Hardware-assisted Run-time Protection
On balancing security and deployability

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

Acknowledgements: Thomas Nyman, Hans Liljestrand, Lachlan J Gunn, Jan-Erik Ekberg
How to thwart run-time attacks?

Run-time attacks are now routine

Software defenses incur security vs. cost tradeoffs

Hardware-assisted defenses are attractive
  Two case studies: HardScope and PARTS/PACStack

Current/future directions
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 \(\text{authenticated} \neq \text{true}\)
then: call \text{unprivileged}()
else: call \text{privileged}()

\[\text{unprivileged}() \{ \ldots \} \]
\[\text{privileged}() \{ \ldots \} \]

Szekeres et al., \textit{SoK: Eternal War in Memory}, IEEE SP (2013)
Classic code-injection

Elias Levy (as Aleph One), *Smashing the stack for fun and profit*, Phrack 7 (1996)
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

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)
Shadow Stack: High-level idea

A → B → C

“Shadow stack”

Adversary tampers with shadow stack
Non-control data attack

$ ./a.out $(perl -e 'print "A"x8 \ 
."\x08\xb0\xc4\x09" )$

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

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

## Selected Research & Vulnerabilities

<table>
<thead>
<tr>
<th>Year</th>
<th>Research Area</th>
<th>Significant Events</th>
</tr>
</thead>
<tbody>
<tr>
<td>2001</td>
<td>ret2libc Solar Designer (Phrack)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Advanced ret2libc Solar Designer (Phrack)</td>
<td></td>
</tr>
<tr>
<td>2005</td>
<td>x86-64 borrowed code chunks exploitation</td>
<td>x86-64 borrowed code chunks exploitation.</td>
</tr>
<tr>
<td></td>
<td>Krahmer</td>
<td></td>
</tr>
<tr>
<td>2007</td>
<td>ROP on x86 Shacham (CCS’07)</td>
<td></td>
</tr>
<tr>
<td>2008</td>
<td>ROP on ATMEL AVR Francillon et al (CCS’08)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ROP on SPARC Buchanan et al (CCS’08)</td>
<td></td>
</tr>
<tr>
<td>2009</td>
<td>ROP Rootkits Hund et al (USENIX Sec. ’09)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ROP on PowerPC FX Lindner (BlackHat USA)</td>
<td></td>
</tr>
<tr>
<td>2010</td>
<td>ROP w/o Returns Checkoway et al (CCS’10)</td>
<td></td>
</tr>
<tr>
<td>2011-12</td>
<td>JIT-ROP Snow et al (IEEE S&amp;P’13)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Blind ROP Bittau et al (IEEE S&amp;P’14)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Out-of-Control Göktas et al (IEEE S&amp;P’14)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Gadget size Matters Göktas et al (USENIX’14)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Stitching Gadgets Davi et al (USENIX’14)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ROP is Still Dangerous Carlini et al (USENIX’14)</td>
<td></td>
</tr>
<tr>
<td>2016</td>
<td>SROP Bosman et al (IEEE S&amp;P’14)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Control-flow Bending Carlini et al (USENIX Sec.’16)</td>
<td></td>
</tr>
</tbody>
</table>

### Dates

- **1988-99**: ROP on x86
- **2001**: ROP on x86 Shacham (CCS’07)
- **2005**: x86-64 borrowed code chunks exploitation Krahmer
- **2007**: ROP on ATMEL AVR Francillon et al (CCS’08)
- **2008**: ROP on SPARC Buchanan et al (CCS’08)
- **2009**: ROP w/o Returns Checkoway et al (CCS’10)
- **2010**: JIT-ROP Snow et al (IEEE S&P’13)
- **2011-12**: Blind ROP Bittau et al (IEEE S&P’14)
- **2016**: SROP Bosman et al (IEEE S&P’14)

### Research Areas

- **Control-flow Bending**
- **Data-Oriented Exploits**
- **DOP**
- **Non-control-data attacks**
- **String-Oriented Programming**

