Samsung Research

In-Kernel Control-Flow Integrity on Commodity OSes using ARM Pointer Authentication

Sungbae Yoo[1*], Jinbum Park[1*], Seolheui Kim[1], Yeji Kim[1], Taesoo Kim[1 2]

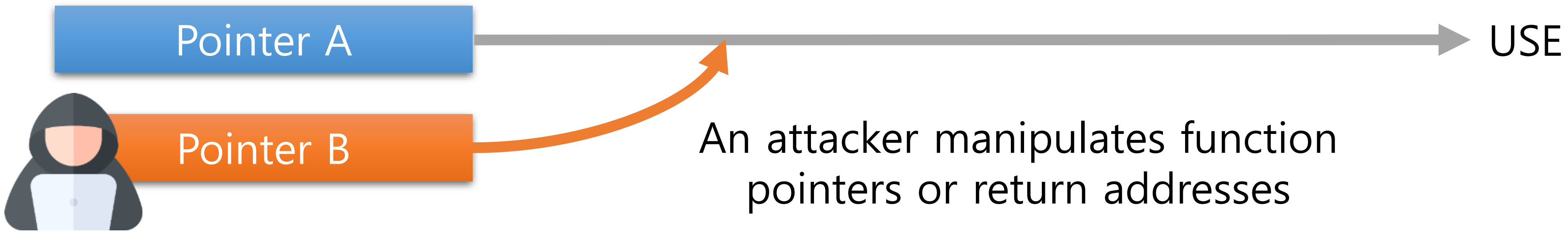
[1] Samsung Research

[2] Georgia Institute of Technology

[*] Primary Co-Authors

Problem: Memory Corruptions are Major Concern in OSes

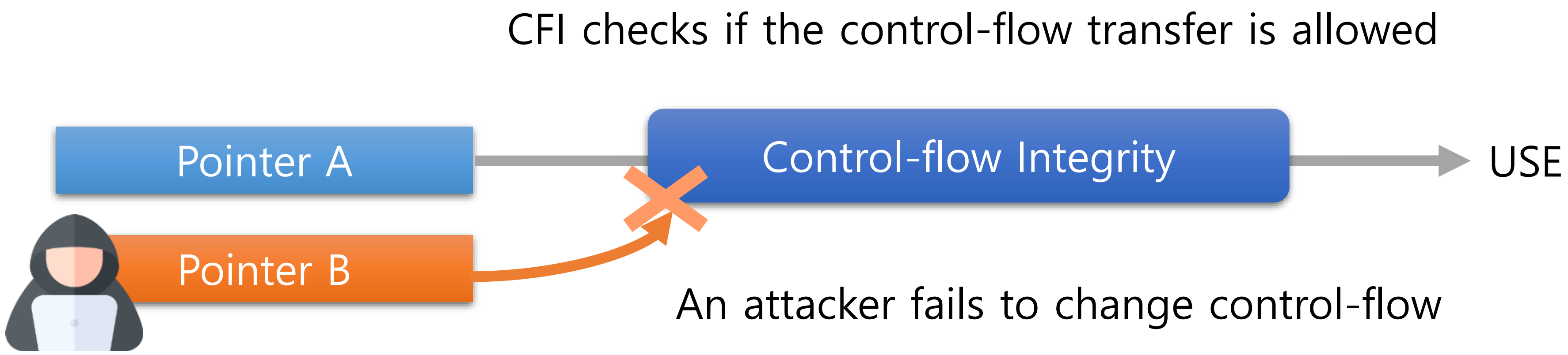
- 189 memory unsafe CVEs in Linux from 2021 to 2022
- Common attack vector: code-reuse attacks



Promising Defense: Control-flow Integrity (CFI)

