DROP THE ROP:

Fine Grained Control-Flow Integrity for The Linux Kernel

João Moreira
Sandro Rigo, Michalis Polychronakis, Vasileios Kemerlis

joao.moreira@lsc.ic.unicamp.br

black hat ASIA 2017

UNICAMP BROWN

```
:/# whoami
```

João Moreira, lvwr, Brazilian...

PhD Candidate @ University of Campinas

UNICAMP

:/# whoami

João Moreira, lvwr, Brazilian...

PhD ~~Candidate~~ @ University of Campinas

Live Patching Engineer @ SUSE

Agenda

- Quick review of Kernel-based ROP
- Control-Flow Integrity
    - Limitations and known issues
- kCFI
    - Implementation
    - Improvements
    - Performance

Memory (un)safety bugs enable code pointer corruption

Memory (un)safety bugs enable code pointer corruption

Control-flow hijacking: Arbitrary code execution

W^X, ASLR

Code-reuse, memory disclosure, ret2usr

Strong Address Space Isolation

ROP

ROP reuses (executable) kernel code

GADGETS, FREELY chained through the stack

0xff8118991d
SMEP Killer
0xff8105b8f0
&payload

```
pop rax
ret
```

0xff8118991d
SMEP Killer
0xff8105b8f0
&payload

```
pop rax
ret
```

0xff8118991d
SMEP Killer
0xff8105b8f0
&payload

```
pop rax    rax = SMEP Killer
ret
```

0xff8118991d
SMEP Killer
0xff8105b8f0
&payload

```
pop rax      rax = SMEP Killer
ret

mov rax,cr4
ret
```

SMEP IS DEAD

THE WALL IS D0WN

0xff8118991d
Turn Off SMEP
0xff8105b8f0
&payload

```
pop rax      rax = SMEP Killer
ret

mov rax,cr4
ret
```

SMEP IS DEAD

THE WALL IS D0WN

0xff8118991d
Turn Off SMEP
0xff8105b8f0
&payload

```
pop rax     rax = SMEP Killer
ret

mov rax,cr4
ret
```

SMEP IS DEAD

PAWNED!

What if we confine indirect branches to safe, previously-computed locations?

Control-Flow Integrity

Paths defined by application's Control-Flow Graph

Different methodologies for computing and enforcing the CFG

What could possibly go wrong?

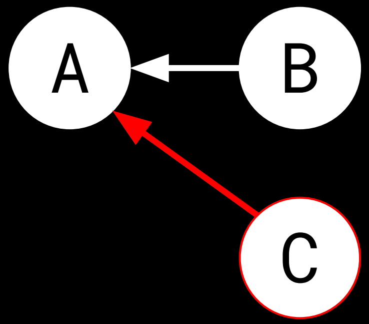
Relaxed permissiveness (granularity)

Coverage

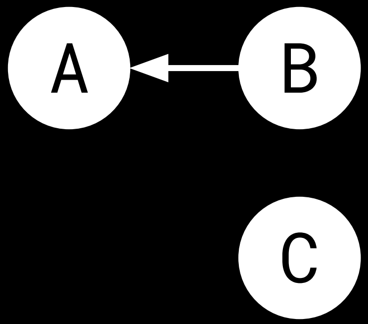
False positives

Granularity issues...

```
<A>:
  ...
  call B;
  ...
```



Coarse-grained CFI



Fine-grained CFI

Coarse-grained: All functions can return to call site **A**

Fine-grained: Only **B** can return to call site **A**

Coarse-grained CFI is known to be **BYPASSABLE**

kCFI

Fine-grained CFI scheme for the Linux kernel

Compiler-based instrumentation (LLVM)

Statically-computed CFGs

Source code + Binary analysis

How to compute a fine-grained CFG?

Backward Edges (returns)

- Functions must return to their respective call sites
- Easy to compute statically

Forward Edges (indirect calls)

- Valid indirect calls targets must be computed
- **Hard:** Complete points-to analysis is infeasible

How to compute a fine-grained CFG?

