Hardware-assisted Run-time Protection

Thomas Nyman‡, N. Asokan†‡

https://asokan.org/asokan/
@nasokan

Acknowledgements: Hans Liljestrand†, Lachlan J Gunn‡, Jan-Erik Ekberg‡, §
†) University of Waterloo, ‡) Aalto University, §) Huawei Technologies
You will be learning

- **Part 1: Memory-related run-time attacks**
  - Common attack techniques against C/C++

- **Part 2: Hardware-assisted defenses**
  - Emerging mechanisms in CotS processors

- **Part 3: Theory of run-time attacks**
  - What are “weird machines”? 
Example: Buffer overflows caused by missing bounds checks.

```c
#include <stdio.h>

int main(int argc, char *argv[]) {
    puts("So... The End...");
    doit(argv[1]);
    puts("or... maybe not?");
    return 0;
}

void doit(char *str) {
    char buf[8];
    char *ptr = buf;

    strcpy(buf, str);
    puts(ptr);
}
```

The code demonstrates how a buffer overflow can occur when bounds checks are missing in the call to `strcpy()`. The executable file generated by the compiler and linker may have issues due to these missing bounds checks.
Run-time behaviour

User

$ ./a.out "Hello !"

```c
#include <stdio.h>

void doit(char *str)
{
    char *ptr = buf;
    strcpy(buf, str);
    puts(ptr);
}
```

```
004843b <doit>:
804843b: 55 push %ebp
804843c: 89 e5 mov %esp,%ebp
804843e: 83 ec 0c sub $0xc,%esp
8048441: 8d 45 f4 lea -0xc(%ebp),%eax
8048444: 89 45 fc mov %eax,-0x4(%ebp)
8048447: 50 push %eax
8048448: e8 ad fe ff ff call 8048300 <strcpy@plt>
8048453: 83 c4 08 add $0x8,%esp
8048456: ff 75 08 pushl -0x4(%ebp)
8048459: e8 b2 fe ff ff call 8048310 <puts@plt>
804845e: 83 c4 04 add $0x4,%esp
8048461: 90 nop
8048462: c9 leave
8048463: c3 ret
```
Control-flow hijacking

```c
void doit(char *str)
{
    char buf[8];
    char *ptr = buf;
    strcpy(buf, str);
    puts(ptr);
}
```

$ ./a.out $(perl -e 'print "A"x8 \
    
    ."\x??\x??\x04\x08" \
    ."A"x4 \
    ."\x64\x84\x04\x08" ;')

```
0xbfffd18a (argv[1])
0xbfffceb4
0xbfffceac
0xbfffceb0
0xbfffcea8
0xbfffcea4
0xbfffcea0
0xbfffce9c

parent stack frame
arguments:
    0xbfffd18a (argv[1])
return address:
    0x08048464 (main)
frame pointer:
    "A" "A" "A" "A"
ptr:
    0x0804???? (data)
    "A" "A" "A" "A"
buf:
    "A" "A" "A" "A"
    0xbffcea0 (ptr)
    0xbfffd18a (str)
strcpy stack frame
```

```
0x8048464:  <main>:
0x804843b:  <doit>:
0x8048300:  <strcpy>:
puts:
```

```
0x8048300:
```
0x8048310:
```
0x8048310:  puts:
```
0x0804843b:  <doit>:
0x08048464:  <main>:
```
lib init:
```

Kernel Space

User Space

Kernel space

Stack (grows down)

Memory Mapping Region

Heap (grows up)

Bss segment

Data segment

Text segment

0x40000000

0xffffffffff

0x00000000

0x00000000

0x8048310:
```
0x8048300:
```

control-flow hijacking

corrupt code pointer / control flow
Memory-related run-time attacks
Memory-related run-time attacks

Software written in memory unsafe languages such as C/C++
• Suffer from various memory-related errors

Memory errors may allow run-time attacks to compromise program behaviour
• Control-flow hijacking / code injection
• Return-Oriented Programming (ROP)
• Non-control-data attacks
• Data-Oriented Programming (DOP)
Run-time attacks compromise program behaviour

(i) Code-injection attack
(ii) Code-reuse attack
(iii) Non-control-data attack

if (authenticated != true)
then: call unprivileged()
else: call privileged()

unprivileged() { ... }
privileged() { ... }
(i) Code-injection attacks

Exploit memory error (e.g. buffer overflow) to:
• Inject shellcode into writable memory (usually stack)
• Corrupt code pointer (usually return address) to redirect execution flow to shellcode

Countermeasures:
• Stack canaries (1990)
  Detect sequential overwrites that corrupt ret. addr.
• W⊕X memory access control policy (2003)
  Prevent execution of shellcode by ensuring that memory pages are either writable or executable

Elias Levy (as Aleph One), Smashing the stack for fun and profit, Phrack 7 (1996)
Szekeres et al., SoK: Eternal War in Memory, IEEE SP (2013)
Classic code-injection

```
$ ./a.out $(perl -e 'print "A"x8 \n."\x??\x??\x04\x08" \n."A"x4 \n."\xb8\xce\xff\xfb" \n."\x80\xcd" ."\x40" \n."\xc0\x31" ."\x80\xcd" \n."\x0b\x0b" ."\xc2\x89" \n."\xc1\x89" ."\xe3\x89" \n."\xe6\xe9\xe6\xe2f\xe68" \n."\xe6\xe8\xe73\xe2f\xe68" \n."\x50" ."\xc0\x31")
```

```
void doit(char *str)
{
    char buf[8];
    char *ptr = buf;
    strcpy(buf, str);
    puts(ptr);
}
```

```
shellcode

0x8048300:   <strcpy>
0x8048464:   <main>
0x804843b:   <doit>
```

- **Library init**
- **0x8048464**: `<main>`
- **0x804843b**: `<doit>`
- **0x8048300**: `<strcpy>`
- **puts**
- **0x8048310**: `puts`
Return-oriented programming (high-level idea)

Sadeghi and Davi. Secure, Trusted and Trustworthy Computing, Runtime attacks. TU Darmstadt 2010
Return-oriented programming

Attacker arranges call stack with code pointers to existing code sequences ("gadgets")
• Given a suitable gadget set, arbitrary return-oriented programs can be constructed
(ii) Code-reuse attacks

