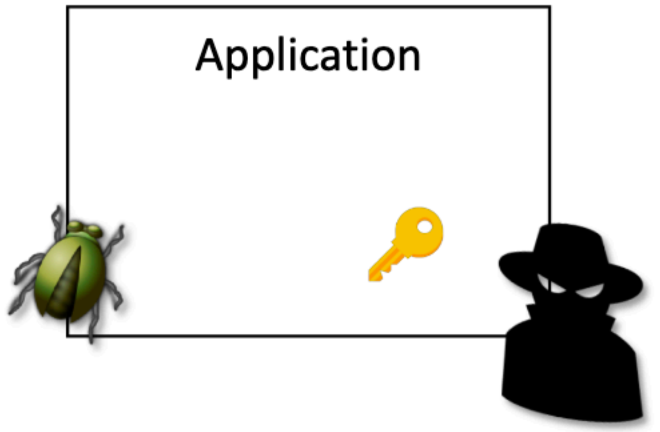
ERIM: Secure, Efficient In-process Isolation with Protection Keys (MPK)

USENIX 2019

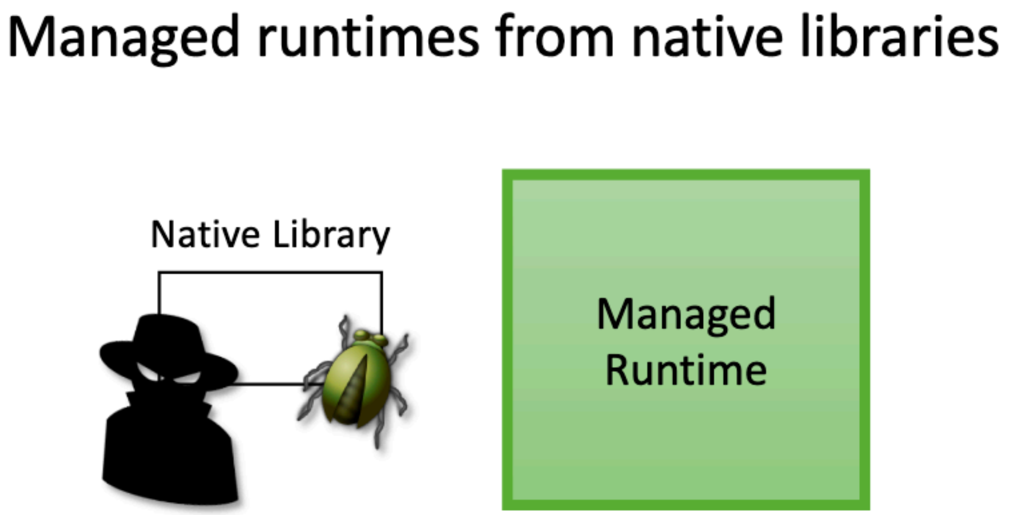
Anjo Vahldiek-Oberwagner
Eslam Elnikety
Nuno O. Duarte
Michael Sammler
Peter Druschel
Deepak Garg
From Max Planck Institute for Software Systems, Saarland Informatics Campus

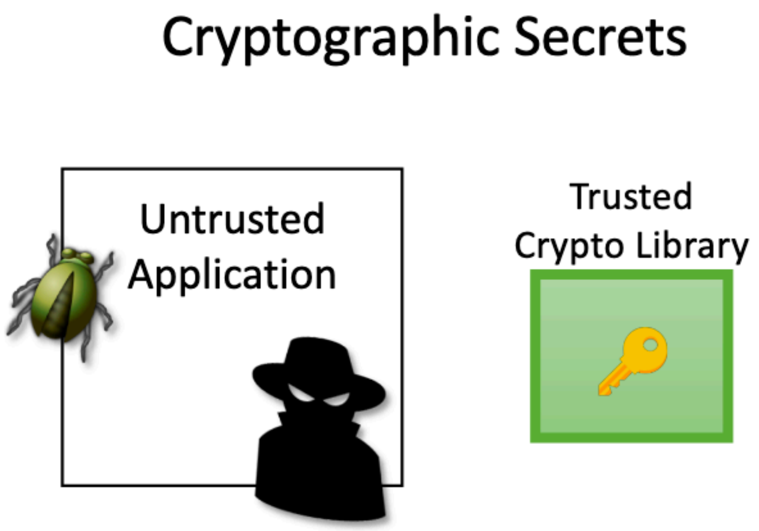
Outline

Introduction



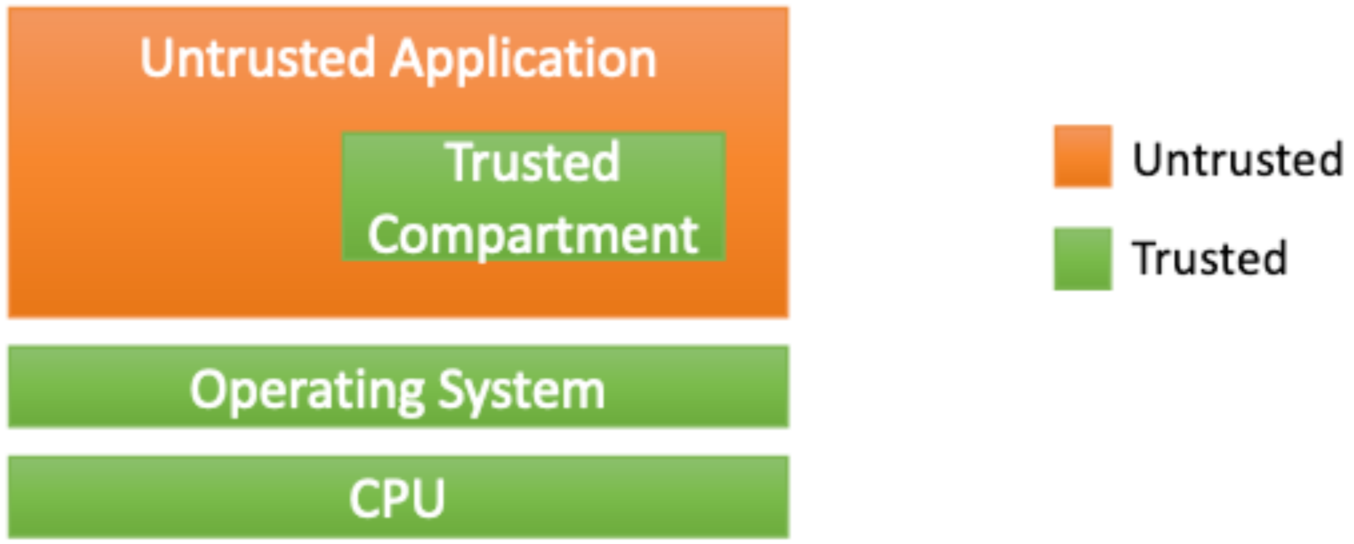
- A single security vulnerability from any component of a process may lead to the loss of its data confidentiality and integrity.
- In-process memory isolation, for instance, Isolating
  - Cryptographic keys in a network server.
  - Managed runtime from unsafe co-linked native library
  - Jump table.





Threat Model

- Attacker's Capabilities:
    - Control-flow hijacks
    - Memory corruption
- Out of scope:
    - Micro-architectural attacks (side channel, row hammer, etc)



Contributions

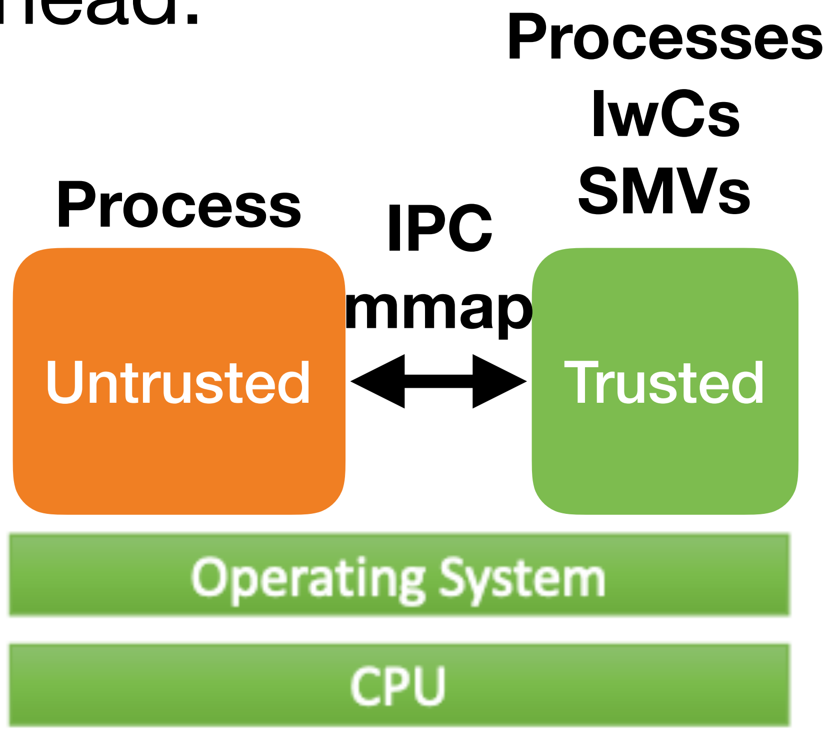
- ERIM, an efficient memory isolation technique that relies on a combination of Intel MPK ISA extension and binary inspection.
- ERIM does not require or assume control-flow integrity.
- A complete rewriting procedure is presented to ensure binaries cannot be exploited to circumvent ERIM.
- ERIM can protect applications with high inter-component switching rates with low overhead, unlike existing techniques.

Background & Related Work

- Most of the following techniques suffer from intolerable overhead in high domain switch rate, or need additional CFI solutions to provide strong security.
    - OS-Based Techniques
    - Virtualization-Based Techniques
    - Language and Runtime Techniques
    - Hardware-Based Trusted Execution Environments

OS-Based Techniques

- Isolation can be easily achieved by placing application components in separate OS processes.
- However, this method has high overhead even with a moderate of cross-component invocation.
- The following approaches have made Isolating long-term signing keys feasible with little overhead:
  - Light-weight contexts (lwCs).
  - Secure memory views (SVMs).
  - Nested kernels.

