|---|---|---|

# Isolating sensitive state with Intel MPK




Domain switch is a user-mode register write: efficient but vulnerable to attack.

# Using ERIM to isolate memory

**Inlined switches**

```
fct_A(…) {
….
switch(Trusted)

access sensitive data

switch(Untrusted)
…
}
```

**Function overwriting**

```
fct_A(…) {
….
}

BUILD_BRIDGE(fct_A);

fct_B(…) {
…
CALL_BRIDGE(fct_A, args);
…
}
```

Function **overloading** via LD_PRELOAD

```
Shared library defines:
fct_A(…) {
f = dlsym(fct_A, …);
switch(Trusted);
ret = f(args);
switch(Untrusted);
return ret;
}
```

# Call Gates

```
  WRPKRU (RW_TRUSTED)

// entry point to trusted

  WRPKRU (DIS_TRUSTED)
  cmp DIS_TRUSTED, EAX
  je continue
  exit
continue:
```

Elevate privileges and transfer to trusted entry point

Remove privileges, check for reduced privileges and return from trusted component

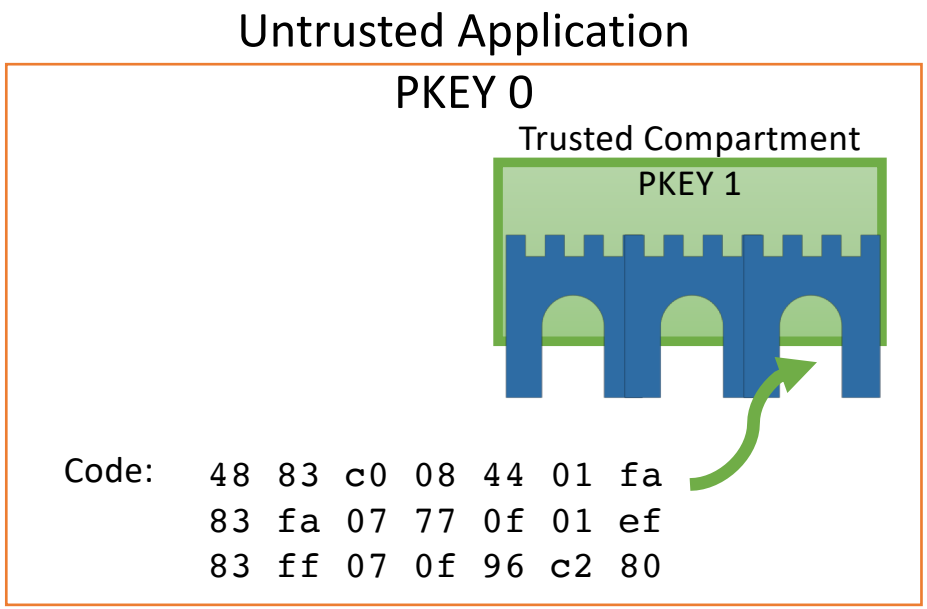
# ERIMized C Program

```
typedef struct secret {
  int number;
} secret;
secret* initSecret() {
  ERIM_SWITCH_T;
  secret * s = malloc(sizeof(secret));
  s->number = random();
  ERIM_SWITCH_U;
  return s;
}
```

```
int compute(secret* s, int m) {
  int ret = 0;
  ERIM_SWITCH_T;
  ret = f(s->number, m);
  ERIM_SWITCH_U;
  return ret;
}
```

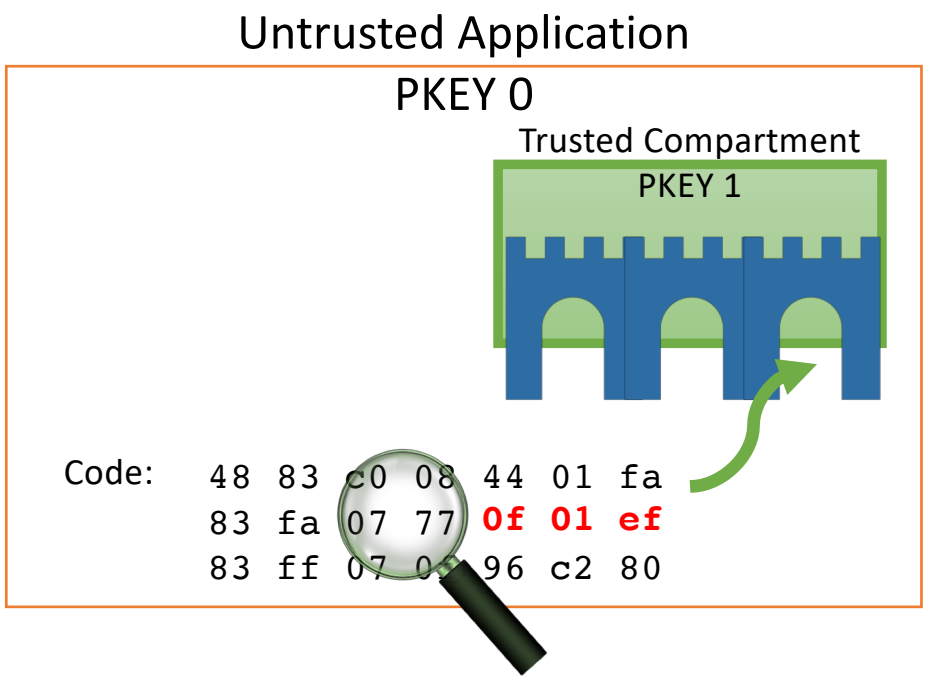
# Overview of ERIM

- Prevent MPK exploitation
  - Safe call gates
  - Prevent execution of permission register updates outside of call gates