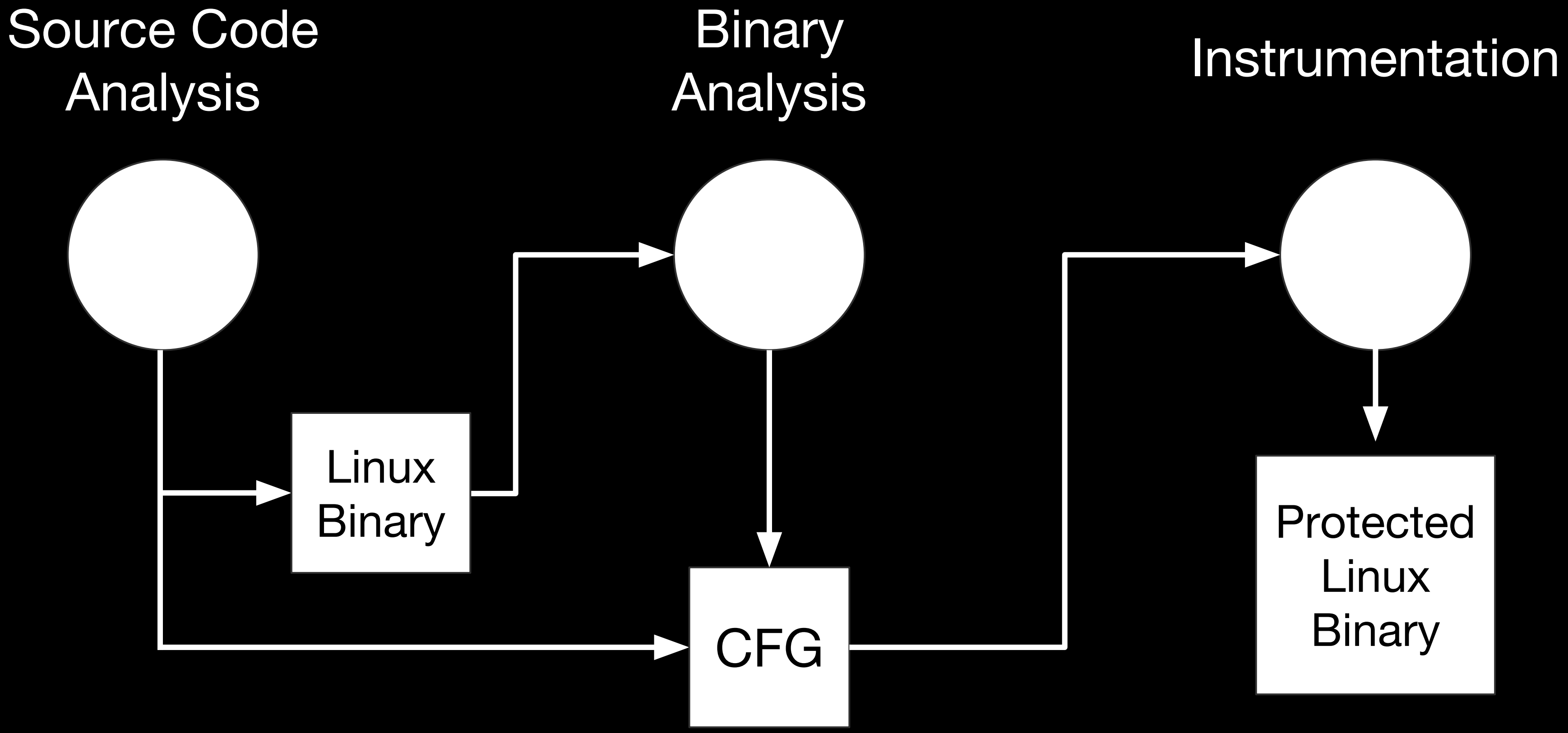
Forward edge computation requires heuristics

kCFI follows the proposal by Abadi et al.:
Pointer and Function prototypes <u>must match</u>!