Exploit memory error without injecting code:
- Corrupt code pointer (usually return address) to redirect execution flow to existing code:
  - Library functions (return-into-libc)
  - Pre-existing instruction sequences (gadgets)

Countermeasures:
- Control-flow Integrity (2005)
  Detect control-flow transfers outside static control-flow graph or mismatched returns (shadow stack)
- Address space randomization (2001)
  Hide locations of useful gadgets in memory

A. Peslyak (as Solar Designer), Getting around non-executable stack (and fix), Bugtraq (1997)
H. Shacham, The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86), ACM CCS (2007)
M. Abadi, Control-flow integrity, ACM CCS (2005)
CFI: High-level idea

- **CFI check at A**: Allowed edges: (A,B), (A,F)
- **CFI check at B**: Allowed edges: (B,A), (B,C), (B,D)
- **CFI check at C**: Allowed edges: (C,B), (C,G)
- **CFI check at D**: Allowed edges: (D,B), (D,G)
- **CFI check at G**: Allowed edges: (G,C), (G,D)
- **CFI check at F**: Allowed edges: (F,A)

**Legend:**
- • • • • • • intended forward-edge in CFG
- ← ← ← ← ← ← intended backward-edge in CFG
- • • • • • • malicious edge not part of CFG
- ○ ○ ○ ○ ○ ○ initial node in CFG
- ○ ○ ○ ○ ○ ○ node in CFG

**CFI violation at D**: Disallowed edge: (F,D)
Shadow Stack: High-level idea

A → B → C

Adversary tampers with shadow stack
Non-control data attack

Program logic that can be influenced as result of memory vulnerability constitute “data-oriented gadgets”

Attacker influences the behavior of benign program code without breaking control-flow integrity

void doit(char *str)
{
    char buf[8];
    char *ptr = buf;
    strcpy(buf, str);
    puts(ptr);
}

libc init:

$ ./a.out $(perl -e 'print "A"x8 \ 
./08\xb0\xc4\x09'' )
Data-oriented Programming

Enables expressive computation via use of “data-oriented gadgets” without diverging from program’s benign control-flow

- Requires a “gadget dispatch” that allows chaining together gadgets at will

Adversary exploits bug
Data-oriented programming

Given a suitable gadget dispatch, an attacker can chain together data-oriented gadgets at will.

Dispatch must be able to chain data-oriented gadgets without violating control-flow.

---

Diagram:
- Loop
- Selector
- Data segment
- Text segment
- Heap (grows up)
- Bss segment
- Stack (grows down)
- Memory Mapping Region
- Kernel space

Corrupt data flow
### Selected Research & Vulnerabilities

<table>
<thead>
<tr>
<th>Year</th>
<th>Research/Exploitation</th>
<th>Vulnerability</th>
</tr>
</thead>
<tbody>
<tr>
<td>2001</td>
<td><strong>Advanced ret2libc</strong>&lt;br&gt;Nergal (Phrack)</td>
<td></td>
</tr>
<tr>
<td>2005</td>
<td><strong>x86-64 borrowed code chunks exploitation</strong>&lt;br&gt;Krahmer</td>
<td>Non-control-data attacks&lt;br&gt;Chen et al (SSYM '05)</td>
</tr>
<tr>
<td>2007</td>
<td><strong>ROP on x86</strong>&lt;br&gt;Shacham (CCS'07)</td>
<td></td>
</tr>
<tr>
<td>2008</td>
<td><strong>ROP on ATMEL AVR</strong>&lt;br&gt;Francillon et al (CCS'08)</td>
<td></td>
</tr>
<tr>
<td>2009</td>
<td><strong>ROP Rootkits</strong>&lt;br&gt;Hund et al (USENIX Sec. '09)</td>
<td></td>
</tr>
<tr>
<td>2010</td>
<td><strong>ROP w/o Returns</strong>&lt;br&gt;Checkoway et al (CCS’10)</td>
<td>String-Oriented Programming&lt;br&gt;Payer (28C3. '11)</td>
</tr>
<tr>
<td>2014</td>
<td><strong>Out-of-Control</strong>&lt;br&gt;Göktas et al (IEEE S&amp;P’14)</td>
<td>Write Once, Pwn Anywhere&lt;br&gt;Yu (BlackHat USA’14)</td>
</tr>
<tr>
<td>2015</td>
<td><strong>Gadget size Matters</strong>&lt;br&gt;Göktas et al (USENIX’14)</td>
<td>Data-Oriented Exploits&lt;br&gt;Hu et al (USENIX Sec.'15)</td>
</tr>
<tr>
<td></td>
<td><strong>Control-flow Bending</strong>&lt;br&gt;Carlini et al (USENIX Sec.'16)</td>
<td>CVE-2016-0034: Angler RCE in Silverlight</td>
</tr>
</tbody>
</table>

Additional notes:
- **CVE-2010-3765**: Nobel Peace Price website 0day
- **CVE-2010-2883**: RCE in Adobe Reader and Acrobat
- **CVE-2011-1938**: RCE in PHP
- **CVE-2012-0003**: RCE in WMP MIDI library
- **CVE-2013-3893**: RCE in Internet Explorer
- **CVE-2014-0160**: Heartbleed vuln. in OpenSSL
- **CVE-2014-9222**: Misfortune cookie in RomPager
- **CVE-2016-0034**: Angler RCE in Silverlight
Taxonomy of Defenses

Adapted from Szekeres et al., *SoK: Eternal War in Memory*, IEEE SP (2013)
Software, coarse-grained

Containers
chroot
Virtual machines

Hardware, coarse-grained
Harvard architecture
TEEs
Memory segments
Protection rings (Exception levels)

Software, fine-grained

Software CFI
Privilege kernels
Memory-safe languages

Hardware, fine-grained
Tagged memory
HW-assisted CFI
Branch target indicators
Pointer Authentication
HW-assisted bounds checks
Fine-grained protection domains
Run-time scope enforcement

Memory Protection (MPU)
Virtual memory (MMU)
W Ξ X
Enclaves

Memory segments
Pointer Authentication Fine-grained protection domains

Part 2.
Hardware-assisted defenses
How to thwart run-time attacks?

