PAC it up

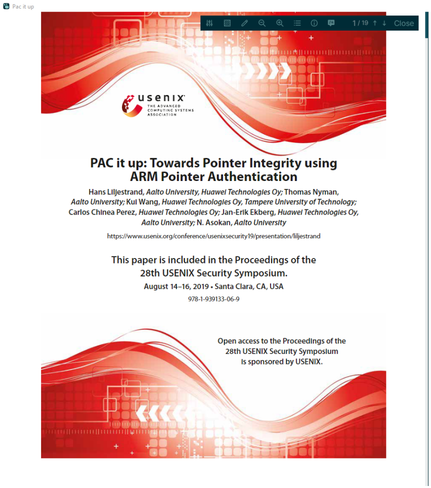
Towards Pointer Integrity using ARM Pointer Authentication

Raghuveer Shivakumar (Raghu)

University of Colorado, Boulder

2022-10-19

PAC it up



usenix THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

PAC it up: Towards Pointer Integrity using ARM Pointer Authentication

Hans Liljestrand, *Aalto University, Huawei Technologies Oy;* Thomas Nyman, *Aalto University;* Kui Wang, *Huawei Technologies Oy, Tampere University of Technology;* Carlos Chinea Perez, *Huawei Technologies Oy;* Jan-Erik Ekberg, *Huawei Technologies Oy, Aalto University;* N. Asokan, *Aalto University*

https://www.usenix.org/conference/usenixsecurity19/presentation/liljestrand

This paper is included in the Proceedings of the 28th USENIX Security Symposium.

August 14–16, 2019 • Santa Clara, CA, USA

978-1-939133-06-9

Open access to the Proceedings of the 28th USENIX Security Symposium is sponsored by USENIX.

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

Description: Pointer authentication based defense mechanism for preserving pointer integrity using LLVM.

Background and Motivation

- `Code Pointer:` *Function Pointers, Saved Return Addresses*
- `Data Pointer:` *Regular Pointers (`int* ptr = &a; // 'a' is a var`)*

Background and Motivation Contd.

- `Common Pointer Attacks:` *ROP, DOP Attack.*

Existing Solutions

- Software based solutions
  - *EMET (Enhanced mitigation experience toolkit) (Performance Overhead)*
- Hardware based solutions
  - *Hardbound: Architectural suport for spatial safety of C (DEVIETTI ET AL), HAFIX - Hardware assisted flow integrity extension (DAVI ET AL), CHERI Capability model (Woodruff et all), etc*
  - Unlikely to be deployed in the near future, require invasive changes to the underlying processor architecture.

Background and Motivation Contd.

- One such hardware attempt: New instruction for pointer authentication
    - PA uses cryptographic message authentication codes (MACs) referred to as `pointer authentication codes (PACs)`

Issues with Pointer Authentication

- Vulnerable to pointer reuse attacks where an authenticated pointer is substituted with another.

Let's understand this in detail.

Control Flow Attack on ARM

- Corrupting a code pointer such as return addr or function ptr can cause a control flow transfer anywhere in the executable memory
    - Applies to all architectures
- Like other RISC processor designs, ARM processors have a dedicated Link Register (LR) that stores the return address. LR is typically set during a function call by the Branch with Link (bl) instruction.
    - Cannot be directly modified, but when the flow is inside a nested function, this will have to placed in the stack so that the nested function can use the LR. During this time an attacker can leverage any memory error to redirect the control flow.

Data Oriented Attacks

- Non-control data attacks.
    - Instead of modifying code pointers they corrupt other program variables that influence program's decision making.

ARM Pointer Authentication

- ARMv8.3-A introduces a new instruction for pointer authentication.
  - `PACXX`
  - `AUTXX`
  - XX can be `GA` or `IA` or `IB` or `DA` or `DB` where `G` stands for generic pointer, `I` stands for instruction pointer, and `D` stands for data pointer.
    - `A` stands for key A and `B` stands for key B.
- Available only in 64-bit ARM mode

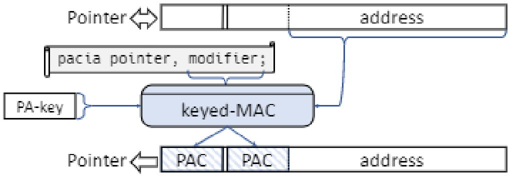


Figure 1: ARM Pointer Authentication

ARM Pointer Authentication Contd.

- Key is usually stored in mm_context structure in the task_struct, which is a part of the process descriptor in the kernel.
  - We saw earlier that in `PACXX` and `AUTXX` instructions, `XX` can be `GA` or `IA` or `IB` or `DA` or `DB`, where `A` and `B` are keys.
  - So, PA provides five different keys for PAC generation

Alignment and Size

- 64 bit ARM processors only use part of the 64 bit address space for virtual address.
  - On a default AArch64 Linux kernel configuration with 39 bit addresses and without address tagging, the PAC size is 24 bits.