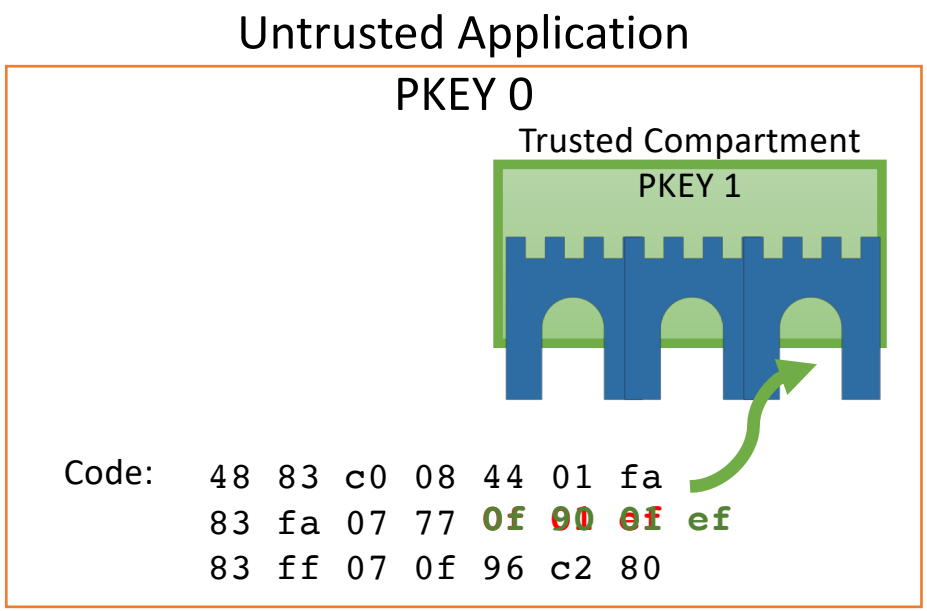
# Overview of ERIM

- Prevent MPK exploitation
  - Safe call gates
  - Prevent execution of permission register updates outside of call gates



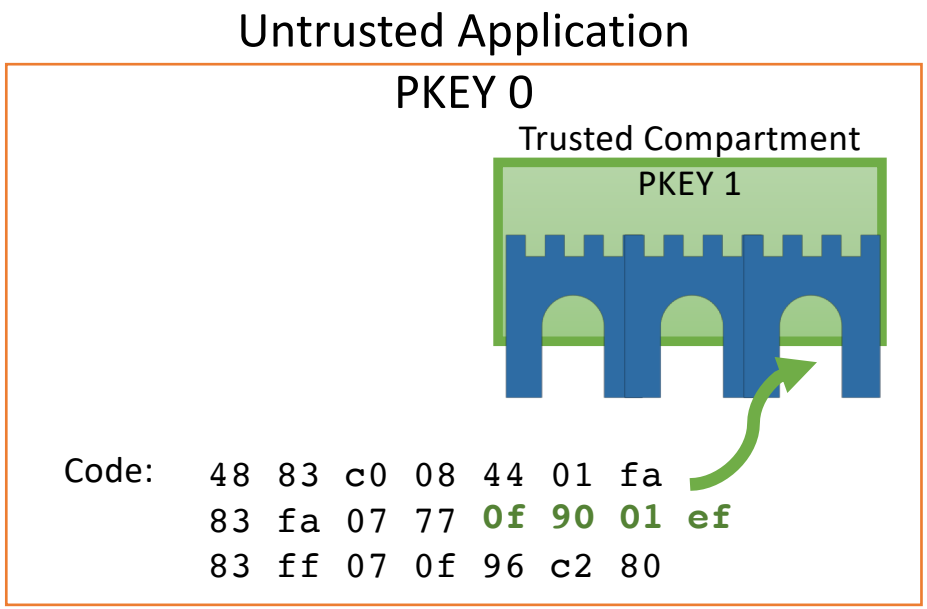
# Overview of ERIM

- Prevent MPK exploitation
  - Safe call gates
  - Prevent execution of permission register updates outside of call gates
- Creating usable binaries
  - Inadvertent PKRU update instruction
  - Rewrite strategy



# Overview of ERIM

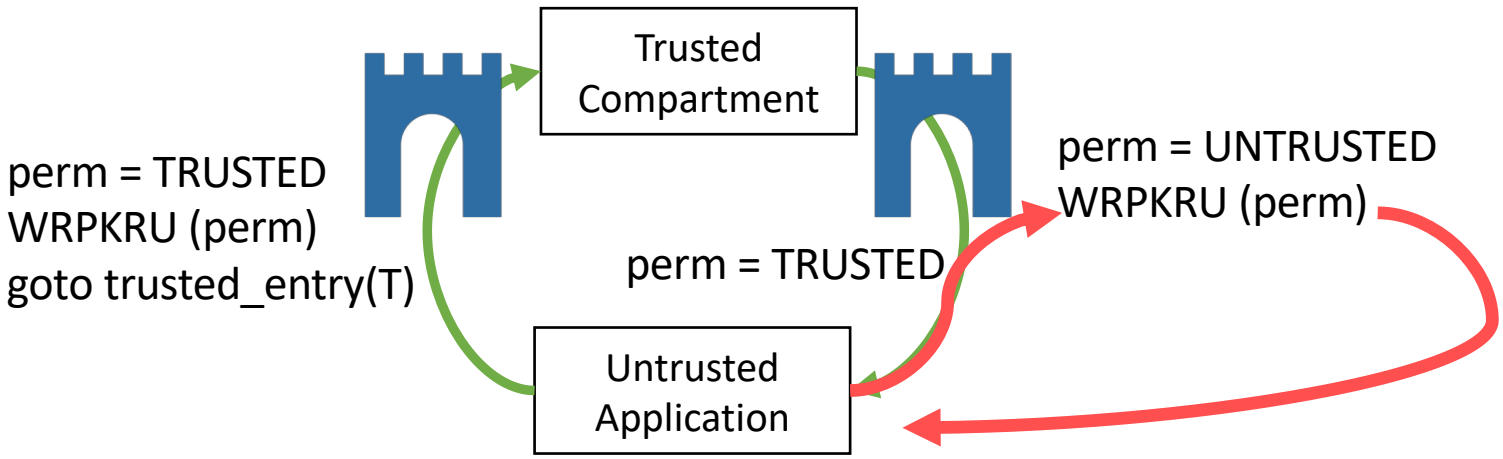
- Prevent MPK exploitation
  - Safe call gates
  - Prevent execution of permission register updates outside of call gates
- Creating usable binaries
  - Inadvertent PKRU update instruction
  - Rewrite strategy
- Evaluation
  - Frequently-switching use cases
  - 10% higher throughput compared to best existing technique



