black hat ASIA 2022

Taking Kernel Hardening to the Next Level

Jinbum Park[1,2], Haehyun Cho[3], Sungbae Yoo[1],
Seolheui Kim[1], Yeji Kim[1], Taesoo Kim[1,4], Bumhan Kim[2]

Samsung Research[1] Samsung Mobile Security[2] Soongsil University[3] Georgia Tech[4]

Introduction

- Memory safety issues are the foremost security problems in today's operating systems.

- A lot of defenses have been proposed to prevent bugs from exploitation. But, all of them is still having a hard time balancing between security and performance.

- Of those defenses, we focus on the two, CFI (Control-Flow Integrity) and UAF (Use-After-Free) Defense and aim to take both to the next level.

Overview (CFI)

NOTE: We deal with ARM Pointer-Authentication based CFI.

- Problem statement
  - A strong security likely breaks compatibility.
  - A wrong compiler implementation can exhibit a severe bug.

- Pain points in the state-of-the-art
  - iOS kernel CFI: Low security for function pointers in C.
  - Other proposals from academia: High security, but breaking compatibility.

- Our new approach
  - PAL, to the rescue of the above pain points, (to appear at USENIX Security 2022), and targets C-based commodity OSes.

Overview (UAF Defense)

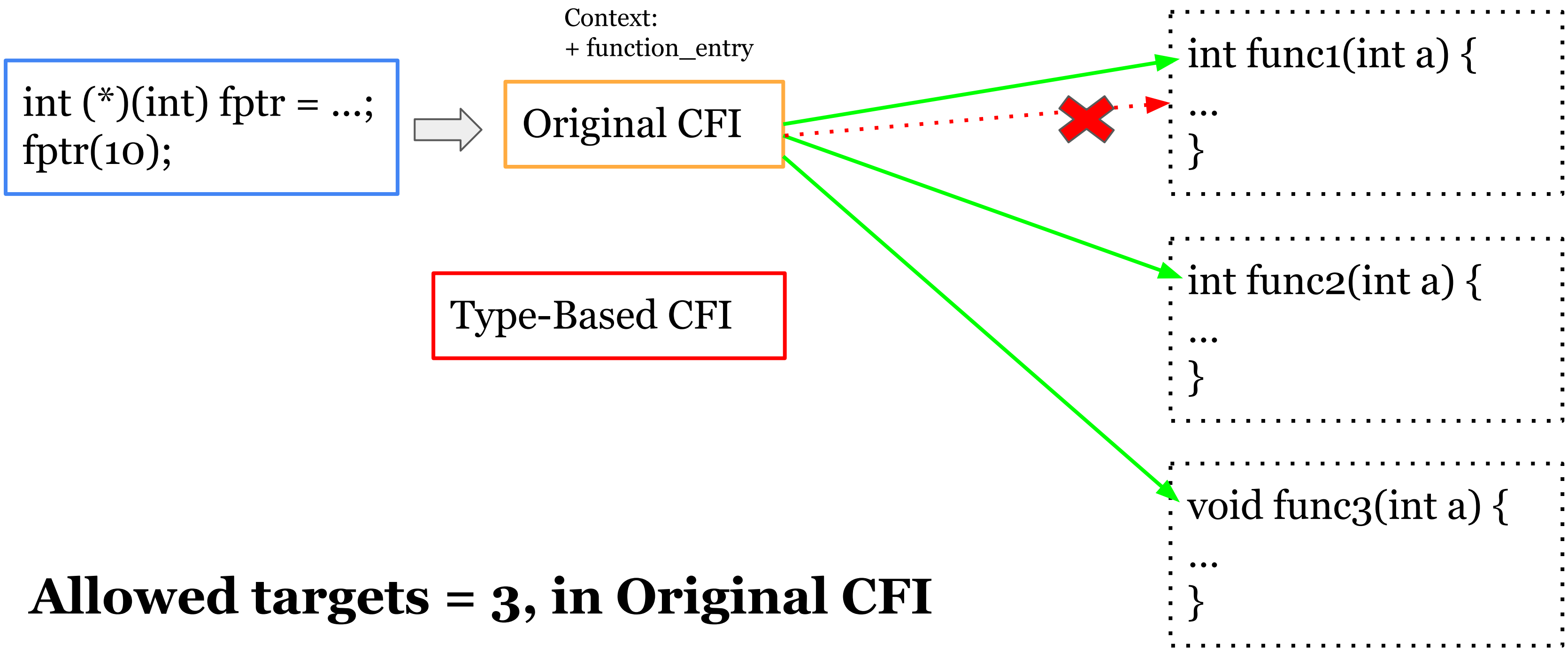
- Problem statement
  - All proposed defenses only care about user-space apps, not kernels.
- Pain points in the state-of-the-art
  - A strong security comes with an unbearable memory or performance overhead.
- Our new approach
  - ViK, to the rescue of the above pain point. (published at ASPLOS 2022)

ARM PA-based Kernel CFI

(PA: Pointer Authentication)

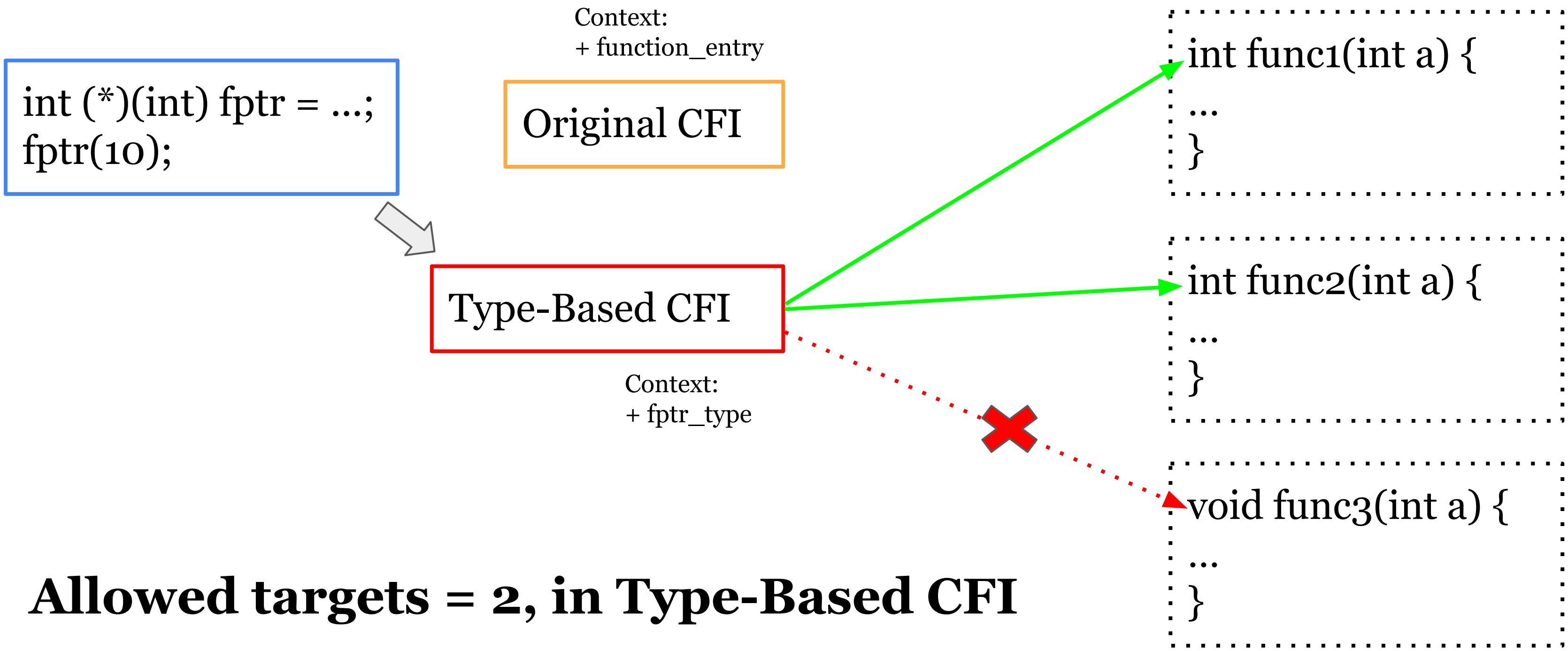
Background

CFI (Control-Flow Integrity)



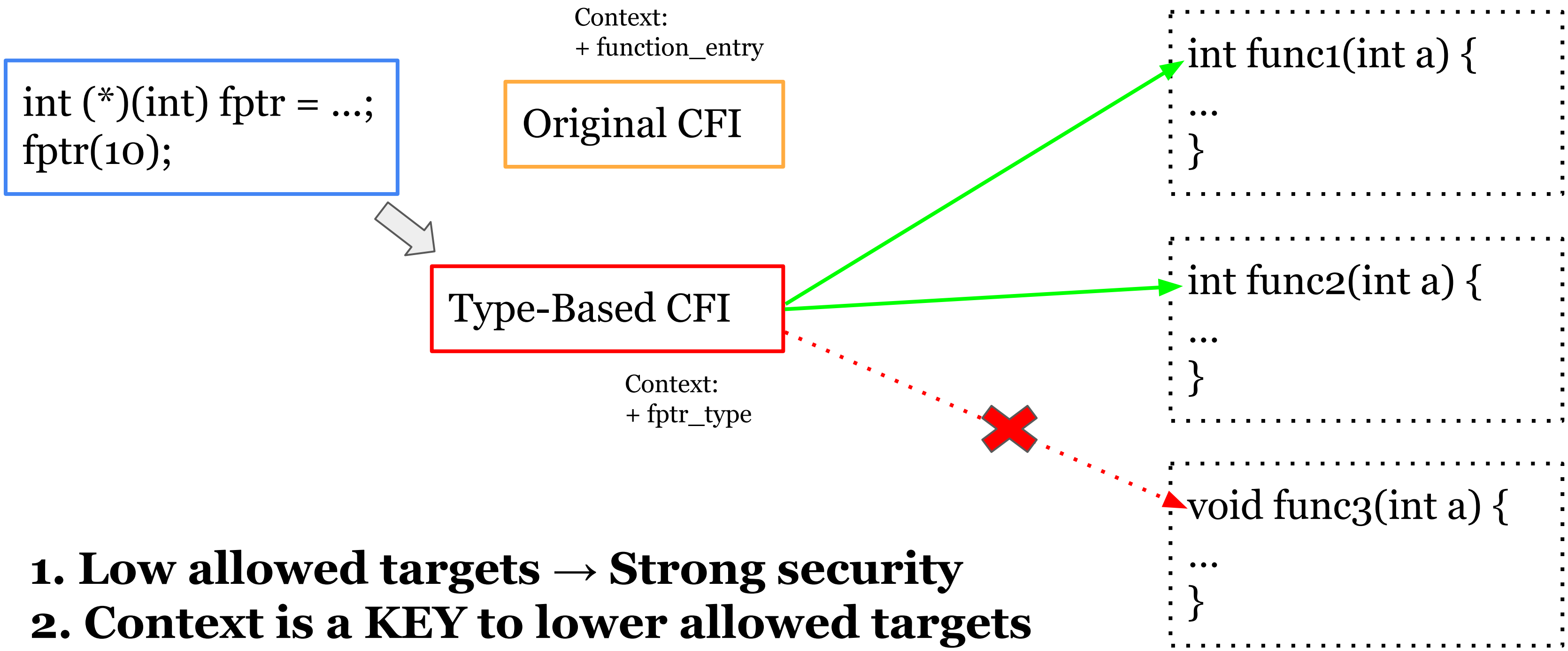
Allowed targets = 3, in Original CFI

CFI (Control-Flow Integrity)



