Hardware-assisted Run-time Protection

N. Asokan‡†

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

Acknowledgements: Thomas Nyman‡, 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

Software Developer

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);
}

missing bounds-checks in call to strcpy!
Run-time behaviour

User

$ ./a.out “Hello !”

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

```
......
08048464 <main>:
......
0804847d: e8 b9 ff ff ff call 804843b <doit>
......
08048482: 83 c4 04       add $0x4,%esp
......
0804848f: 83 c4 04       add $0x4,%esp
......
```

```
......
0804843b: 55             push %ebp
0804843c: 89 e5          mov %esp,%ebp
0804843e: 83 ec 0c       sub $0xc,%esp
08048441: 8d 45 f4       lea -0xc(%ebp),%eax
08048444: 89 45 fc       mov %eax,-0x4(%ebp)
08048447: ff 75 08       pushl 0x8(%ebp)
0804844a: 8d 45 f4       lea -0xc(%ebp),%eax
0804844d: 50             push %eax
0804844e: e8 ad fe ff ff call 8048300 <strcpy@plt>
08048453: 83 c4 08       add %eax,0x8
08048456: ff 75 fc       pushl -0x4(%ebp)
08048459: e8 b2 fe ff ff call 8048310 <puts@plt>
0804845e: 83 c4 04       add $0x4,%esp
08048461: 90             nop
08048462: c9             leave
08048463: c3             ret
```

```
......
0804843b <doit>:
......
08048444: 89 45 fc       mov %eax,-0x4(%ebp)
08048447: ff 75 08       pushl 0x8(%ebp)
0804844a: 8d 45 f4       lea -0xc(%ebp),%eax
0804844d: 50             push %eax
0804844e: e8 ad fe ff ff call 8048300 <strcpy@plt>
08048453: 83 c4 08       add %eax,0x8
08048456: ff 75 fc       pushl -0x4(%ebp)
08048459: e8 b2 fe ff ff call 8048310 <puts@plt>
0804845e: 83 c4 04       add $0x4,%esp
08048461: 90             nop
08048462: c9             leave
08048463: c3             ret
```

```
......
parent stack frame
......
arguments: 0xbffeeab (argv[1])
return address: 0x8048482 (saved eip)
frame pointer: 0xbffeeb0 (saved ebp)
ptr: 0xbffeea0 (%ebp)
buf: 0xbfffce9c
ptr:
buf:
arguments: 0xbffeeab (ptr)
0xbffeea8
ptr:
buf:
arguments: 0xbffeeab (ptr)
0xbffeea8
ptr:
buf:
arguments: 0xbffeeab (ptr)
......
```
Control-flow hijacking

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

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

libc init:

08048464: <main>:

0804843b: <doit>:

0x8048300: <strcpy>:

8048310: puts:

parent stack frame

arguments:

return address:

frame pointer:

ptr:

buf:

0xbfffd18a (argv[1])
0xbfffceb8 (saved ebp)
0xbfffceac
0xbfffce0c
0xbfffce9c
0xbfffce00
0xbfffce80
0xbfffce40
0xbfffce8c
0xbfffcebc
0xbfffce44
0xbfffce08
0xbfffceb0
0xbfffceb4
0xbfffceb8
0xffffffff
0x08040000
0x00000000
0x40000000
0xc0000000

Kernel Space

User Space

Stack (grows down)

Memory Mapping Region

Heap (grows up)

Bss segment

Data segment

Text segment

corn code pointer / control flow

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

libc init:

08048464: <main>:

0804843b: <doit>:

0x8048300: <strcpy>:

8048310: puts:

parent stack frame

arguments:

return address:

frame pointer:

ptr:

buf:

0xbfffd18a (argv[1])
0xbfffceb8 (saved ebp)
0xbfffceac
0xbfffce0c
0xbfffce9c
0xbfffce00
0xbfffce80
0xbfffce40
0xbfffce8c
0xbfffcebc
0xbfffce44
0xbfffce08
0xbfffceb0
0xbfffceb4
0xbfffceb8
0xffffffff
0x08040000
0x00000000
0x40000000
0xc0000000

Kernel Space

User Space

Stack (grows down)

Memory Mapping Region

Heap (grows up)

Bss segment

Data segment

Text segment

corn 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

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

2. unprivileged() { ... }

3. privileged() { ... }
   ...