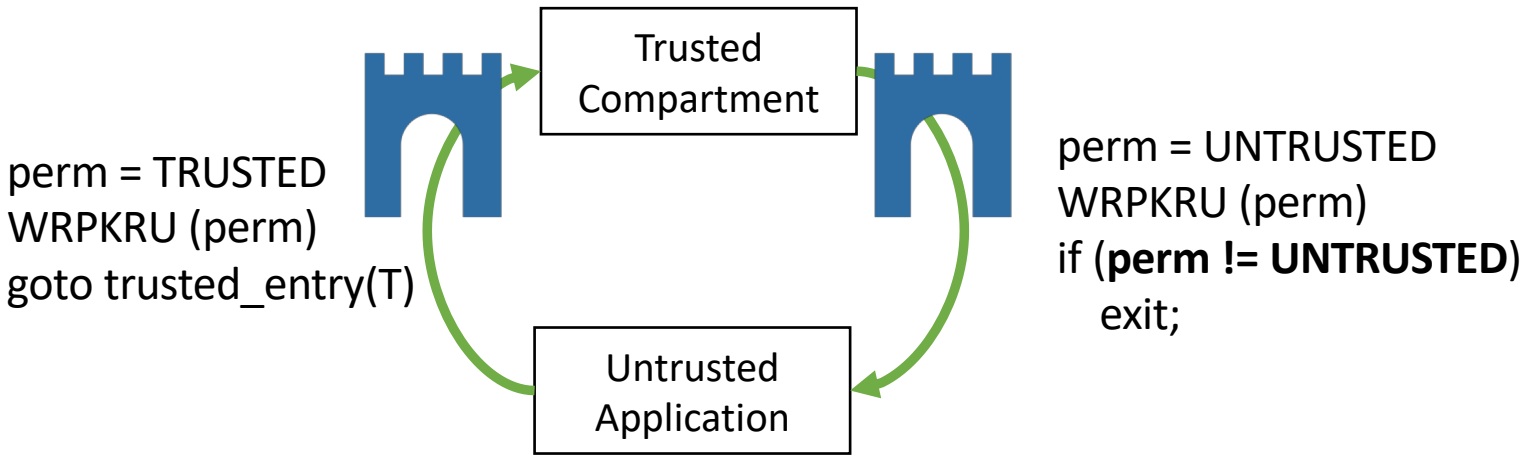
# Updating the permission in PKRU register

- WRPKRU
  - Write EAX into PKRU
- XRSTOR
  - If **bit 9** of EAX is set
  - Load PKRU register from specified memory address