Allowed targets = 2, in Type-Based CFI

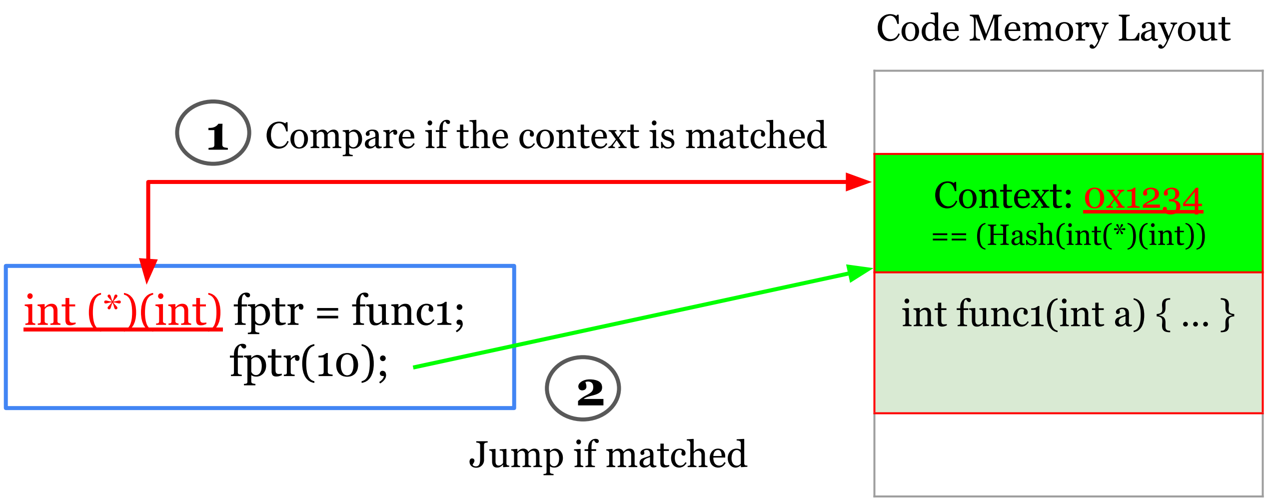
CFI (Control-Flow Integrity)



1. Low allowed targets → Strong security
2. Context is a KEY to lower allowed targets

ARM PA (Pointer Authentication)

Type-based CFI implementation without ARM PA



Downside:

- Every indirect call demands one more memory access to the stored context.

ARM PA (Pointer Authentication)

Type-based CFI implementation with ARM PA (Sign)

:: int func1(int a) {}

```
int (*)(int) fptr = func1;
....
....
fptr(10);
```

→ GEN: pacia func1, hash(int(*)(int)) → fptr = 0xabcd....c000

pacxx function_pointer, context, where xx is a key selector.

→ **QARMA (function_pointer) with (context, xx_key) => pac + pointer**

ARM PA (Pointer Authentication)

Type-based CFI implementation with ARM PA (Auth)

:: int func1(int a) {}

```
int (*)(int) fptr = func1;      →  GEN:  pacia func1, hash(int(*)(int))  →  fptr = 0xabcd....c000
....                                        ↕ Context matched!                  ↓
....
fptr(10);                       →  USE:  autia fptr, hash(int(*)(int))  →  fptr = 0x0000....c000

                                                                          →  fptr(10);
```

autxx function_pointer, context, where xx is a key selector.

→ **QARMA (function_pointer) with (context, xx_key) => pointer**

ARM PA (Pointer Authentication)

Type-based CFI implementation with ARM PA (Auth)

:: void func1(int) {}

```
int (*)(int) fptr = func1;   →  GEN:  pacia func1, hash(void(*)(int))  →  fptr = 0xbcde....c000
....                                                                     ↓
....                          Context mismatched!
fptr(10);                     →  USE:  autia fptr, hash(int(*)(int))   →  fptr = 0xcab9....c000
                                                                         →  fptr(10); → Crash!
```

If key and context are not matched between GEN and USE, system crash arises!

Pain point-1:
A poor security (a low CFI precision)

Two aspects of Context

- A good context helps improving security while not sacrificing compatibility.

- Two aspects of context
  - Unique: more unique, more secure
  - Invariant: if invariant, likely no compatibility issue

- We have to find a good context considering these two aspects.

Type-based Kernel CFI

- PARTS (USENIX Security 19) proposes a type-based CFI using ARM PA for the first time.
- Android kCFI (kernel CFI) also uses a type-based CFI.
- Context evaluation
  - Unique? → Not that much.. (e.g., TROP (ACSAC 2018))
  - Invariant? → Yes! i.e., no compatibility issue

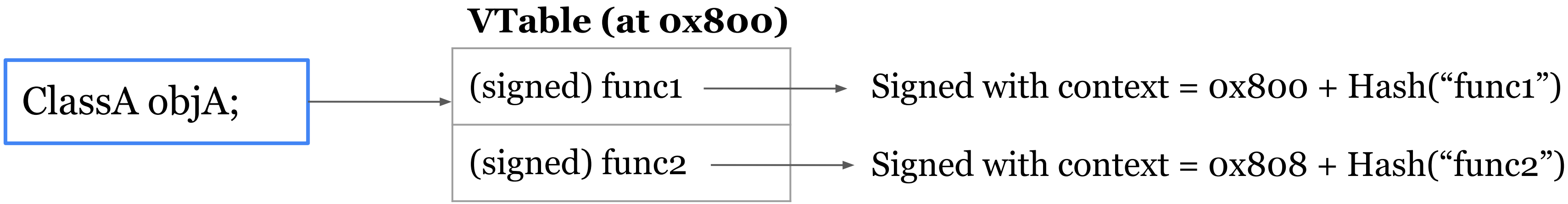
iOS Kernel CFI (PA-based)

- iOS Kernel is made up of different languages, C++ and C and Objective-C.

- iOS Kernel CFI uses fine-grained contexts for C++ and Objective-C (i.e., strong security), but not for C.
  - This is why iOS Kernel CFI is not applicable to C-based OSes. (Linux)

iOS Kernel CFI (PA-based)

How iOS CFI deals with its C++ function pointers (VTable)



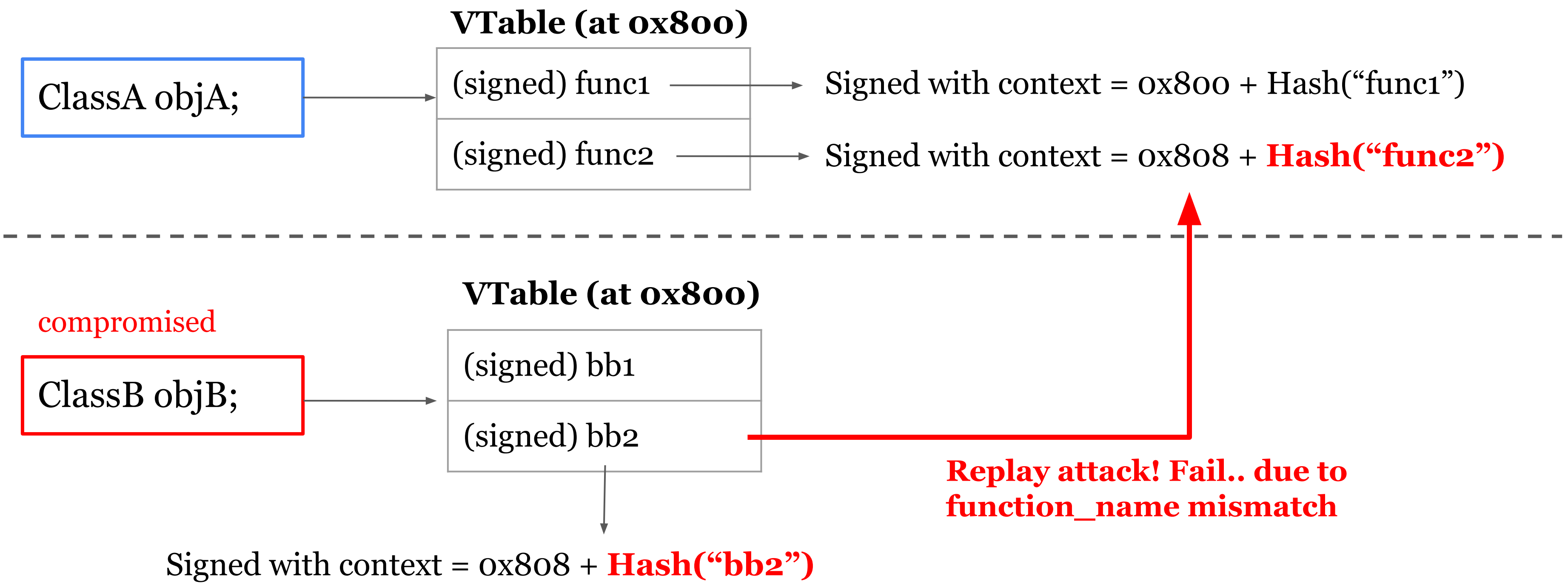
The context of a VTable entry
= Storage Address + Hash(function_name)

So powerful combination of dynamic and static context-!