Run-time attacks are now routine

Software defenses incur security vs. cost tradeoffs

Hardware-assisted defenses are attractive
Protect against run-time attacks without incurring a significant performance penalty
Design new hardware-security mechanisms

Example: HardScope

*Enforce variable visibility rules at run time*

Mitigate effects of attacks that corrupt data-plane information

Digital design, FPGA realization, compiler instrumentation, extensive analysis

Deployment challenge:

- Required the addition of 7 new instructions to the RISC-V ISA

Nyman et al. [HardScope: Hardening Embedded Systems Against Data-Oriented Attacks](https://www.dac-conf.org). DAC 2019
Hardware assisted defenses in CotS processors

**ARMv8-A mechanisms**
- Pointer Authentication (PA)
- Memory Tagging Extension (MTE)
- Branch Target Identification (BTI)

**Intel x86_64 mechanisms**
- Memory Protection eXtension (MPX)
- Memory Protection Keys (PKU)
- Control-flow Enforcement Technology (CET)
ARMv8-A mechanisms
Pointer Integrity: memory safety for pointers

Ensure **pointers** in memory remain **unchanged**

- **Code pointer integrity** implies CFI
  - Control-flow attacks manipulate code pointers

- **Data pointer integrity**
  - Reduces data-only attack surface

Kuznetsov et al. “**Code-Pointer Integrity**”, USENIX OSDI 2014
ARMv8.3-A Pointer Authentication

General purpose hardware primitive **approximating pointer integrity**
- Ensure pointers in memory remain unchanged

**Introduced in ARMv8.3-A specification** (2016) **to be improved in ARM-8.6-A** (2020)
- First compatible processors 2018 (Apple A12 / iOS12)
- Support in Linux 5.0
- Instrumentation support in GCC 7.0 ( -msign-return address, deprecated in GCC 9.0 -mbranch-protection=pac-ret[+leaf] GCC 9.0 and newer)
ARMv8.3-A PA – PAC Generation

Adds Pointer Authentication Code (PAC) into unused bits of pointer

- Keyed, tweakable MAC from pointer address and 64-bit modifier
- PA keys protected by hardware, modifier decided where pointer created and used

# ARMv8.3-A PA – Key management and instructions

## Keys for PAC generation and verification

<table>
<thead>
<tr>
<th>APIAKey_EL1</th>
<th>Key A for instruction address PACs</th>
</tr>
</thead>
<tbody>
<tr>
<td>APIBKey_EL1</td>
<td>Key B for instruction address PACs</td>
</tr>
<tr>
<td>APDAKey_EL1</td>
<td>Key A for data address PACs</td>
</tr>
<tr>
<td>APDBKey_EL1</td>
<td>Key B for data address PACs</td>
</tr>
<tr>
<td>APGAKey_EL1</td>
<td>Key for generic authentication</td>
</tr>
</tbody>
</table>

## PA Instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PAC&lt;i</td>
<td>d&gt;&lt;a</td>
</tr>
<tr>
<td>AUT&lt;i</td>
<td>d&gt;&lt;a</td>
</tr>
<tr>
<td>PACGA &lt;Xd&gt; &lt;Xn&gt; &lt;Xm&gt;</td>
<td>Calculate generic PAC for data in $Xn$ using modifier in $Xm$</td>
</tr>
<tr>
<td>XPAC&lt;i</td>
<td>d&gt; &lt;Xd&gt;</td>
</tr>
<tr>
<td>BRA&lt;a</td>
<td>b&gt; &lt;Xn&gt; &lt;Xm&gt;</td>
</tr>
<tr>
<td>BLRA&lt;a</td>
<td>b&gt; &lt;Xn&gt; &lt;Xm&gt;</td>
</tr>
<tr>
<td>RETA&lt;a</td>
<td>b&gt; &lt;Xn&gt; &lt;Xm&gt;</td>
</tr>
<tr>
<td>ERETA&lt;a</td>
<td>b&gt; &lt;Xn&gt; &lt;Xm&gt;</td>
</tr>
<tr>
<td>LDRA&lt;a</td>
<td>b&gt; &lt;Xt&gt; &lt;Xn&gt;</td>
</tr>
</tbody>
</table>

Operate on **instruction** keys only
Operate on **data** keys only
PA-based protection schemes

PA instructions are **primitives**, assembled to form **protection schemes**

Two main components:
- When are pointers “PACed” and “unPACed”?
- Which modifier is used at a given point?

What should the modifier be for a given pointer?
- For **security**: using many different modifiers makes **replay attacks harder**
- For **functionality**: large numbers of modifiers are **hard to keep track of**
Example: -mssign-return-address

Deployed in GCC 5.0 and LLVM/Clang 7.0

```
func {
  pacia LR, SP
  str LR...
  ...
  ...
  ldr LR
  autia LR, SP
  ret
}
```

- pacia – add PAC
- autia – authenticate

Stack

Deployed in GCC 5.0 and LLVM/Clang 7.0

Qualcomm “Pointer Authentication on ARMv8.3”, whitepaper 2017
PA return address protection as a canary

The signed return address effectively is a canary:
  • Any overflow that corrupts the return address is detected

More powerful than -stack-protector canaries:
  • Does not require reference value
  • Can be bound to contextual information (e.g., the SP value)
  • Protects return address against arbitrary writes

Also has similar weaknesses:
  • Existing return addresses can be reused

Liljestrand et al. PAC it up: Towards Pointer Integrity using ARM Pointer Authentication USENIX Security (2019)
PA only approximates fully-precise pointer integrity

Adversary may reuse PACs

```
... /* func1() */
br1 %func1
...
/* func2() */
br1 %func2
...

func1 {
pacia LR, SP
str LR
...
}

func2 {
pacia LR, SP
str LR
...
ldr LR ←
autia LR, SP
ret }
```

pacia – add PAC
autia – authenticate
Modifier: based on pointer type
- Assigned at compile-time based on C type
- “this pointer really points to this type of data or function”

On-load, on-branch authentication
- Branching with combined auth+branch instruction (lbraa)
- Iterating an array uses only one authentication

// ptr = …
…
mov Xmod, #type_id
pacia Xptr, Xmod

PACed only on pointer creation!