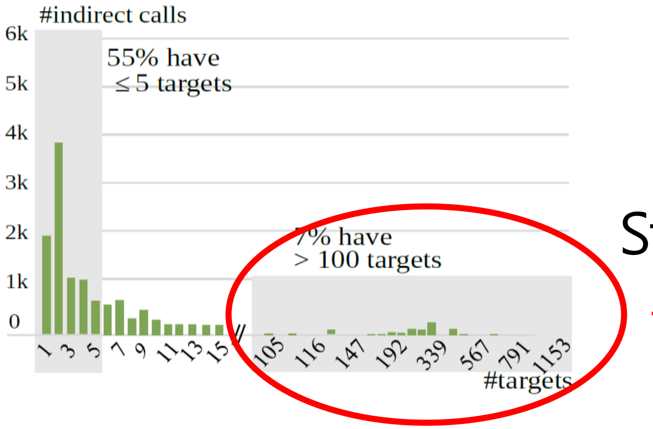
Samsung Research

- Ensures control-flow transfers remain intact at runtime



State-of-the-art of CFI for Commodity OSes

- Type-based CFI
  - Google's



Still shows large number of allowed target
→ Problem: too course-grained

- Hardware-based CFI
  - iOS Kernel, PARTS, PATTER

Examining Pointer Authentication on the iPhone XS

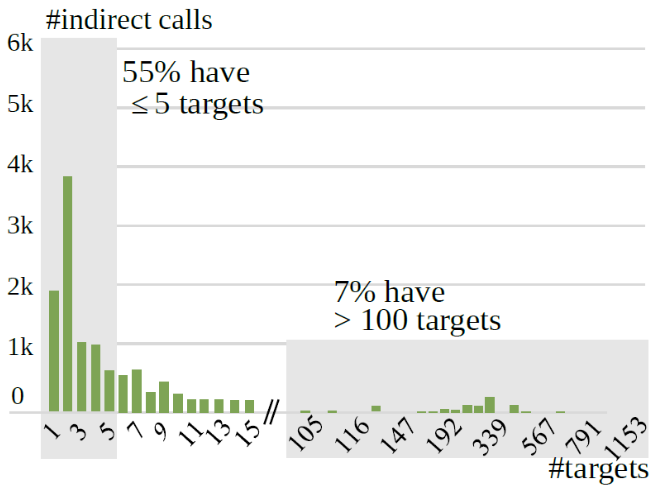
Posted by Brandon Azad, Project Zero

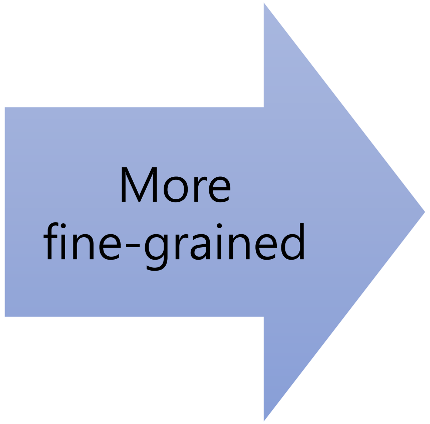
Several vulnerabilities to misuse HW
→ Problem: too error-prone

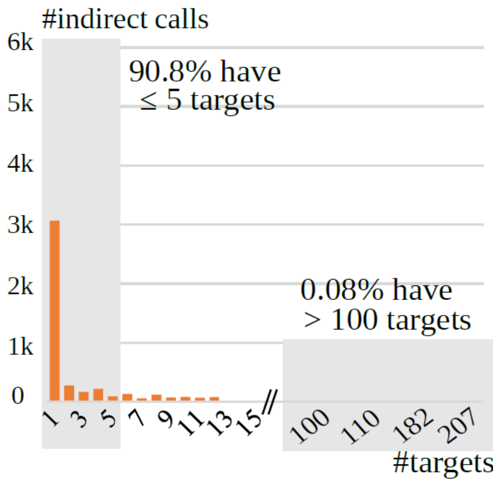
Our Approach: Fine-grained CFI with Hardware Support

- Key idea: Leveraging the common design idioms in OSes

➢ Approach 1: Adopting the latest HW-based protection
➢ Approach 2: Static validator to avoid mistakes



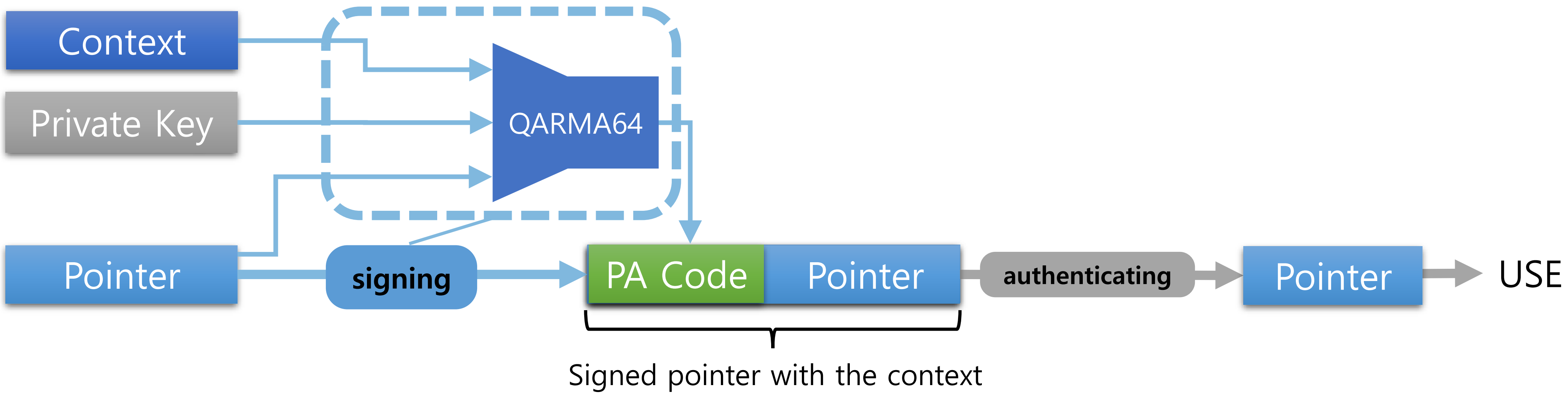




Key Enabler: ARM Pointer Authentication (PA)

