Aalto University

UNIVERSITY OF WATERLOO

Hardware-assisted runtime protection: Pointer Authentication and beyond

Lachlan J. Gunn
Aalto University

https://gunn.ee/

@lachlan_gunn

Acknowledgements: Thomas Nyman (Aalto University), Hans Liljestrand, N. Asokan (University of Waterloo)

Android Security and Privacy Research Summit
Mountain View, CA, USA, 28 February 2020

Motivation: Run-time Attacks

Software written in memory unsafe languages such as C/C++

- Suffer from various memory-related errors

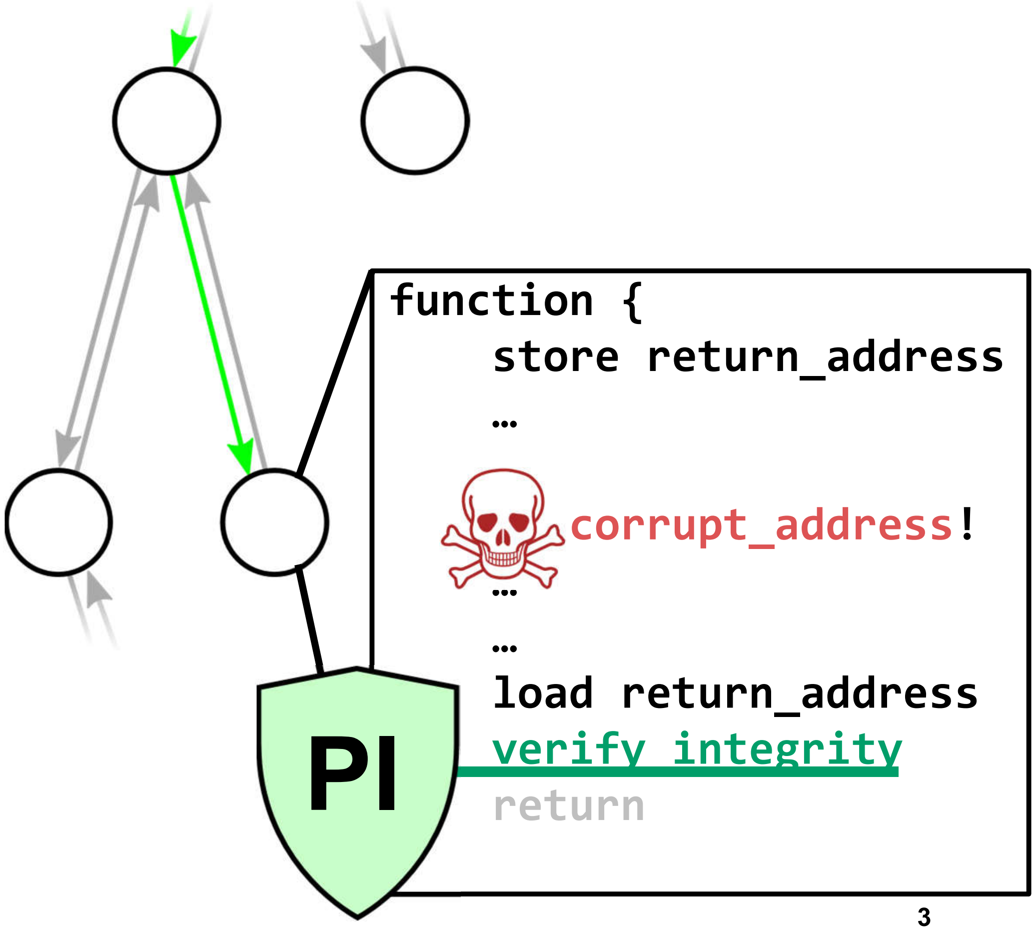
Memory errors may allow run-time attacks to compromise program behaviour

- *Control-flow hijacking / code injection*
- *Return-Oriented Programming* (ROP)
- *Non-control-data attacks*
- *Data-Oriented Programming* (DOP)

Pointer Integrity: memory safety for pointers

Ensure pointers in memory remain unchanged

- **Code pointer integrity implies CFI**
  - E.g. return-address pointers, function pointers
  - Control-flow attacks manipulate code pointers
- **Data pointer integrity**
  - Reduces data-only attack surface