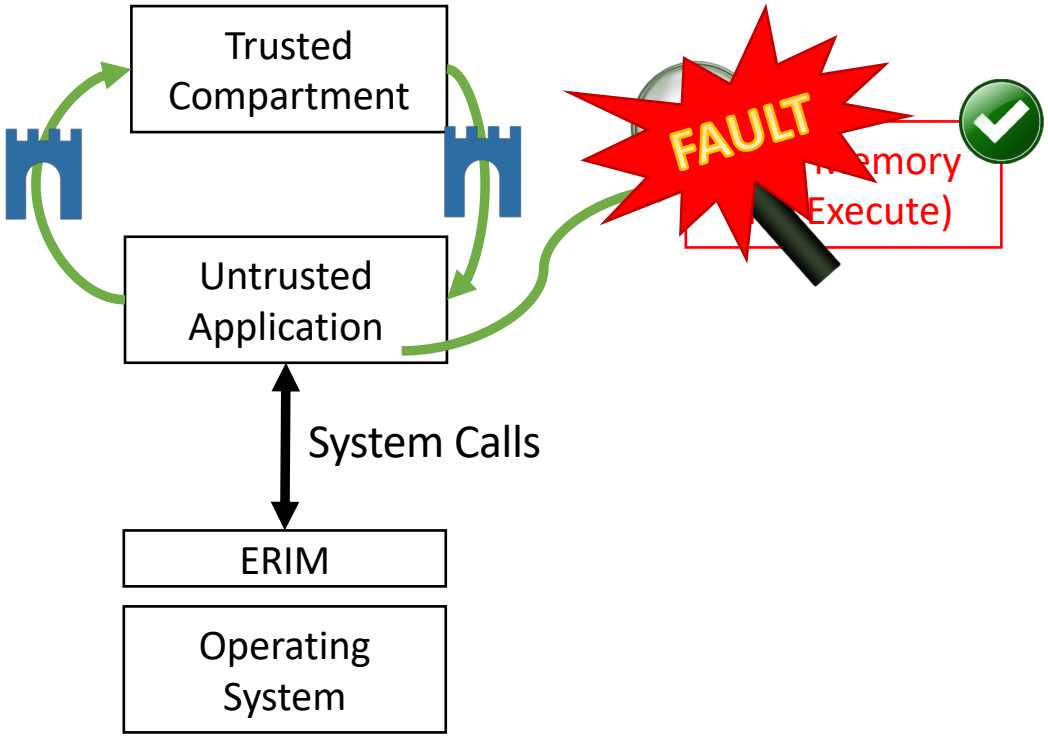
# Safe switching using **call gates**



# Safe switching using **call gates**



# Prevent execution of WRPKRU/XRSTOR outside of call gates




Prevent execution of unvetted pages by

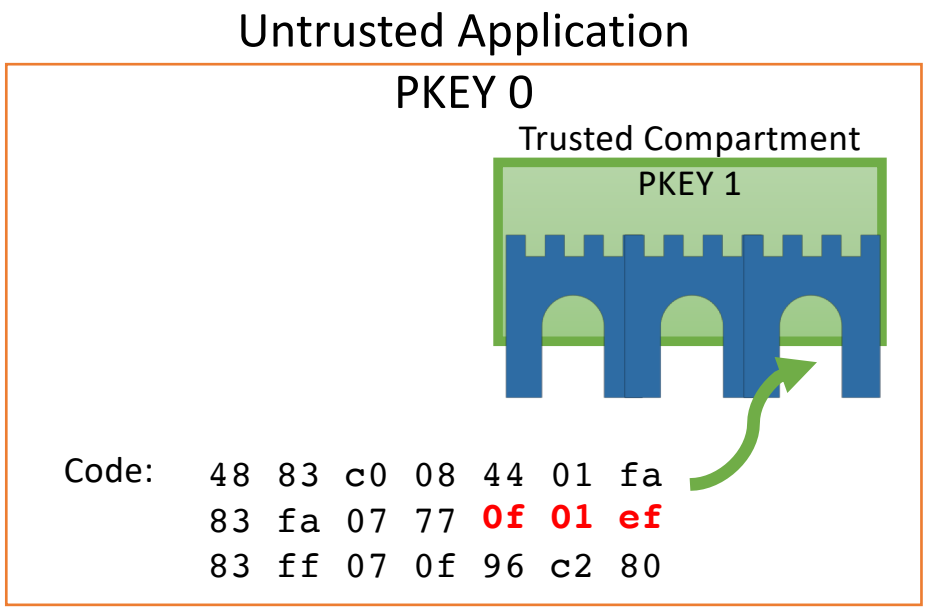
1) Monitoring system calls and removing the execute permission

2) ERIM's fault handler scans memory pages and ensures:
   - WRPKRU is part of a call gate
   - XRSTOR is followed by

```
if(eax | 0x100)
        exit();
```

# Overview of ERIM

- Prevent MPK exploitation
  - Safe call gates
  - Prevent execution of permission register updates outside of call gates
- Creating usable binaries
  - Inadvertent PKRU update instruction
  - Rewrite strategy
- Evaluation
  - Frequently-switching use cases
  - 10% higher throughput compared to best existing technique



# How frequent are inadvertent WRPKRUs/XRSTORs?

| | Debian 8 | Ubuntu 14 | Ubuntu 16 | Gentoo | Gentoo Gold |
|---|---|---|---|---|---|
| **Elf files** | 56035 | 58548 | 69907 | 9940 | 9940 |
| **Elf files with WRPKRU/XRSTOR** | 665 | 603 | 720 | 73 | 34 |
| **Executable WRPKRU/XRSTOR** | 4244 | 1147 | 2105 | 124 | 46 |
| **WPKRU/XRSTOR in code** | 481 | 276 | 384 | 41 | 31 |
| **Disassembled by Dyninst** | 420 | 215 | 332 | 32 | 24 |
| **Inter-instruction** | 30 | 29 | 44 | 5 | 5 |
| **Intra-instruction** | 390 | 186 | 288 | 27 | 19 |

# Creating usable binaries

- ERIM halts executables with inadvertent WRPKRUs/XRSTORs

**Inter-Instruction WRPKRU**

| Instruction 1 | Instruction 2 |
|---|---|
| …**0F** | **01EF**… |

**Intra-Instruction WRPKRU**

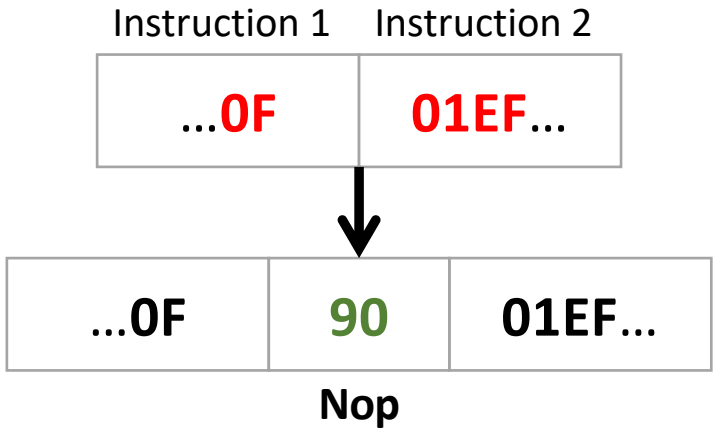
| Instruction 1 |
|---|
| 01**0F01EF**0000 |

→Eliminate inadvertent WRPKRU/XRSTOR by **binary rewriting** at
**compile** time,
**runtime** prior to enabling execute permission,
or via **static** binary rewriting for pre-compiled binaries