Adversary exploits bug

shellcode
(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

```sh
$ ./a.out $(perl -e 'print "A"x8 \n."\x??\x??\x04\x08" \n."A"x4 \n."\xb8\xc0\x04\x08" \n."\x80\xcd" ."\x40" \n."\xc0\x31" ."\x80\xcd" \n."\x00\xb0" ."\xc2\x89" \n."\xc1\x89" ."\xe3\x89" \n."\xe6\xe6\xe2\x86\x92\xe6" ."\xe6\xe6\xe2\x86\x92\xe6" \n."\x50" ."\xc0\x31")
```

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

```
void libc init:
0x8048464: <main>:
0x804843b: <doit>:
0x8048300: <strcpy>:

parent stack frame:
shellcode

0xbfffc0 (ptr)

strcpy stack frame
```

```
0xbfffc18a (str)
0xbfffd18a
```
Return-oriented programming (high-level idea)
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

Adversary exploits bug
(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

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 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 E
Allowed edges: (E,A)

CFI check at F
Allowed edges: (F,A)

CFI check at G
Allowed edges: (G,C), (G,D)

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

A → B → C

“Shadow stack”

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

$ ./a.out $(perl -e 'print "A"x8 \ ."\x08\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 program

loop
selector

... copy()
...

#2

#1

#3

store()
...

load()
...
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.
<table>
<thead>
<tr>
<th>Year</th>
<th>Research &amp; Vulnerabilities</th>
</tr>
</thead>
<tbody>
<tr>
<td>1988-99</td>
<td><strong>ret2libc</strong>&lt;br&gt;Solar Designer (Phrack)</td>
</tr>
<tr>
<td></td>
<td><strong>Format string vulnerabilities</strong>&lt;br&gt;Anders (Bugtraq. 1999)</td>
</tr>
<tr>
<td>2001</td>
<td><strong>Advanced ret2libc</strong>&lt;br&gt;Nergal (Phrack)</td>
</tr>
<tr>
<td>2005</td>
<td><strong>x86-64 borrowed code chunks exploitation</strong>&lt;br&gt;Krahmer</td>
</tr>
<tr>
<td>2007</td>
<td><strong>ROP on x86</strong>&lt;br&gt;Shacham (CCS’07)</td>
</tr>
<tr>
<td>2008</td>
<td><strong>ROP on ATMEL AVR</strong>&lt;br&gt;Francillon et al (CCS’08)</td>
</tr>
<tr>
<td></td>
<td><strong>ROP on SPARC</strong>&lt;br&gt;Buchanan et al (CCS’08)</td>
</tr>
<tr>
<td>2009</td>
<td><strong>ROP Rootkits</strong>&lt;br&gt;Hund et al (USENIX Sec. ’09)</td>
</tr>
<tr>
<td></td>
<td><strong>ROP on PowerPC</strong>&lt;br&gt;FX Lindner (BlackHat USA)</td>
</tr>
<tr>
<td></td>
<td><strong>ROP on ARM / iOS</strong>&lt;br&gt;Miller et al (BlackHat Europe)</td>
</tr>
<tr>
<td>2010</td>
<td><strong>ROP w/o Returns</strong>&lt;br&gt;Checkoway et al (CCS’10)</td>
</tr>
<tr>
<td></td>
<td><strong>CVE-2010-3765</strong>: Nobel Peace Price website 0day</td>
</tr>
<tr>
<td></td>
<td><strong>CVE-2010-2883</strong>: RCE in Adobe Reader and Acrobat</td>
</tr>
<tr>
<td>2011-12</td>
<td><strong>JIT-ROP</strong>&lt;br&gt;Snow et al (IEEE S&amp;P’13)</td>
</tr>
<tr>
<td></td>
<td><strong>CVE-2013-3893</strong>: RCE in Internet Explorer</td>
</tr>
<tr>
<td>2013</td>
<td><strong>Blind ROP</strong>&lt;br&gt;Bittau et al (IEEE S&amp;P’14)</td>
</tr>
<tr>
<td></td>
<td><strong>CVE-2014-0160</strong>: Heartbleed vuln. in OpenSSL</td>
</tr>
<tr>
<td></td>
<td><strong>CVE-2014-9222</strong>: Misfortune cookie in RomPager</td>
</tr>
<tr>
<td></td>
<td><strong>Data-Oriented Exploits</strong>&lt;br&gt;Hu et al (USENIX Sec.’15)</td>
</tr>
<tr>
<td>2015</td>
<td><strong>Gadget size Matters</strong>&lt;br&gt;Göktas et al (USENIX’14)</td>
</tr>
<tr>
<td></td>
<td><strong>CVE-2014-0016</strong>: Angler RCE in Silverlight</td>
</tr>
<tr>
<td>2016</td>
<td><strong>SROP</strong>&lt;br&gt;Bosman et al (IEEE S&amp;P’14)</td>
</tr>
<tr>
<td></td>
<td><strong>CVE-2016-0034</strong>: Angler RCE in Silverlight</td>
</tr>
<tr>
<td></td>
<td><strong>Control-flow Bending</strong>&lt;br&gt;Carlini et al (USENIX Sec.’16)</td>
</tr>
<tr>
<td></td>
<td><strong>DOP</strong>&lt;br&gt;Hu et al (IEEE S&amp;P ’16)</td>
</tr>
</tbody>
</table>
Taxonomy of Defenses