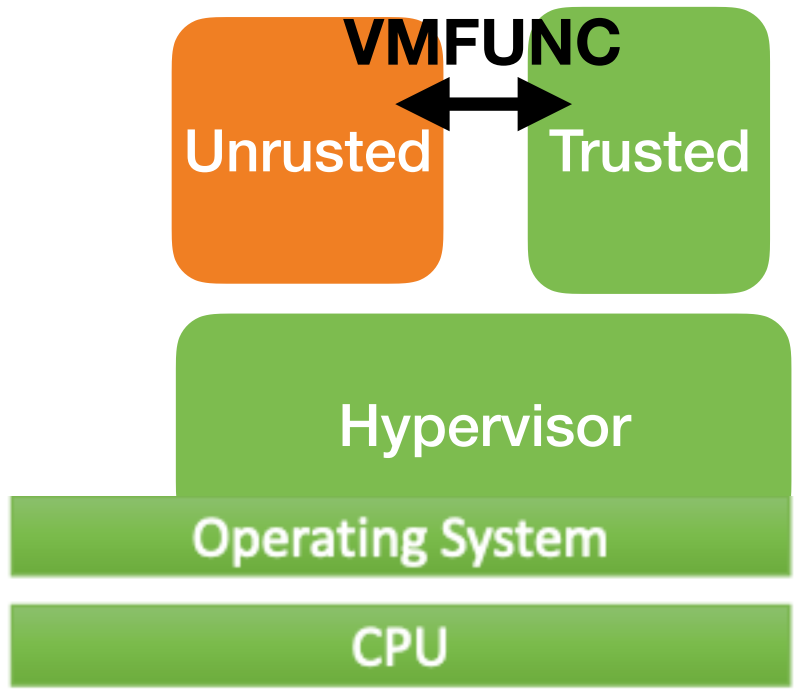


Virtualization-Based Techniques

- In-process data encapsulation can be provided by a hypervisor.
- Intel VT-x x86 virtualization ISA extensions:
  - Several researches use VMFUNC to switch extended page tables.
  - SIM relies on VT-x to isolate a security monitor within a untrusted guest.
- TrustVisor uses a thin hypervisor and nested page tables to support isolation.

Virtualization-Based Techniques (Cont'd)

- Nexen decompose Xen hypervisor into isolated components and a security monitor. Control of MMU is restricted to the monitor.
- In addition to the overhead of VMFUNC itself, these techniques incur overheads on TLB misses and syscalls (extended page tables & hypercalls).



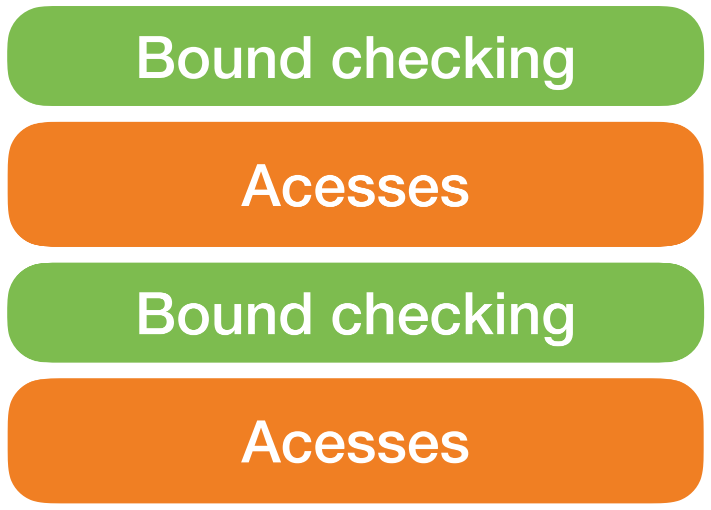
Language and Runtime Techniques

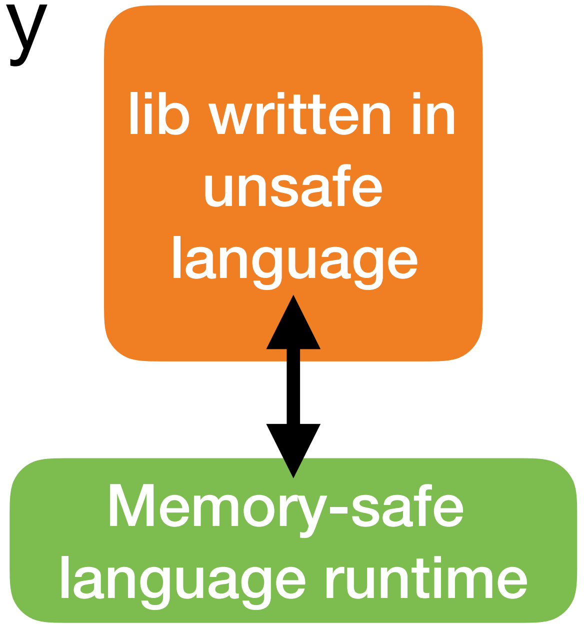
- Memory isolation can be provided as part of a memory-safe programming language.

- Software Fault Isolation (SFI) provides memory isolation in unsafe languages using runtime access checks inserted by compiler or by rewriting binaries.

  - Even with Intel MPX support, the overhead of bound checks is order of tens percent points in many applications.

  - Control flow integrity.





Hardware-Based Trusted Execution Environments

- Intel SGX and ARM TrustZone allow components of applications to execute with hardware-enforced isolation.
- While these method can isolate data even from the OS, switching overheads are high.
  - Intel SGX: It costs around 10K of cycles to switch between components (ECALLs).

Hardware-Based Trusted Execution Environments (Cont’d)

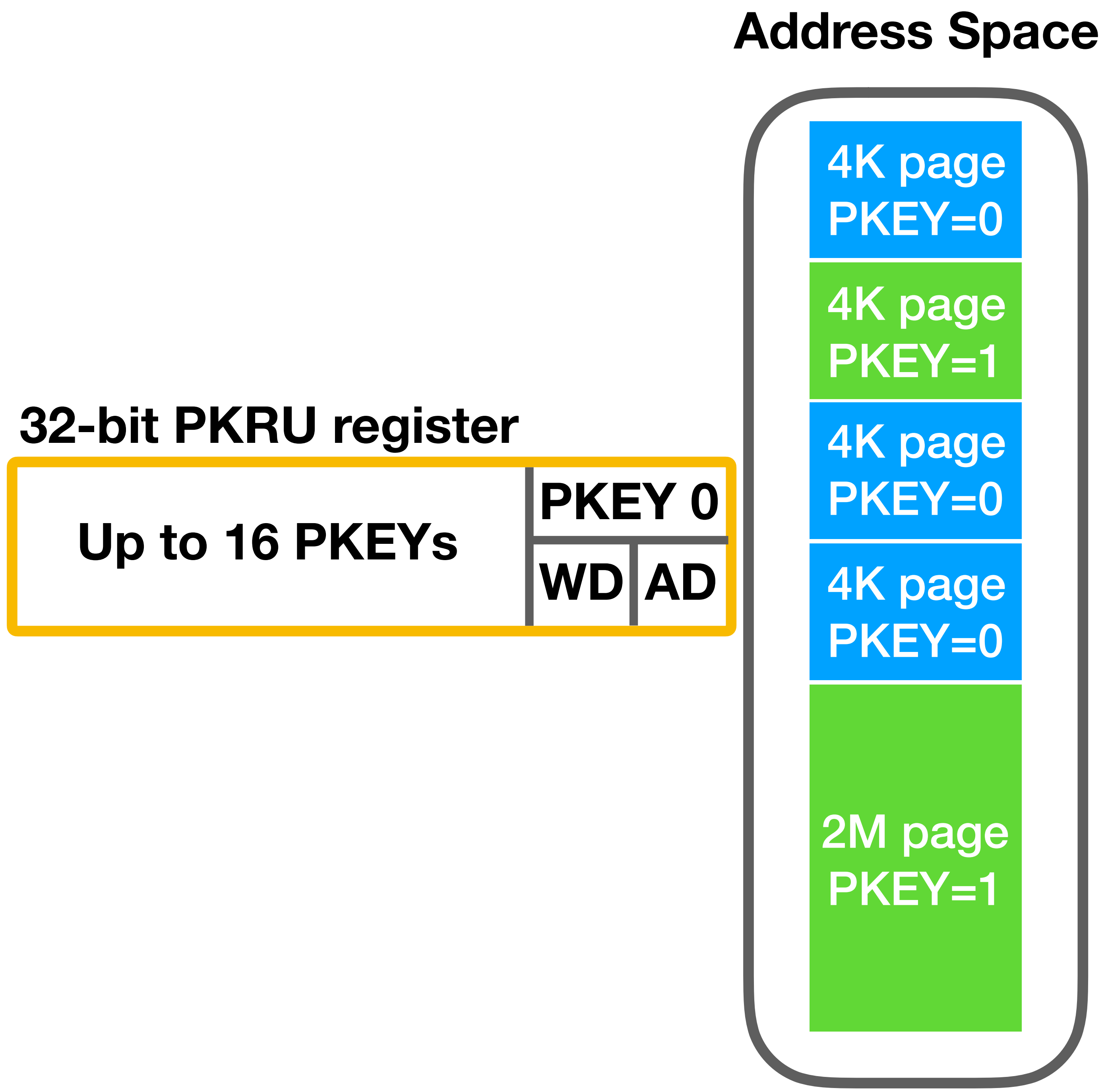
- ARM memory domains:
  - Domain switching is a privileged instruction. (syscall)
- MPK-based techniques:
  - MemSentry is implemented as a pass in LLVM compiler toolchain, providing a general framework for data encapsulation.
  - However, it does not defense against control flow attacks that misuse PKRU-updating instructions.

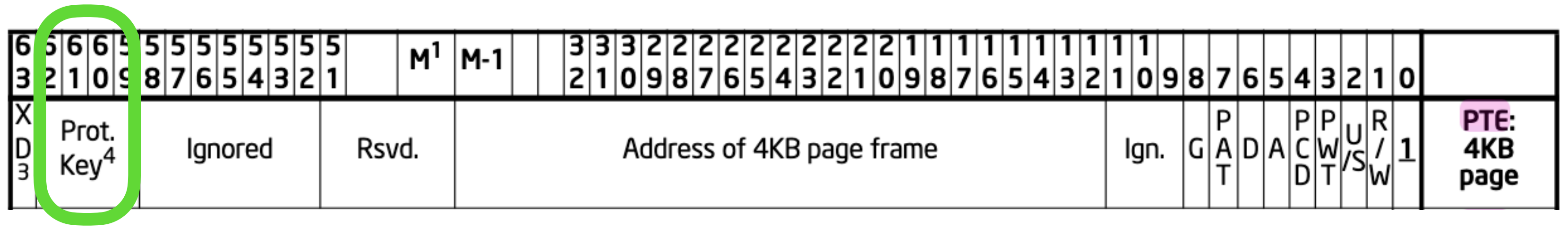
Design

- Intel Memory Protection Key (MPK)
- Design Overview
- Threat Model
- Call Gates
- Binary Inspection
- Binary Rewriting
- Developing ERIM Applications

Intel Memory Protection Key (MPK)

- MPK is a memory tagging ISA extension available on Skylake server CPUs.
- It tags memory pages with a 4 bits PKEY.
- States in PKRU register determine the data access right jointly with record on the page table.