### Significant Events

- **CVE-1999-1416**: DoS & RCE in Solaris Answerbook 2
- **CVE-2010-2883**: RCE in Adobe Reader and Acrobat
- **CVE-2010-3765**: Nobel Peace Price website 0day
- **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-2014-0922**: Misfortune cookie in RomPager
- **CVE-2014-9222**: Misfortune cookie in RomPager
- **CVE-2016-0034**: Angler RCE in Silverlight
- **CVE-2016-0034**: Angler RCE in Silverlight

### References

- **Anders (Bugtraq. 1999)**
- **Chen et al (SSYM. ’05)**
- **Carlini et al (USENIX Sec.’16)**
- **Göktas et al (IEEE S&P’14)**
- **Hund et al (USENIX Sec. ’09)**
- **FX Lindner (BlackHat USA)**
- **Hund et al (USENIX Sec. ’09)**
- **JIT-ROP Snow et al (IEEE S&P’13)**
- **Carlini et al (USENIX’14)**
- **Göktas et al (USENIX’14)**
- **Bittau et al (IEEE S&P’14)**
- **Bosman et al (IEEE S&P’14)**
- **Bittau et al (IEEE S&P’14)**
- **Göktas et al (USENIX Sec.’16)**

### Additional Information

- **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

### Exploits

- **Non-control-data attacks**
- **String-Oriented Programming**
- **DOP**
Taxonomy of Defenses

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

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

3. Exploit Payload
   - Execute modified code
   - Inject attacker-controlled code
   - Execute gadget / code fragment
   - Execute data-oriented gadget
   - Use of corrupt data

4. Exploit Dispatch
   - Information leak
   - Code-injection attack
   - Execute gadget / code fragment
   - Use of corrupt data

5. Exploit Execution
   - Binary Attestation
   - Exfiltration
   - Control-flow attack
   - Data-oriented attack

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

Virtual machines

Harvard architecture

TEEs

Memory segments

Protection rings (Exception levels)

Hardware, coarse-grained

Virtual memory (MMU)

W \oplus X

Memory Protection (MPU)

Enclaves

Software, fine-grained

Software CFI

Privilege kernels

Memory-safe languages

Tagged memory

HW-assisted CFI

Branch target indicators

Fine-grained protection domains

Pointer Authentication

HW-assisted bounds checks

Run-time scope enforcement

Hardware, fine-grained

Memory segments

Pointers Authentication

Fine-grained protection domains

Memory-safe languages

Software, coarse-grained

Containers

Virtual machines

Harvard architecture

TEEs

Memory segments

Protection rings (Exception levels)
Protect against run-time attacks without incurring a significant performance penalty
Hardware-assisted run-time protection

Two case studies:

• **HardScope**: minimal CPU extensions for hardware-assisted scope enforcement

• **PARTS and PACStack**: Run-time safety using ARM Pointer Authentication
HardScope

Hardware-assisted Run-time Scope Enforcement

Joint work with TU Darmstadt
Motivation: Run-time Attacks

Memory corruption vulnerabilities in C / C++ can allow an attacker to access to:

- Control-data, e.g. return address stored on call stack (control-flow hijacking)
- Decision-making data, e.g. user id used for authorization decisions (data-oriented attack)
- Sensitive data, e.g. cryptographic keys (information leakage)

Access to unintended data

Compile-time variable visibility rules make references to unintended variables less likely

⇒ Enforcing variable scope also at run-time would reduce potential of memory attacks
Challenges

Lexical scope only known at compile-time
• In C / C++, variable visibility information not available at run-time

Granularity of enforcement
• Effective compartmentalization requires fine granularity for subjects (code) and objects (data)

Context-sensitive access
• Same code may operate under different set of access rules depending on caller

Pervasiveness
• Efficiently mediate all memory accesses
Design
HardScope: High-level Idea

Instrument program during compilation to:
• Split code up into distinct *execution contexts* (common environment for function or block)
• Associate each execution context with *storage regions* (data memory accessed)

Modify underlying hardware with HardScope instructions to:
• Accumulate rules for storage regions
• Track changes in execution context
• Track dynamic data flows
• Enforce accesses to storage regions
New Instructions