// *ptr
…
1dr Xptr, <memory>
mov Xmod, #type_id
autda Xptr, Xmod
<something> [Xptr]

Authenticated on use

// ptr();
…
…
…
mov Xmod, #type_id
lbraa Xptr, Xmod

Liljestrand et al. PAC it up: Towards Pointer Integrity using ARM Pointer Authentication USENIX Security (2019)
Authenticated Call Stack: high-level idea

Chained MAC of authentications tokens cryptographically bound to return addresses

- Provides modifier ($auth$) bound to all previous return addresses on the call stack
- Statistically unique to control-flow path
  - prevents reuse
  - allows precise verification of returns

\begin{align*}
auth_0 &= H_k(ret_0, 0) \\
auth_1 &= H_k(ret_1, auth_0) \\
& \vdots \\
auth_i &= H_k(ret_i, auth_{i-1}) \\
& \vdots \\
auth_n &= H_k(ret_n, auth_{n-1})
\end{align*}

$auth_i$, $i \in [0, n - 1]$ bound to corresponding return addresses, $ret_i$, $i \in [0, n]$, and $auth_n$

Mitigation of hash-collisions: PAC masking

- **Challenge**: PAC collisions occur on average after $1.253 \times 2^{b/2}$ return addresses
  - For $b=16$ $n = 321$ addresses

- **Solution**: Prevent recognizing collisions by masking each auth
  - pseudo-random mask generated using $\text{pacib}(0\times0, auth_{i-1})$

<table>
<thead>
<tr>
<th>Attack</th>
<th>w/o Masking</th>
<th>w/ Masking</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reuse previous auth collision</td>
<td>1</td>
<td>$2^{-b}$</td>
</tr>
<tr>
<td>Guess auth to existing call-site</td>
<td>$2^{-b}$</td>
<td>$2^{-b}$</td>
</tr>
<tr>
<td>Guess auth to arbitrary address</td>
<td>$2^{-2b}$</td>
<td>$2^{-2b}$</td>
</tr>
</tbody>
</table>

Maximum probability of success for different attacks
ARMv8.5-A Memory Tagging Extension

Ensures memory accesses are safe by comparing tag in pointer with tag in memory

Introduced in ARMv8.5-A specification (announced September 2018)
- Support in Linux proposed July 2019
- Stack Tagging will become available in LLVM 9
- Heap Tagging support planned

ARM. Armv8.5-A Memory Tagging Extension, whitepaper 2019
ARM. Opensource support for Armv8.5-A Memory Tagging Extension. 2019
ARMv8.5-A MTE

Address tags stored in top byte of pointer
- uses existing top-byte ignore feature

Allocation tags stored by transparently by hardware and cached
- 4-bit tag per 16-byte granule of memory

Mismatch between tags reported either:
- synchronously (precise check during testing), or
- asynchronously (imprecise checks after deployment)

ARM. Armv8.5-A Memory Tagging Extension, whitepaper 2019
Example: Stack Tagging

- Choose **random tag** on function entry
- For each slot in stack frame choose **tag at an offset** to initial tag
- Accesses using immediate offset from SP are unchecked

```c
int main(int argc, char *argv[]) {
    ...
    doit(argv[1]);
    ...
    return 0;
}

void doit(char *str) {
    char buf[16];
    char ptr = buf;
    strcpy(ptr, str);
    puts(ptr);
}

void strcpy(char *str) {
    ...
}
```
# ARMv8.5-A MTE – Instructions

## MTE Instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IRG</td>
<td>Insert random address tag to address in register</td>
</tr>
<tr>
<td>GMI</td>
<td>Manipulate excluded set of tags for IRG</td>
</tr>
<tr>
<td>ADDG/SUBG</td>
<td>Arithmetic on addresses with tags for creating pointer to objects on stack</td>
</tr>
<tr>
<td>SUBP(S)</td>
<td>56-bit subtract allowing address tag in top byte of pointers to be ignored</td>
</tr>
<tr>
<td>LDG/STG/STZG</td>
<td>Get or set allocation tags for granule (STGZ also initializes data to zero)</td>
</tr>
<tr>
<td>ST2G/STZ2G</td>
<td>Like STG or STZG but operate on two granules of memory at a time</td>
</tr>
<tr>
<td>STGP</td>
<td>Store both tag and data to memory</td>
</tr>
<tr>
<td>LDGM/STGM/STZGM</td>
<td>Bulk tag manipulation for initializing or serializing tags by system software</td>
</tr>
</tbody>
</table>


ARM. [Armv8.5-A Memory Tagging Extension](https://www.arm.com), whitepaper 2019
ARMv8.5-A Branch Target Identification

Hardware-assisted CFI similar to Intel CET Indirect Branch Tracking

Introduced in ARMv8.3-A specification (2016)
• Support for Linux proposed May 2019
• Instrumentation support in GCC 9.0 (-mbranch-protection=standard|bt1)
Indirect branches to **guarded code regions** require **marker instructions**

- compiler places marker potential indirect branch targets
- two classes of targets: **calls** and **jumps** (RET instructions not restricted by BTI)

### Branch sources

<table>
<thead>
<tr>
<th>BTI call type branches</th>
<th>BTI jump type branches</th>
</tr>
</thead>
<tbody>
<tr>
<td>BLR ...</td>
<td>Indirect function calls</td>
</tr>
<tr>
<td>BR &lt;x16</td>
<td>x17&gt;</td>
</tr>
</tbody>
</table>

| BR ... (except x16|x17) Branches to jump tables |

<table>
<thead>
<tr>
<th>BTI Marker Instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>BTI &lt;c</td>
</tr>
<tr>
<td>BRK</td>
</tr>
<tr>
<td>HLT</td>
</tr>
<tr>
<td>PACIASP / PACIBSP</td>
</tr>
</tbody>
</table>

Branch Target Identification for c=calls, j=jumps, cj=calls or jumps

Breakpoint Instruction

Halting breakpoint

Create PAC for Instruction address in LR using key A/B and SP as modifier

---

Adapted from Szekeres et al., *SoK: Eternal War in Memory*, IEEE SP (2013)
Intel x86_64 mechanisms
Intel Memory Protection Extension

Run-time checks for memory accesses to detect pointer bounds violations

Deployed in SkyLake microarchitecture (2015)
- Support in Linux 3.9, removal proposed May 2019
- Instrumentation support in GCC 5.0, removed in 9.0 (-fcheck-pointer-bounds)

Intel MPX – Example

```c
struct obj { char buf[100]; int len };
obj* a[10];

for (i=0; i<M; i++) {
    total += a[i]->len;
}
```

