PAC it up: Towards Pointer Integrity using ARM Pointer Authentication

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

†) Aalto University, ‡) Huawei Technologies
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
  - Prevents all known Data-Oriented Programming (DOP) attacks

```c
function {
    store return_address
    ...
    corrupt_address!
    ...
    load return_address
    verify_integrity
    return
}
```
Pointer Authentication in ARMv8.3-A

- General purpose hardware primitive approximating pointer integrity
- 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
Example: PA-based return address signing

Deployed as -msign-return-address in GCC and LLVM/Clang

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

STACK

- pacia – add PAC
- autia – authenticate

Qualcomm “Pointer Authentication on ARMv8.3”, whitepaper 2017
PA prevents arbitrary pointer injection

- Modifiers do not need to be confidential
  - Visible or inferable from the code section / binary

- Keys are protected by hardware and set by kernel
  - Attacker cannot generate PACs

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

pacia – add PAC
autia – authenticate
PA only approximates fully-precise pointer integrity

Adversary may reuse PACs

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

Not necessarily unique!

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

pacia – add PAC
autia – authenticate
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
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**
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?

```
... /* 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

Authenticated immediately on load
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)
Take aways

ARM PA can efficiently protect pointers and is (going to be) widely available

How to optimally minimize scope for reuse attacks?
• For return addresses: PACStack (arXiv:1905.10242)
• For other types of pointers?

Use other emerging hardware primitives for run-time protection?
• For instance: Memory tagging, Branch target indication