# Rewriting inadvertent WRPKRUs/XRSTORs

Devise rewrite rules for inadvertent WRPKRUs

**Inter-Instruction:**

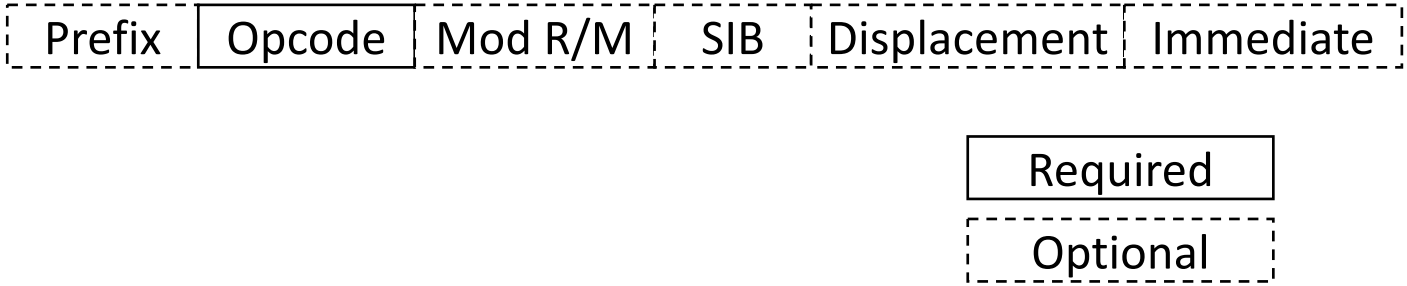


# Rewriting inadvertent WRPKRUs/XRSTORs

Devise rewrite rules for inadvertent WRPKRUs

**Intra-instruction WRPKRU**

Simplified x86 instruction format:



# Rewriting inadvertent WRPKRUs/XRSTORs

Devise rewrite rules for inadvertent WRPKRUs

Example rewrite rule:

add ecx, [**ebx** + **0x01EF0000**]

| Opcode | Mod R/M | Displacement |
|---|---|---|
| 0x01 | **0x0F** | 0x01EF0000 |

→ push eax;
mov eax, ebx;
add ecx, **[eax + 0x01EF0000]**;
pop eax;

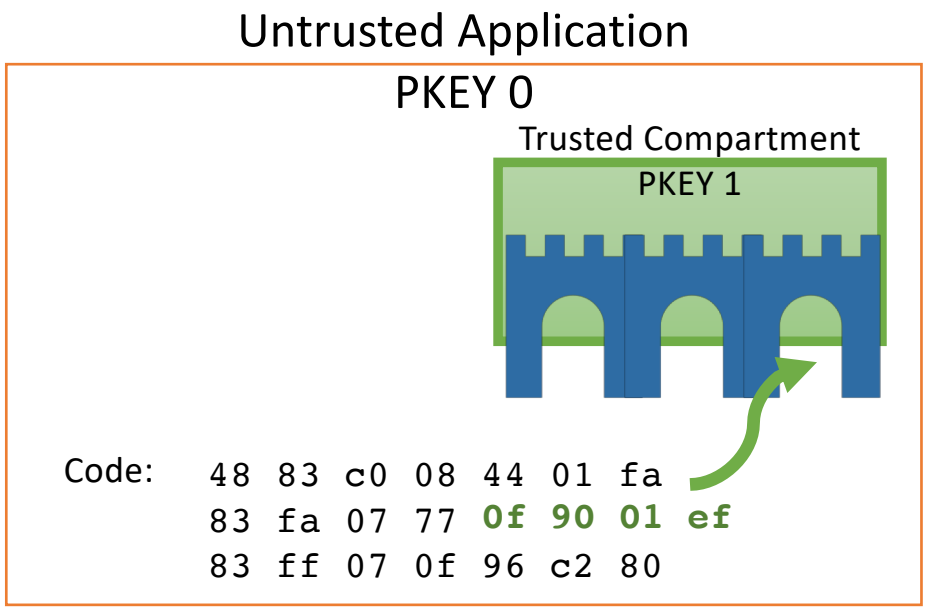
| Opcode | Mod R/M | Displacement |
|---|---|---|
| 0x01 | **0x07** | 0x01EF0000 |

# Creating safe binaries: Rewrite Rules

| Overlap with | Cases | Rewrite strategy | ID | Example |
|---|---|---|---|---|
| Opcode | Opcode = WRPKRU | Insert privilege check after WRPKRU | 1 | |
| Mod R/M | Mod R/M = 0x0F | Change to unused register + move command | 2 | add ecx, [ebx + 0x01EF0000] → mov eax, ebx; add ecx, [eax + 0x01EF0000]; |
| | | Push/Pop used register + move command | 3 | add ecx, [ebx + 0x01EF0000] → push eax; mov eax, ebx; add ecx, [eax + 0x01EF0000]; pop eax; |
| Displacement | Full/Partial sequence | Change mode to use register | 4 | add eax, 0x0F01EF00 → (push ebx;) mov ebx, 0x0F010000; add ebx, 0x0000EA00; add eax, ebx; (pop ebx;) |
| | Jump-like instruction | Move code segment to alter constant used in address | 5 | call [rip + 0xffef010f] → call [rip + 0xffef0100] |
| Immediate | Full/Partial sequence | Change mode to use register | 6 | add eax, 0x0F01EF → (push ebx;) mov ebx, 0x0F01EE00; add ebx, 0x00000100; add eax, ebx; (pop ebx;) |
| | Associative opcode | Apply instruction twice with different immediates to get equivalent effect | 7 | add ebx, 0x0F01EF00 → add ebx, 0x0E01EF00; add ebx, 0x01000000 |