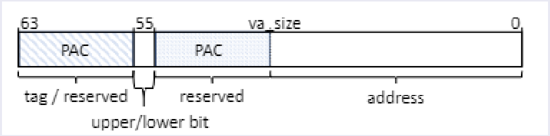
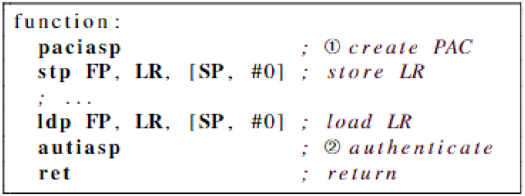


Figure 2: PAC Alignment

Example Use Case - Qualcomm's return address signing



```
function:
  paciasp                 ; ① create PAC
  stp FP, LR, [SP, #0]    ; store LR
  ; ...
  ldp FP, LR, [SP, #0]    ; load LR
  autiasp                 ; ② authenticate
  ret                     ; return
```

Figure 3: Qualcomm's return address signing

SP is the stack pointer and LR is the link register.

Back to Issues with Pointer Authentication - Pointer Reuse Attack

- Now that we have seen how pointer authentication works, let's understand how pointer authentication can be bypassed using pointer reuse attack.
  - [**Hint:** Replay Non-unique SP; PA doesn't garantee uniqueness of modifiers]
  - func1:

```
func1:
         paciasp   ;Generate PAC for SP
         stp FP, LR, [SP, #0]    ; LR is stored in the stack
         ...
         ldp FP, LR, [SP, #0]    ; LR is loaded from the stack
         autiasp  ; Authenticate SP
         ret
func2:
         paciasp   ;Generate PAC for SP
         ...
```

  - Attacker can reuse the authenticated return addresses value from one function when a different vulnerable function executes with a matching SP value.

Malicious use of PACs

1. Unauthenticated pointer value before PAC creation: get an arbitrary authenticated pointer for any context with the same modifier and PA key.
2. Control the PA modifier value: get an authenticated pointer for a context with the same PA key, but with an attacker-chosen modifier.
3. Both: get arbitrary authenticated pointers for a context with attacker-chosen modifier, and the same PA key.

Solution: The program must not contain PA creation instructions with attacker controlled input (Basically PA + CFI)

Threat Model

- A powerful attacker with arbitrary memory read and write capabilities restricted only by DEP
- The attacker can thus read any program memory and write to non-code segment
- The attacker has no control of higher privilege levels, i.e., an attacker targeting a user space process cannot access the kernel or higher privilege levels.
- The attacker cannot infer the PA keys, as they are in registers not directly readable from user space
- The attacker's ability to read arbitrary memory precludes the use of randomization-based defenses that cannot withstand information disclosure

Goals

- Enhanced scheme for pointer signing that enforces pointer integrity for all code and data pointers.
- Runtime type safety to constrain pointer substitution attack by ensuring the pointer is of the correct type.
  - Implemented using PARTS - Pointer Authentication Run-Time Safety, a compiler instrumentation framework that leverages PA to realize our proposed defenses.

Requirements

- **[R1]** Pointer Integrity: Detect/prevent the use of corrupted code and data pointers.
- **[R2]** PA-attack resistance: Resist attempts to control PAC generation, and pointer reuse attacks.
- **[R3]** Compatibility: Allow protection of existing programs without interfering with their normal operation.
- **[R4]** Performance: Minimize run-time and memory over-head and gracefully scale in relation to the number of protected pointers and dereferences/calls.

Implementation

- **PARTS** - Pointer Authentication Run-Time Safety
  - A compiler enhancement that emits PA instructions to sign pointers in memory as required. Specifically it protects:
    - return addr, local/global/static ptrs, pointers in c-structs.

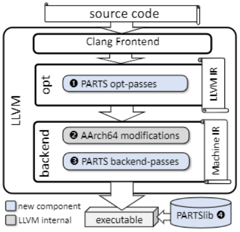


Figure 3: PARTS architecture.

Figure 4: PARTS

Implementation Contd.

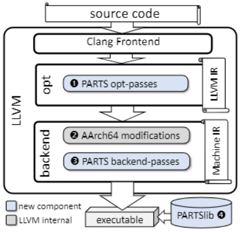


Figure 3: PARTS architecture.

Figure 5: PARTS

[R1] Pointer Integrity: *Satisfied*

- ④ PARTSLib - Programs may contain pointers which are initialized by the compiler, e.g., defined global variables. However, PAC values for authenticated pointers cannot be calculated before program execution, as PA keys are not known until runtime.
  - PARTSlib, processes the relocated variables and invokes the generated initializer function to ensure that any defined pointers are furnished with a PAC.

Implementation Contd.