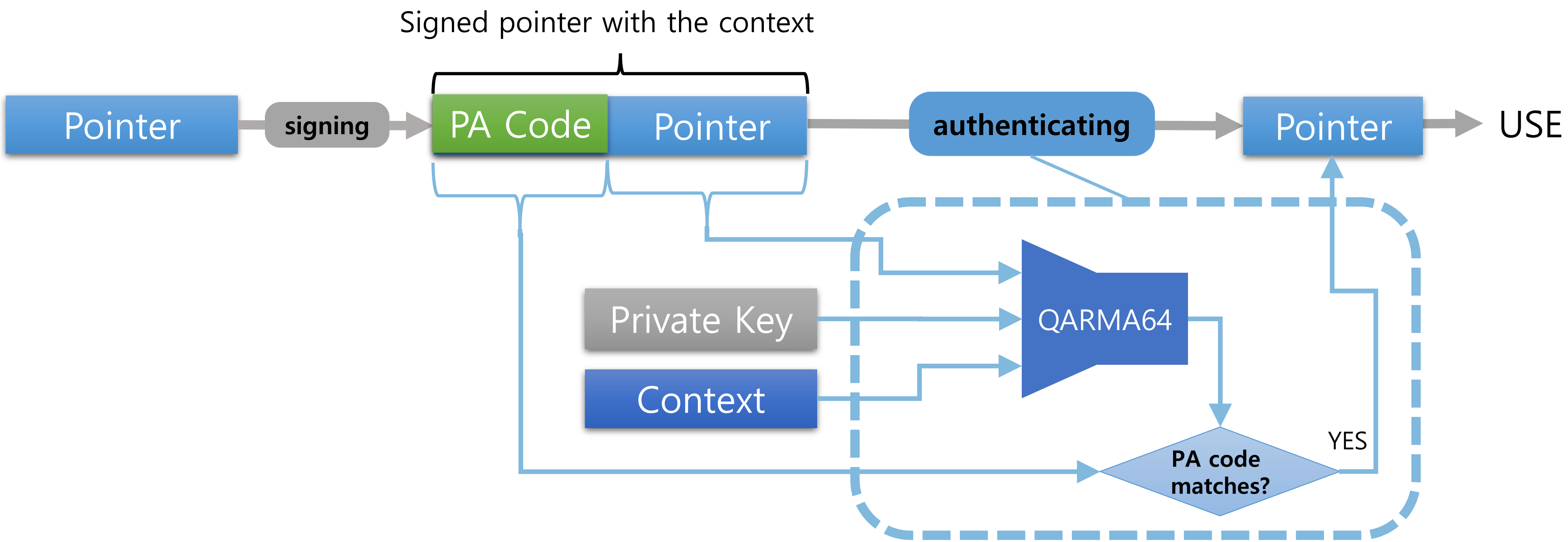
- ARM PA ensures the integrity of pointers at runtime

- PAC signs a pointer



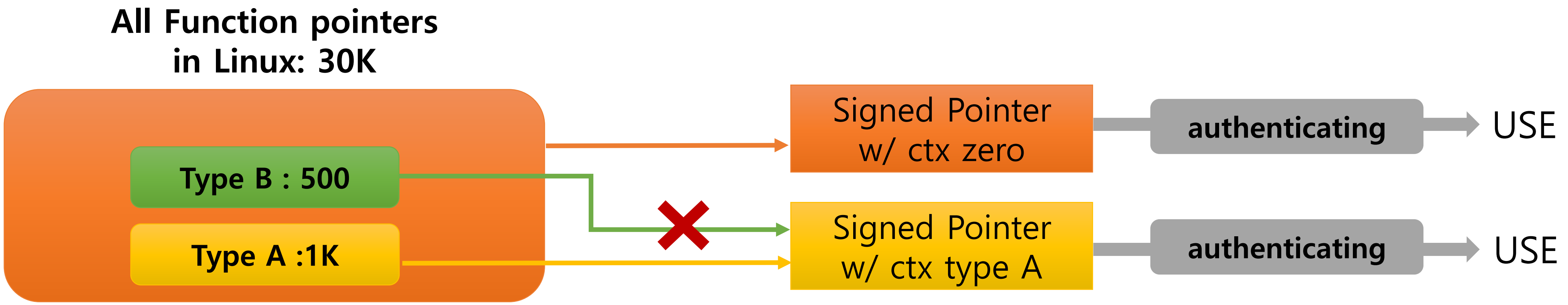
Key Enabler: ARM Pointer Authentication (PA)

- ARM PA ensures the integrity of pointers at runtime
- AUT checks the integrity of a pointer and restores the pointer



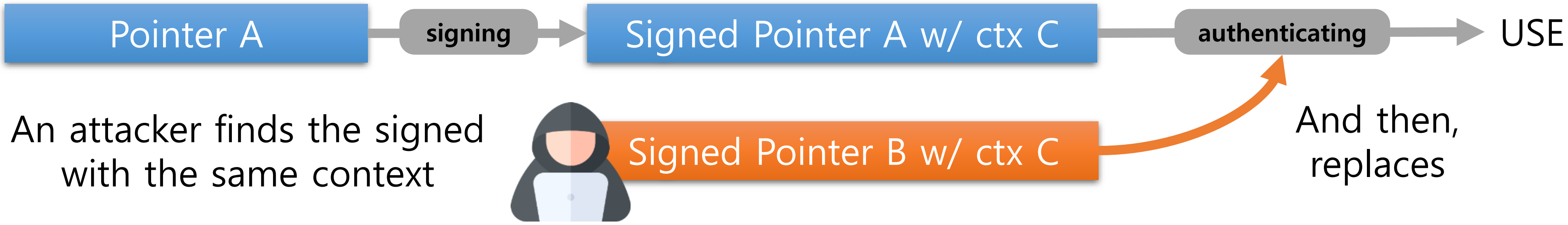
How to properly Set “context” for Better Precision?