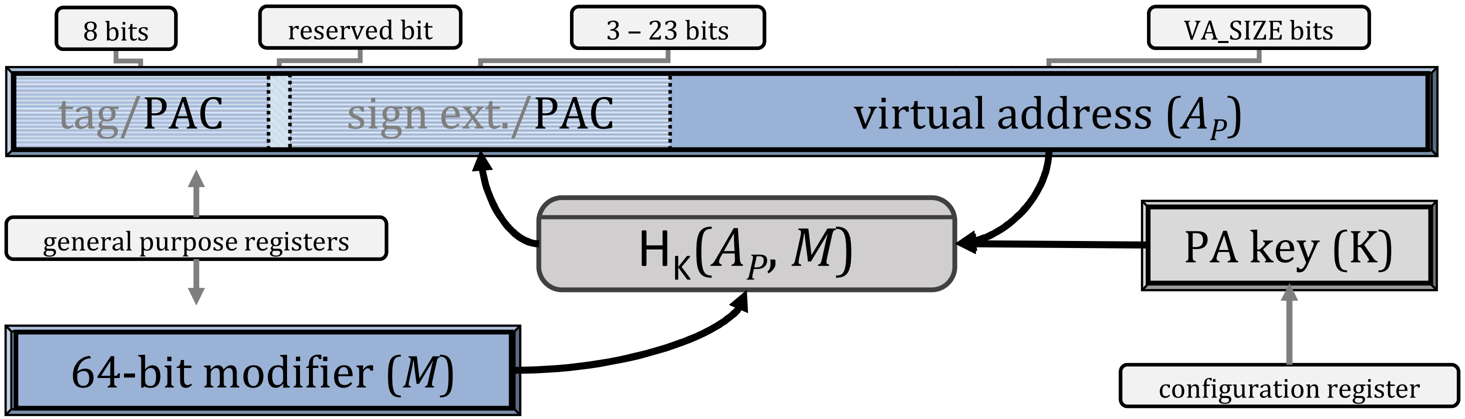


Kuznetsov et al. "Code-Pointer Integrity", USENIX OSDI 2014

ARMv8.3-A Pointer Authentication

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



ARM. Arm® Architecture Reference Manual Armv8, for Armv8-A architecture profile. Version E.a. July 2019