Functions are clustered by prototype

```
void function(){
...
float (*fptr)(int);
...
}
```

```
float dog(int a);
float cat(int a);
int fish();
```


Source Code
Analysis
Binary
Analysis
Instrumentation
Linux
Binary
CFG
Protected
Linux
Binary

return instrumentation

```
<main>:
...
1: callq  <f1>
2: nopl   0xdeadbeef

<f1>:
...
1: mov    (%rsp),%rcx
2: cmpl   $0xdeadbeef,0x4(%rcx)
3: je     7
4: push   %rcx
5: callq  <ret_violation_handler>
6: pop    %rcx
7: retq
```

indirect call instrumentation

```
<main>:
...
1: cmpl   $0xc001c0de,0x4(%rcx)
2: je     6
3: push   %rcx
4: callq  <call_violation_handler>
5: pop    %rcx
6: call   *%rcx

<f1>:
1: nopl   0xc001c0de
...

<f2>:
1: nopl   0xc001c0de
...
```

So… is this approach really fine-grained?

Well, it is fine-grained,

but **we can do better!**

The presented scheme is prone to a problem that we call

Transitive Clustering Relaxation

Valid targets for indirect calls are clustered

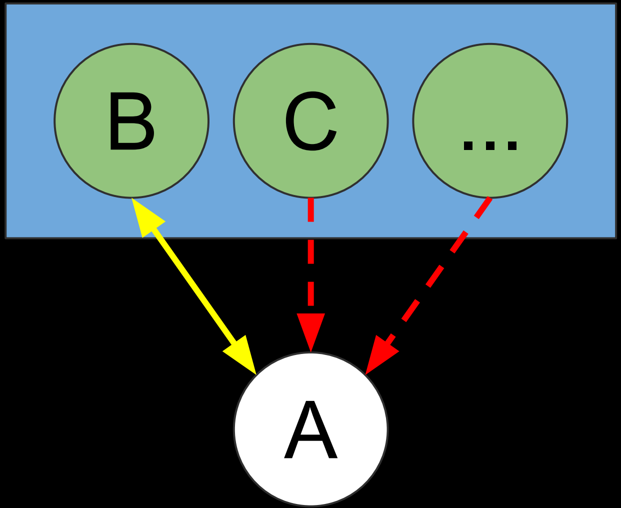
Same tags on call sites and prologues

A directly calls B

B has the same prototype of C

C can return to B’s call site in A

```
<A>:
call b
tag 0xdeadbeef

<Z>:
if(something) ptr = &B
else ptr = &C
call ptr
tag 0xdeadbeef

<B>:
check 0xdeadbeef
ret

<C>:
check 0xdeadbeef
ret
```



In our code base, only for ‘`void()`’, we have 10645 call sites to 4484 `void()` functions

Other prototypes add to that

So yes, this is overly permissive

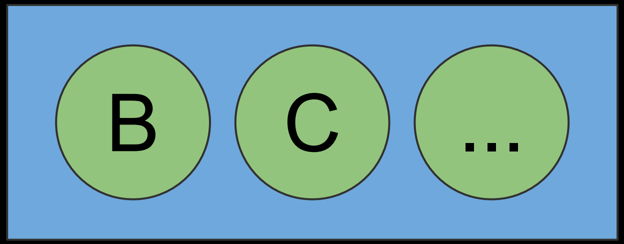
kCFI fixes Transitive Clustering Relaxation
through **Call Graph Detaching** (CGD)

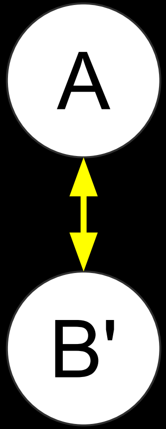
Functions callable both directly and indirectly are cloned