# Overview of ERIM

- Prevent MPK exploitation
  - Safe call gates
  - Prevent execution of permission register updates outside of call gates
- Creating usable binaries
  - Inadvertent PKRU update instruction
  - Rewrite strategy
- Evaluation
  - Frequently-switching use cases
  - 10% higher throughput compared to best existing technique



# Prototype implementation

- ERIM userspace library
  - Call gates
  - Memory allocator for trusted component overloading malloc-like functions
  - Memory inspection (exclude unsafe WRPKRU/XRSTOR)
- Prevent execution on pages with unsafe WRPKRUs/XRSTOR
  a) P-Trace and seccomp BPF userspace monitor
  b) Linux Security Module
- Remove inadvertent WRPKRUs/XRSTORs
  - Static binary rewrite tool based on DynInst

# Evaluation

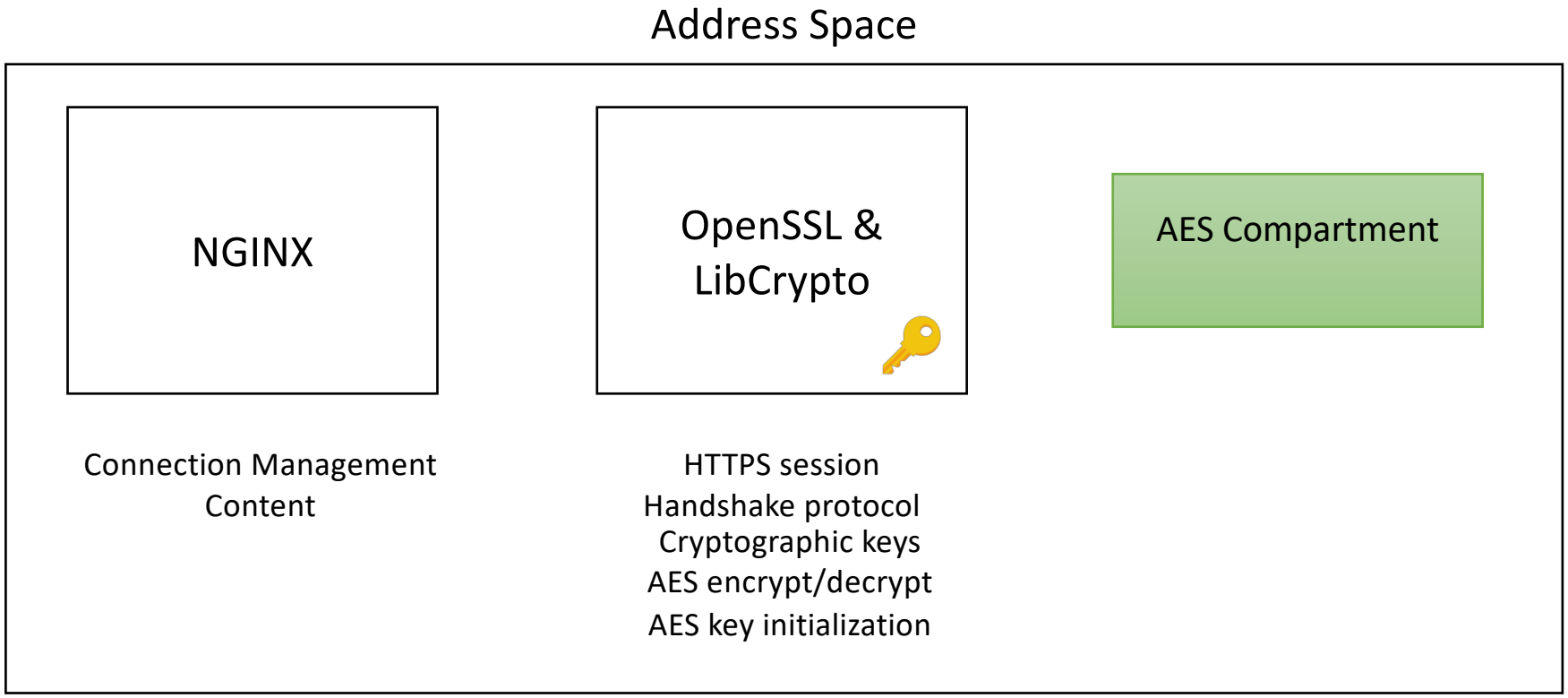
How frequent are inadvertent WRPKRUs/XRSTORs?

- Inspected about 200,000 executable files of 5 Linux distributions
- Found 1213 **inadvertent** WRPKRU/XRSTOR in binary code
- DynInst disassembled 1,023
- 100% rewrite success

What is ERIM's overhead in frequently-switching use cases?