- Naïve solution: using zero
  - # allowed targets : 30K – in Linux
- Strawman solution: using type
  - Max. # allowed targets : 1K - int (*) (struct platform_device *) in Linux



Attack Vectors: Replaying or Substitution

- Re-uses an indirect call with the same context



Solution: Using more Idiom in Kernel Objects

- An example of actual code in Linux

```
int __init arch_timer_register(void) {
  …
  err = request_percpu_irq(ppi,
                           arch_timer_handler_phys,
                           "arch_timer", …);
  …
}
```

How can we make the context for this as unique as possible?

Function pointer type

```
Int request_percpu_irq(unsigned int irq,
                       irq_handler_t handler,
                       const char *devname, …) {
  struct irqaction *action = …;
  action->name = devname;
  action->handler = handler;
  …
}
```

Structure name

Constant field using as a identifier

```
struct irqaction {
  irq_handler_t handler;
  const char *name;
  …
}
```

Solution: Using more Idiom in Kernel Objects

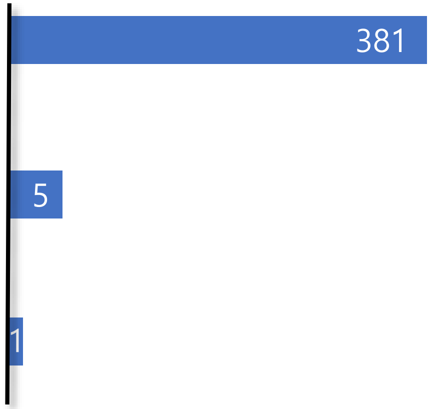
- Unique, Invariant, Movable (compatible with memcpy)

```
struct irqaction {
  irq_handler_t handler;
  const char *name;
}
```

```
void func1() {
  struct irqaction *o = …;
  o->name = "o1";
  o->handler = &target;
}
```

Layer	Context
Type	Irqhandler_t
⊕	
Object	struct.irqaction
⊕	
Objbind	o->name ("o1")

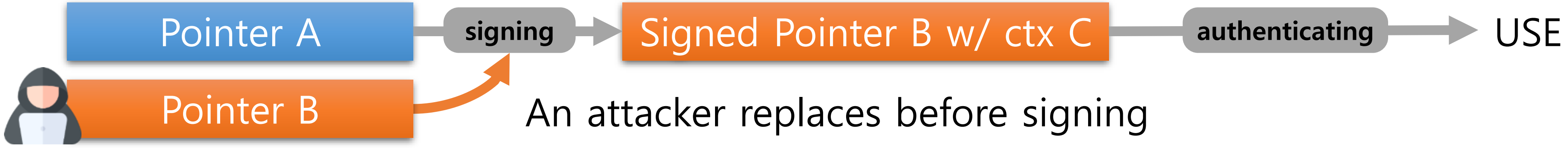
PAC(&target, context)