Direct calls to function are replaced by calls to clone

Clone has unique tags, different from cluster tags

```
<A>:
call b_clone
tag 0xdeadc0de

<Z>:
if(something) ptr = &B
else ptr = &C
call ptr
tag 0xdeadbeef

<B>:
check 0xdeadbeef
ret

<C>:
check 0xdeadbeef
ret

<B_clone>:
check 0xdeadc0de
ret
```





Allowed call sites reduced to 220 for indirectly called ‘`void()`’ functions

Directly invoked callees return to their exclusive call sites

No more transitiveness

It is also important to support **Assembly** code

…otherwise it raises false alerts and, even worse, becomes a clear target

We support Assembly through **Lua**-based automatic source-code rewriting

(plus very few handcrafted fixes)

We evaluated performance with 3 benchmarks

Instrumented SPEC2006 (~2%)

Instrumented kernel running LMbench (~8%)

Instrumented kernel running Phoronix (~2%)

Details are available on white-paper or in the bonus-slides, just ask in the end :-)

Fine-grained CFI is not perfect either ...

Control-Flow Bending [USENIX SEC ‘16]

Control Jujutsu [CCS ‘16]

Non-control data attacks [Black Hat Asia 2017]

Yet, the complexity behind these methods shows how relevant CFI is in raising the bar for attacks!

DEMO!

Black Hat Sound Bytes

Fine-grained CFI in the OS context is achievable

CFI can be used to provide a meaningful level of protection, pushing attackers towards more constrained and complex exploitation techniques

Current existing methods for refining the granularity of CFI can (and must) be improved

DROP THE ROP:

Fine Grained Control-Flow Integrity for The Linux Kernel

João Moreira

Sandro Rigo, Michalis Polychronakis, Vasileios Kemerlis

joao.moreira@lsc.ic.unicamp.br

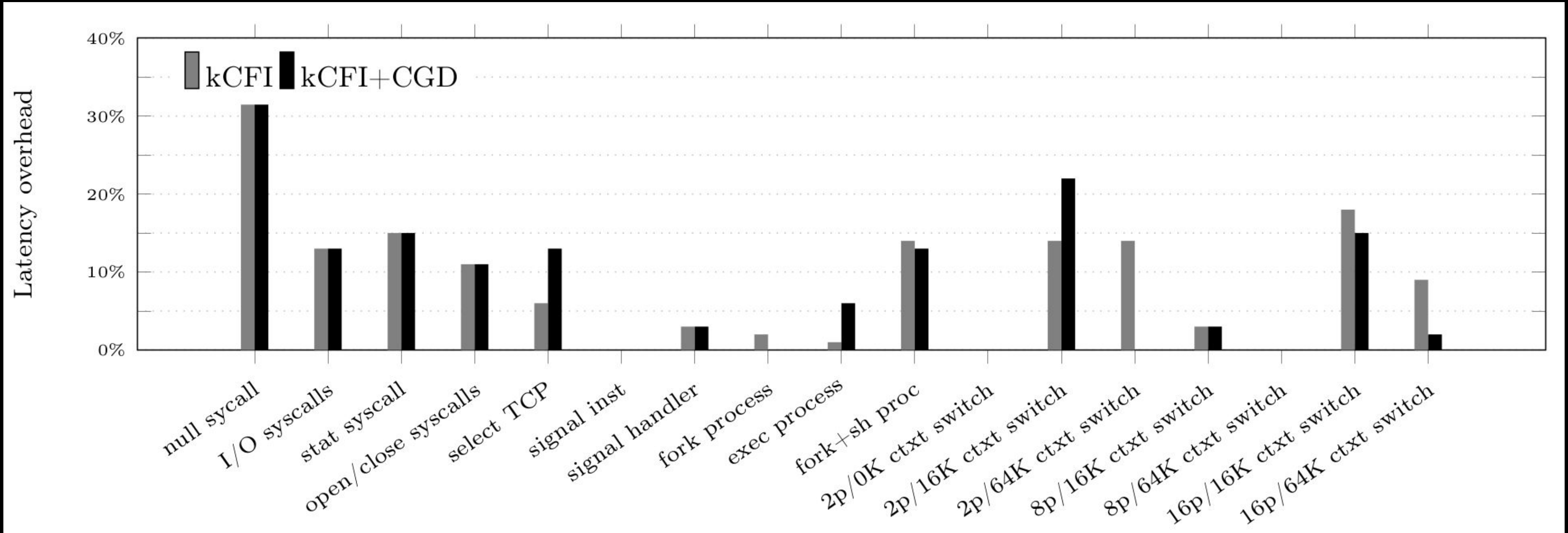
blackhat ASIA 2017