black hat ASIA 2022

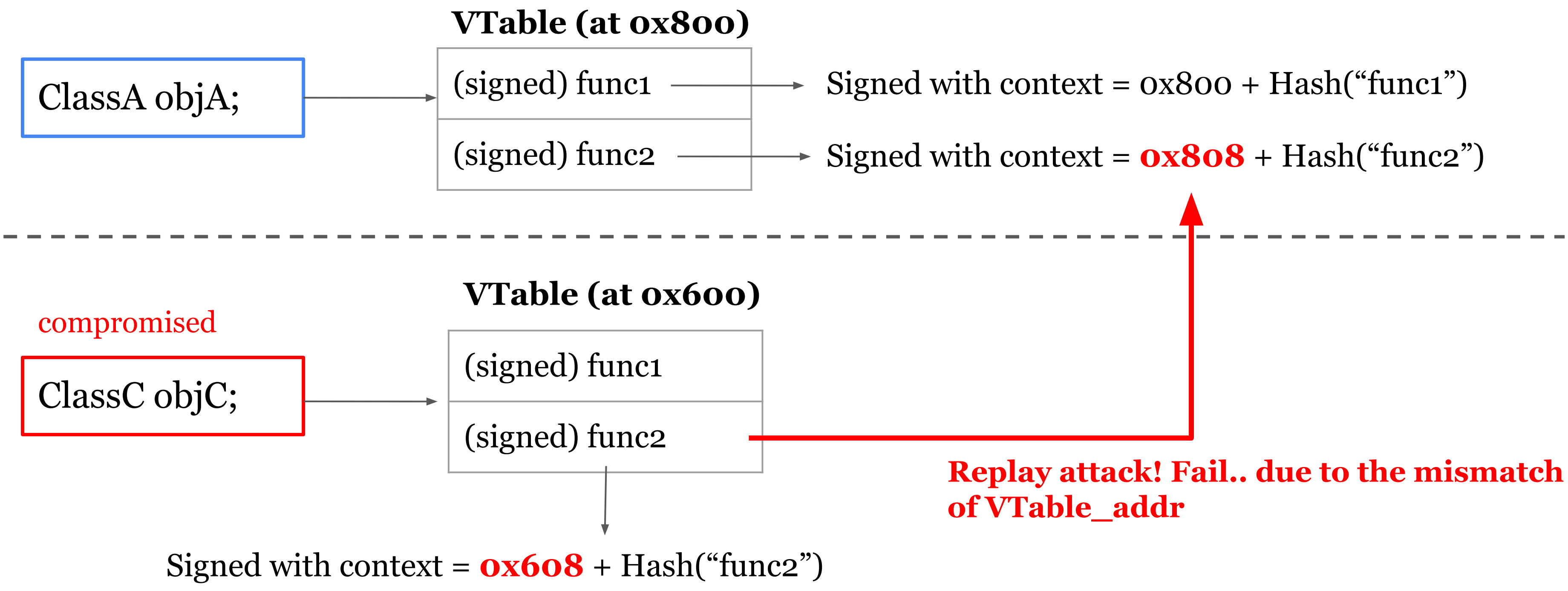
iOS Kernel CFI (PA-based)

How iOS CFI deals with its C++ function pointers (VTable)



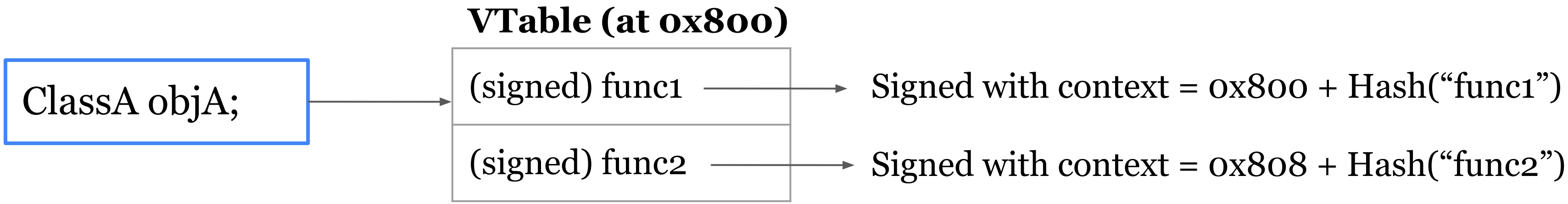
iOS Kernel CFI (PA-based)

How iOS CFI deals with its C++ function pointers (VTable)



iOS Kernel CFI (PA-based)

How iOS CFI deals with its C++ function pointers (VTable)



Context evaluation:

- Hash(function_name): unique (within a class) and invariant! (perfect!)
- Storage address: unique (within an address system) but not invariant! (what problem could come up?)

iOS Kernel CFI (PA-based)

Applying this technique to C function pointers

C++ Class

ClassA objA;

VTable

(signed) func1
(signed) func2

=

C Struct

```
Struct ClassA {
    void (*func1)(int);
    void (*func2)(int);
    .....
    .....
}
```

C++ class and C struct look very similar,
so it seems that we can use it for C struct as well!

iOS Kernel CFI (PA-based)

Problem in C function pointers

```
void func1() {
    struct obj *dst = ..., * src = ...;
    dst->fp = &target1;
    src->fp = &target2;
    ....
    memcpy(dst, src, ...);
    ....
    dst->fp();
}
```

Storage address as context!

GEN: PAC(&target1, &dst->fp, key-0)

GEN: PAC(&target2, &src->fp, key-0)

iOS Kernel CFI (PA-based)

Problem in C function pointers (Cont)

```
void func1() {
    struct obj *dst = ..., * src = ...;
    dst->fp = &target1;
    src->fp = &target2;
    ....
    memcpy(dst, src, ...);
    ....
    dst->fp();
}
```

GEN: PAC(&target1, &dst->fp, key-0)

GEN: PAC(&target2, &src->fp, key-0)

After memcpy(),
dst->fp = src->fp = PAC(&target1, **&src->fp**, key-0)

iOS Kernel CFI (PA-based)

Problem in C function pointers (Cont)

```
void func1() {
    struct obj *dst = ..., * src = ...;
    dst->fp = &target1;
    src->fp = &target2;
    ....
    memcpy(dst, src, ...);
    ....
    dst->fp();
}
```

GEN: PAC(&target1, &dst->fp, key-0)

GEN: PAC(&target2, &src->fp, key-0)

After memcpy(),
dst->fp = src->fp = PAC(&target1, **&src->fp**, key-0)

Context mismatch-!

USE: AUT(&target2, **&dst->fp**, key-0)

A dynamic context (not invariant) would break compatibility in memory-related functions-!

iOS Kernel CFI (PA-based)

Problem in C function pointers (Cont)

```
void func1() {
    struct obj *o1 = ..., * o2 = ...;
    o1->fp = &target1;
    o2->fp = &target2;
    ....
    memcpy(o2, o1, ...);
    ....
    o2->fp();
}
```

A naive solution? Re-signing-!

```
memcpy(void *dst, void *src, ..) {
    // Solution:
    // Re-sign–
    //     o2->fp = PAC(&target1, &o2->fp, key-0)
}
```

USE: AUT(&target2, **&o2->fp**, key-0)

BUT.. it's infeasible to identify its object type correctly..

iOS Kernel CFI (PA-based)

Wrap-up and Takeaways

- Use of static context solely (i.e., type-based CFI) is not secure.
- A decent combination of dynamic (invariant) and static context promises a better security.
- But, use of dynamic context is likely prone to compatibility issues, especially in C-based OSes.

Solution-1:
Multi-Layer Context Generation

Multi-Layer Context Generation

A new combination of static and dynamic contexts

- Two static contexts
  - typesig
  - objtype
- Two dynamic contexts
  - objbind: plays a crucial role in our system!
  - retbind (not discussed today)

Multi-Layer Context Generation

(static) typesig: base-line context (same to type-based CFI)

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

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

Layer	**Context**
typesig	irqhandler_t

Multi-Layer Context Generation

(static) objtype

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

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

Layer	Context
typesig	irqhandler_t
objtype	struct.irqaction

Multi-Layer Context Generation

(dynamic) objbind: blends a specific field value

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

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

Layer	Context
typesig	irqhandler_t
objtype	struct.irqaction
objbind	o->name ("o1")

GEN: PAC(&target, **context**, key-o)

Objbind

What's behind objbind

- We found there are common OS design patterns beneficial to bring out a good context for CFI.

- OS design patterns we found
  - A lot of structs has a field that is unique as well as invariant.

Objbind

What's behind objbind: unique

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

This field likely differently initialized for different codes.

It certainly helps enhance the security level of CFI!

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

```
void func2() {
    struct irqaction *o = …;
    o->name = "o2";
    o->handler = &target2;
}
```

```
void func3() {
    struct irqaction *o = …;
    o->name = "o3";
    o->handler = &target3;
}
```

Objbind

What's behind objbind: invariant

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

→ Invariant: const value-!

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

→ This field is initialized at the time of its object creation, and is likely not changed until its object is freed.

(1) Easy to maintain and
(2) Likely no compatibility issue

Objbind

+ memcpy-compatible

```
void func1() {
    struct irqaction *dst = ..., * src = ...;
    dst->name = “dst”;  src->name = “src”;
    dst->handler = &target1;
    src->handler = &target2;
    ....
    memcpy(dst, src, ...);
    ....
    dst->handler();
}
```

Objbind as context!

GEN: PAC(&target1, **dst->name**, key-0)

GEN: PAC(&target2, **src->name**, key-0)

Objbind

+ memcpy-compatible

```
void func1() {
    struct irqaction *dst = ..., * src = ...;
    dst->name = “dst”;  src->name = “src”;
    dst->handler = &target1;
    src->handler = &target2;
    ....
    memcpy(dst, src, ...);
    ....
    dst->handler();
}
```

GEN: PAC(&target1, dst->name, key-0)

GEN: PAC(&target2, src->name, key-0)

After memcpy(),
dst->handler = PAC(&target2, **src->name**, key-0)

Objbind

+ memcpy-compatible

```
void func1() {
    struct irqaction *dst = …, * src = …;
    dst->name = “dst”;  src->name = “src”;
    dst->handler = &target1;
    src->handler = &target2;
    ….
    memcpy(dst, src, …);
    ….
    dst->handler();
}
```

GEN: PAC(&target1, dst->name, key-0)

GEN: PAC(&target2, src->name, key-0)

After memcpy(),
dst->handler = PAC(&target2, **src->name**, key-0)

Still matched!

USE: AUT(&target2, **src->name**, key-0)

No memcpy-compatible issue arises!

Multi-Layer Context Generation

Wrap-up and Takeaways

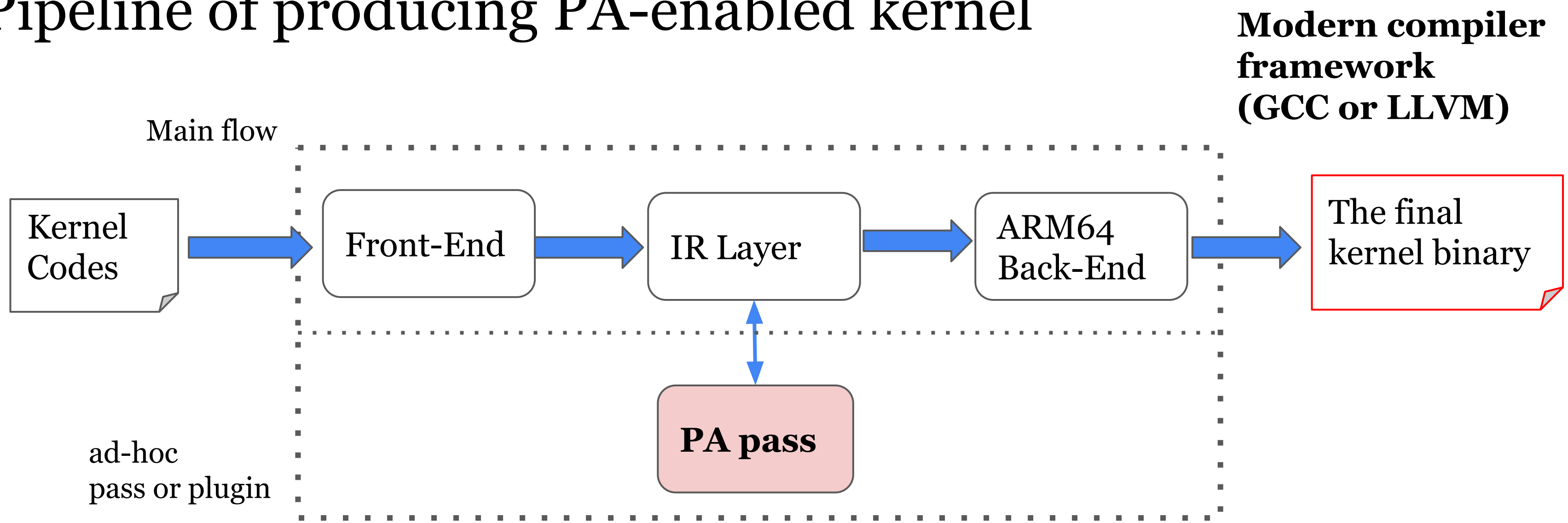
- A CFI scheme can make use of design patterns in C-based OSes, to enhance CFI security without compatibility issues.