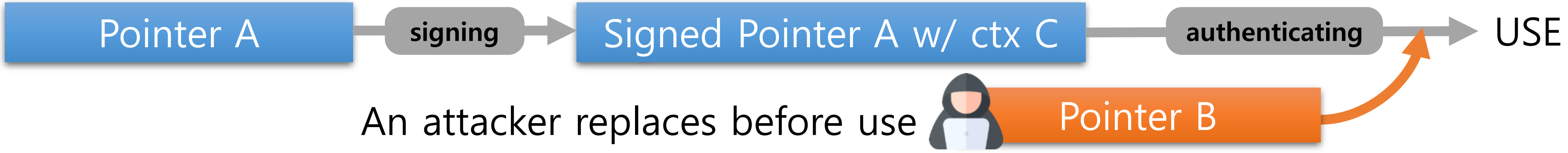
The number of allowed targets

Two Other Attack Vectors: Forging and TOCTOU

- Forging attack
  - Generates a signed pointer using signing gadgets



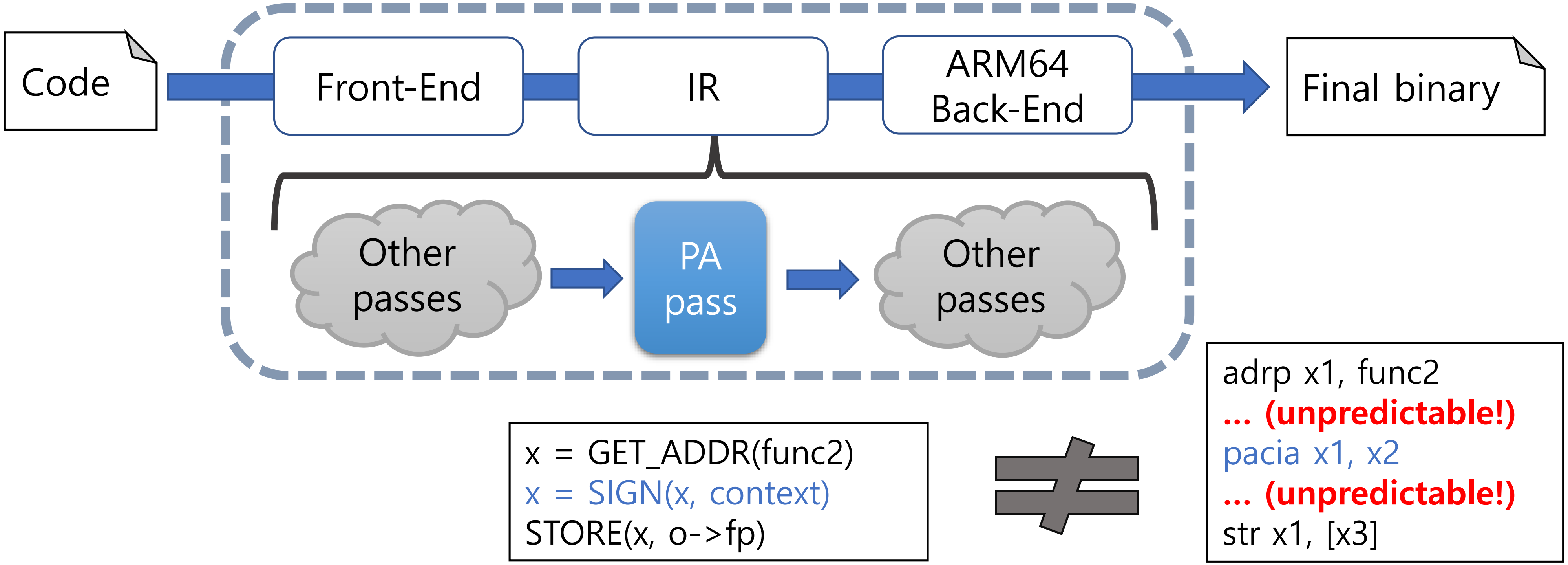
- Time-of-check to time-of-use (TOCTOU)
  - Manipulates spilled and restored pointers before it uses



Problem: Complex optimization passes in the compiler inadvertently causes the bugs!

Problem: Complex Optimization Passes in Compiler

- Highly sophisticated modern compiler frameworks
  - Unpredictable produced binaries
    - Optimizations could spill out registers to memory



Static Validator: Correctness Check of the Final Binary

1. Complete protection
   : All indirect branches have to be authenticated before use

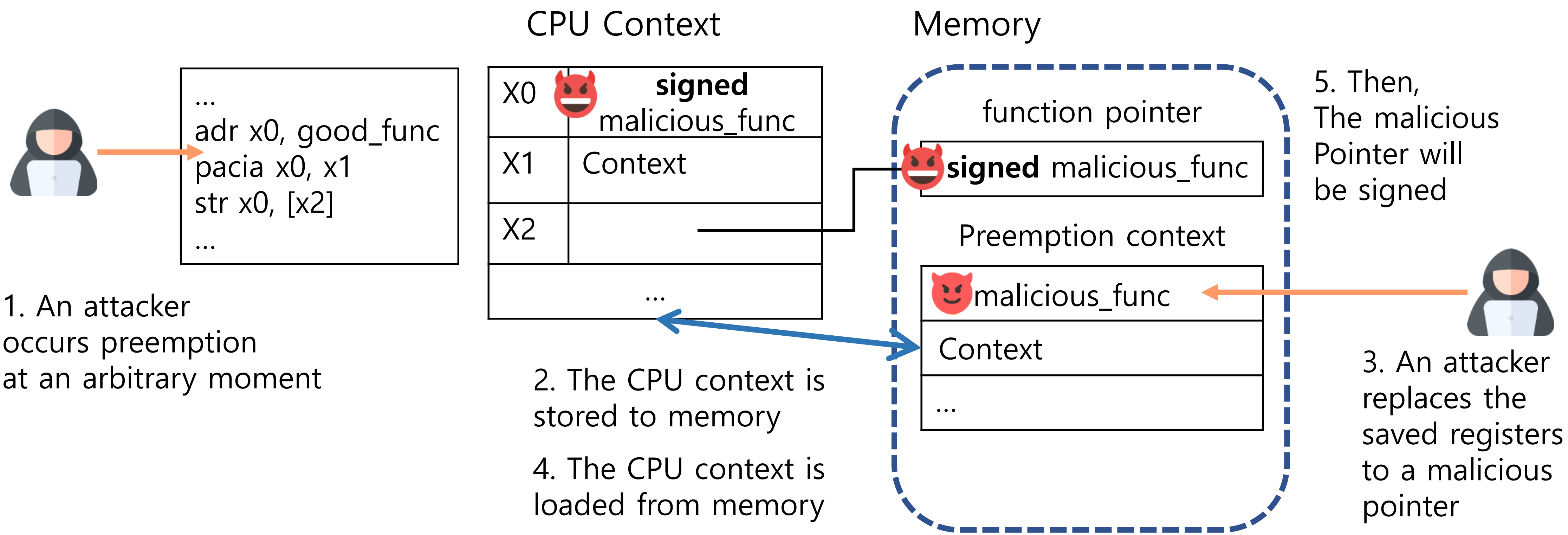
2. No time-of-check-time-of-use (TOCTOU)
   : Raw pointers after PA instructions are never stored back in memory