UNICAMP

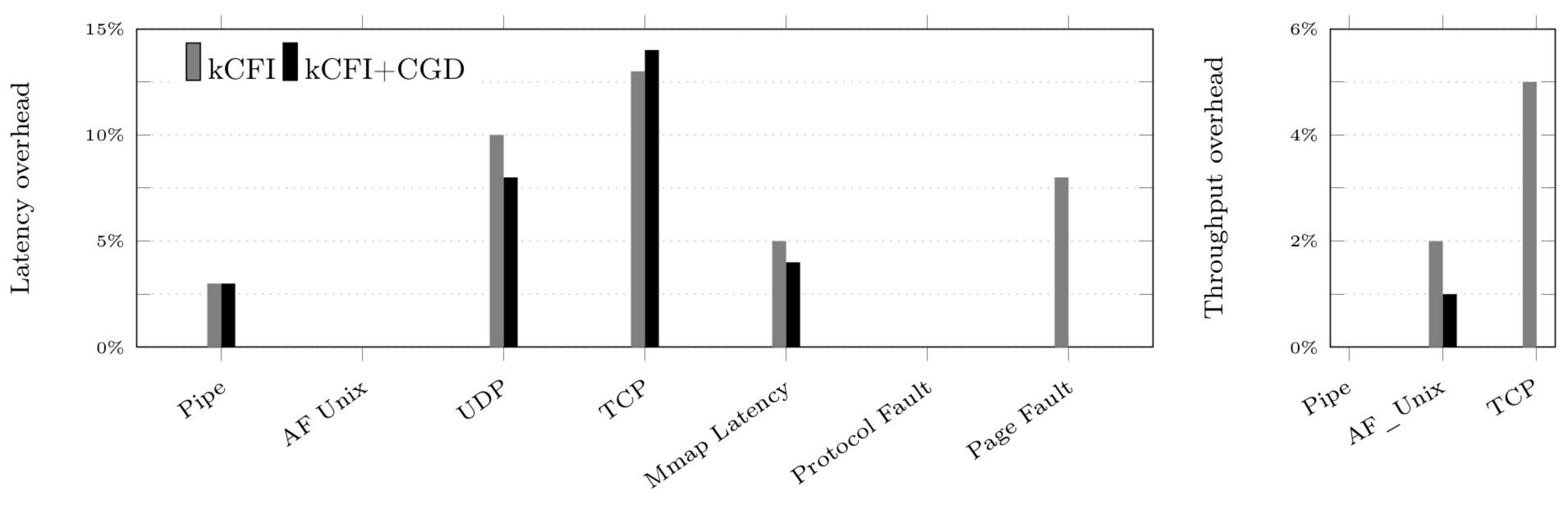
BROWN

BONUS SLIDES

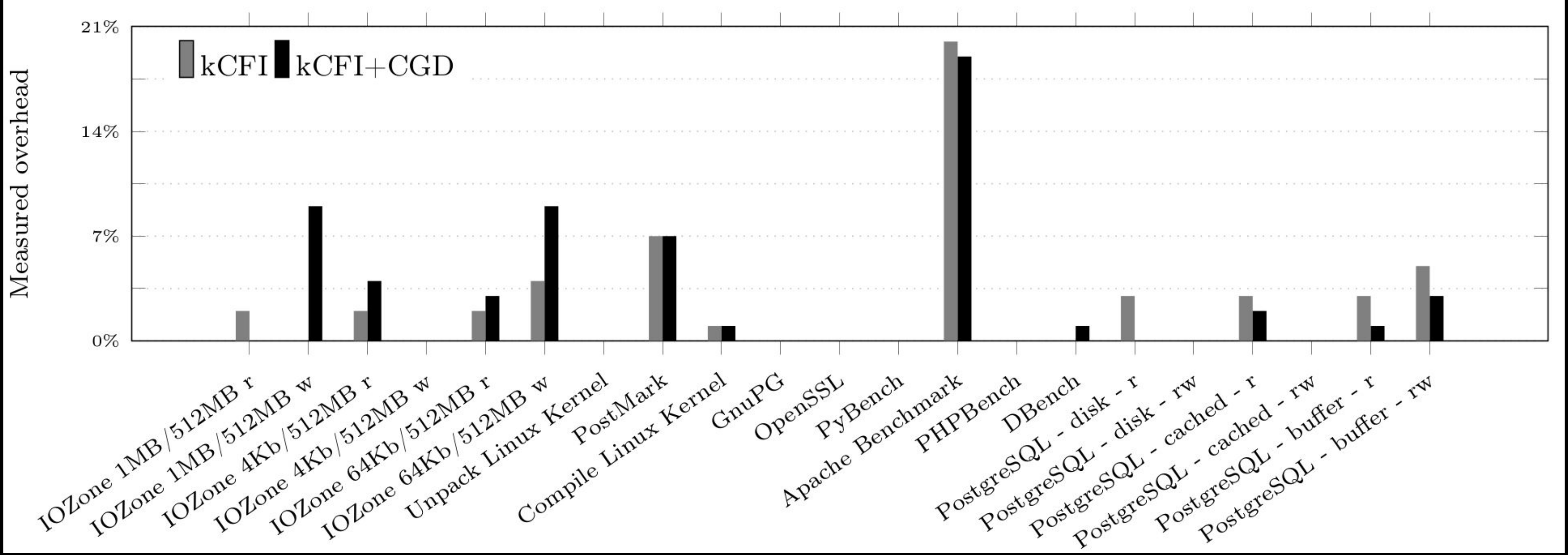
Performance Overhead (LMbench)



Performance Overhead (LMbench)



Performance Overhead (Phoronix)