1: obj* a[10] // Array of pointers to objs
2: total = 0
3: for (i=0; i<M; i++):
4:    ai = a + i // Pointer arithmetic on a
5:    objptr = load ai // Pointer to obj at a[i]
6:    lenptr = objptr + 100 // Pointer to obj.len
7:    len = load lenptr
8:    total += len // Total length of all objs
## Intel MPX – Bound Check Instructions

### MPX Registers

| BND0 – BND03 | Bound Registers storing 64-bit LowerBound (LB) and 64-bit UpperBound (UB) |

### MPX Instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Format</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BNDMK</td>
<td>&lt;Reg&gt;  &lt;Addr&gt; &lt;offset&gt;</td>
<td>Create LB (Addr) and UB (Addr + offset) in Reg</td>
</tr>
<tr>
<td>BNDCL</td>
<td>&lt;Reg&gt;  &lt;Addr&gt;</td>
<td>Check Addr against LB in Reg</td>
</tr>
<tr>
<td>BNDCU</td>
<td>&lt;Reg&gt;  &lt;Addr&gt;</td>
<td>Check Addr against UB in Reg in 1’s compliment form</td>
</tr>
<tr>
<td>BNDCN</td>
<td>&lt;Reg&gt;  &lt;Addr&gt;</td>
<td>Check Addr against UB in Reg not in 1’s compliment form</td>
</tr>
<tr>
<td>BNDMV</td>
<td>&lt;Reg&gt;  &lt;Reg</td>
<td>Addr&gt;</td>
</tr>
<tr>
<td>BNDMV</td>
<td>&lt;Addr&gt; &lt;Reg&gt;</td>
<td>Store LB and UB from Reg to Addr</td>
</tr>
<tr>
<td>BNDLDX</td>
<td>&lt;Reg&gt;  &lt;SIB&gt;</td>
<td>Load LB and UB from bound directory</td>
</tr>
<tr>
<td>BNDSTX</td>
<td>&lt;SIB&gt;  &lt;Reg&gt;</td>
<td>Store LB and UB to bound directory</td>
</tr>
</tbody>
</table>

Intel. [Intel® 64 and IA-32 Architectures Software Developer Manuals](https://ark.intel.com), Volume 1. Chapter 17.4. May 2019
**Intel MPX – Example**

```c
struct obj { char buf[100]; int len }
obj* a[10]
1: for (i=0; i<M; i++) {
2:     total += a[i]->len;
3: }
```

```c
1: obj* a[10] // Array of pointers to objs
2: a_b = bndmk a, a+79 // Make bounds [a, a+79]
3: total = 0
4: for (i=0; i<M; i++):
5:     ai = a + i // Pointer arithmetic on a
6:     bndcl a_b, ai // LowerBound check of a[i]
7:     bndcu a_b, ai+7 // UpperBound check of a[i]
8:     objptr = load ai // Pointer to obj at a[i]
9:     objptr_b = bndldx ai // Bounds for pointer at a[i]
10: lenptr = objptr + 100 // Pointer to obj.len
11:     bndcl objptr_b, lenptr // Check LowerBound of obj.len
12:     bndcu objptr_b, lenptr+3 // Check UpperBound of obj.len
13:     len = load lenptr
14:     total += len // Total Length of all objs
```
Intel MPX – Bound Directory

Figure 17-4. Bound Paging Structure and Address Translation in 64-Bit Mode

Intel. Intel® 64 and IA-32 Architectures Software Developer Manuals. Volume 1. Chapter 17.4.3.1 May 2019
Intel MPX – Limitations

Study by Oleksenko et al. identified the following limitations:

- overhead comparable to software-based (up to 4x slowdown, ~50% overhead on average)
- no protection against temporal memory safety errors (e.g. use-after-free)
- no support for multithreading, can lead to unsafe data races between threads
- no support for common C/C++ idioms due to memory layout restrictions
- conflicts with other ISA extensions (Intel TSX, SGX)
- instrumentation incurs significant performance penalty (> 15%) even if MPX not available

Susceptible to Bounds Check Bypass due to Meltdown speculative execution attack

- exploits lazy handling of raised bound range ( #BR) exception

Canella et al. A Systematic Evaluation of Transient Execution Attacks and Defenses. USENIX Sec ‘19
Adapting MPX for kernel code

Bound Directory cannot be used in kernel code

- kernel cannot handle page faults at arbitrary points within its own execution
- pre-allocating the bound directory and bound tables not feasibly due to memory overhead
  bound directory for 64-bit kernel is $2^{28}$ 64-bit entries = 2 Gbytes
  each bound table $2^{17}$ 32-byte entries = 4 Mbytes

Solution: dynamically determine pointer bounds using existing kernel metadata

https://ssg.aalto.fi/research/projects/kernel-hardening/

Intel Protection Keys for Userspace

User-level memory access-control mechanism at page granularity
- Associates each memory page with a 4-bit protection key kept in page table entry
- Access control rules for protection keys maintained by userspace code in PKRU register

Deployed in SkyLake microarchitecture (server configuration, 2015)
- Support in Linux 4.6 (known as MPK)
- Userspace support in GNU C Library (glibc) 2.27, GCC 9.1 (pkey_mprotect)
## PKU Registers

<table>
<thead>
<tr>
<th>PKRU</th>
<th>Protection Key Right Register</th>
</tr>
</thead>
</table>

### PKU Instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RDPKRU</td>
<td>Read PKRU value to EAX</td>
</tr>
<tr>
<td>WRPKRU</td>
<td>Write EAX value to PKRU</td>
</tr>
</tbody>
</table>

**Figure 2-9. Protection Key Rights Register for User Pages (PKRU)**

Write disable, access disable
Intel Control-flow Enforcement Technology

Hardware-assisted Control-Flow Integrity (CFI) to prevent control-flow hijacking

Not yet deployed (announced 2016, updated specification May 2019)
- Support in Linux proposed June 2018 [mem-mgmt, usrermode SHSTK, IBT]
- Runtime support in glibc 2.28, Instrumentation support in GCC 8.0 (-fcf-protection)

Intel CET – Shadow Stack

Mechanism for protecting return address stored on the call stack
• introduces second stack used exclusively for copies of return addresses
• return address popped from both stacks on return and compared

Writes to shadow stack restricted to control-flow and management instructions
• shadow stack pages protected by page table protections (additional ”shadow stack” attr)
• page protection also prevents overflow and underflow of shadow stack