- Our paper includes more features integral to make up a PA-based Kernel CFI. Check out the full paper!
  - Context analyzer: identifying the best objbind field automatically
  - Kernel infrastructure: key management, preemptive hijacking prevention, brute-force attack mitigation

Pain point-2:
A complicated compiler behavior

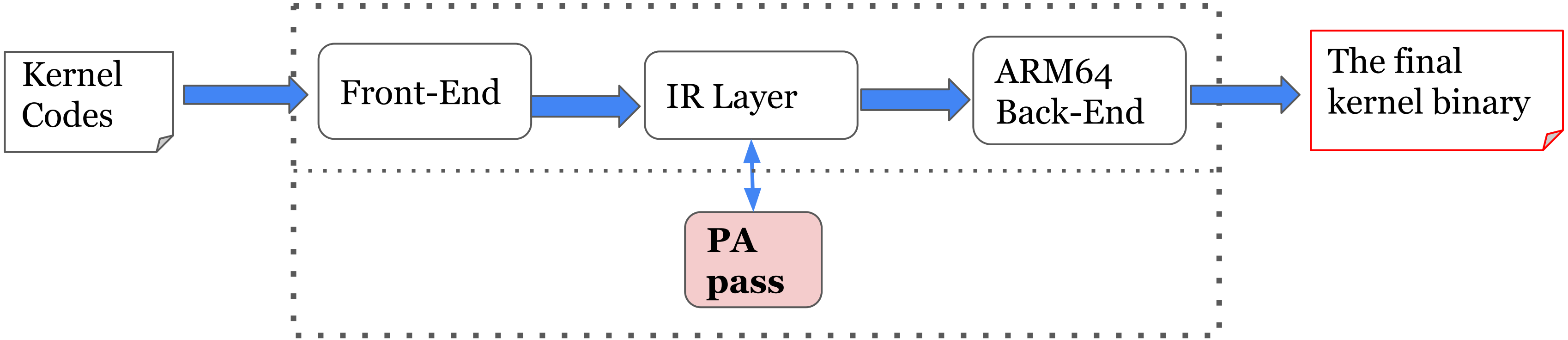
A complicated compiler behavior

Pipeline of producing PA-enabled kernel



A complicated compiler behavior

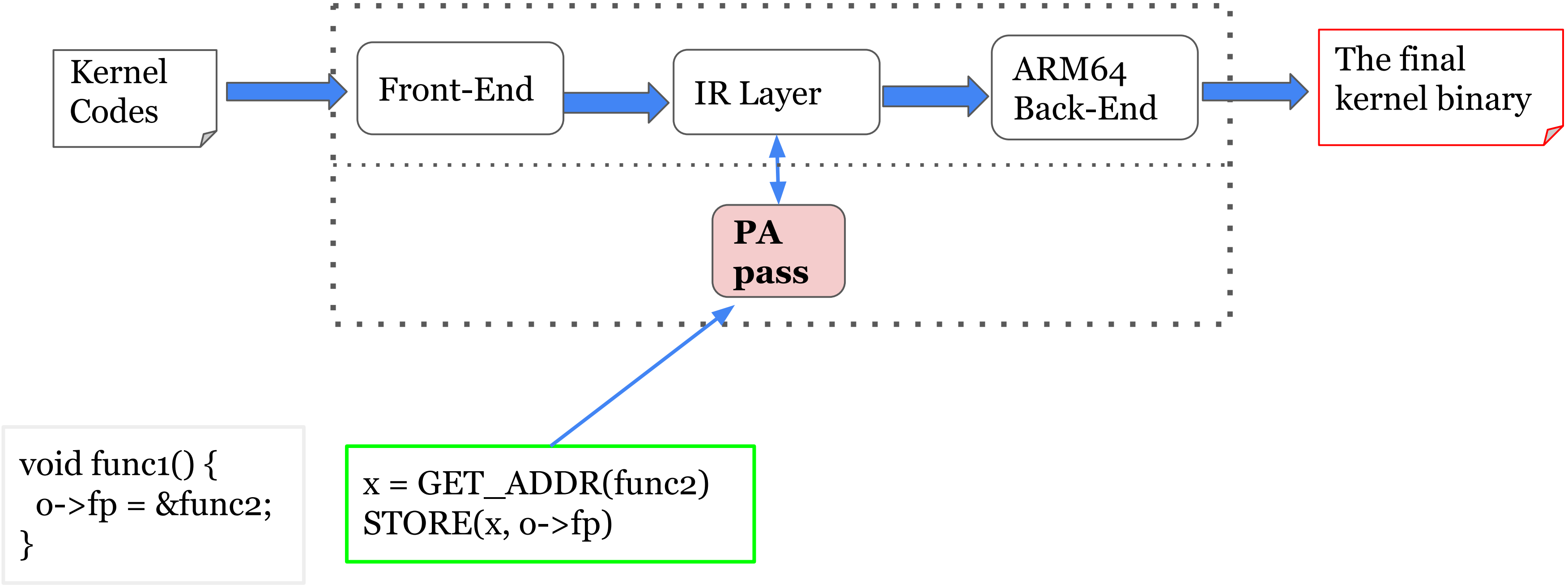
The gap between **expectations** and reality




```
void func1() {
  o->fp = &func2;
}
```

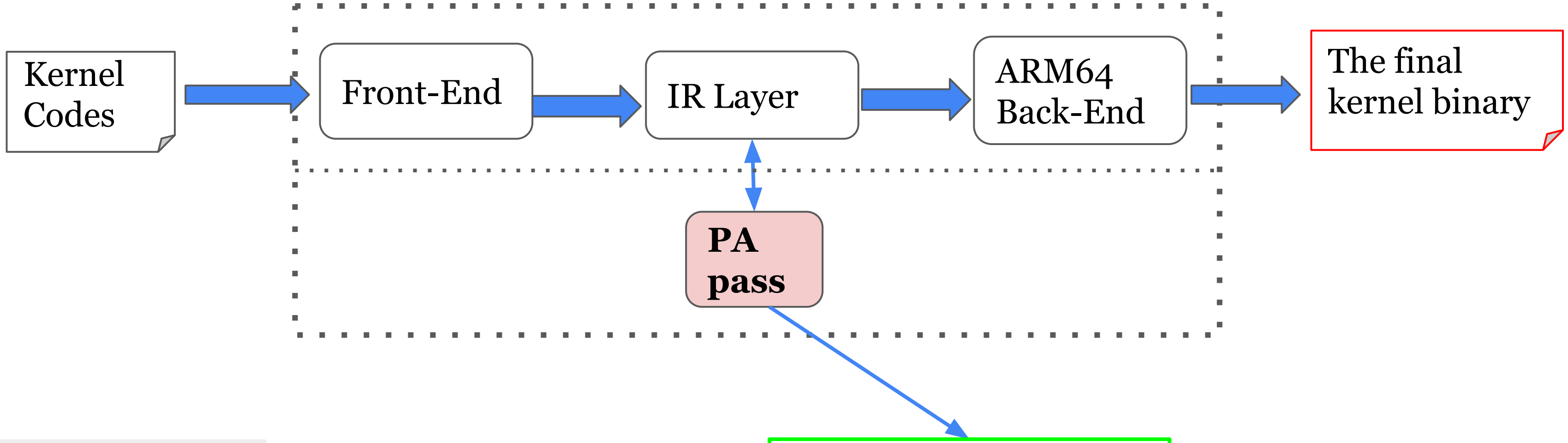
A complicated compiler behavior

The gap between **expectations** and reality



A complicated compiler behavior

The gap between **expectations** and reality



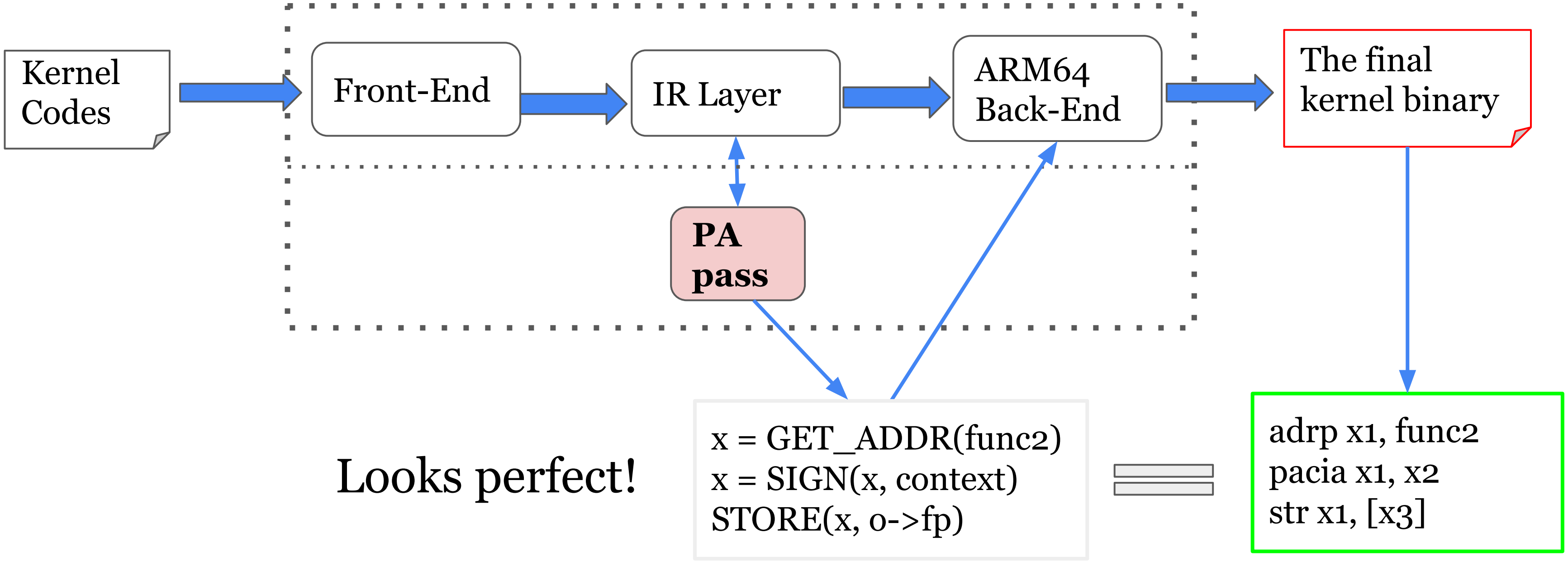

```
void func1() {
  o->fp = &func2;
}
```

```
x = GET_ADDR(func2)
STORE(x, o->fp)
```

```
x = GET_ADDR(func2)
x = SIGN(x, context)
STORE(x, o->fp)
```