Space Overhead

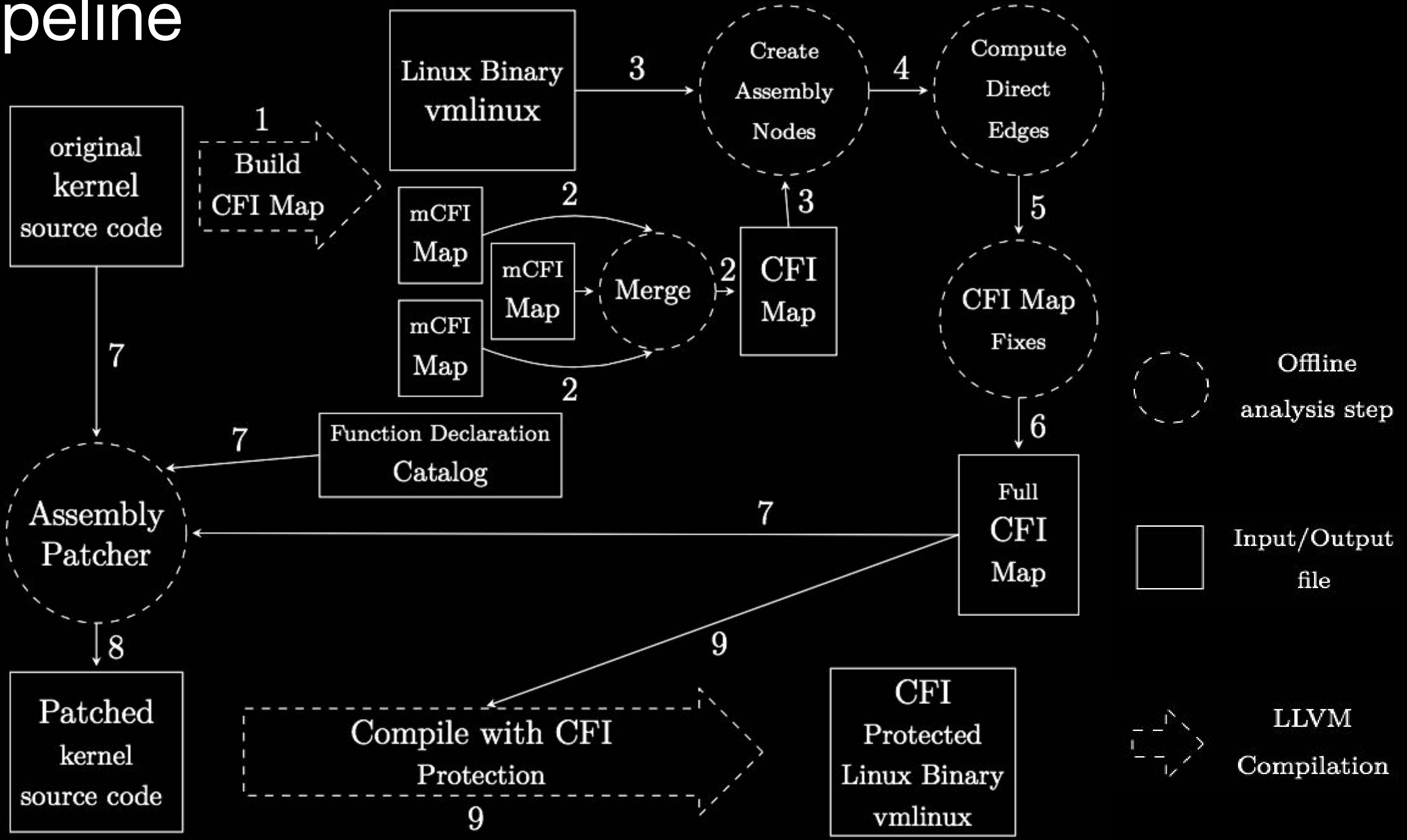
kCFI: 2% space overhead (718MB/705MB)

kCFI+CGD: 4% space overhead (732MB/705MB)

Code base: 132,972 functions

No. of cloned functions: 17,779 functions (~7.5%)

kCFI Pipeline

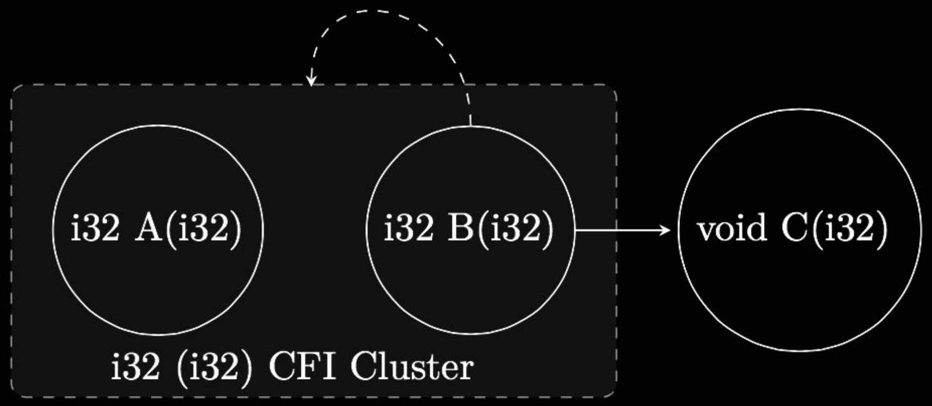


CFI Map (1/2)