3. No signing oracle
   : There must be no gadget that signs an attacker-chosen pointer

4. No unchecked control-flow change
   : All direct modifications of program counter register must be validated

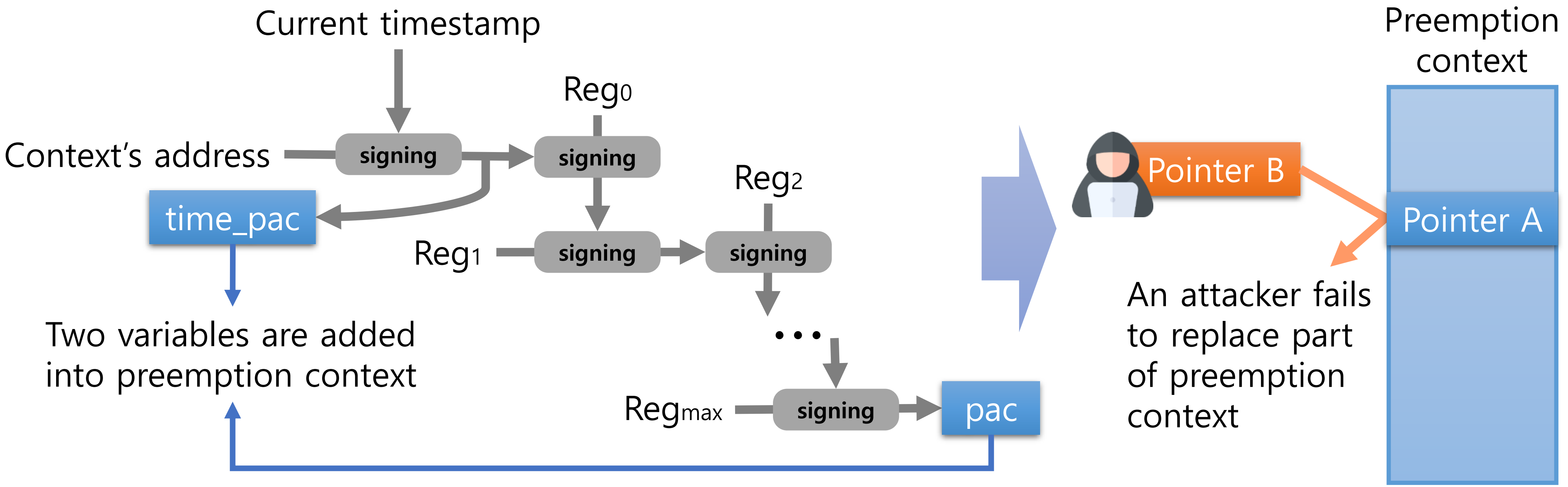
Problem: Preemption Hijacking Attack

- Attackers can occur preemption when they want in kernel
- Preemption context save/restore can be used as a signing oracle



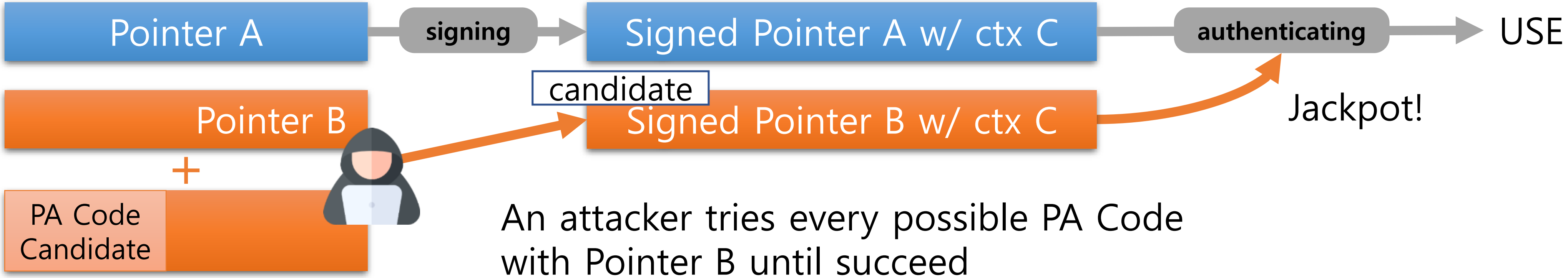
Solution : Preemption Context Protection

- Whole preemption context signing via key-chaining technique
  - Prevents substitution attack to part of preemption context



Another Attack Vector : Brute-forcing in Kernel

- Enumerates all possible PA code bits (generally 2^{15})