New architectural register: Shadow Stack Pointer
• Cannot be directly encoded as source, destination or memory operand by instructions
Intel CET – Shadow Stack Instructions

### Shadow Stack Registers

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SSP</td>
<td>Shadow Stack Pointer</td>
</tr>
<tr>
<td>IA32_PL3_SSP - IA32_PL0_SSP</td>
<td>MSRs for storing SSP value on privilege level transition</td>
</tr>
</tbody>
</table>

### Shadow Stack Instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CALL/ICALL</td>
<td>Push return address to call and shadow stack and branch</td>
</tr>
<tr>
<td>RET/IRET</td>
<td>Compare return addresses in both stacks and return if addresses match</td>
</tr>
</tbody>
</table>

**Shadow Stack Management Instructions**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>INCSSP</td>
<td>Increment shadow stack by ( n ) frames</td>
</tr>
<tr>
<td>RDSSP</td>
<td>Read SSP value to ( Reg )</td>
</tr>
<tr>
<td>SAVEPREVSSP</td>
<td>Save SSP to ”shadow stack restore” token on current shadow stack</td>
</tr>
<tr>
<td>RSTORSSP</td>
<td>Restore SSP from ”shadow stack restore” token at ( Addr )</td>
</tr>
<tr>
<td>WRSS</td>
<td>Write to current shadow stack</td>
</tr>
<tr>
<td>WRUSS</td>
<td>Write to user shadow stack (privileged mode only)</td>
</tr>
<tr>
<td>SETSSBSY</td>
<td>Verify ”supervisor shadow stack” token and set busy bit</td>
</tr>
<tr>
<td>CLRSSBSY</td>
<td>Verify ”supervisor shadow stack” token at ( Addr ) and clear busy bit</td>
</tr>
</tbody>
</table>

Intel CET – Indirect Branch Tracking

Prevents diverting indirect CALL/JMP to invalid targets
• typical attack vector in call/jmp-oriented programming attacks
• achieves only weak CFI guarantees (single class of targets)

Requires indirect JMP / CALL to target specific marker instructions
• compiler places marker at all potential indirect branch targets
• new control-protection exception (#CP) raised otherwise

IBT Marker Instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ENDBRANCH32</td>
<td>Marker instruction in 32-bit mode</td>
</tr>
<tr>
<td>ENDBRANCH64</td>
<td>Marker instruction in 64-bit mode</td>
</tr>
</tbody>
</table>

Taxonomy of Defenses

Adapted from Szekeres et al., *SoK: Eternal War in Memory*, IEEE SP (2013)
Comparison
## Intel MPX vs. ARMv8.5-A MTE

<table>
<thead>
<tr>
<th>Feature</th>
<th>Intel MPX</th>
<th>ARMv8.5-A MTE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Spatial error protection</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Temporal error protection</td>
<td>×</td>
<td>✓</td>
</tr>
<tr>
<td>Enforcement model</td>
<td>Deterministic</td>
<td>Probabilistic (16 classes)</td>
</tr>
<tr>
<td>Memory Overhead</td>
<td>High</td>
<td>?</td>
</tr>
<tr>
<td>Run-time Overhead</td>
<td>Moderate to High</td>
<td>?</td>
</tr>
</tbody>
</table>
## Intel CET ShadowStack vs. ARMv8.3-A PA

<table>
<thead>
<tr>
<th>Feature</th>
<th>Intel CET Shadow Stack</th>
<th>ARMv8.3-A PA</th>
</tr>
</thead>
<tbody>
<tr>
<td>Return address protection</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Indirect branch protection</td>
<td>×</td>
<td>capable*</td>
</tr>
<tr>
<td>Data pointer protection</td>
<td>×</td>
<td>capable*</td>
</tr>
<tr>
<td>Enforcement model</td>
<td>Deterministic</td>
<td>Probabilistic**</td>
</tr>
<tr>
<td>Immune to pointer reuse</td>
<td>✓</td>
<td>×</td>
</tr>
<tr>
<td>Memory Overhead</td>
<td>Low to Moderate</td>
<td>N/A</td>
</tr>
<tr>
<td>Run-time Overhead</td>
<td>? (likely low)</td>
<td>Low</td>
</tr>
</tbody>
</table>

*) Liljestrand et al. [PAC It Up: Towards Pointer Integrity using ARM Pointer Authentication](https://www.usenix.org/conference/usenixsecurity19/presentation/liljestrand), USENIX Security’19

**) Liljestrand et al. [PACStack: an Authenticated Call Stack](https://www.dropbox.com/s0f7pfxjthv7srja/pacstack.pdf?dl=0), Preprint 2019.
# Intel IBT vs. ARMv8.3-A BTI

<table>
<thead>
<tr>
<th>Feature</th>
<th>Intel IBT</th>
<th>ARMv8.3-A BTI</th>
</tr>
</thead>
<tbody>
<tr>
<td>Indirect branch protection</td>
<td>✔ (one class)</td>
<td>✔ (two classes)</td>
</tr>
<tr>
<td>Enforcement model</td>
<td>Deterministic</td>
<td>Deterministic</td>
</tr>
<tr>
<td>Memory Overhead</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>Run-time Overhead</td>
<td>? (likely low)</td>
<td>? (likely low)</td>
</tr>
</tbody>
</table>

## Explanation
- **Indirect branch protection**: Intel IBT supports branch target indirection table (BTI) protection with one class, while ARMv8.3-A BTI supports two classes.
- **Enforcement model**: Both Intel IBT and ARMv8.3-A BTI use a deterministic enforcement model.
- **Memory Overhead**: Both have no additional memory overhead (N/A).
- **Run-time Overhead**: The overhead is likely low for both technologies.
A theory of run-time attacks
von Neumann architecture

Architecture for a stored-program computer
- Realizes (theoretical) concept of universal Turing machine
- Instructions and data stored in memory
- Operates by changing internal state, i.e., instructions read and modify some data.