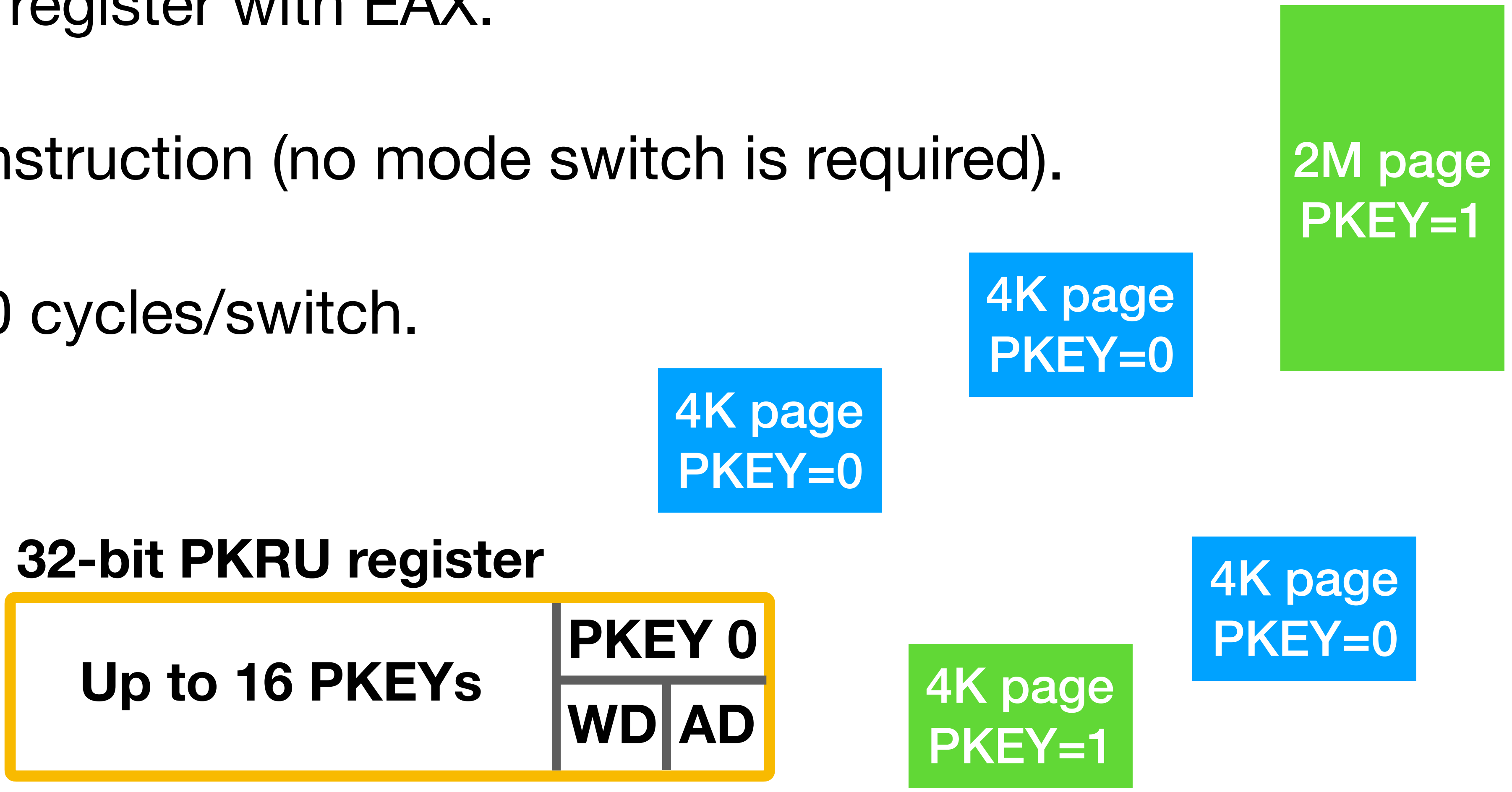




63	62 61 60 59	58 57 56 55 54 53 52 51	M[1]	M-1 … 32 31 … 12	11 10 9	8	7	6	5	4	3	2	1	0	
XD3	Prot. Key[4]	Ignored	Rsvd.	Address of 4KB page frame	Ign.	G	PAT	D	A	PCD	PWT	U/S	R/W	1	**PTE: 4KB page**

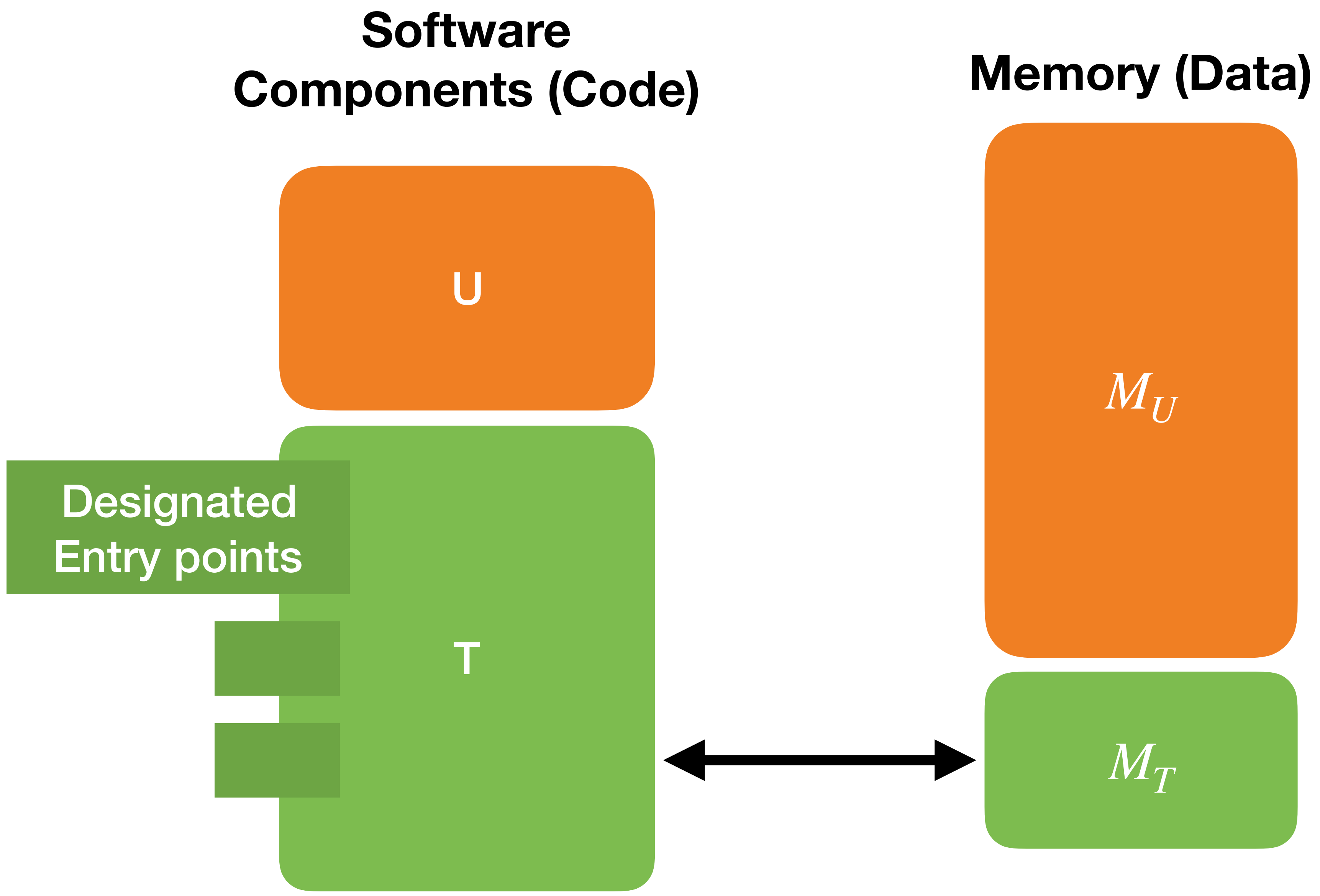
MPK (Cont'd)

- Each CPU core has a PKRU register.
- To modify the access permission of a set of pages marked by same PKEY:
  - WRPKRU:
    - writes PKRU register with EAX.
    - user-space instruction (no mode switch is required).
    - Takes 11-260 cycles/switch.
  - XRSTROE



Design Overview

- Goals:



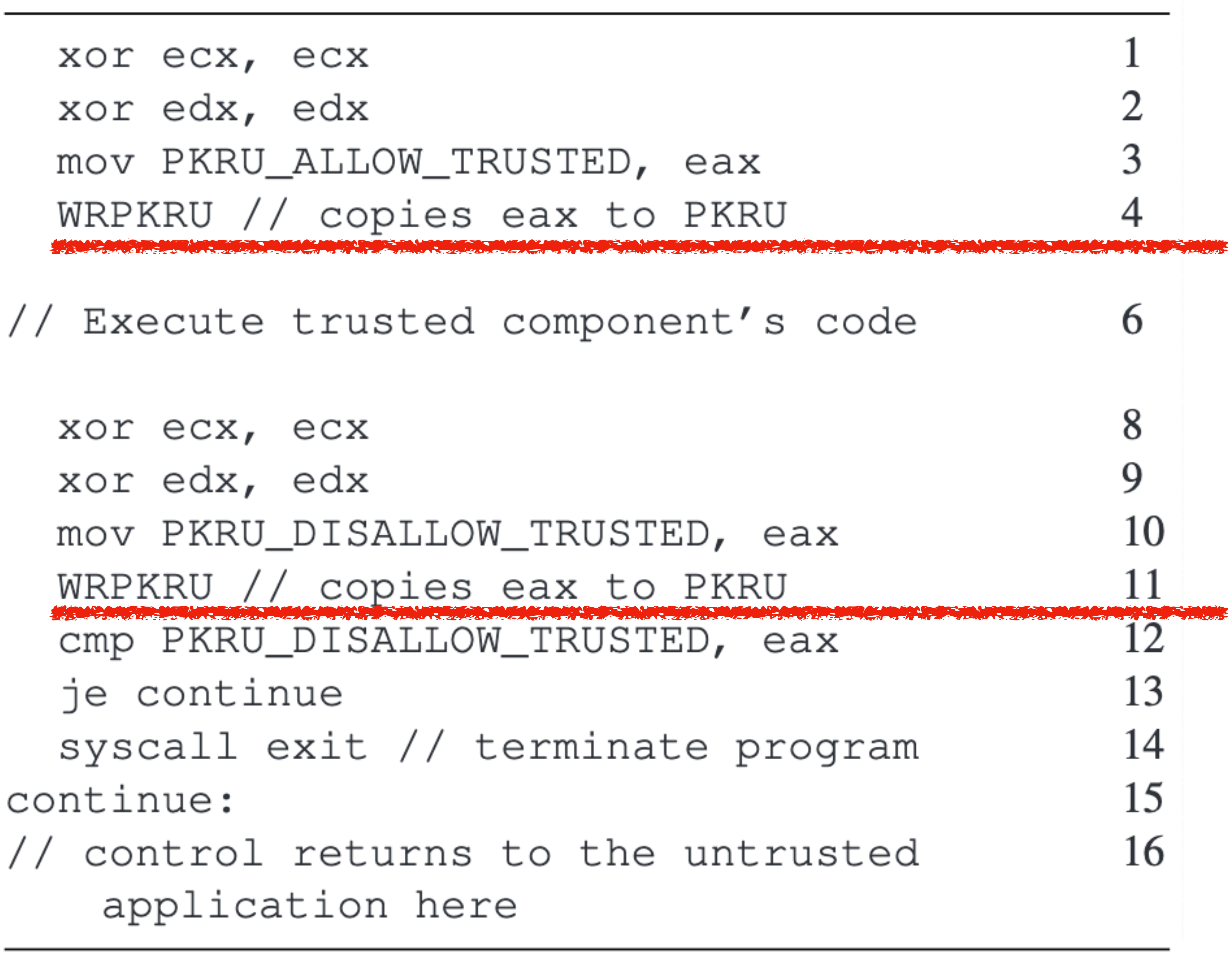
Design Overview (Cont'd)

- Preventing exploitation: Occurrences of WRPKRU instruction sequence on executable pages may be exploited by control flow hijacks.
  - By binary inspections, the author states that an occurrence of WRPKRU is safe if it is immediately followed by:
    - A pre-designated entry point of T .
    - A sequence of instructions that checks that the permissions set by WRPKRU do not include access to M_T
- Creating safe binaries: the author uses a binary rewriting scheme that rewrite any unsafe occurrence of WRPKRU.

Threat Model

- There is no assumption about the untrusted component U
- ERIM assumes that the trusted component T's binary does not have such vulnerabilities and explicit information leaks.
- ERIM assumes that the kernel enforces standard DEP.
- Side-channel, row hammer attacks and micro architectural attacks are beyond the scope of this work.
- Current prototype is incompatible with apps that simultaneously use MPK for other purposes.

Call Gates



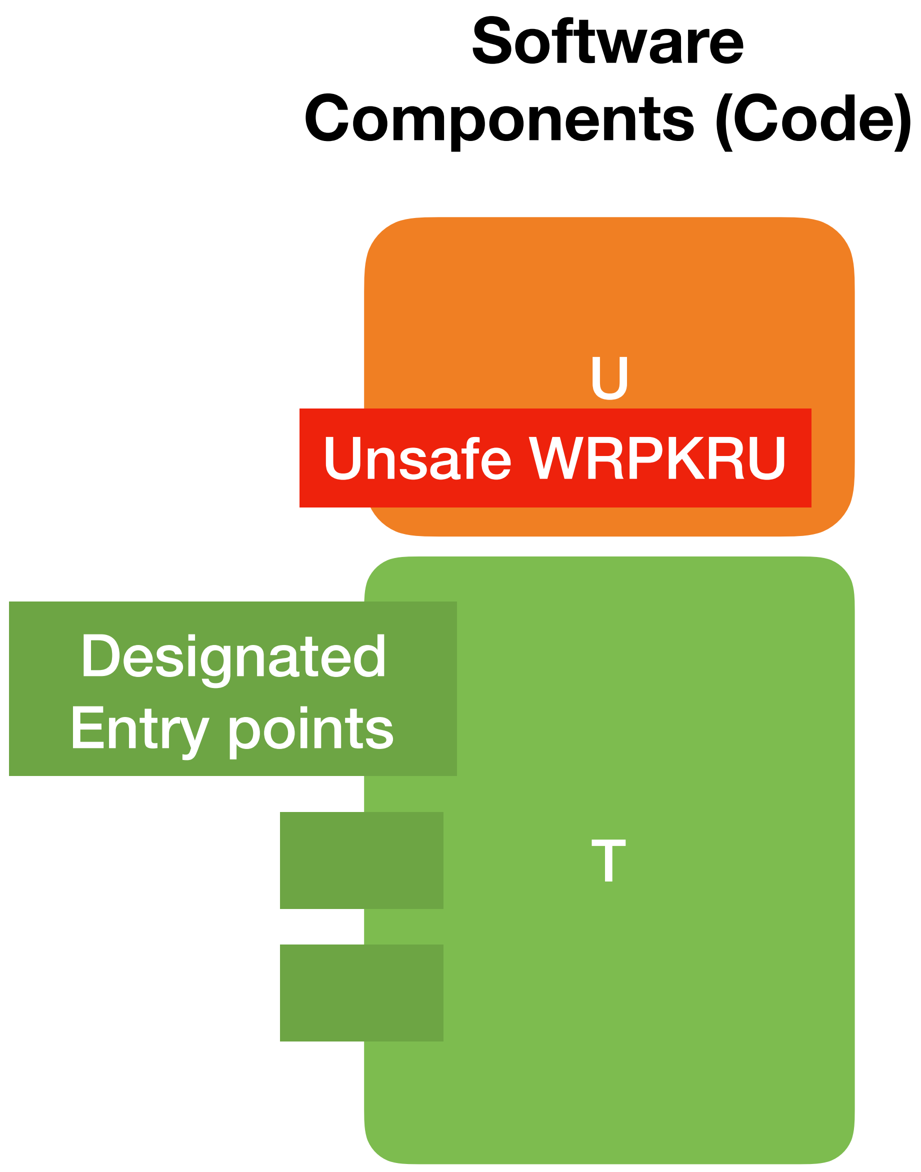
```
  xor ecx, ecx
  xor edx, edx
  mov PKRU_ALLOW_TRUSTED, eax
  WRPKRU // copies eax to PKRU

// Execute trusted component's code

  xor ecx, ecx
  xor edx, edx
  mov PKRU_DISALLOW_TRUSTED, eax
  WRPKRU // copies eax to PKRU
  cmp PKRU_DISALLOW_TRUSTED, eax
  je continue
  syscall exit // terminate program
continue:
// control returns to the untrusted
   application here
```

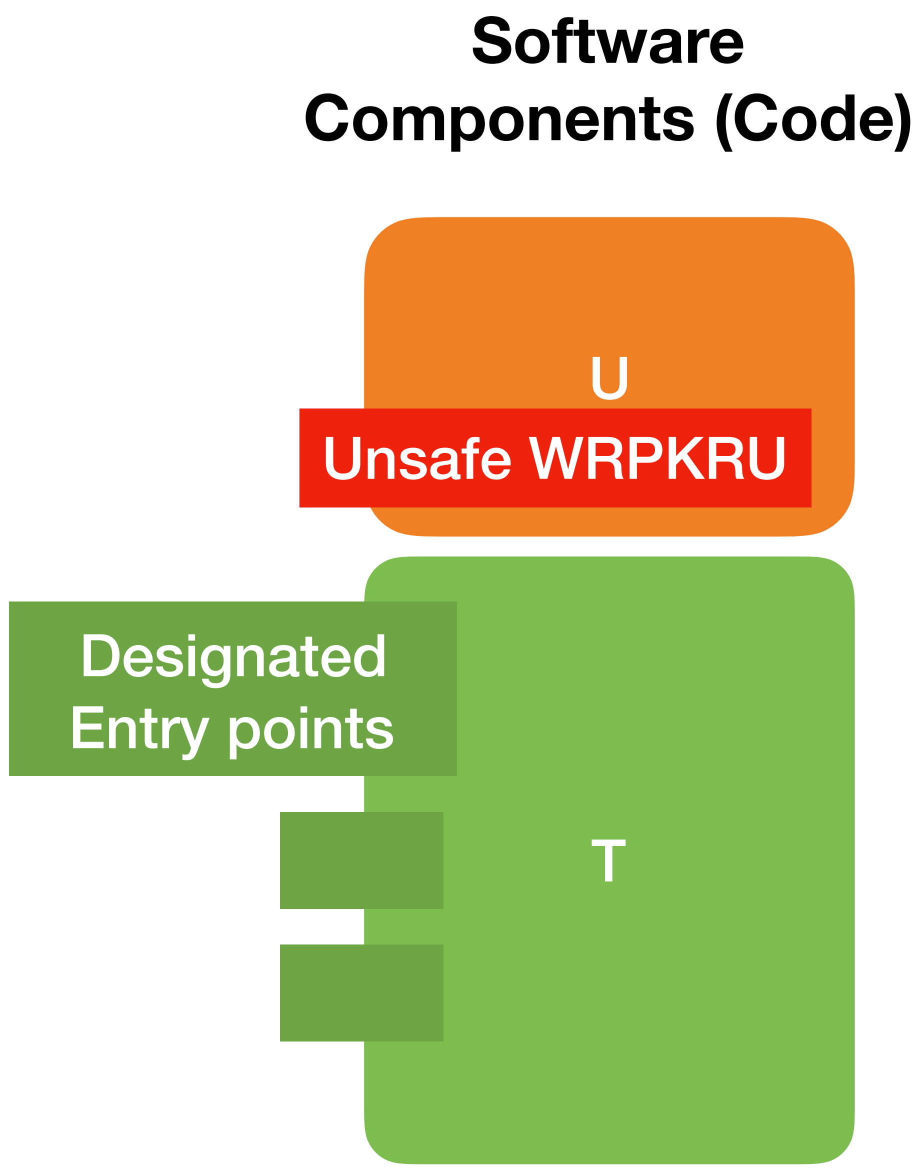
Unsafe Occurrence

- An occurrence of WRPKRU is considered “unsafe” if it is not immediately followed by:
  - a call to a designated entry point.
  - a check to confirm that it does not guarantee access to M_T .