1. Memory vulnerability
   - Out-of-bounds pointer
   - Dangling pointer
   - Format string vulnerability
   - Unintended Read
   - Unintended Write

2. Integrity violation
   - Exfiltrate data
   - Modify code
   - Modify control-data
   - Modify non-control-data

3. Exploit Payload
   - Interpret exfiltrated data
   - Inject attacker-controlled code
   - Inject attacker-controlled address
   - Control-flow Integrity
   - Use of corrupt data

4. Exploit Dispatch
   - Indirect jump to corrupt address
   - Return to corrupt address
   - W⊕X Policy

5. Exploit Execution
   - Execute modified code
   - Execute injected code fragment
   - Execute gadget / code fragment
   - Execute data-oriented gadget

From Thomas Nyman's doctoral dissertation, *Towards Hardware-assisted Run-time Protection, 2020* (Figure 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)

Memory Protection (MPU)

Virtual memory (MMU)

W ⊕ X

Enclaves

Software, fine-grained

Software CFI

Privilege kernels

Memory-safe languages

Hardware, fine-grained

Tagged memory

Branch target indicators

HW shadow stack

Pointer Authentication

HW-assisted bounds checks

Fine-grained protection domains

Run-time scope enforcement

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. DAC 2019
Hardware assisted defenses in CotS processors

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

**Intel x84_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), improved in ARMv8.6-A (2020)

- First compatible processors 2018 (Apple A12 / iOS12)
- Userspace support in Linux 4.21, enhancements in 5.0, in-kernel support in 5.7
- 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

```
tag/PAC  sign ext./PAC  virtual address (A_P)
```

```
8 bits  reserved bit  3 – 23 bits  VA_SIZE bits
```

```
general purpose registers
```

```
64-bit modifier (M)
```

```
H_K(A_P, M)
```

```
PA key (K)
```

```
configuration register
```
ARMv8.3-A PA – Key management and instructions

Keys for PAC generation and verification

<table>
<thead>
<tr>
<th>Key</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>APIAKey_EL1</td>
<td>Key A for instruction address PACs</td>
</tr>
<tr>
<td>APIBKey_EL1</td>
<td>Key B for instruction address PACs</td>
</tr>
<tr>
<td>APIDAKey_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;</td>
<td>d</td>
</tr>
<tr>
<td>AUT&lt;</td>
<td>d</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;</td>
<td>d&gt; &lt;Xd&gt;</td>
</tr>
<tr>
<td>BRA&lt;</td>
<td>a</td>
</tr>
<tr>
<td>BLRA&lt;</td>
<td>a</td>
</tr>
<tr>
<td>RETA&lt;</td>
<td>a</td>
</tr>
<tr>
<td>ERETA&lt;</td>
<td>a</td>
</tr>
<tr>
<td>LDRA&lt;</td>
<td>a</td>
</tr>
</tbody>
</table>

APIAKey_EL1: Key A for instruction address PACs
APIBKey_EL1: Key B for instruction address PACs
APDAKey_EL1: Key A for data address PACs
APDBKey_EL1: Key B for data address PACs
APGAKey_EL1: Key for generic authentication

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: -msign-return-address

Deployed in GCC 5.0 and LLVM/Clang 7.0

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

Risk of PAC reuse!