A complicated compiler behavior

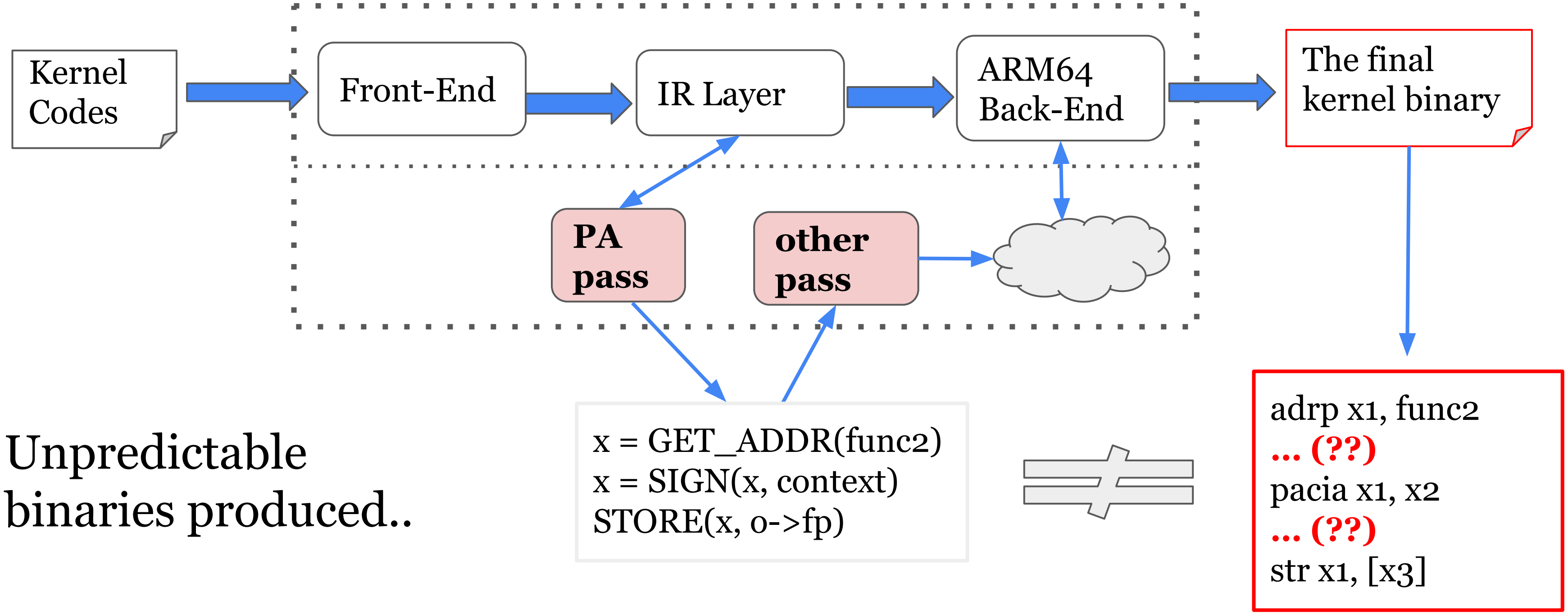
The gap between **expectations** and reality



Looks perfect!

A complicated compiler behavior

The gap between expectations and **reality**



Unpredictable binaries produced..

A complicated compiler behavior

When it turns out problematic

- We assume attackers who can corrupt memory but not registers.
- The aim of attackers is to make an arbitrarily signed pointer using the signing code.

A secure sequence (expectation)

```
(L1) adrp x1, func2
(L2) pacia x1, x2
(L3) str x1, [x3]
```

The raw pointer (x1) never spills onto memory,
and it's guaranteed that a pointer stored on memory is signed.

A complicated compiler behavior

When it turns out problematic

- We assume attackers who can corrupt memory but not registers.
- The aim of attackers is to make an arbitrarily signed pointer using the signing code.

An insecure sequence (reality)

```
(L1) adrp x1, func2
(L2) str x1, [sp]
(L3) ....
(L4) ldr x1, [sp]
(L5) pacia x1, x2
(L6) str x1, [x3]
```

(L1) loads the raw address of func2 into x1.
(L2) stores x1 onto the stack memory.
(L3) .... imagines a stack vulnerability here ....
attackers put an arbitrary pointer in the stack memory.
(L4) loads the attacker-chosen pointer
(L5) signs the attacker-chosen pointer

A complicated compiler behavior

Wrap-up and Takeaways

- Modern compiler frameworks are so complicated that you cannot expect what you did still remains as secure in the final binary. (even if you did great)

- The insecure sequences attributed to the compiler issue could be exploited to disarm CFI defenses as entirely.

Solution-2: Static Validator

Static Validator

- It checks if the final kernel binary respects a set of security rules, thereby ensuring all sequences of PA instructions in kernel are secure.

- It performs a binary-level static analysis on a whole-kernel binary. (intra-procedural)

- We ran static validator on three kernel binaries.
  - iOS kernel binary
  - Linux kernel binary compiled by PARTS (academic paper)
  - Linux kernel binary compiled by our PA pass

Static Validator

Four principles that kernel must respect

1. Complete protection (P1)
    - All indirect branches have to be authenticated before use.
2. No time-of-check-time-of-use (TOCTOU) (P2)
    - Raw pointers after PA instructions are never stored back in memory.
3. No signing oracle (P3)
    - There must be no gadget that signs an attacker-chosen pointer.
4. No unchecked control-flow change (P4) (Not discussed)
    - All direct modifications of program counter register must be validated.

Static Validator

Found violation of P1 (Complete protection)

Expectation

```
bgmac_chip_reset(x0, ...) {
...
   L1:  mov   x19, x0
   L2:  ldr    x21, [x19, x8]
   L3:  autib x21, x9
   L3:  blr    x21
...
}
```

Reality

```
bgmac_chip_reset(x0, ...) {
...
   L1:  mov   x19, x0
   L2:  ldr    x21, [x19, x8]
   — no authentication –
   L3:  blr    x21
...
}
```

- From: PARTS
- Violation: an indirect branch happens without authentication at L3
- Consequence: attackers can make an arbitrary control-flow transition

Static Validator

Found violation of P2 (No TOCTOU)

Expectation

```
sort(.., .., void (*swap_func)()) {
...
   L1:  autib   x2, x0
   L2:  blr     x2
...
}
```

Reality

```
sort(.., .., void (*swap_func)()) {
...
   L1:  autib   x2, x0    // x2: swap_func
   L2:  stp     x1, x2, [x29, 144]
...
   L3:  ldr     x2, [x29, 144]
   L4:  blr     x2
}
```

- From: PAL during development
- Violation: a raw pointer is spilled onto the memory
- Consequence: attackers can make an arbitrary control-flow transition

Static Validator

Found violation of P3 (No signing oracle)

Expectation

```
UNDEFINED(...) {
...
   L1: adrp   x21, 0xffff....ab00
   L2: pacia   x22, x8
...
}
```

Reality

```
UNDEFINED(.., .., x2) {
...
   L1: mov    x19, x2
   L2: ldr     x21, [x19, 240]
   L3: pacia  x21, x8
...
}
```

- From: iOS Kernel
- Violation: signs a pointer that comes from memory
- Consequence: attackers can make an arbitrary signed pointer

Static Validator

Found violation of P3 (No signing oracle) (ADVANCED)

Expectation

```
usb_stor_CB_transport(...) {
...
   L1: adrp    x22, 0xffff....ab00
   L2: pacia   x22, x23
...
}
```

Reality

```
usb_stor_CB_transport(...) {
...
   L1: adrp    x22, 0xffff....ab00
   L2: bl      usb_stor_msg_common
   L3: pacia   x22, x23
...
}
```

Why is it problematic??

Static Validator

Found violation of P3 (No signing oracle) (ADVANCED)

Reality

```
usb_stor_CB_transport(...) {
...
   L1:  adrp    x22, 0xffff....ab00
   L2:  bl      usb_stor_msg_common
   L3:  pacia   x22, x23
...
}
```

```
usb_stor_msg_common(...) {
   L1:  stp     x22, x21, [sp, 48]
...
   L2:  ldp     x22, x21, [sp, 48]
   L3:  ret
}
```

- From: PARTS
- Violation: signs a pointer that comes from memory
- Consequence: attackers can make an arbitrary signed pointer

Static Validator

Results

We confirmed

- 15 violations in PARTS-applied linux kernel binary
- 5 violations in iOS kernel binary
- 7 violations in PAL-applied linux kernel binary (during dev)

NOTE

- Violation does not mean Exploitable. There are many variables involved in exploitability. (e.g., the context of inter-procedural stuffs)

Static Validator

Wrap-up and Takeaways

- Don’t trust the compiler you’re relying on. Instead, you should trust a binary-level validator that runs at the end of the kernel-build procedure.

UAF Defense
(UAF: Use-After-Free)

Exploiting UAF

Step-1: creating a dangling pointer
Step-2: allocating an object to overlap with the freed victim object
Step-3: dereferencing the dangling pointer

To defend against UAF attacks, it suffices to stop the attack at any of these three steps.

Pain point:
No one cares about Kernel UAF defenses- Why?

No one cares about kernel UAF

WHY?

- Size: OS kernel is huge in size
- Low-level: in most cases, OS kernel is placed at the bottom of entire software stack

Existing approaches

1. Pointer invalidation
    a. prevent the creation of dangling pointer. (Step-1)
2. Safe memory allocation
    a. prevent the reallocation of freed object (Step-2)
3. Access validation
    a. check if a pointer dereferencing is valid (Step-3)

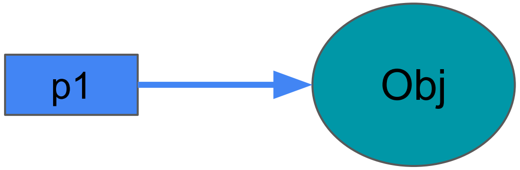
Existing approaches

Pointer invalidation (No dangling pointer)

C++ Smart Pointer (similar to Rc/Arc in Rust)

```
func(...) {
    shared_ptr<Obj> p1(new Obj());
    shared_ptr<Obj> p2;
     ...
    p2 = p1;
}  // end
```

Reference count: 1



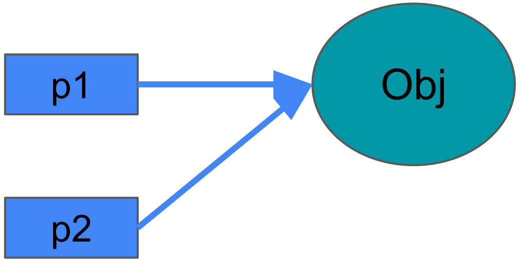
Existing approaches

Pointer invalidation (No dangling pointer)

C++ Smart Pointer (similar to Rc/Arc in Rust)

```
func(...) {
    shared_ptr<Obj> p1(new Obj());
    shared_ptr<Obj> p2;
     ...
    p2 = p1;
}  // end
```

Reference count: 2



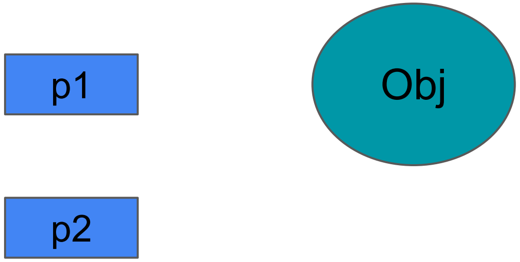
Existing approaches

Pointer invalidation (No dangling pointer)

C++ Smart Pointer (similar to Rc/Arc in Rust)

```
func(...) {
    shared_ptr<Obj> p1(new Obj());
    shared_ptr<Obj> p2;
     ...
    p2 = p1;
}  // end
```

Reference count: 0, Deallocate Obj!