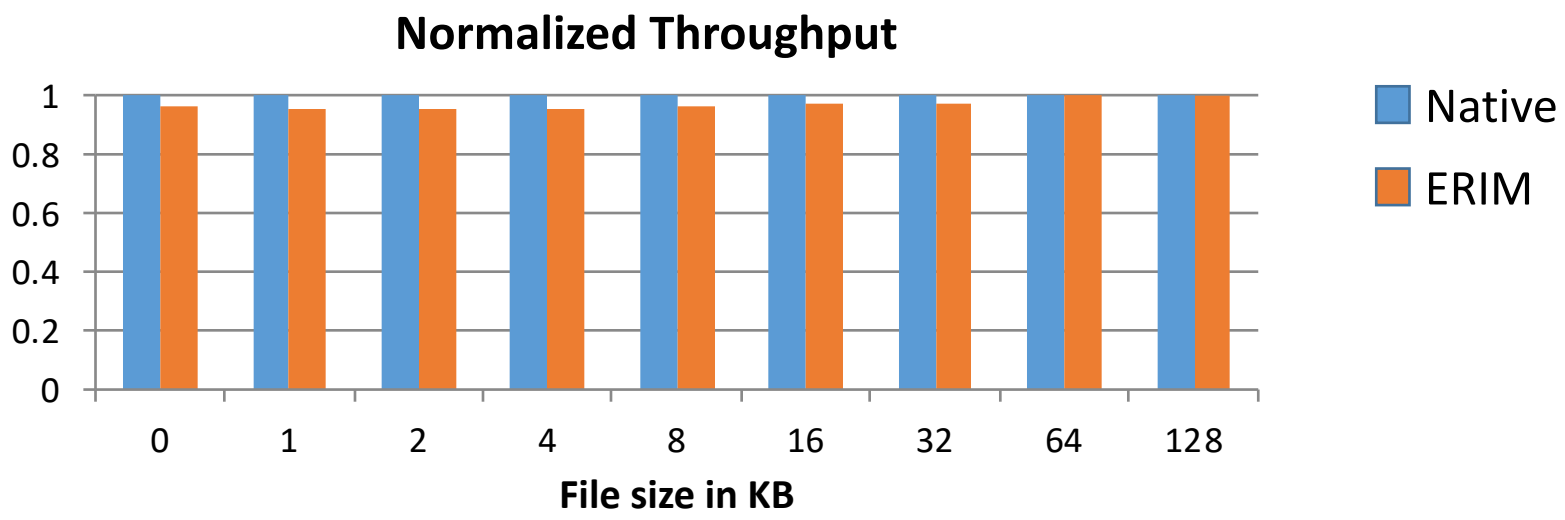
- Isolating **session keys** in Nginx
- Isolating a **managed runtime** (node.js) from native libraries
- Isolating **in-memory state** of reference monitors (CPI/CPS)