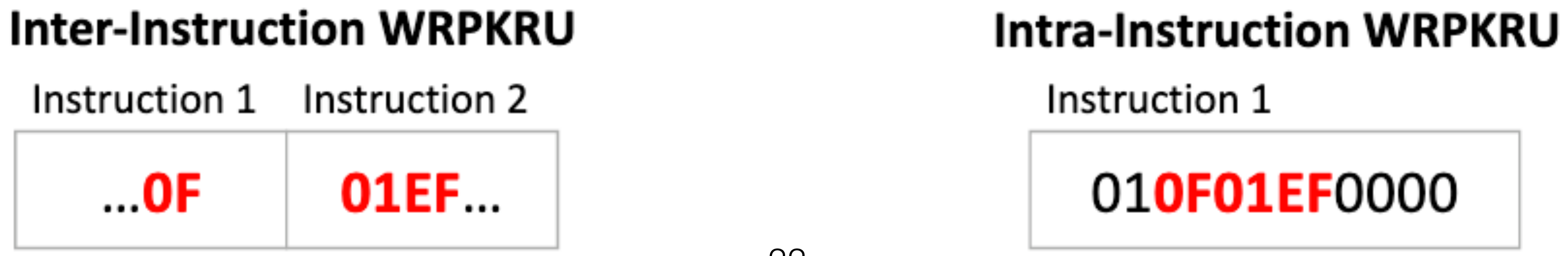
Binary Inspection

- An inspection function that verifies that a sequence of pages does not contain unsafe occurrences.
  - Symbol table is needed to determine the entry points.
- An interception mechanism that prevents U from mapping executable pages without inspection.
  - Using ptrace, bpf, or a LSM.



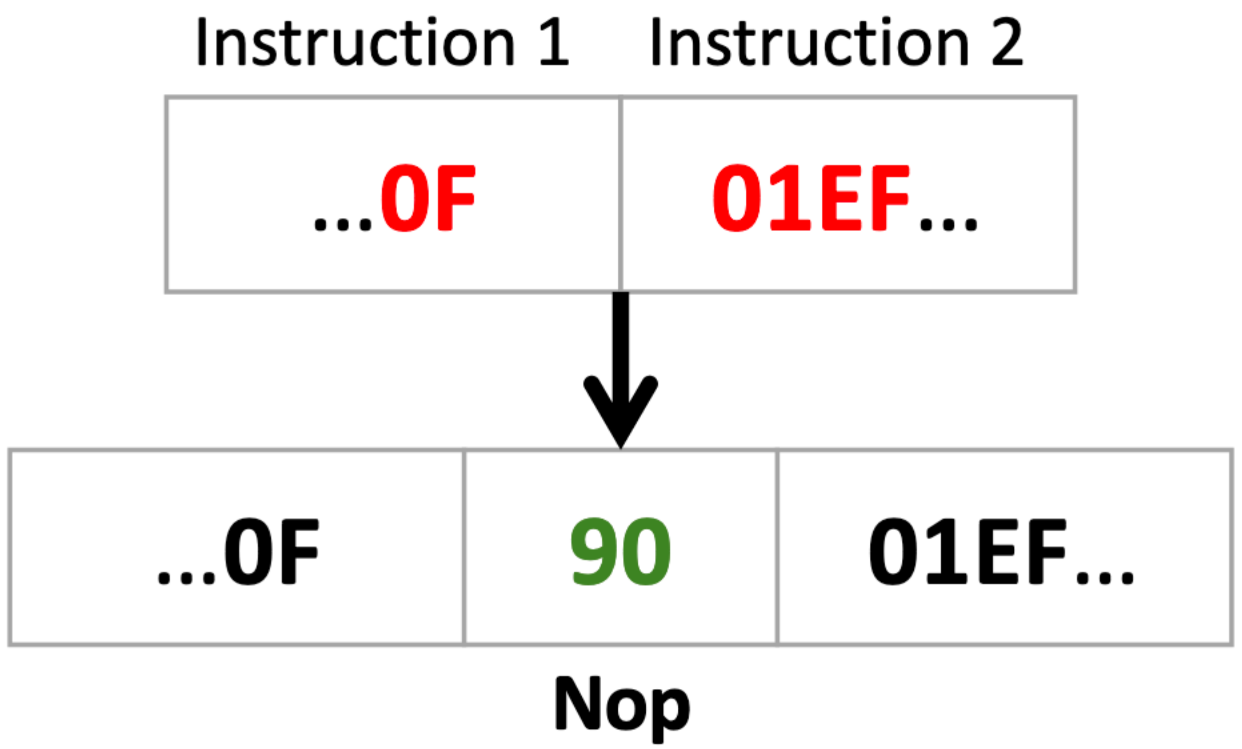
Binary Rewriting

- Byte sequence of WRPKRU: 0x0F01EF
- This sequence may:
  1. span two or more instructions.
  2. Appear entirely within a longer instruction.
- Eliminate unsafe occurrences of WRPKRU by binary rewriting at
  - compile time.
  - runtime prior to the execution.
  - Static binary rewriting for pre-compiled binaries.



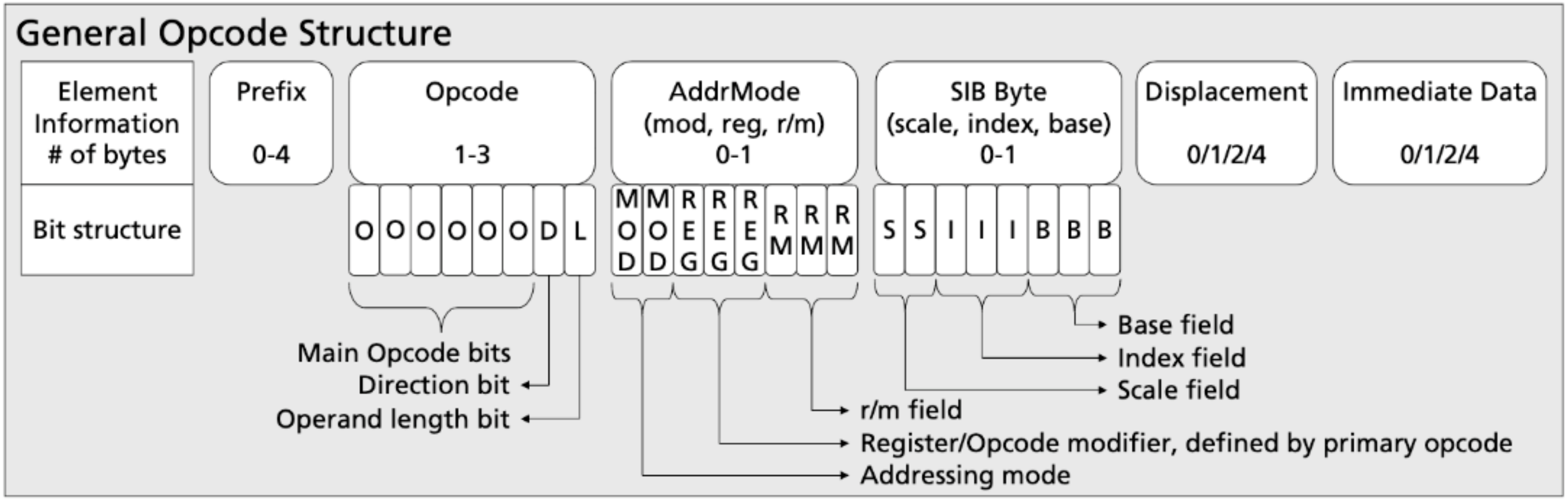
Rewriting Scheme (1)

- Rewriting rule for inter-instruction WRPKRU:



Rewriting Scheme (2)

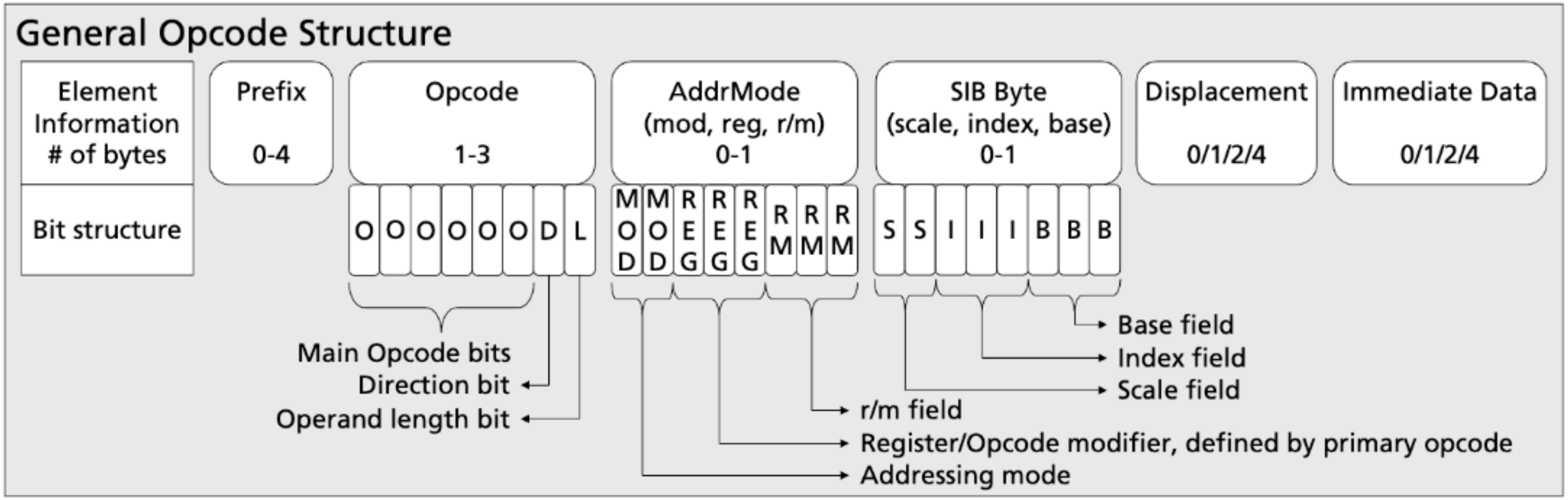
- If the sequence appears within an instruction, the rewriting rule depends on where WRPKRU locates.