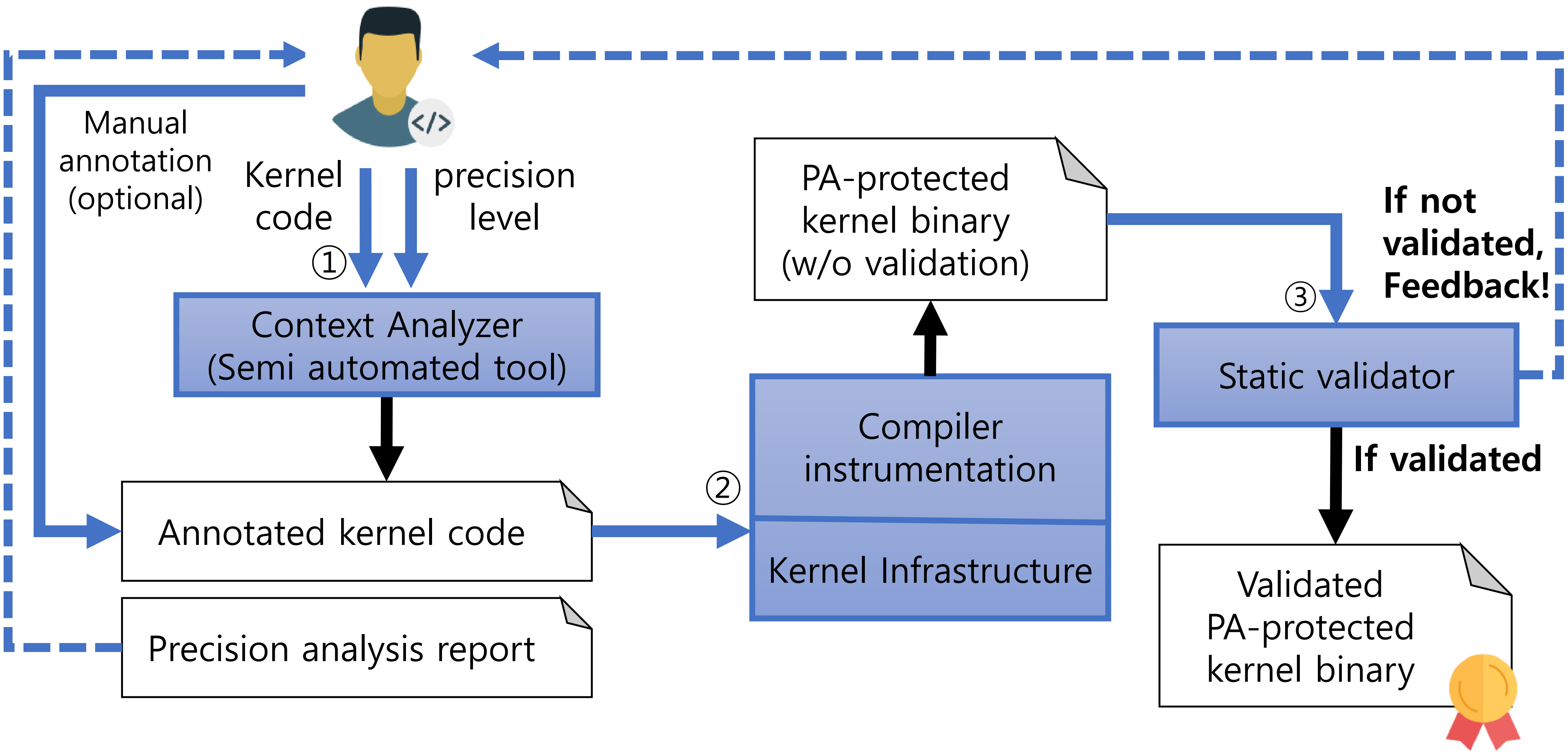


An attacker tries every possible PA Code with Pointer B until succeed

Solution: If an attack is detected, ~~just panicking~~ inadequate for kernel giving delays with increasing exponentially

System Overview: PAL

Samsung Research



Implementation

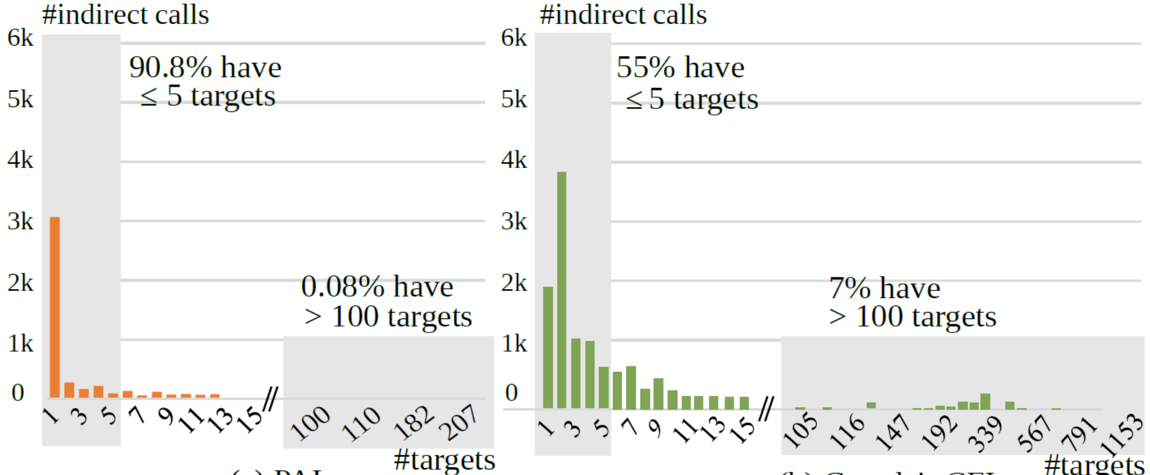
- Applied to Linux(Tizen, Apple M1 mini), FreeBSD

- PAL
  - GCC plugin (forward-edge) : 3,632 LoC (C++)
  - GCC (backward-edge) : 127 LoC changes
  - Static validator : 848 LoC (Python)
  - Context analyzer : 1943 Loc (C++)