Computer (circa 1945)

With a large addressable memory, different memory types (e.g. SRAM, DRAM flash etc.) and I/O map onto single memory space
Programs as intended finite state machines

Design of program $p$ can be modeled as (potentially very large) finite state machine$^†,‡$

- The intended finite state machine (IFSM) describes the intended function of $p$
- To execute the IFSM on real-world computers, $p$ is realized as a software emulator for the IFSM

$$\theta = (Q, i, F, \Sigma, \Delta, \delta, \sigma)$$

The IFSM represents a bug-free version of $p$

$p$ is a (potentially faulty) emulator for the IFSM

$p$ runs on a processor $cpu$

$†)$ or a finite state transducer if output is possible

$‡)$ non-equivalence of FSM/FST to a Turing machine does not matter as any real-world computing device has finite memory

$§)$ $Q =$ set of states, $i =$ initial state

$F =$ final state, $\Sigma, \Delta =$ input and output alphabets

state transition function $\delta: Q \times \Sigma \to Q$,

output function $\sigma: Q \times \Sigma \to \Delta$
$Q^p_{cpu} = Q_{cpu}^{IFS} \cup Q_{cpu}^{trans}$

$Q_{cpu}^{IFS}$: concrete states of target machine that map to a state in the IFSM

$Q_{cpu}^{trans}$: benign transitory states that occur during emulation of an edge in the IFSM; part of intended transitions

Two different perspectives of $\theta$

State 1 \(\rightarrow\) State 2 \(\rightarrow\) State 3 \(\rightarrow\) State 4 \(\rightarrow\) State 5

Program $p$

Data (user’s “program”)

Program $p$ (“data” from user’s perspective)

What is a “weird state”? 

$Q_{cpu} = Q_{IFS}^M \cup Q_{trans} \cup Q_{weird}$

$Q_{IFS}^M$: concrete states of target machine that map to a state in the IFSM

$Q_{trans}^M$: benign transitory states that occur during emulation of an edge in the IFSM; part of intended transitions

$Q_{cpu}$: set of states in $Q_{cpu}$ not in $Q_{IFS}^M$ nor $Q_{trans}^M$

Weird states arise unintentionally and have no meaningful interpretation in the IFSM

Reaching a weird state

Intuitively: a bug has occurred when cpu enters a weird state

Vulnerability
• method of moving $p$ to a weird state (accessible to attacker)

Exploitation; run-time attack
• process of choosing $q_i$, entering $q_{init}$ and programming resulting “weird machine” in order to violate security properties of the IFSM

Weird machines

A weird machine is a computational device where IFSM transitions operate on weird states

\[ \theta = (Q, i, F, \Sigma, \Delta, \delta, \sigma) \]

- \( Q \) = set of states, \( i \) = initial state
- \( F \) = final state, \( \Sigma, \Delta \) = input and output alphabets
- state transition function \( \delta: Q \times \Sigma \rightarrow Q \), output function \( \sigma: Q \times \Sigma \rightarrow \Delta \)

Instruction stream depends on input
- weird machine programmed through carefully crafted input to \( p \) once \( q_{init} \) has been entered

Emergent instruction set
- attacker (programmer of the weird machine) must discover the (often unwieldy) semantics of instructions

Unknown state space
- depends heavily on \( p \) and \( q_{init} \)

Unknown computational power
- greater complexity of the IFSM may yield greater number of instructions, but whether or not the instructions are usable is difficult to predict

Recall:
\[ \psi = \left( Q_{cpu}, q_{init}, Q_{IFSM} \cup Q_{trans}, \Sigma', \Delta', \delta', \sigma' \right) \]

Possible sources of weird states

Human error when program $p$ is developed
- Memory-related errors, e.g.,
  - spatial errors (buffer overflows)
  - temporal errors (use-after-free)
- Logic errors, e.g., integer overflow

Hardware faults when $p$ is executed
- Probabilistically deterministic hardware
- Fault injection, e.g., Rowhammer

Transcription errors when $p$ is transmitted over error-prone medium
- Hardware failure, e.g., hard drive

Y. Kim et al., *Flipping Bits in Memory Without Accessing Them: An Experimental Study of DRAM Disturbance Errors*, ISCA (2014)
Modelling the attacker

**Arbitrary program-point, chosen-bitflip**
  can stop $p$ anywhere, flip any one bit in memory, and continue

**Arbitrary program-point, chosen-bitflip, registers**
  same as above, but cannot modify/access registers

**Fixed program-point, chosen-bitflip, registers**

**Fixed program-point, sequential memory rewriting, registers**
  *classical buffer overflow*

...  

**Arbitrary program-point, arbitrary memory-rewriting, registers**
  *most powerful adversary*

---

Defining security

Depends on the desired security goal of $\theta$ and $p$: e.g., not disclose sensitive information $s$

Attacker defines $\theta_{\text{exploit}}$ to (adapatively) interact with $\theta_{\text{weird}}$

Attacker wins if $s$ is in the output of $\theta_{\text{weird}}$ with a higher probability than random

New hardware-assisted defenses are emerging and are (going to be) widely available.

How to utilize available primitives **effectively**?
- Towards pointer integrity with PA ([USENIX SEC ’19](https://ssg.aalto.fi/research/projects/harp/))

How to deal with **downsides**?
e.g. **optimally minimize scope for PA reuse attacks**?
- For return addresses: PACStack ([arXiv:1905.10242](https://ssg.aalto.fi/research/projects/harp/))
- For other types of pointers?

How do different hardware primitives compare?

How can we formalize run-time attacks and defenses?
Acknowledgments

Icons on slides 3, 4, 5, 15, 16, 17, 19, 21, 22, 23 and 24 made by Good Ware from www.flaticon.com licensed by CC 3.0 BY

The PHP logo on slide 25 made by Colin Viebrock licensed by CC BY-SA 4.0

The BSD daemon on slide 25 is copyright of Marshall Kirk McKusick

All product and company names and logos are trademarks™ or registered ® trademarks of their respective holders. Use of them does not imply any affiliation with or endorsement by them.
PARTS: Towards Pointer Integrity using ARM Pointer Authentication

Hans Liljestrånd†, Thomas Nyman†, Kui Wang‡, Carlos Chinea Perez‡, Jan-Erik Ekberg‡, N. Asokan†

†) Aalto University, ‡) Huawei Technologies
Can PI be realized in practice?

Can we use ARM v8.3-A PA?

But, PA is vulnerable to pointer reuse!