Rewriting Scheme (2-1)

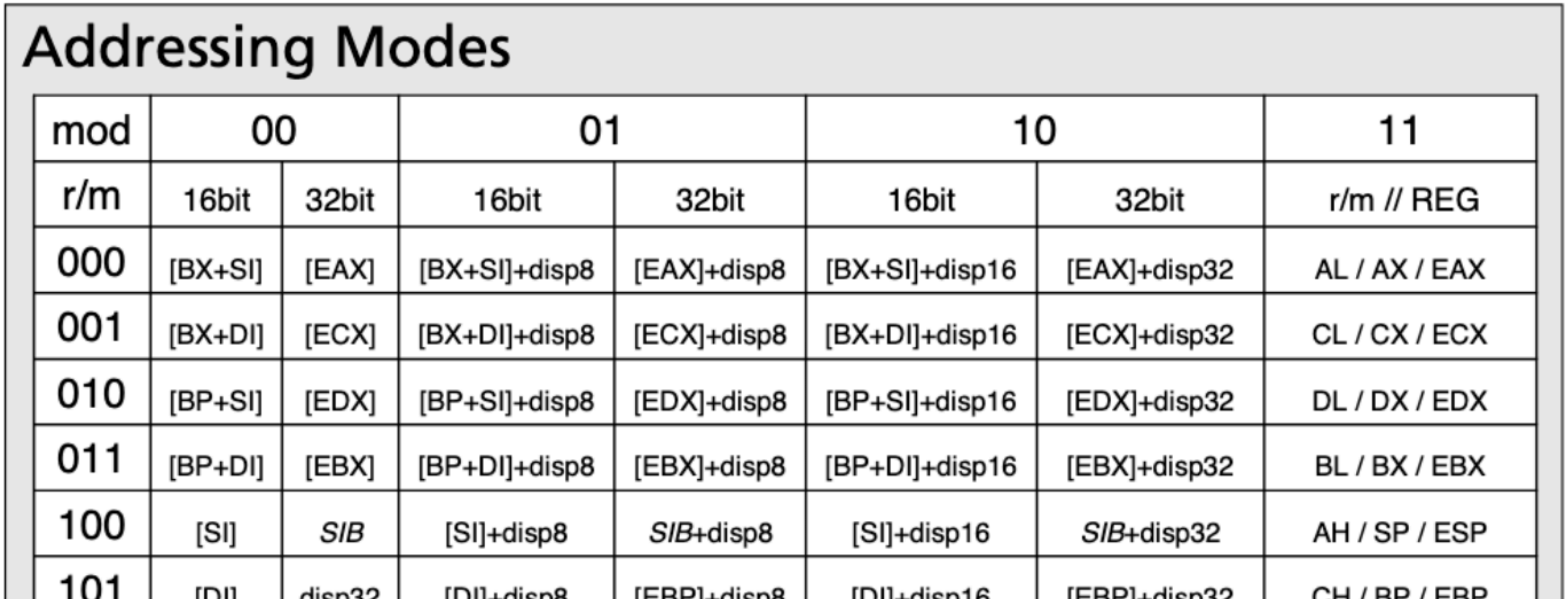
- If WRPKRU appears in the entire Opcode sequence, the instruction itself is WRPKRU.

  ➢ Insert the corresponding checks.



Rewriting Scheme (2-2)

- If WRPKRU overlaps with AddrMode:
  - Change to a free register
  - push/pop



Addressing Modes

mod	00		01		10		11
r/m	16bit	32bit	16bit	32bit	16bit	32bit	r/m // REG
000	[BX+SI]	[EAX]	[BX+SI]+disp8	[EAX]+disp8	[BX+SI]+disp16	[EAX]+disp32	AL / AX / EAX
001	[BX+DI]	[ECX]	[BX+DI]+disp8	[ECX]+disp8	[BX+DI]+disp16	[ECX]+disp32	CL / CX / ECX
010	[BP+SI]	[EDX]	[BP+SI]+disp8	[EDX]+disp8	[BP+SI]+disp16	[EDX]+disp32	DL / DX / EDX
011	[BP+DI]	[EBX]	[BP+DI]+disp8	[EBX]+disp8	[BP+DI]+disp16	[EBX]+disp32	BL / BX / EBX
100	[SI]	*SIB*	[SI]+disp8	*SIB*+disp8	[SI]+disp16	*SIB*+disp32	AH / SP / ESP
101	[DI]	disp32	[DI]+disp8	[EBP]+disp8	[DI]+disp16	[EBP]+disp32	CH / BP / EBP

Rewriting Scheme (2-3)

- If WRPKRU overlaps with displacement or immediate fields:

```
call [rip+0x0F01EF00]
```

Move the code code segment

```
add eax, 0x0F01EF00
```

```
push ebx
mov  ebx, 0x0F010000
add  ebx, 0x0000EF00
add  eax, ebx
pop  ebx
```

Implementations

- Dynamic:
  - Use direct jump to perform in-place rewriting.
- Static:
  - Use Dyninst to disassemble and rewrite those occurrences.
    - 1213 occurrences were found and rewritten over 204k of binaries.

Developing ERIM Applications

- Binary-only approach
    - LD_PRELOAD
    - T must be a dynamic symbol in the binary.
- Source approach
- Compiler approach
    - makes arbitrary inlining possible.

Developing ERIM Applications

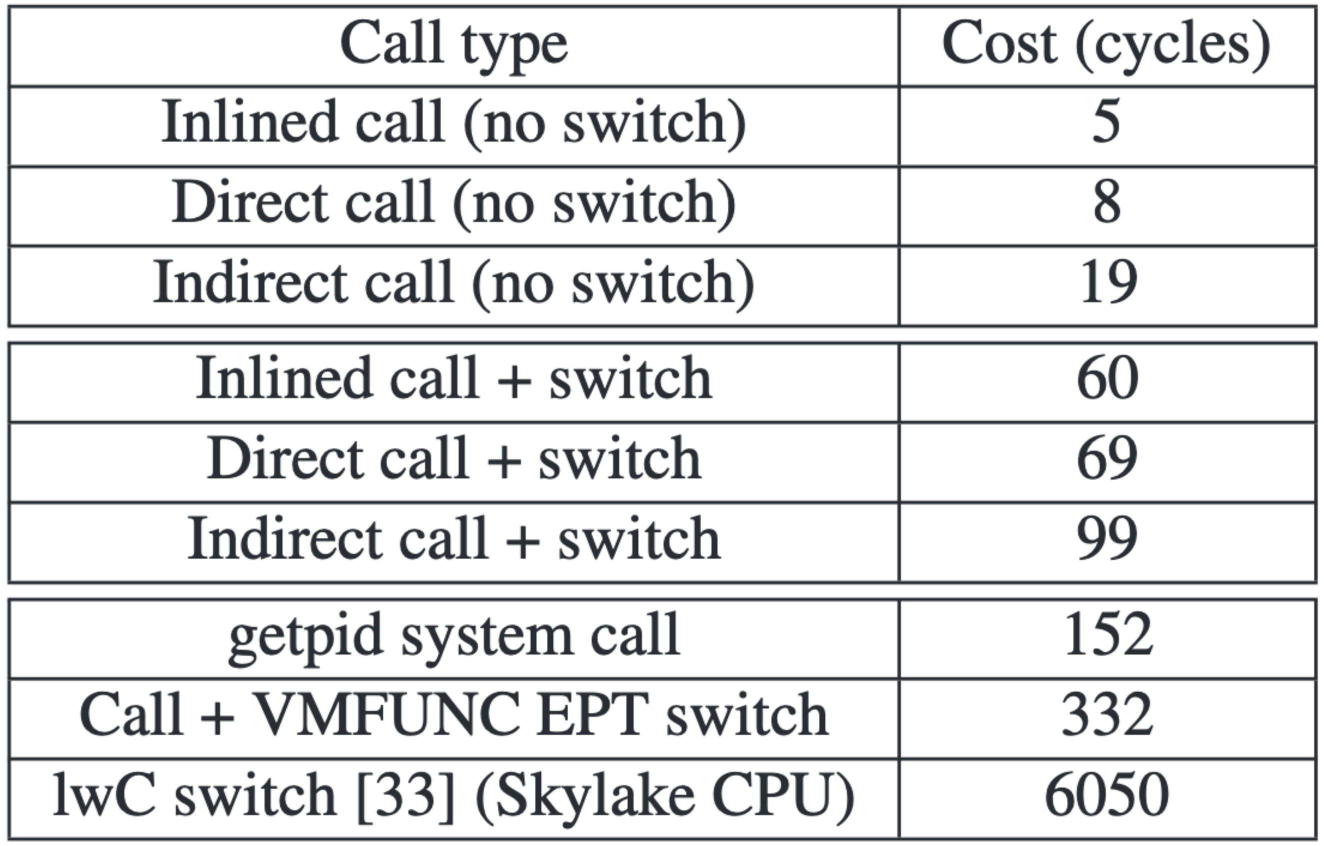
```
typedef struct secret {
  int number; } secret;
secret* initSecret() {
  ERIM_SWITCH_T;
  secret * s = malloc(sizeof(secret));
  s->number = random();
  ERIM_SWITCH_U;
  return s;
}
int compute(secret* s, int m) {
  int ret = 0;
  ERIM_SWITCH_T;
  ret = f(s->number, m);
  ERIM_SWITCH_U;
  return ret;
}
```

Evaluation

- Microbenchmarks
- Use Cases
- Comparing to Existing Techniques

Microbenchmarks

- Switch cost:



Call type	Cost (cycles)
Inlined call (no switch)	5
Direct call (no switch)	8
Indirect call (no switch)	19
Inlined call + switch	60
Direct call + switch	69
Indirect call + switch	99
getpid system call	152
Call + VMFUNC EPT switch	332
lwC switch [33] (Skylake CPU)	6050