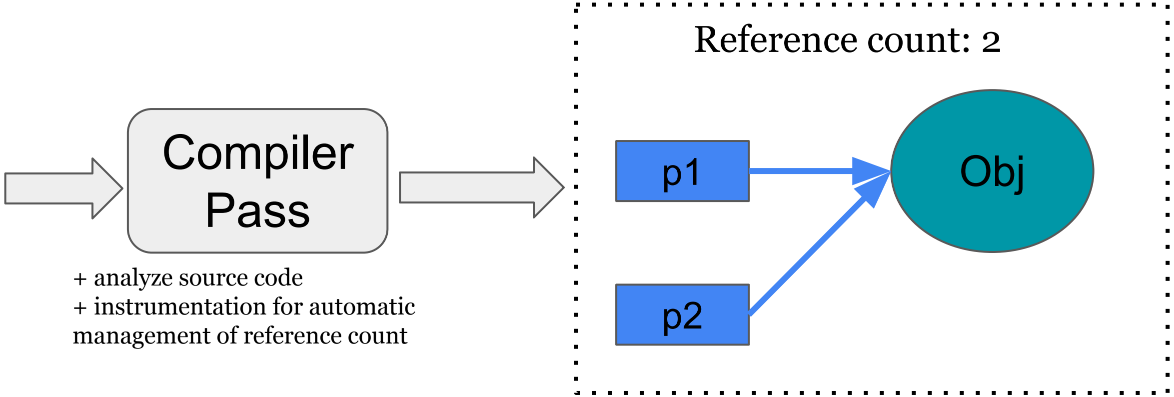
- If we perfectly manage a reference count for an object, no dangling pointer will occur.
- Problem? → Developers have to explicitly turn all pointers into smart pointers, which is unrealistic.

Existing approaches

Pointer invalidation (No dangling pointer)

CRCount (NDSS 2019)

```
func(...) {
    Obj *p1 = new Obj();
    Obj *p2 = p1;
     ...
}  // end
```



Solution?

→ an automatic reference count management using a compiler instrumentation

Existing approaches

Pointer invalidation (No dangling pointer)

CRCount (NDSS 2019)

```
unsigned long u1 = 0;

func(...) {
    Obj *p1 = new Obj();
    ...
    u1 = (unsigned long)p1;
     ...
}
```

How to deal with it?
incrementing a count or not?

Problem?

→ There are cases in which an automatic management does not work well, and such cases are commonly found in OS kernel due to its huge size.

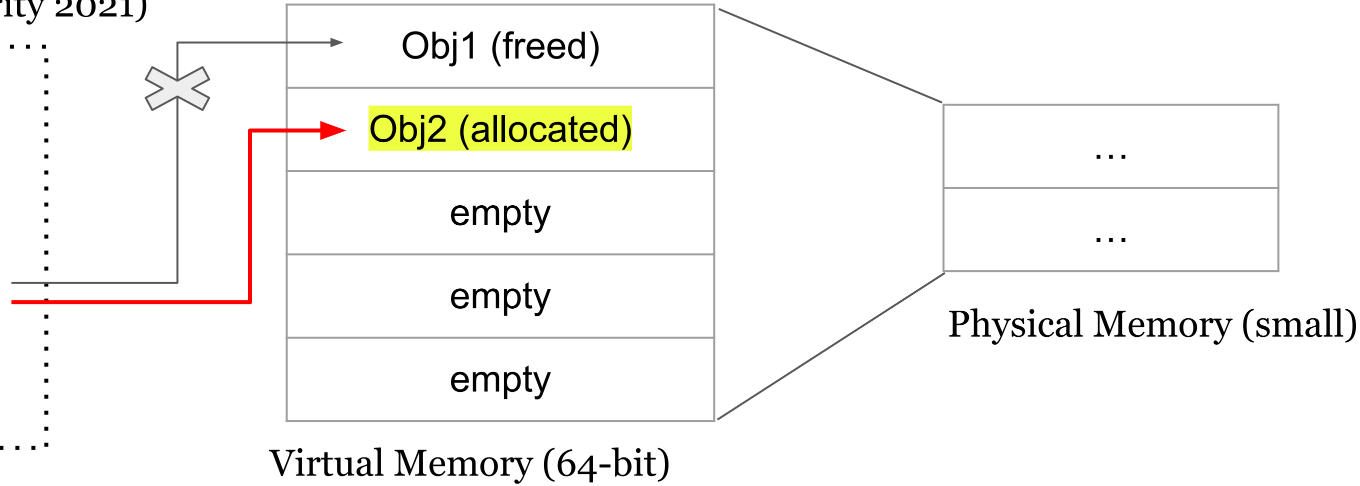
Existing approaches

Safe memory allocation (No reallocation)

FFmalloc (USENIX Security 2021)

```
func(...) {
    Obj *p1 = malloc();
    free(p1);
    ...
    Obj *p2 = malloc();
     ...
}
```

User-space app



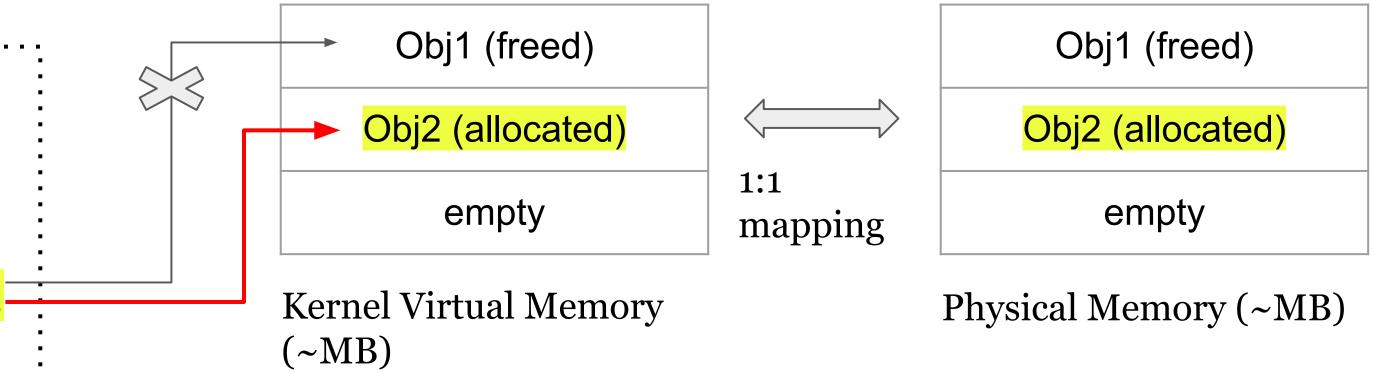
Never allows the reallocation of a freed object!
It works out in practice for user apps, thanks to the large size of virtual memory.

Existing approaches

Safe memory allocation (No reallocation)

```
func(...) {
    Obj *p1 = kmalloc();
    kfree(p1);
    ...
    Obj *p2 = kmalloc();
     ...
}
```

Kernel code



An allocation in kernel directly takes up a part of physical memory, bring on out-of-memory issues in a short time.

Existing approaches

Access validation

```
func(...) {
    Obj *p1 = malloc();
    ...
    p1->val = 10;
}
```

Random ID : 0xabcd

Pointer-side ID (stored in place)

p1 = 0xabcd110022003300

Object-side ID (stored in a separate table)

Object address (Key)	ID (Value)
0x110022003300	0xabcd
…	…

NOTE: this is a simplified illustration of mapping table

Compare if a pointer-side is equivalent to an object-side ID

Existing approaches

Access validation

```
func(...) {
    Obj *p1 = malloc();
    free(p1);
    Obj *p2 = malloc();
    ...
    p1->val = 10;
}
```

Pointer-side ID (stored in place)

p1 = 0xabcd110022003300

Object-side ID (stored in a separate table)

Object address (Key)	ID (Value)
0x110022003300	0x1234
…	…

NOTE: this is a simplified illustration of mapping table

In case of invalid access-ID mismatch!

Existing approaches

Access validation

```
func(...) {
    Obj *p1 = malloc();
    free(p1);
    Obj *p2 = malloc();
    ...
    p1->val = 10;
}
```

Pointer-side ID (stored in place)

p1 = 0xabcd110022003300

Object-side ID (stored in a separate table)

Object address (Key)	ID (Value)
0x110022003300	0x1234
...	...

NOTE: this is a simplified illustration of mapping table

Problem?

→ a pointer dereference demands N additional memory accesses (N = 2 or 3), bring on substantial performance downgrade.

Existing approaches

Wrap-up and Takeaways

- Pointer invalidation
  - It's infeasible to implement a perfect static analysis for a huge kernel.
- Secure memory allocation
  - Readily reach out-of-memory, when applied to kernels
- Access validation
  - Bring on a large performance downgrade

Solution:
Object ID inspection through base identifier

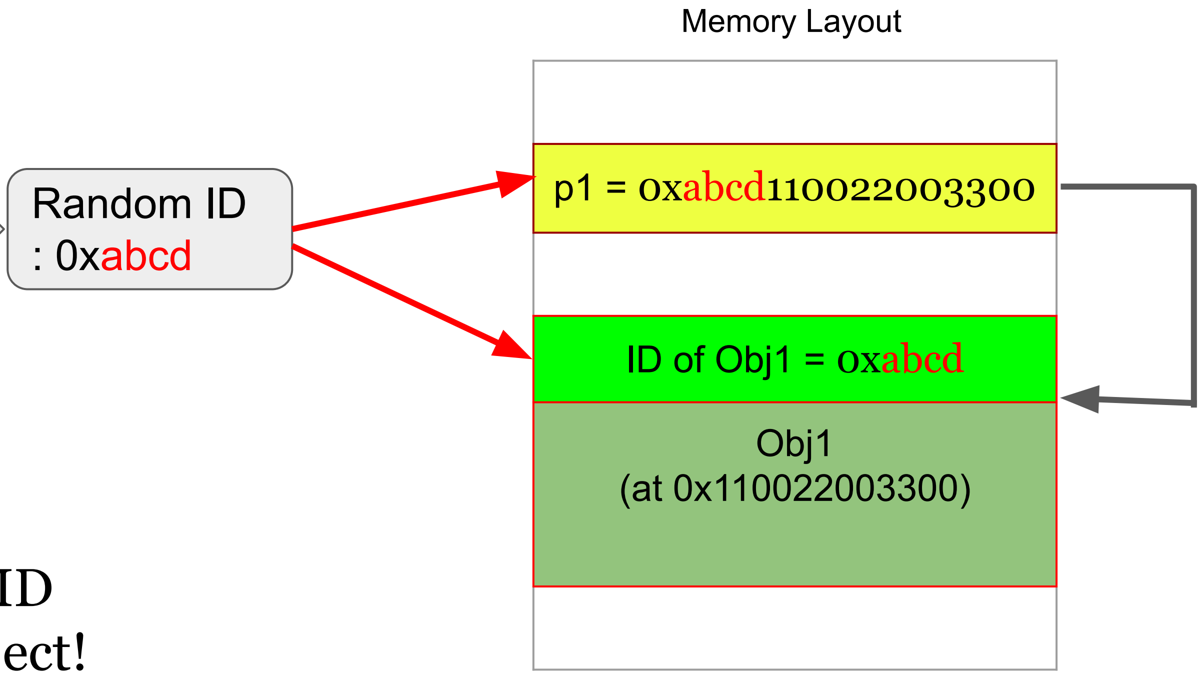
Goal

- Optimizing Access Validation Approach
  - AS-IS: three more memory loads are required to obtain an object-side ID.
  - TO-BE: Just one memory load is needed to obtain an object-side ID.