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

Last year: 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-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
…
ldr Xptr, <memory>
mov Xmod, #type_id
autda Xptr, Xmod
<something> [Xptr]
```

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

Authenticated on use

`pacda`	– add PAC with data A-key
`autda`	– authenticate
`pacia`	– add PAC with instr A-key
`lbraa`	– authenticate and branch

[LNWPEA19] PAC it up: Towards Pointer Integrity using ARM Pointer Authentication. USENIX Security (2019)

PACStack: Authenticated Call Stack

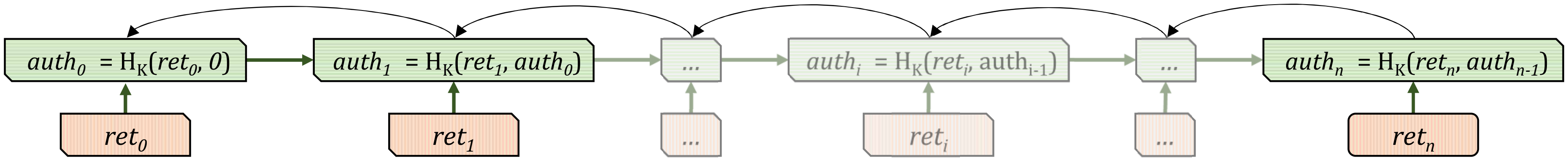
Hans Liljestrand[2,3], Thomas Nyman[1], Lachlan J Gunn[1], Jan-Erik Ekberg[1,2], N. Asokan[3]

[1] Aalto University, [2] Huawei Technologies, [3] University of Waterloo

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
- Head of the chain stored safely in a register



$auth_i$, $i \in [0, n-1]$ 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*$2^{b/2}$ return addresses

- For b = 16 bits: n = 321 addresses

Solution: Prevent *recognizing* collisions by masking each *auth*

- pseudo-random mask generated using `pacib(0x0,` $auth_{i-1}$)

Attack	w/o Masking	w/ Masking
Reuse previous auth collision	1	2^{-b}
Guess auth to existing call-site	2^{-b}	2^{-b}
Guess auth to arbitrary address	2^{-2b}	2^{-2b}

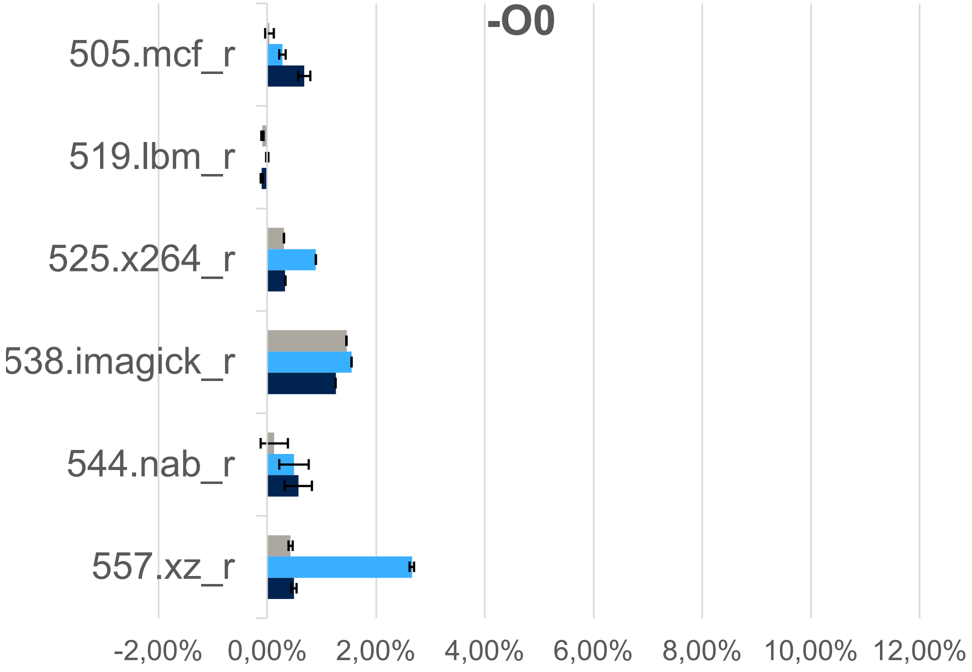
Maximum probability of success for different attacks

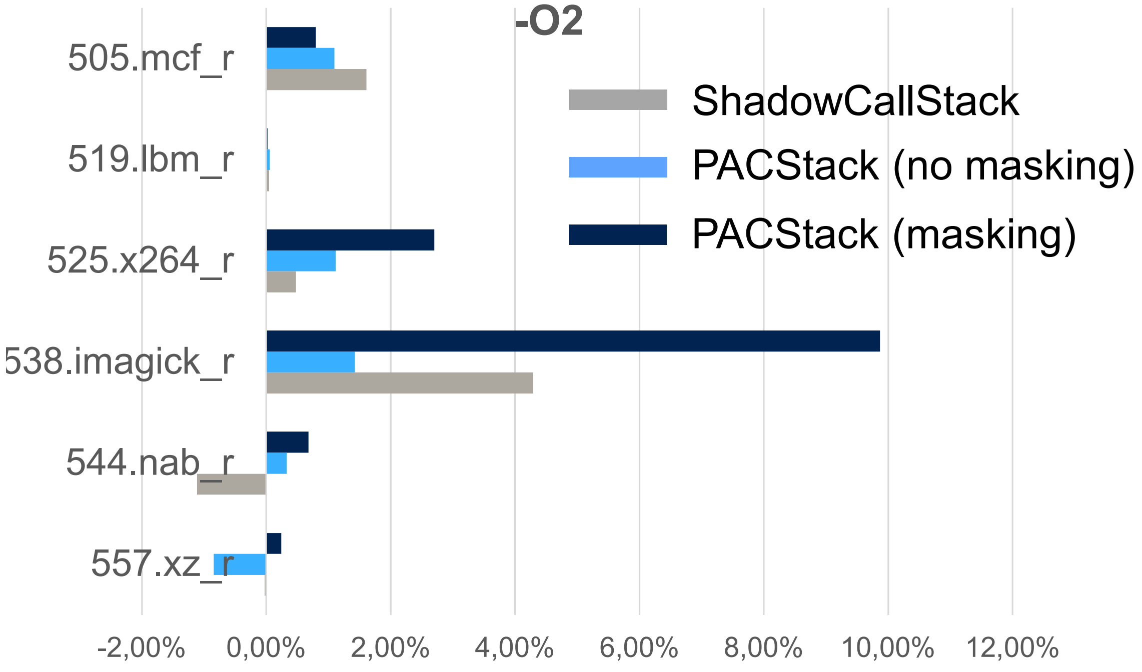
Evaluation: SPEC CPU 2017 C-language benchmarks

Performance overhead based on 4-cycles per PA instruction

- without masking < 0.5% (geo.mean)
- with masking < 1% (geo.mean)

- without masking < 1% (geo.mean)
- with masking < 2.5% (geo.mean)





Next steps & future work

Attack surface:

- How prevalent are pointer reuse vulnerabilities in real-world programs?

High-level behavioral guarantees:

- Can we ensure that a compromised program behaves correctly (or crashes)?

Does PA synergize with upcoming features [1]?

- Memory tagging
- Branch-target indication

[1] ARM A-Profile Architecture Developments 2018: Armv8.5-A
[2] Intel Control-flow Enforcement Technology Specification, May 2019

Takeaways

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

How to utilize available primitives effectively?

e.g. minimize scope for PA reuse attacks?

- For return addresses: PACStack (arXiv:1905.10242)
- For other types of pointers: PARTS (arXiv:1811.09189)

How to use other emerging hardware primitives, e.g.

- memory tagging
- branch-target indication

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