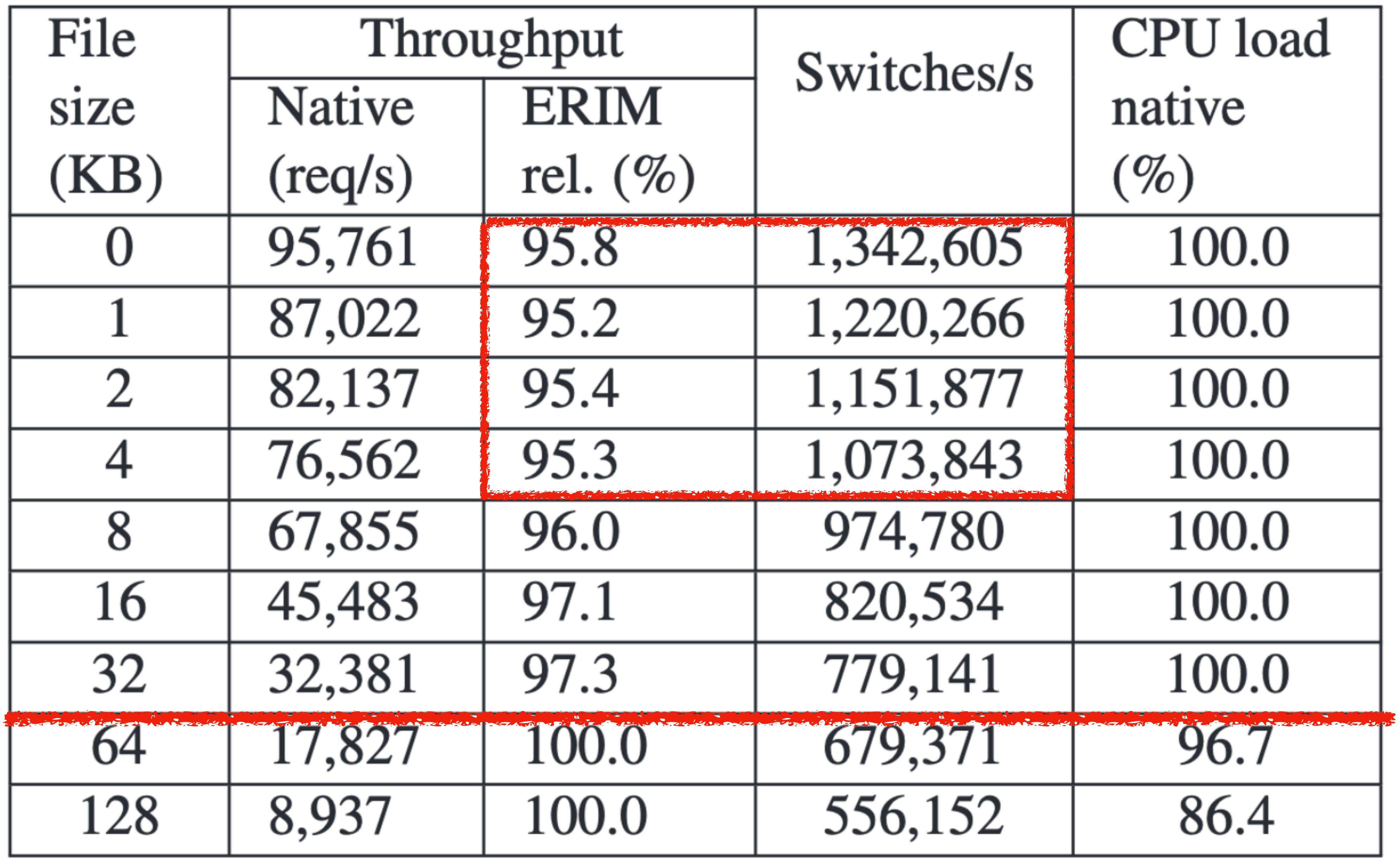
- Binary inspection:
    - 3.5~6.2 microseconds per page.

Use Cases

- Protecting session keys in NGINX
- Isolating managed runtimes
- Protecting sensitive data in CPI/CPS

Protecting session keys in NGINX

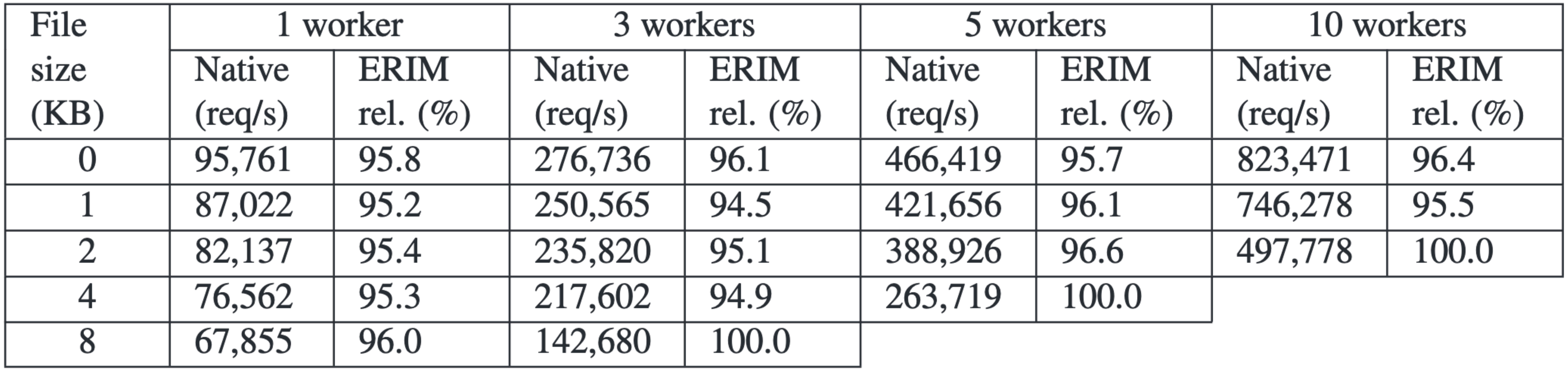
- Single worker:



File size (KB)	Throughput		Switches/s	CPU load native (%)
	Native (req/s)	ERIM rel. (%)		
0	95,761	95.8	1,342,605	100.0
1	87,022	95.2	1,220,266	100.0
2	82,137	95.4	1,151,877	100.0
4	76,562	95.3	1,073,843	100.0
8	67,855	96.0	974,780	100.0
16	45,483	97.1	820,534	100.0
32	32,381	97.3	779,141	100.0
64	17,827	100.0	679,371	96.7
128	8,937	100.0	556,152	86.4

Protecting session keys in NGINX

- Scaling with multiple workers:

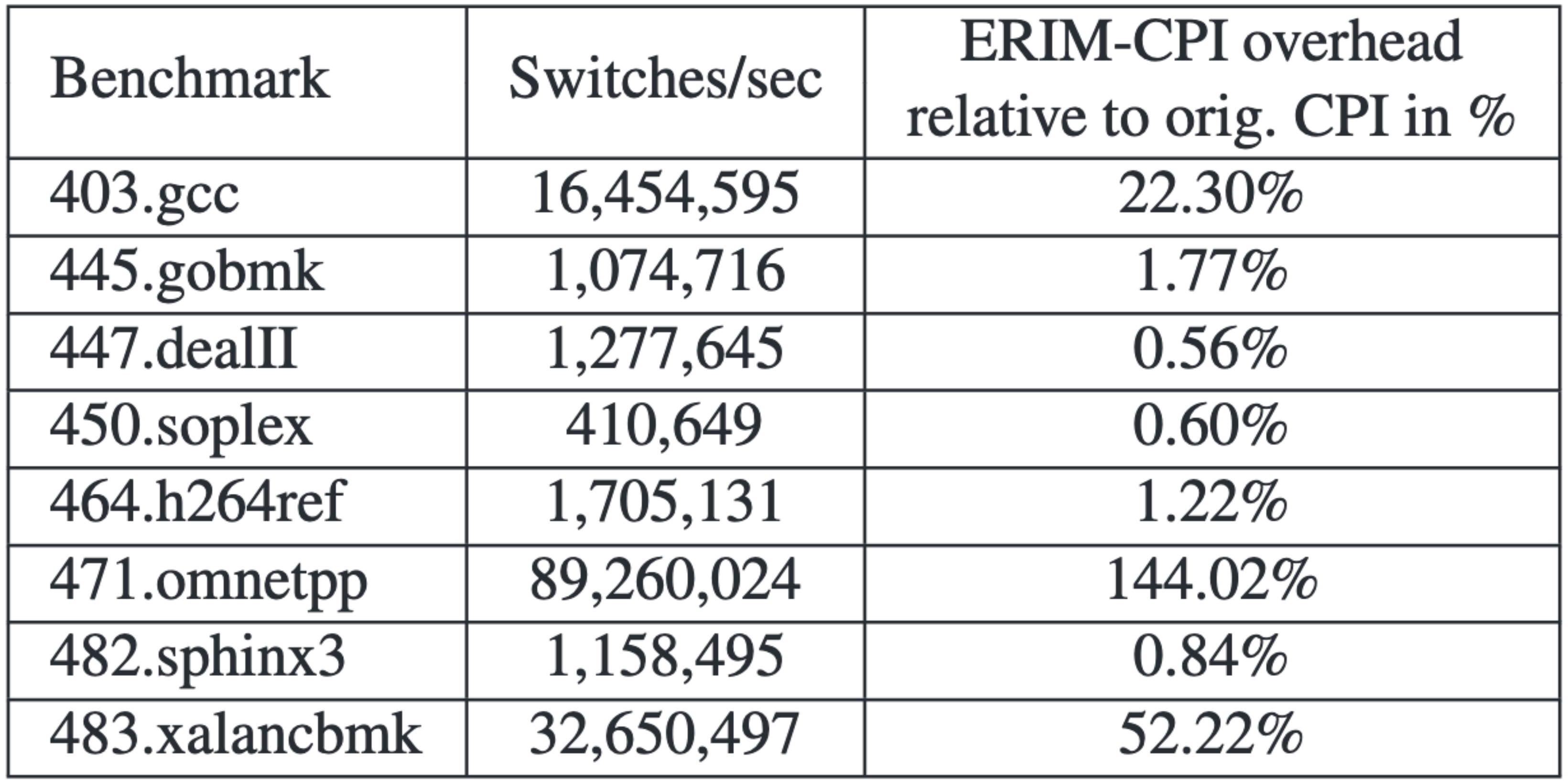


File size (KB)	1 worker		3 workers		5 workers		10 workers	
	Native (req/s)	ERIM rel. (%)	Native (req/s)	ERIM rel. (%)	Native (req/s)	ERIM rel. (%)	Native (req/s)	ERIM rel. (%)
0	95,761	95.8	276,736	96.1	466,419	95.7	823,471	96.4
1	87,022	95.2	250,565	94.5	421,656	96.1	746,278	95.5
2	82,137	95.4	235,820	95.1	388,926	96.6	497,778	100.0
4	76,562	95.3	217,602	94.9	263,719	100.0		
8	67,855	96.0	142,680	100.0				