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, Protecting the stack with PACed canaries. SysTEX@SOSP 2019
PA only approximates fully-precise pointer integrity

Adversary may reuse PACs

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

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

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

 PAC only approximates fully-precise pointer integrity

Not necessarily unique!

pacia – add PAC
autia – authenticate
PARTS

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-use or on-load authentication
- Branching with combined auth+branch instruction (blraa)
- Iterating an array uses only one authentication

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

\[
\begin{align*}
auth_0 &= H_K(ret_0, 0) \\
auth_1 &= H_K(ret_1, auth_0) \\
& \vdots \\
auth_n &= H_K(ret_n, auth_{n-1}) \\
\end{align*}
\]

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

Liljestrand et al. PACStack: an Authenticated Call Stack. Usenix Security (2021)
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 $pacib(\theta \times \theta, auth_{i-1})$

<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
- Can prevent sequential buffer overflows, and (with high probability) other memory errors

Introduced in ARMv8.5-A specification (announced in 2018), no hardware currently
- Userspace support in Linux 5.10, to be enabled via PROT_MTE flag in mmap()
- Stack tagging in LLVM 9.0, heap tagging support planned
- Experimental support in Android 11 via LLVM’s cudo memory allocator

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

Address tags stored in top 4-bits of a pointer
• uses existing top-byte ignore (TBI) feature

Allocation tags stored 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)
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[])
{
    ... do it(argv[1]); ... return 0;
}

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

void strcpy(char *str)
{
    ...
}
```
**LLVM MemTagSanitizer**

Random base tag for each stack frame
- Slots sequentially tagged to minimize tag book-keeping
- Uses Stack Safety Analysis to optimize instrumentation

Globals tagging requires loader support to assign initial tags
Heap tagging planned via the new secure Scudo allocator

Provides:
- Deterministic prevention of sequential overflows
- Probabilistic detection of use-after-free and non-sequential out-of-bounds
  - In the general case: $1 - 2^{-4} \approx 0.94$ chance of detection

E. Stepanov et al., Memory tagging in LLVM and Android, LLVM Developers’ Meeting (2020)
LLVM, MemTagSanitizer, online documentation
LLVM, Stack Safety Analysis, online documentation
LLVM Stack Safety Analysis

Introduced by Kuznetsov et al. but also used to optimize MTE instrumentation

Memory safety loosely defined as:
A memory object is safe, if all pointers derived from it are guaranteed to only access the memory object itself.

Does not preclude safe object corruption
• by unrelated unsafe memory accesses
• by within-allocation memory corruption

Algorithm finds access range of pointers
• If range is within allocated memory, then allocation is provably safe
• Local analysis
  • Determines local use ranges for allocations and function arguments
• Global analysis
  • Merges ranges from function arguments
  • Runs until fixed point reached

Pointers in memory assumed unsafe!

V. Kuznetsov et al., Code-pointer integrity, OSDI (2014)
R. Gil et al., There’s a hole in the bottom of the C: on the effectiveness of allocation protection, SecDev (2018)
MTE (and MemTagSanitizer) challenges

Tags are **corruptible**
- Random tags prevent hard-coding
- Adversary can inject tagged pointers
  - Safe memory tags always known!
  - Guessing probability $2^{-4}$ with short 4-bit tags

**Analysis is not MTE aware**
- Assume **pointers in memory are unsafe**

**Unclear security properties**
- Probabilistic, but sometimes not
- No hard guarantees with tag corruption

LLVM, [MemTagSanitizer](https://www.llvm.org/docs/MemTagSanitizer), online documentation
Our goal

Prevent tag forgery
  • _Enforce_ any pointer loaded from _unsafe memory_ is recognized as _unsafe_

Leverage MTE-awareness in safety analysis
  • Introduce MTE-specific _protected_ domain (in addition to safe / unsafe domains)
  • Prove/make a _larger set_ of allocations as safe (better optimization)

Provide clearer security guarantees
  • Allow programmer to _indicate_ variables that must remain safe
  • Provide _concrete guarantees_ for variables designated as safe by the compiler
Preventing tag corruption

Tags can be enforced to achieve **hard guarantees**!

Always **set one tag-bit on load from memory**
- Prevents injection of “safe” tags (costs one tag-bit)

Alternatively use **ARM Pointer Authentication**
- Probabilistic but does not reserve one tag-bit

→ pointers in **safe memory can remain uncorrupted**!

// Load ptr1 from ptr2
char *ptr1 = *ptr2;

// Apply ptr2 tag bits to ptr1
ptr1 = ptr1 | (tag & ptr2);
MTE-aware analysis

MTE can be used to prevent sequential overflows
- Surround with different tag (either other allocation or dedicated memory guard)

Can be treated as if memory safe:
New protected memory safe domain
- violation will lead to crash
- tagging of allocation itself can be omitted

Analysis checks that for any sequential access
- \(\text{start_range} \subseteq \text{allocated_memory}\)
- \(\text{max_step} < \text{memory_guard_size}\)
Analysis of in-memory pointers

Pointer safety requires storage location is safe
- Must prove safety of pointer within storage
  - In addition to allocation-based safety
- Must find all subsequent loads of pointer
  - Requires point-to analysis in general case
  - Non-linear data-flow through globals / heap