The first attempt

The first attempt we did

```
func(...) {
    Obj *p1 = kmalloc();
    ...
    p1->val = 10;
}
```

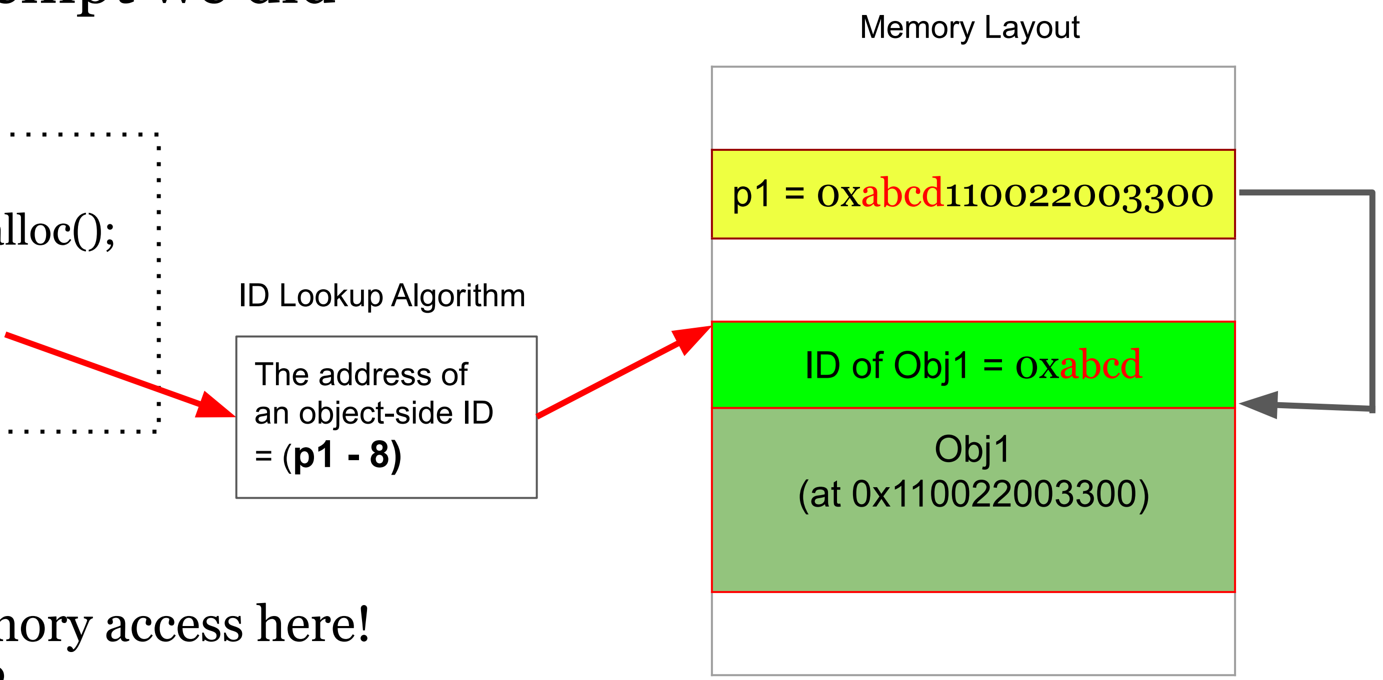


Place an object-side ID
at the start of the object!

The first attempt

The first attempt we did

```
func(...) {
    Obj *p1 = kmalloc();
    ...
    p1->val = 10;
}
```



Only one memory access here!
Any problem?

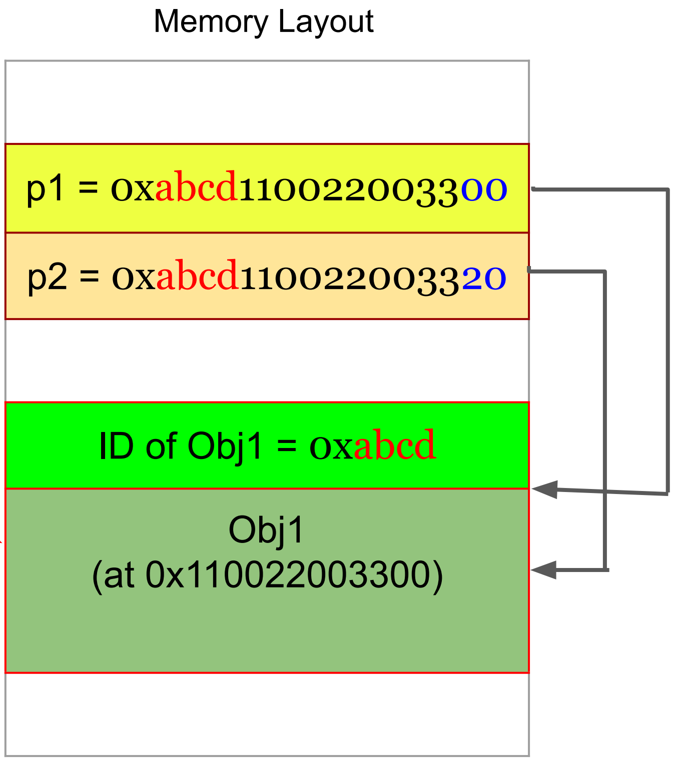
The first attempt (Problem)

ID lookup for the middle of a pointer

```
func(...) {
    Obj *p1 = kmalloc();
    int *p2 = &p1->val;
    ...
    *p2 = 10;
}
```

ID Lookup Algorithm

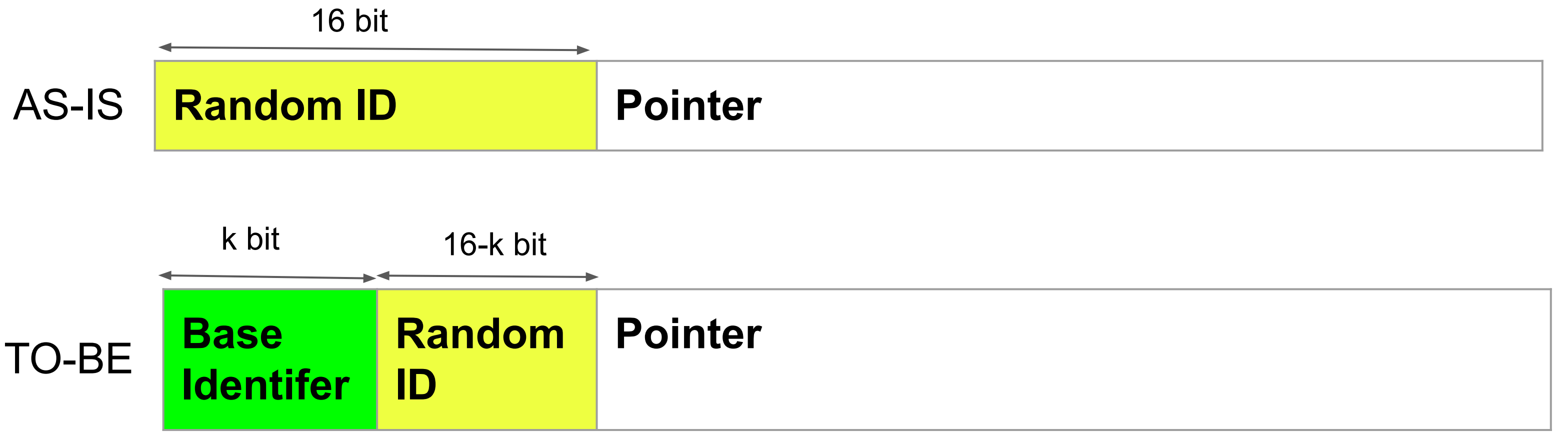
The address of an object-side ID = (**p2 - 8)**



Result in an incorrect ID lookup-!
Solution?

Base Identifier

Base Identifier

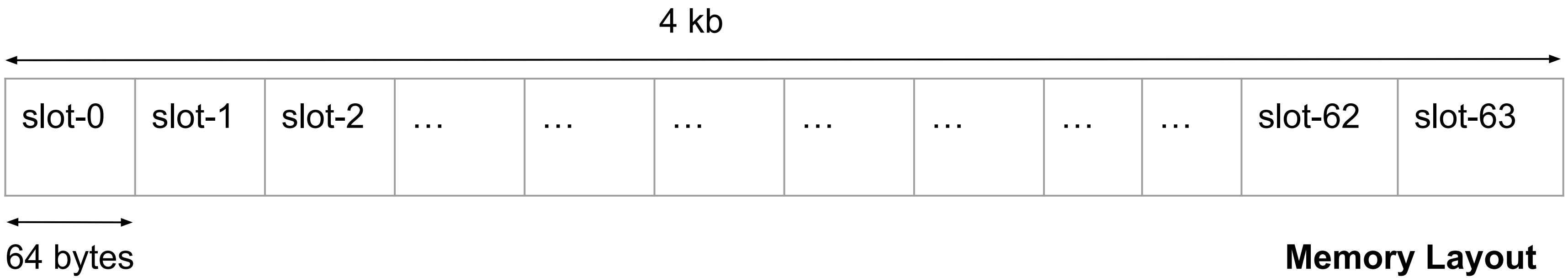


Base Identifier: an auxiliary data that helps the ID lookup process.
takes k bit, where k is typically 6. (i.e., 10 bit for random id)

Base Identifier

How it works under two assumptions

Assumption-1: Every object is limited up to 4kb in size. (2^M bytes, M = 12)
Assumption-2: Every object is aligned with 64 bytes. (2^N bytes, N = 6)
Base Identifier: (M - N) bit, 6 bit, is used to express a slot index.