During run-time, **7 new instructions** configure HardScope-hardware with access rules

- Scope Block instructions mark points of *domain transitions*, e.g. function call / return
- Storage Region (SR) instructions *whitelist memory regions* for current domain, e.g. stack frame
- Delegation instructions gives callee/caller access to SRs e.g. arguments, return values

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sbent</td>
<td>Scope Block ENter</td>
<td>Mark transition into new domain</td>
</tr>
<tr>
<td>sbxit</td>
<td>Scope Block eXIT</td>
<td>Mark transition out of domain</td>
</tr>
<tr>
<td>sradd</td>
<td>Storage Region ADD</td>
<td>Set base and limit for new storage region</td>
</tr>
<tr>
<td>srdda</td>
<td>Storage Region ADD (reverse operands)</td>
<td></td>
</tr>
<tr>
<td>srdel</td>
<td>Storage Region DELete</td>
<td>Revoke access to storage region</td>
</tr>
<tr>
<td>srdlg</td>
<td>Storage Region DeLeGate</td>
<td>Delegate existing SR to callee / caller</td>
</tr>
<tr>
<td>srdsup</td>
<td>Storage Region Delegate SUBregion</td>
<td>Delegate subregion to an existing SR</td>
</tr>
</tbody>
</table>
Storage Region Stack

Stack-oriented storage for accumulated access rules
- Stores the bounds of each used storage region (e.g. stack variable, heap object, global variable)
- Frames created upon domain entry (sbent → push)
- Frames discarded upon domain exit (sbxit → pop)

Actively enforced rules in topmost frame
- Memory accesses matched only against active rules
- Subsequent frame store inactive rules for inactive domains
- Function-level enforcement mirrors structure of call stack

Maintained in protected memory
- Rules only modifiable by HardScope instructions
Storage Region Stack Hardware Design

Active and delegated storage region rules stored in register banks

- Allows enforcement without slowing down loads / stores as active rules cached for fast access
- Cache management amortized over several instructions on execution context change
Function-granularity compartmentalization

Functions separated into distinct execution contexts

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

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

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

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

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

void strcpy(char *str) {
    ...
}
```
Return-state compartmentalization

Function prologue and epilogue separated into own execution context

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

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

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

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

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

void strcpy(char *str) {
    ...
}
```
Implementation
Proof-of-Concept Implementation

Proof-of-Concept ISA extension for RISC-V processor
• Software simulation implemented for Spike ISA simulator
• Integrated with PULPino SoC on ZedBoard FPGA

GCC Plug-in for automatic HardScope instrumentation
• Function granularity enforcement for local, global, and static variables, function arguments and return values

Only 3.2% performance overhead in CoreMark embedded benchmarks
• 11% binary size increase due to instrumentation
• 32 entries in register banks (CoreMark used only up to 23)
• 574 byte memory overhead*

*) Maximum SRS depth: 71 entries over 11 frames encoded using 64 bits per SR entry + 4 bits per frame for the number of entries

HardScope benefits

+ Adjustable granularity of enforcement
e.g. module-, function-, code-block-compartmentalization
+ Can provide resilience against multiple classes of attacks
  e.g. ROP, DOP

HardScope limitations

- Currently only supports single-threaded C programs
  Additions to hardware design needed to support concurrency

- Currently manual annotations needed to instrument dynamic data structures
  Coarse-granularity enforcement can be provided via wrappers

- Assumes programs minimize variable scope and module interdependence
  Programs without logical structure benefit less and consume more SRS resources

- Secure memory size (needed for storing info about storage regions) fixed at synthesis time
  Optimal size may be difficult to determine

- Hardware changes seen as too invasive
HardScope: Thwarting DOP with Hardware-assisted Run-time Scope Enforcement
DAC 2019 (phew!)