```c
struct s { char buff[32]; char *ptr; }; // sizeof(struct s) = 40

struct s store;
struct s DATA;
store.ptr = &DATA;

for (i = 0; i < 40; ++i)
  store.buff[i] = get_char();

char *c = &store;
func(store.ptr);
```

Where is `&DATA` used?

store is safe

Is store pointer safe?

R. Gil et al., *There's a hole in the bottom of the C: on the effectiveness of allocation protection*, SecDev (2018)
MTE-aware analysis with in-memory pointers

Conservative approximation of pointer-safety
• Check in-allocation bounds based on type
• Assumes non-local stores unsafe
• Assumes non-typed use is unsafe

Allows lightweight data-flow analysis
• Without dependence on full points-to analysis

```
struct s { char buff[32]; char *ptr; };
// sizeof(struct s) = 40

struct s store;
struct s DATA;

store.ptr = &DATA;
for (i = 0; i < 40; ++i)
    store.buff[i] = get_char();

&DATA leaked to global
&store.buff out of bounds
Is store pointer safe?

g_ptr = store.ptr; // store in global
char *c = &store;
func(store.ptr);

Need to merge function use with DATA use

type information lost
```
ARMv8.5-A Branch Target Identification

Hardware-assisted CFI similar to Intel CET Indirect Branch Tracking

Introduced in ARMv8.5-A specification (2016)

- Support in Linux 5.8
- Instrumentation support in GCC 9.0 (-mbranch-protection=standard|bt)

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)

<table>
<thead>
<tr>
<th>Branch sources</th>
<th>BTI call 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>

<table>
<thead>
<tr>
<th></th>
<th>BTI jump type branches</th>
</tr>
</thead>
<tbody>
<tr>
<td>BR ... (except x16</td>
<td>x17)</td>
</tr>
</tbody>
</table>

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

Taxonomy of Defenses

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, removed in 5.6
- Instrumentation support in GCC 5.0, removed in 9.0 (`-fcheck-pointer-bounds`)

struct obj { char buf[100]; int len };

obj* a[10];

1: for (i=0; i<M; i++) {
2: total += a[i]->len;
3: }

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

<table>
<thead>
<tr>
<th>Register</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BND00 – BND03</td>
<td>Bound Registers storing 64-bit LowerBound (LB) and 64-bit UpperBound (UB)</td>
</tr>
</tbody>
</table>

## MPX Instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BNDMK &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 &lt;Reg&gt; &lt;Addr&gt;</td>
<td>Check Addr against LB in Reg</td>
</tr>
<tr>
<td>BNDCU &lt;Reg&gt; &lt;Addr&gt;</td>
<td>Check Addr against UB in Reg in 1’s compliment form</td>
</tr>
<tr>
<td>BNDCN &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 &lt;Reg&gt; &lt;Reg</td>
<td>Addr&gt;</td>
</tr>
<tr>
<td>BNDMV &lt;Addr&gt; &lt;Reg&gt;</td>
<td>Store LB and UB from Reg to Addr</td>
</tr>
<tr>
<td>BNDLDX &lt;Reg&gt; &lt;SIB&gt;</td>
<td>Load LB and UB from bound directory</td>
</tr>
<tr>
<td>BNDSTX &lt;SIB&gt; &lt;Reg&gt;</td>
<td>Store LB and UB to bound directory</td>
</tr>
</tbody>
</table>

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: a_b = bndmk a, a+79 // Make bounds [a, a+79]
3: total = 0
4: for (i=0; i<M; i++):
   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 feasible 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 5.3 (pkey_mprotect)
PKU Registers

PKRU
Protection Key Right Register

![Figure 2-9. Protection Key Rights Register for User Pages (PKRU)](image)

**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>

Intel Control-flow Enforcement Technology

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

Deployed in Tiger Lake microarchitecture (mobile CPUs, 2020)
- Linux support proposed in 2018 [mem-mgmt, usermode SHSTK, IBT] (currently at v15)
- Runtime support in glibc 2.28, instrumentation support in GCC 8.0 (-fcf-protection)
- Enabled by default in Fedora 28 and Ubuntu 19.10 onwards

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 – 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