Our work: Design PA-assisted Run-time Safety (PARTS)

• Return address signing ≈ backward-edge CFI
• Code pointer signing ≈ forward-edge CFI
• Data pointer signing ≈ data-flow integrity for pointers
• Mitigates pointer reuse with run-time type safety
Our goal: Strengthen PA-based protection

1) Expand scope of PA protection to all pointers

2) Mitigate reuse attacks
Design
On choosing a PAC modifier

Without modifier all signed pointers are interchangeable

Ideally modifiers should be unique for each pointer and pointer value
• must be available at both creation and authentication
• must not be modifiable by attacker

Strawman design choices:
• Using unique static modifiers only
  • But cannot work for pointers assigned conditionally or re-assigned at run-time
• Using a nonce as a modifier
  • But needs to be stored securely
Expands scope of PA protection

- Return address signing
- Code pointer signing
- Data pointer signing

Mitigates pointer reuse by binding

- return addresses to the function definition
- code and data pointers to the pointer type
Hardening return address signing

SP as modifier is convenient
- It changes at run-time and has same value at pac / aut
- But reuse possible when SP values coincide

Modifier: SP + function-id
- ID assigned at compile-time
- Prevent cross-function reuse

```
func {
  mov Xmod, SP
  mov Xmod, #f_id, #lsl_16
  pacia LR, Xmod
  ...
  ...
  mov Xmod, SP
  mov Xmod, #f_id, #lsl_16
  retab Xmod
}
```

pacib – add PAC with instr A-key
retab – authenticate and return

Mashtizadeh et al. “Cryptographically Enforced Control Flow Integrity”, ACM CCS 2015
Code pointer signing

Modifier: based on pointer type
- type_id assigned at compile-time

Uses on-use (i.e., on-branch) authentication
- Branches use combined auth+branch instr. (lbraa)
- No intermediate authentication

// void (*Xptr)(void) =
...

mov Xmod, #type_id

pacia Xptr, Xmod

PACed only on pointer creation!

// Xptr();
...

mov Xmod, #type_id

lbraa Xptr, Xmod

Authenticated on use

pacia – add PAC with instr A-key
lbraa – authenticate and branch
Data pointer signing

Modifier: based on pointer type
• type_id assigned at compile-time

Uses on-load authentication
• Improves performance
  • e.g. only one authentication when iterating arrays
• Register allocation causes a challenge
  • e.g., how to handle register spills securely?

Listing:
```asm
/* data *Xptr */
mov Xmod, #type_id
pacda Xptr, Xmod
str Xptr, <memory>
...

... use(ptr); */
ldr Xptr, <memory>
mov Xmod, #type_id
autda Xptr, Xmod
...
```

`pacda` – add PAC with data A-key
`autda` – authenticate and branch
Implementation and evaluation
PARTS implementation

LLVM 6.0 (now 8.0) based instrumentation

• **Using opt for high-level instrumentation**
  • Using LLVM intrinsics for pointer type handling

• **AArch64 backend modifications**
  • Lower intrinsics to HW-specific instructions
  • Recognizing and protecting register spills
Evaluation: nbench benchmarks

Functional evaluation on ARM FVP simulator for correctness

Estimated performance overhead based on 4-cycles per PA instruction

- Return address signing < 0.5% (geo.mean)
- Code pointer signing < 0.5% (geo.mean)
- Data pointer signing ~19.5% (geo.mean)
PACStack: Authenticated Call Stack

Hans Liljestrand†‡, Thomas Nyman†, Lachlan J Gunn†, Jan-Erik Ekberg†‡, N. Asokan†

†) Aalto University, ‡) Huawei Technologies
Challenge: return address protection

Recall: Using the **SP value as modifier only reduces set of interchangeable pointers**
- still **allows reuse attacks when SP values coincide**

**PARTS** further narrows the scope of modifier values
- but **cannot completely prevent reuse attacks**
Authenticated Call Stack: high-level idea

Chained MAC of authentications tokens cryptographically bound to return addresses

- Provides modifier (*auth*) bound to all previous return addresses on the call stack
- Statistically unique to control-flow path
  - prevents reuse
  - allows precise verification of returns

\[
\text{auth}_0 = H_k(\text{ret}_0, 0) \quad \text{auth}_1 = H_k(\text{ret}_1, \text{auth}_0) \quad \ldots \quad \text{auth}_i = H_k(\text{ret}_i, \text{auth}_{i-1}) \quad \ldots \quad \text{auth}_n = H_k(\text{ret}_n, \text{auth}_{n-1})
\]

\(\text{auth}_i, i \in [0, n - 1]\) bound to corresponding return addresses, \(\text{ret}_i, i \in [0, n]\), and \(\text{auth}_n\)
PACStack: ACS implementation using PA

Two variants:

1. Generate 32-bit \( auth \) with pacga instruction and store on stack

2. Generate 16-bit \( auth \) with pacib instruction and embed in PAC-bits

- Topmost \( auth_n \) always stored securely in dedicated CPU register
Mitigation of hash-collisions: PAC masking

- **Challenge:** PAC collisions occur on average after $1.253 \times 2^{b/2}$ return addresses
  - For $b=16$ $n=321$ addresses

- **Solution:** Prevent recognizing collisions by masking each auth
  - pseudo-random mask generated using $\text{paci}(\emptyset \emptyset, auth_{i-1})$

<table>
<thead>
<tr>
<th>Attack</th>
<th>w/o Masking</th>
<th>w/ Masking</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reuse previous auth collision</td>
<td>1</td>
<td>$2^{-b}$</td>
</tr>
<tr>
<td>Guess auth to existing call-site</td>
<td>$2^{-b}$</td>
<td>$2^{-b}$</td>
</tr>
<tr>
<td>Guess auth to arbitrary address</td>
<td>$2^{-2b}$</td>
<td>$2^{-2b}$</td>
</tr>
</tbody>
</table>

Maximum probability of success for different attacks
Evaluation: SPEC CPU 2017 C-language benchmarks

Estimated performance overhead based on 4-cycles per PA instruction
- without masking < 0.5% (geo.mean)
- with masking < 1% (geo.mean)
New hardware-assisted defenses are emerging and are (going to be) widely available.

How to utilize available primitives effectively?
e.g. expand scope of PA protection to all pointers?
• Towards pointer integrity with PA (USENIX ’19)

How to deal with downsides?
e.g. optimally minimize scope for PA reuse attacks?
• For return addresses: PACStack (arXiv:1905.10242)
• For other types of pointers?

Use other emerging hardware primitives for run-time protection?
e.g. capability enforcement (CHERI)