- Protecting sensitive data in CPI/CPS

Protecting sensitive data in CPI/CPS

- Overhead becomes noticeable when switching rate exceed 10^6

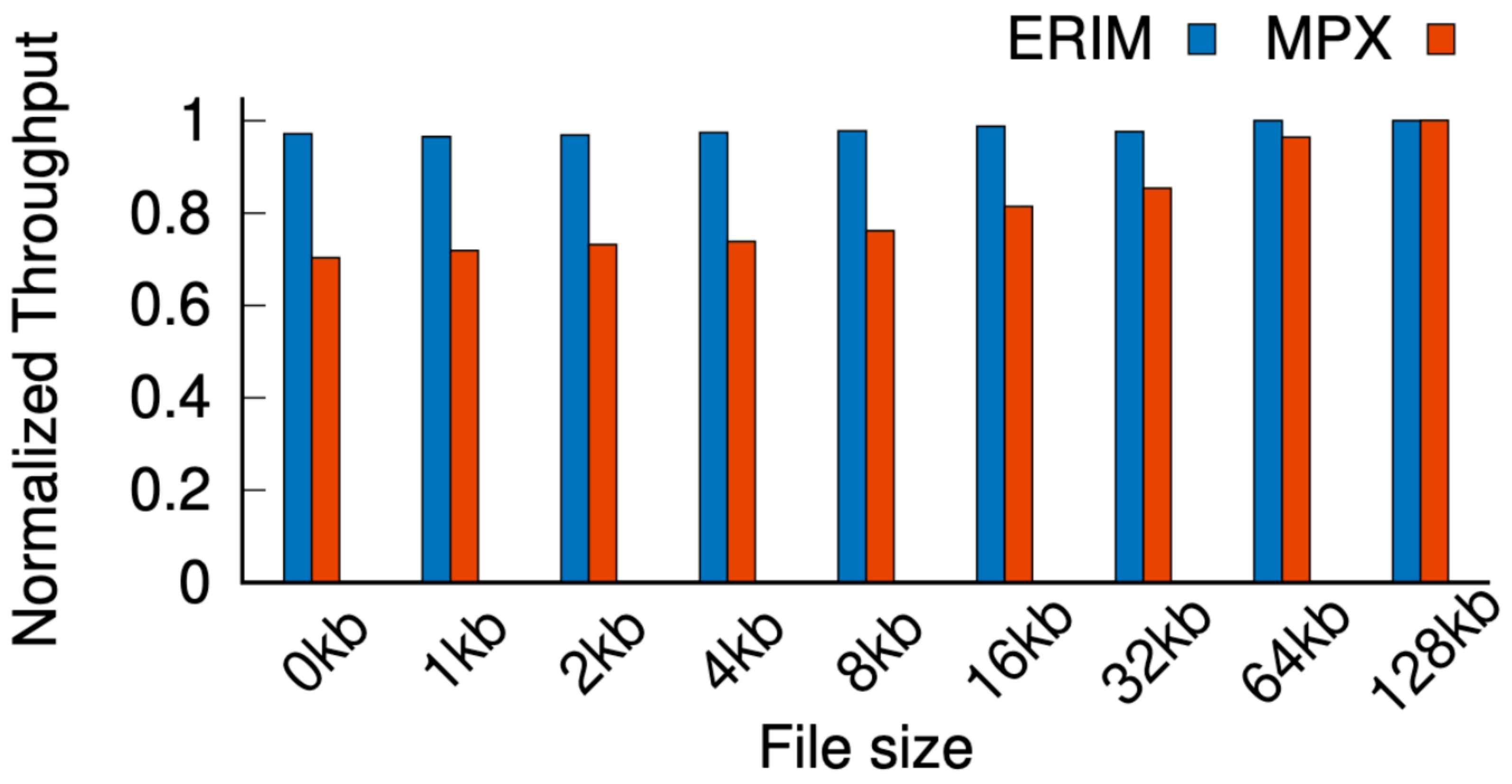


Benchmark	Switches/sec	ERIM-CPI overhead relative to orig. CPI in %
403.gcc	16,454,595	22.30%
445.gobmk	1,074,716	1.77%
447.dealII	1,277,645	0.56%
450.soplex	410,649	0.60%
464.h264ref	1,705,131	1.22%
471.omnetpp	89,260,024	144.02%
482.sphinx3	1,158,495	0.84%
483.xalancbmk	32,650,497	52.22%

Comparing to Existing Techniques

- SFI unsung MPX
    - Hardware bound checking
- VMFUNC
    - Hypervisor-based extended page table
- LwC
    - Separate address space in the same process

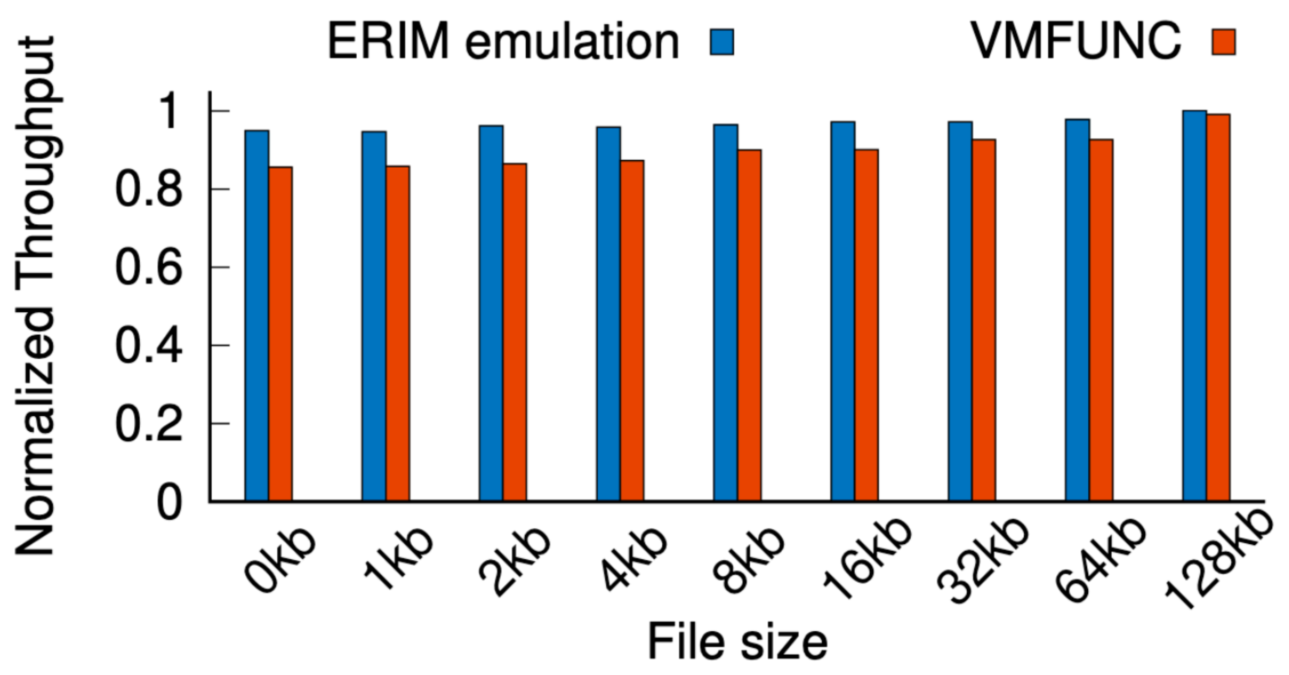
Comparing to Existing Techniques

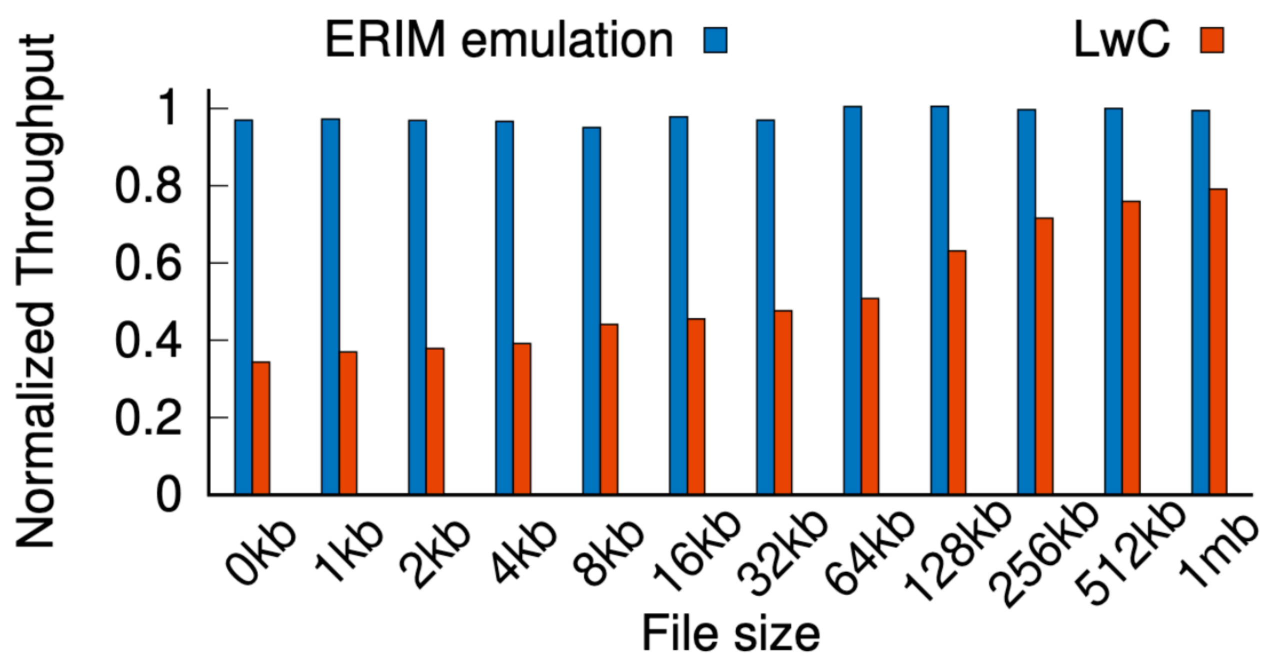


- MPX imposes an overhead during the execution of NGINX (compartment U)

- ERIN imposes an overhead on component switches.

Comparing to Existing Techniques





- VMFUNC EPT switch is faster than an OS process switch
- The use of EPT also induces an overhead on all syscalls and page walks in the VMFUNC isolation.

Conclusion

- ERIM provides hardware-enforced isolation with an overhead of less than 1% for every 100k switches/s between components.
    - ERIM switch cost is up to two orders of magnitude lower than that of kernel-page table isolation, and
    - Up to 3-5x lower than that of VMFUNC-based isolation.