# Use case: Session Key Isolation

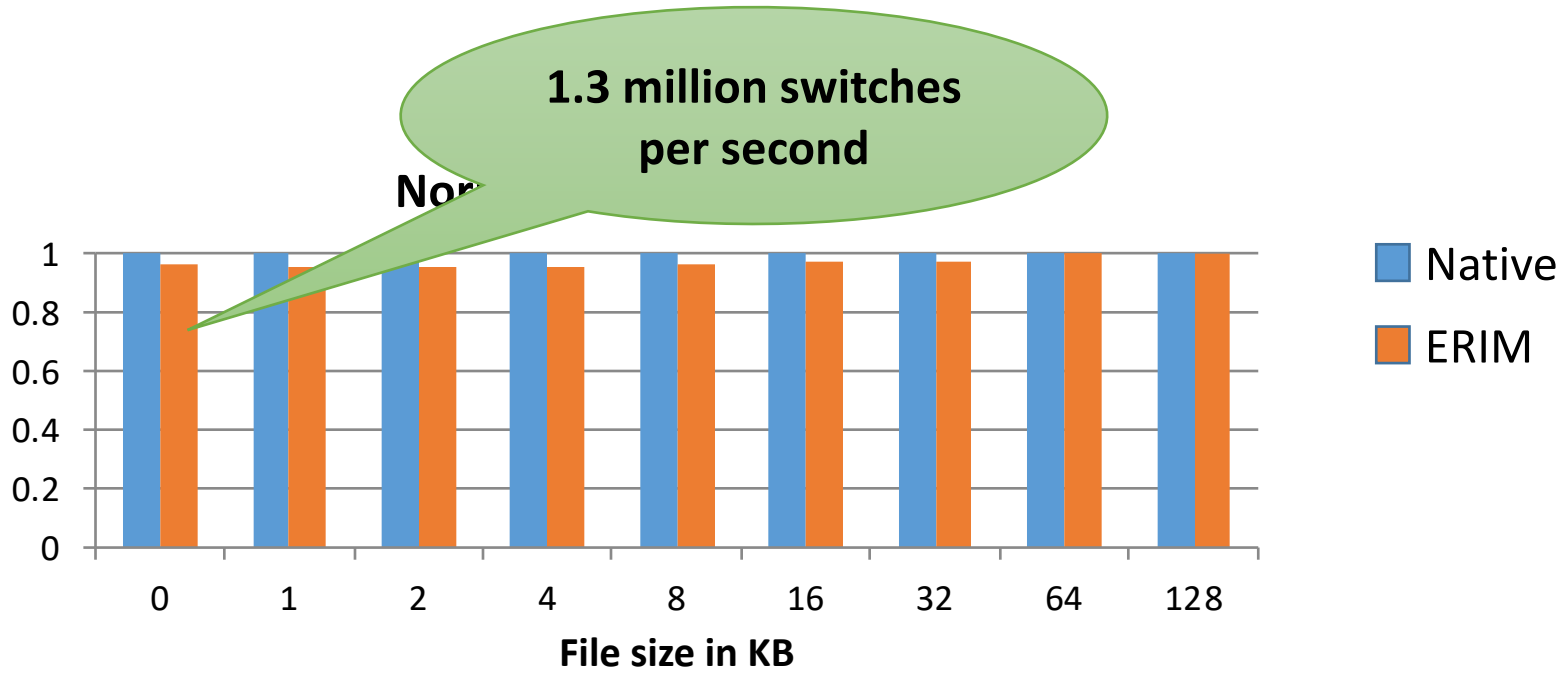


# Nginx Throughput with protected session keys

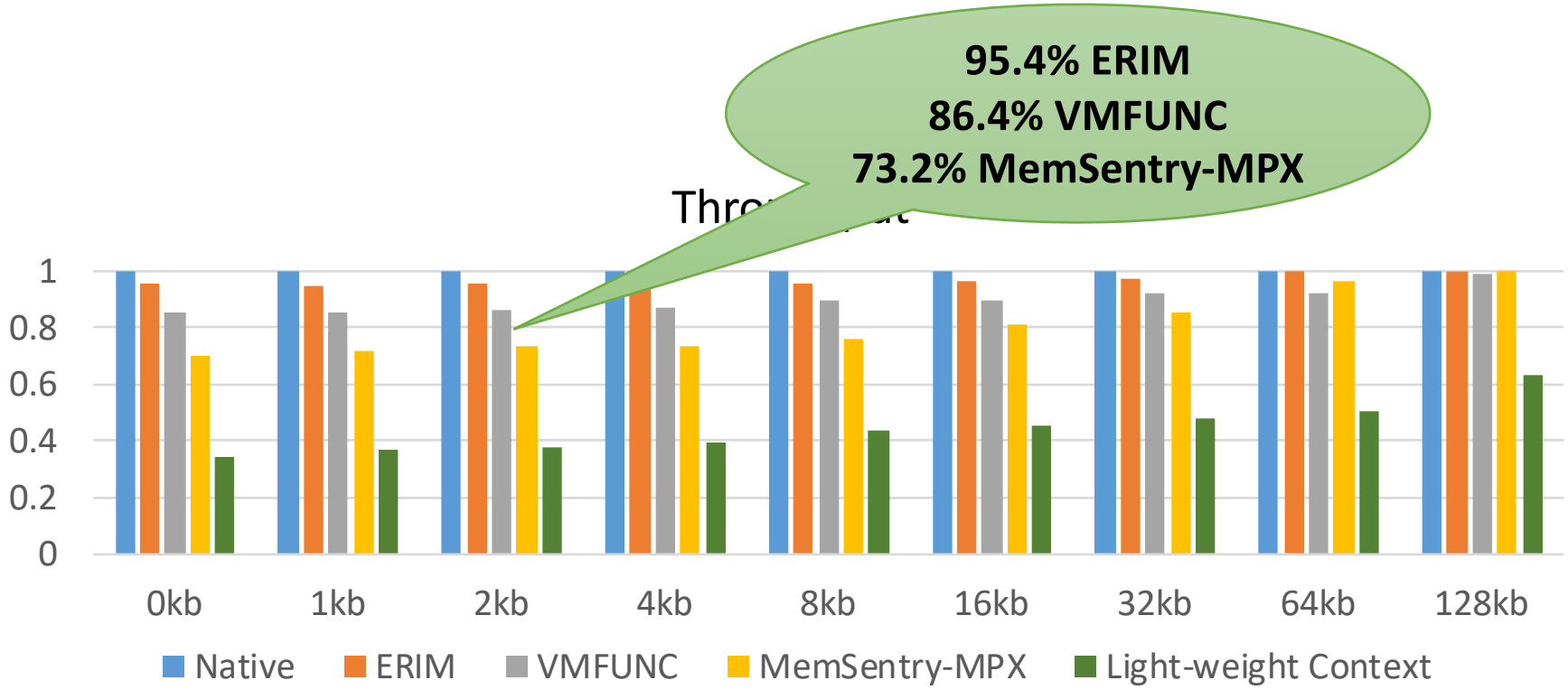
ERIM throughput within 5% of native.



# Nginx Throughput with protected session keys

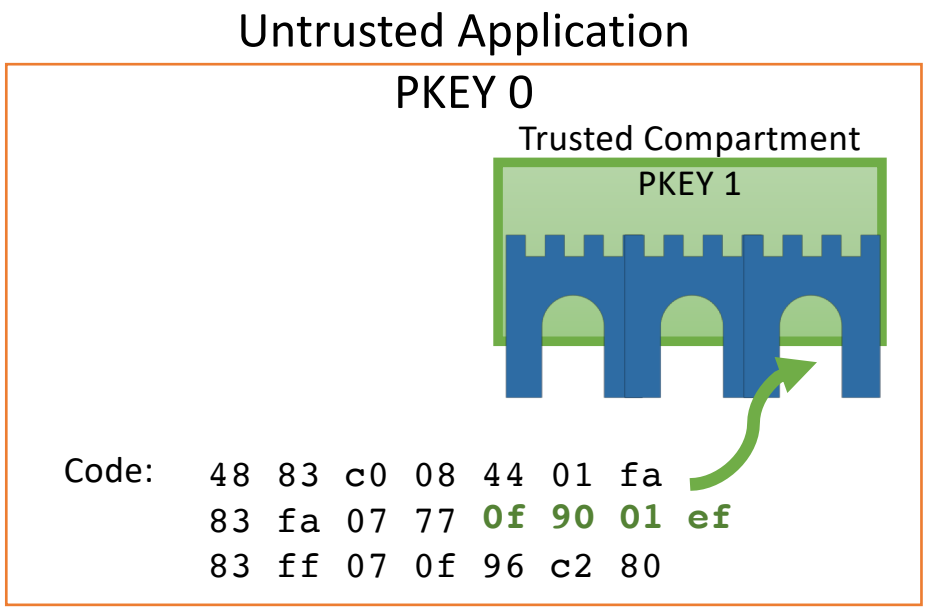


# Comparison to Prior Art



# Summary

- Prevent MPK exploitation
  - Safe call gates
  - Prevent execution of permission register updates outside of call gates
- Creating usable binaries
  - Inadvertent PKRU update instruction
  - Rewrite strategy
- Evaluation
  - Frequently-switching use cases
  - 10% higher throughput compared to best existing technique



# Thank you!

## ERIM: Secure, Efficient in-process Isolation with Memory Protection Keys

**Anjo Vahldiek-Oberwagner**, Eslam Elnikety, Nuno O. Duarte,
Michael Sammler, Peter Druschel, Deepak Garg

Code available at **https://gitlab.mpi-sws.org/vahldiek/erim**