Toolchain, emulator and code samples:
https://github.com/runtime-scope-enforcement/

https://arxiv.org/abs/1705.10295
Protect against run-time attacks without incurring a significant performance penalty or major deployment hurdles
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)
PARTS and PACStack

Run-time safety using ARM Pointer Authentication
Joint work with Huawei Helsinki Platform Security Team
Deploying new hardware extensions is difficult

But CPU vendors are rolling out several different hardware security mechanisms

How can we use these mechanisms for run-time protection?
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

![Diagram of PAC generation process]

General purpose registers, configuration register, virtual address registration, reserved bit, 8 bits, 3 – 23 bits, 64-bit modifier (M), virtual address (A_P), sign ext./PAC, tag/PAC, PA key (K), VA_SIZE bits.
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 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 }
```

```
...ab08
...ab10
...ab18
...ab20
...ab28
...ab30
...ab38
...ab40
...ab48
...ab50
```

**pacia** – add PAC

**autia** – authenticate
PA-assisted Run-time Safety (PARTS)

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

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

\[ auth_i = H_k(ret_i, auth_{i-1}) \]

\[ auth_0 = H_k(ret_0, 0) \]

\[ auth_1 = H_k(ret_1, auth_0) \]

\[ auth_n = H_k(ret_n, auth_{n-1}) \]

\[ auth_i, \ i \in [0, \ n-1] \] 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*.
  - A pseudo-random mask is generated using $\text{pacib}(\emptyset \times \emptyset, auth_{i-1})$.

<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

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

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

Mismatch between tags reported either:
• synchronously (precise check during testing, leads to an MTE fault), or
• asynchronously (imprecise checks, kept track of in a status register, checked by kernel)
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 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
MTE (and MemTagSanitizer) challenges

Tags are **corruptible**

- Random tags prevent hard-coding
- Adversary can inject tagged pointers
  - Wild-card tags can be used to circumvent MTE checking!
  - Guessing probability $2^{-4}$ with short 4-bit tags

Used with generic LLVM optimization (to avoid unnecessary tagging) but analysis not MTE-aware: assumes **pointers in memory are unsafe**

Cannot guarantee **memory-safety in the standard adversary model**: adversary who
- has read/write access to entire process memory
- can exploit some memory vulnerability read from or write to any address
- can repeat attacks

*(How) Can we use MTE to provide post-deployment run-time memory safety?*
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|bti)
Taxonomy of Defenses

Adapted from Szekeres et al., *SoK: Eternal War in Memory*, IEEE SP (2013)
Hardware assisted defenses in CotS processors

<table>
<thead>
<tr>
<th>ARMv8-A mechanisms</th>
<th>Intel x84_64 mechanisms</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pointer Authentication (PA)</td>
<td>Memory Protection eXtension (MPX)</td>
</tr>
<tr>
<td>Memory Tagging Extension (MTE)</td>
<td>Memory Protection Keys (PKU)</td>
</tr>
<tr>
<td>Branch Target Identification (BTI)</td>
<td>Control-flow Enforcement Technology (CET)</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><strong>Low to Moderate</strong></td>
<td>N/A</td>
</tr>
<tr>
<td>Run-time Overhead</td>
<td>? (likely low)</td>
<td>Low</td>
</tr>
</tbody>
</table>


Custom solution vs. generic building block

PA can be used for:

- Return address protection
- Indirect branch protection
- Data pointer protection
- Strengthening Stack Canaries
- Authenticated data structures

originally Qualcomm design, PACStack** (2)
PARTS (1)
PARTS (1)
PCan (2)
work-in-progress (4)

2) Liljestrand et al., PACStack: an Authenticated Call Stack. Usenix Security (2021)
3) Liljestrand et al., Protecting the stack with PACed Canadies, ACM SysTex (2019)
4) Ghorshi et al., work-in-progress (2022)
Takeaways

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

How to deal with downsides?
- e.g., optimally minimize scope for PA reuse attacks?
- For return addresses: PACStack (Usenix SEC ‘21)
- e.g., using MTE for post-deployment run-time safety? (wip)

Are there other novel uses? (USENIX SEC ’19, SysTex ‘19, wip)

How do different hardware primitives compare?

How can we formalize run-time attacks and defenses?

https://ssg.aalto.fi/research/projects/harp/
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.

Slide 6 (Return-oriented programming (high-level idea) is by Luca Davi.)