- Infrastructure
  - Linux: 491 LoC changes
  - FreeBSD: 258 LoC changes

Evaluation – Comparing with other approaches

- Google's - Allowed targets for indirect calls



(a) PAL (b) Google's CFI

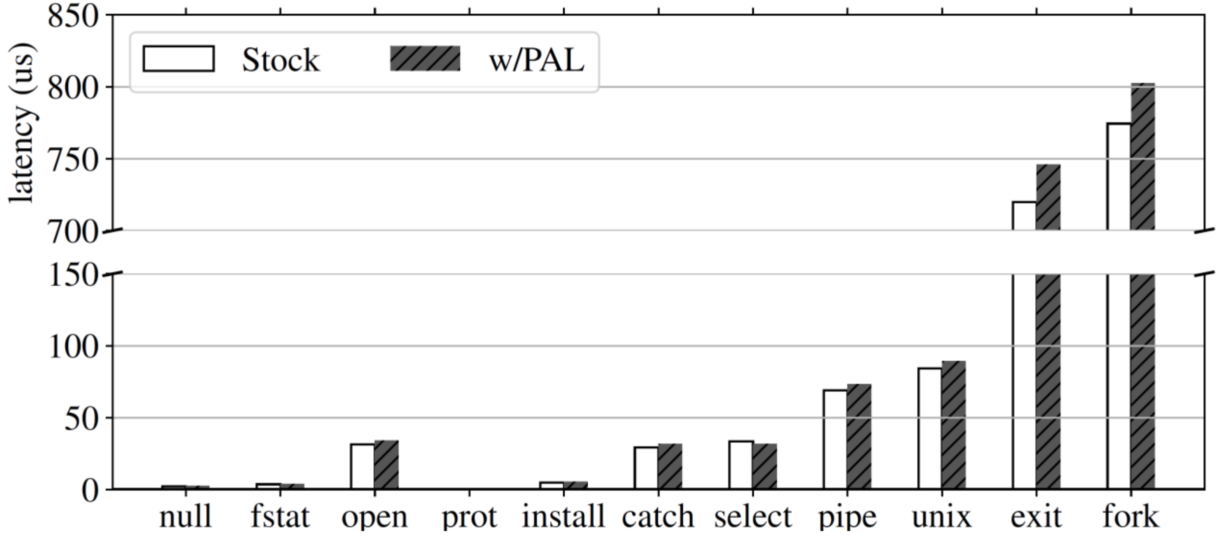
#targets	Google's	PAL		
		+Type	+Object	+objbind
≤5	55.0%	84.9%	88.6%	90.8%
>100	7.0%	2.8%	1.6%	0.08%
Max	1,153	35,264	30,622	207

- iOS kernel – Indirect calls sharing the same context

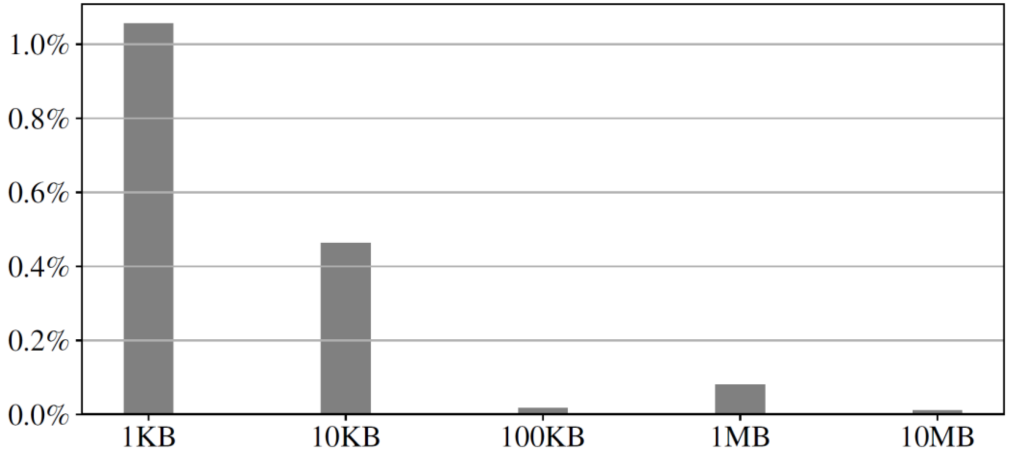
#contexts	iOS Kernel	PAL
≤5	62.2 %	94.9 %
>100	21.2 %	0.0 %
Max	6,513	70

Evaluation - Performance

- Micro-benchmark : LMBench
  - Latency: 0-3μs (median. 7%)



- Macro-benchmark : Apache
  - RPi3: 1.06%, Mac mini: 0.75%



- Binary increase

	5.12.0-rc-1/Mac mini	4.19.49/RPi3	FreeBSD/Qemu
Stock	123.5 MB	19.9 MB	5.9 MB
w/ PAL	130.7 MB	23.0 MB	6.4 MB
Overhead	7.2 / 5.8%	3.1 / 15.6%	0.5 / 8.5%

Conclusion

- PAL is a new in-kernel CFI based on ARM PA
  - Leverage the common design idioms in OSes
  - Check the correctness of the final binary
- PAL considers kernel's characteristics such as preemption
- PAL is fully evaluated on real HW supporting ARM PA
  - Negligible overhead in most workloads

Samsung Research

Thank you

Contacts :
sungbae.yoo@samsung.com / ysbnim@gmail.com
jinb.park@samsung.com / jinb.park7@gmail.com

Source code (To be released)
https://github.com/SamsungLabs/PALinux