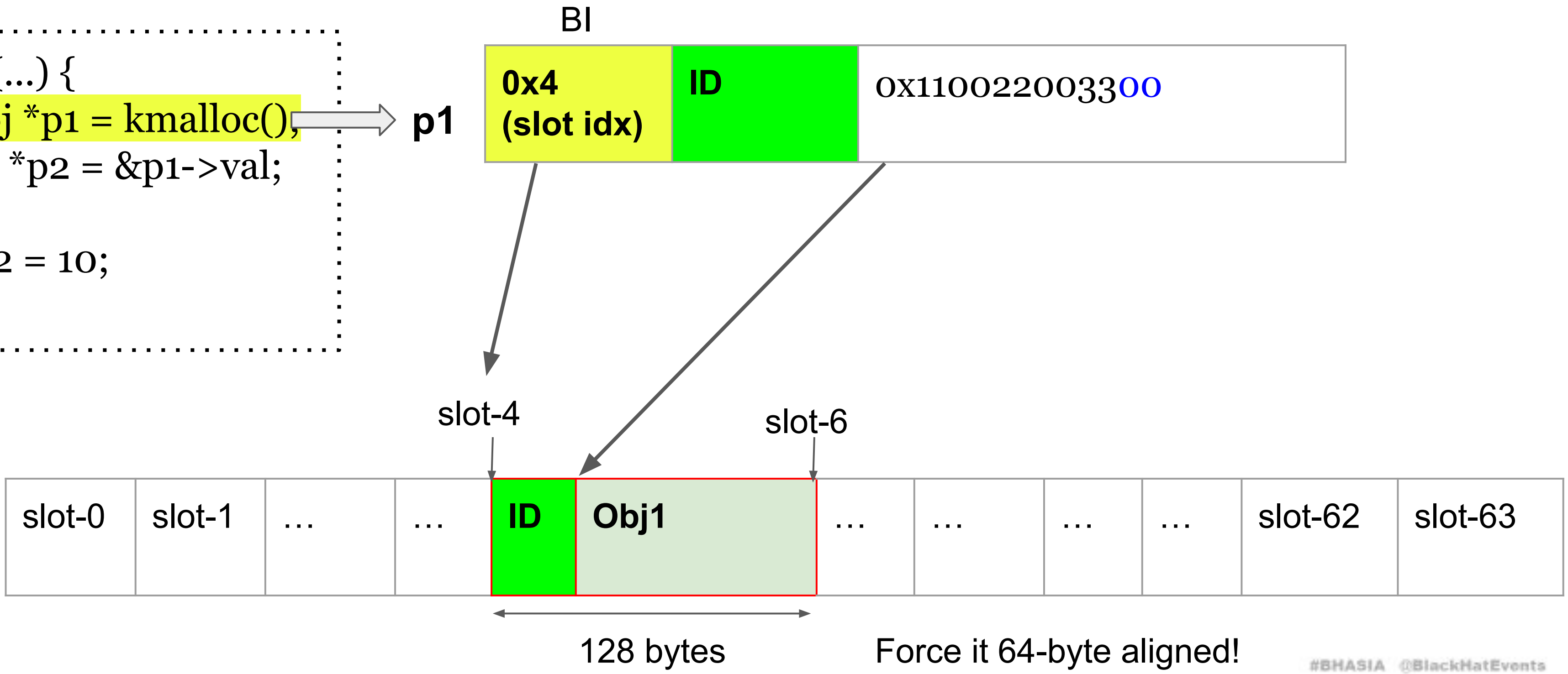


Memory Layout

Base Identifier

How it works under two assumptions

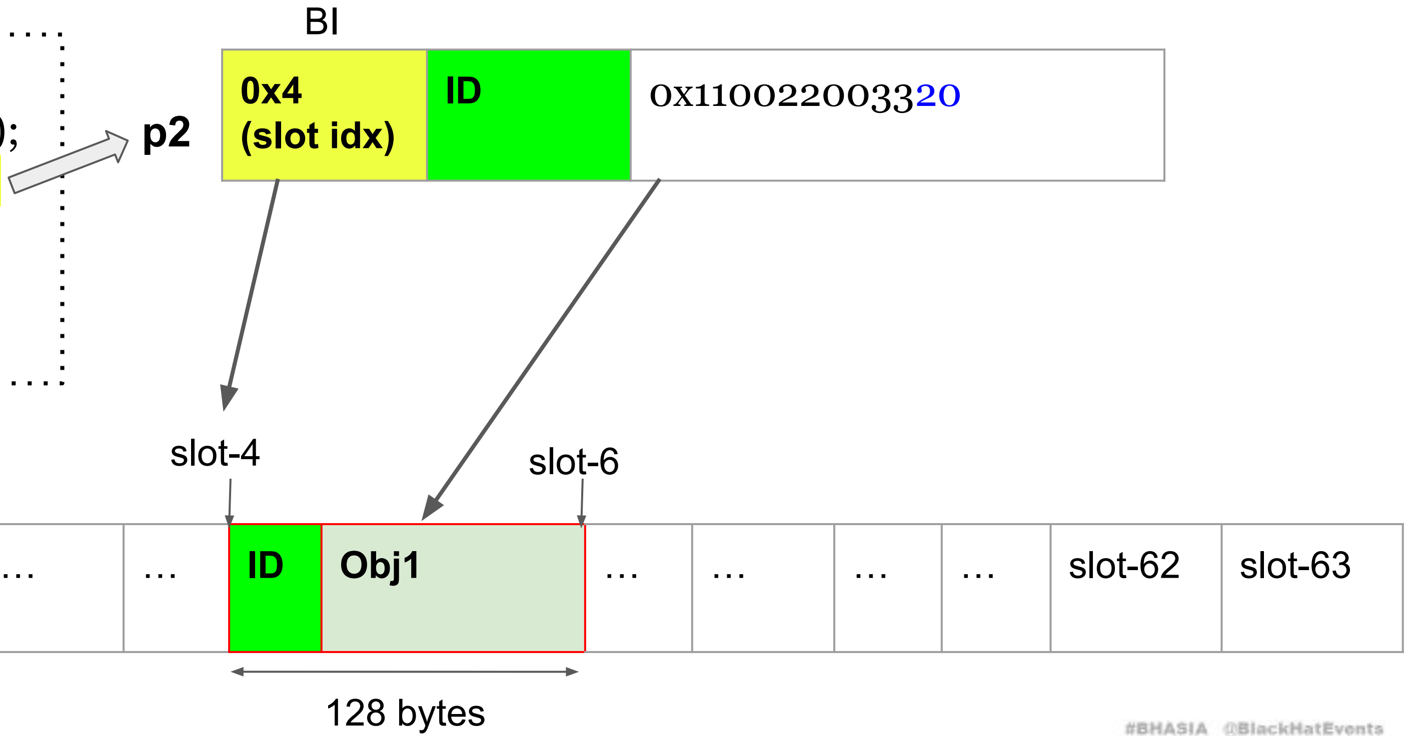
```
func(...) {
    Obj *p1 = kmalloc();
    int *p2 = &p1->val;
    ...
    *p2 = 10;
}
```



Base Identifier

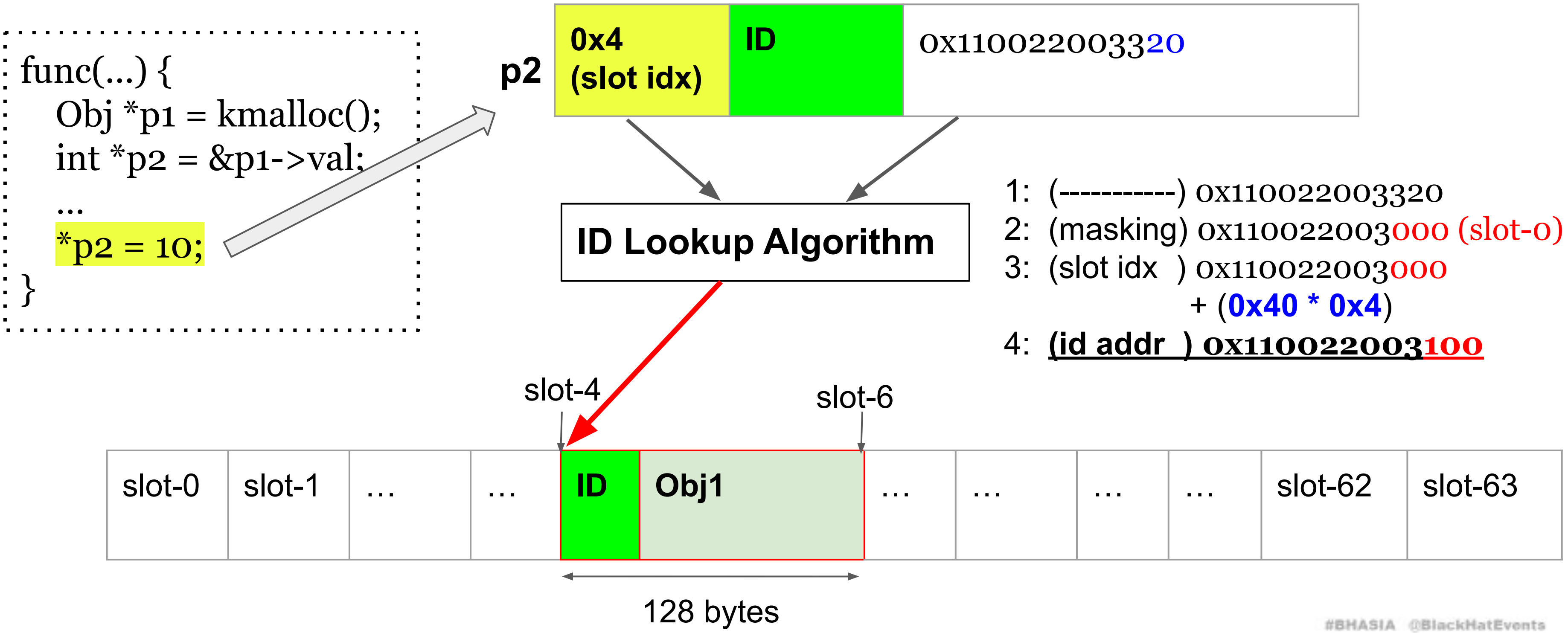
How it works under two assumptions

```
func(...) {
    Obj *p1 = kmalloc();
    int *p2 = &p1->val;
    ...
    *p2 = 10;
}
```



Base Identifier

How it works under two assumptions



Evaluation

Evaluation

- We also design several static analyses to eliminate inspections for UAF-safe pointers. (Not discussed in this talk. Check out the full paper for detail)
- **LMBench result (i.e., syscall latency)**
  - Ubuntu kernel (x86_64): + **20.71**%
  - Android kernel (arm64): + **19.86** %

Evaluation

Evaluation

- We also developed a performance-first variant using ARM TBI, for ARM boards only.
  - **Performance**: + **1–2 %** overhead
  - **Security**: lowered as being not able to inspect the middle pointer.
  - (Not discussed today in detail as well)

Base Identifier

Wrap-up and Takeaways

- It’s possible to build an efficient UAF protection for kernels as entirely, and we are the first one who’s demonstrated it!