(a) Example source code.

```
#pragma weak A = A_Alias

int A(int x){
   return x*x;
}
int B(int y){
   int(*f)(int);
   f = &A;
   C(30);
   return 7 * f(y);
}
void C(int z){
   while(1){ };
}
int A_Alias(int x){
}
```

(b) Resulting CFI Map.



CFI Map (2/2)

(c) Resulting CFI Map data structure.

Nodes				
Identifier	Name	Prototype	Module	Return tag
290f2fd5	A	i32 (i32)	ex.c	1dc2aaf0
7d63f629	B	i32 (i32)	ex.c	6e28b9d1
6ba8458b	C	void (i32)	ex.c	164e44a8

Clusters			
Identifier	Prototype	Entry-point tag	Return tag
6a8597ea	i32 (i32)	69e1b040	46068a5c

Edges			
Identifier	From	To	Type
7dcdc019	7d63f629	6a8597ea	indirect
7728cc01	7d63f629	6ba8458b	direct

Aliases	
Identifier	Alias
290f2fd5	A_alias

Special Cases: Syscalls

All must return to same site: i.e., the syscall dispatcher

Some have very common prototypes: e.g., `i64 (void)`

If clustered, syscalls result in a large CFG relaxation

Solution: **Secondary Tags**

```
mov     (%rsp),%rdx
cmpl    $0x138395,0x4(%rdx)
je      9
cmpl    $0x11deadca,0x4(%rdx)
je      9
push    %rdx
callq   <kcfi_vhndl>
pop     %rdx
retq
```

Special Cases: Alternative Calls

Kernel does crazy stuff, like patching itself

(e.g, replaces callees based on available CPU features)

kCFI fixes this behavior by **clustering replaceable functions**

No CFG harm: only one of the alternative functions is used in each kernel run

Special Cases: Assembly (1/2)

Automatically handling inline Assembly is hard!

Requires patching the (kernel) source code

```
#define __put_user_x(size, x, ptr, __ret_pu)                  \
   asm volatile("call __put_user_" #size "\nnopl 0x00dead04"  \
   : "=a" (__ret_pu)                                          \
   : "0" ((typeof(*(ptr)))(x)), "c" (ptr) : "ebx")
```

Special Cases: Assembly (2/2)

The prototype of indirect calls in Assembly cannot be trivially inferred :(

Indirect calls missed:

- 6 calls used only during boot
- 5 calls that happen through **verified** tables
- 5 calls are based on data that need to be moved to `.rodata`

Attacks on Fine-grained CFI (1/2)

Control Jujutsu + Control-Flow Bending

Non-control-data attacks may allow arbitrary computation

Not demonstrated in kernel context

`printf() vs. printk()`

(but, of course, this doesn’t mean that they are impossible)

Attacks on Fine-grained CFI (2/2)

Attacks on **backward edges**

Defeatable through shadow stacks

In absence of a shadow stack, CGD raises the bar

Attacks on **forward edges**

Control Jujutsu examples are not feasible under kCFI heuristics

CFI can use composite methods to build tighter CFGs

CET: Control-Flow Enforcement Technology

Hardware shadow stack implementation (awesome)

Coarse-grained forward-edge CFI (not awesome)

Feature not yet available on Intel CPUs

Compatibility and performance are unknown

DROP THE ROP:

Fine Grained Control-Flow Integrity for The Linux Kernel

João Moreira

Sandro Rigo, Michalis Polychronakis, Vasileios Kemerlis

joao.moreira@lsc.ic.unicamp.br

black hat ASIA 2017

UNICAMP

BROWN