1. Memory vulnerability
   - Out-of-bounds pointer
   - Dangling pointer
   - Format string vulnerability
   - Unintended Read
   - Unintended Write
   - Memory-safety

2. Integrity violation
   - Exfiltrate data
   - Modify code
   - Modify control-data
   - Code Integrity

3. Exploit Payload
   - Interpret exfiltrated data
   - Inject attacker-controlled code
   - Inject attacker-controlled address
   - Instruction Set Randomization

4. Exploit Dispatch
   - Indirect jump to corrupt address
   - Return to corrupt address
   - Control-flow Integrity

5. Exploit Execution
   - Execute modified code
   - Execute injected code fragment
   - Control-flow attack
   - W\(\oplus\)X Policy

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></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>

## Intel IBT vs. ARMv8.3-A BTI

<table>
<thead>
<tr>
<th></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>
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
Design of program $p$ can be modeled as (potentially very large) **finite state machine**\(^\dagger,\dagger\)

- 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

\(^\dagger\) or a finite state transducer if output is possible

\(^\dagger\) 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 \(\sigma: Q \times \Sigma \to \Delta\)
**cpu states**

\[ Q_{cpu}^p = Q_{IFSM} \cup Q_{trans} \]

\( Q_{IFSM} \): concrete states of target machine that map to a state in the IFSM

\( Q_{trans} \): benign transitory states that occur during emulation of an edge in the IFSM; part of intended transitions

Two different perspectives of $\theta$

Program $p$

Program $p$ ("data" from user’s perspective)

Data (user’s “program”)

State 1 $\rightarrow$ State 2 $\rightarrow$ State 3 $\rightarrow$ State 4 $\rightarrow$ State 5

input $\rightarrow$ instruction $\rightarrow$ instruction $\rightarrow$ instruction $\rightarrow$ input

State 1 $\rightarrow$ input $\rightarrow$ instruction $\rightarrow$ instruction $\rightarrow$ input $\rightarrow$ State 5

Two different perspectives of $\theta$

Program $p$

Data (attacker’s “program”)

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

What is a “weird state”?

\[ Q_{cpu} = Q^{IFSM}_{cpu} \cup Q^{trans}_{cpu} \cup Q^{weird}_{cpu} \]

\( Q^{IFSM}_{cpu} \): concrete states of target machine that map to a state in the IFSM

\( Q^{trans}_{cpu} \): benign transitory states that occur during emulation of an edge in the IFSM; part of intended transitions

\( Q^{weird}_{cpu} \): set of states in \( Q_{cpu} \) not in \( Q^{IFSM}_{cpu} \) nor \( Q^{trans}_{cpu} \)

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 \( q_{\text{init}} \) and programming resulting “weird machine” in order to violate security properties of the IFSM

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

\[ \theta_{\text{weird}} = (Q_{\text{cpu}}^{\text{weird}}, q_{\text{init}}, Q_{\text{cpu}}^{\text{IFS M}} \cup Q_{\text{cpu}}^{\text{trans}}, \Sigma', \Delta', \delta', \sigma') \]

Instruction stream depends on input
• weird machine programmed through carefully crafted input to \( p \) once \( q_{\text{init}} \) 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 \( q_{\text{init}} \)

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

\( Q \) = 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 \( \sigma: Q \times \Sigma \rightarrow \Delta \)

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

Takeaways

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)

How to deal with downsides?
e.g. optimally minimize scope for PA reuse attacks?
• For return addresses: PACStack (Usenix SEC ‘21)
• For other types of pointers?

How do different hardware primitives compare?

We have open postdoc and graduate student positions. Talk to me!
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.

Slide 11 (Return-oriented programming (high-level idea) is by Luca Davi.