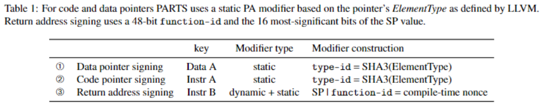


Table 1: For code and data pointers PARTS uses a static PA modifier based on the pointer's *ElementType* as defined by LLVM. Return address signing uses a 48-bit `function-id` and the 16 most-significant bits of the SP value.

		key	Modifier type	Modifier construction
①	Data pointer signing	Data A	static	`type-id` = SHA3(ElementType)
②	Code pointer signing	Instr A	static	`type-id` = SHA3(ElementType)
③	Return address signing	Instr B	dynamic + static	SP | `function-id` = compile-time nonce

Figure 6: Reuse Attack Preventation

- Type ID is a truncated 64-bit SHA-3 hash of the pointer's LLVM ElementType
  - ElementType represents the IR level data type and distinguishes between basic data types, but does not retain typedef or other information from the frontend (i.e., clang).
  - For return addr signing, the modifier is combined with a unique prpg-based function-id, which is garanteed to be unique for each function within the module

[R2] PA-attack resistance: *Satisfied*

Implementation Contd.

- Basically handling pointer conversion in the case of data pointer integrity
- **Simple Heuristic:** If the operation is load, use source object type; otherwise use dest object type

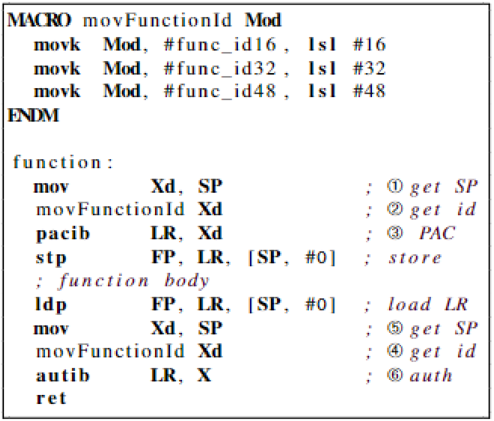
[R3] Compatibility: *Satisfied*

Instrumentation

- In-line instrumention
    - Does not require storage of separate run-time metadata.
- No explicit error handling is added by PARTS; instead, an authentication failure will set specific high-order bits in the pointer causing segmentation fault on the subsequent dereference or call using the pointer that failed authentication.

Instrumentation Contd.

- Return address signing



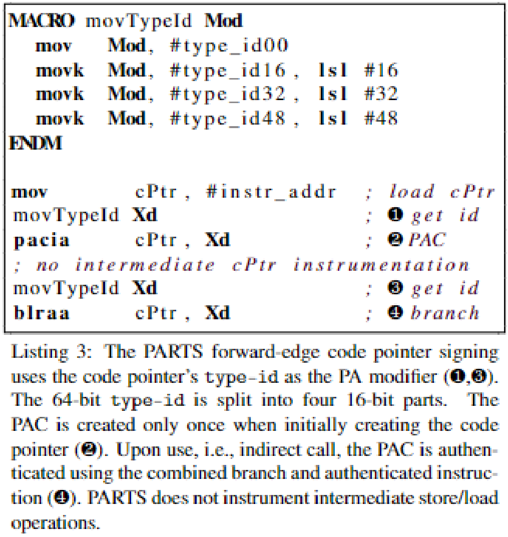
```
MACRO movFunctionId Mod
   movk  Mod, #func_id16 , lsl #16
   movk  Mod, #func_id32 , lsl #32
   movk  Mod, #func_id48 , lsl #48
ENDM

function:
   mov           Xd, SP                 ; ① get SP
   movFunctionId Xd                     ; ② get id
   pacib         LR, Xd                 ; ③ PAC
   stp           FP, LR, [SP, #0]       ; store
   ; function body
   ldp           FP, LR, [SP, #0]       ; load LR
   mov           Xd, SP                 ; ⑤ get SP
   movFunctionId Xd                     ; ④ get id
   autib         LR, X                  ; ⑥ auth
   ret
```

Listing 2: The PARTS return address signing binds the PAC to the SP (①,⑤) and unique function id (②,④). The PA modifier is in register `Xd` during PAC creation (③) and authentication (⑥). The 48-bit `func-id` is split into three 16-bit parts, each moved individually to `Xd` by left-shifting.

Instrumentation Contd.

- Code pointer signing



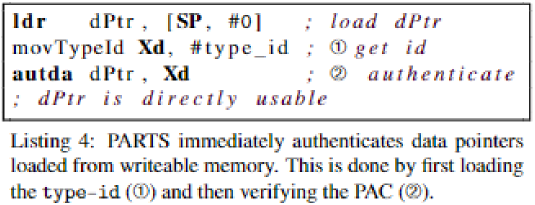
```
MACRO movTypeId Mod
  mov    Mod, #type_id00
  movk   Mod, #type_id16 , lsl #16
  movk   Mod, #type_id32 , lsl #32
  movk   Mod, #type_id48 , lsl #48
ENDM

mov         cPtr , #instr_addr   ; load cPtr
movTypeId Xd                     ; ❶ get id
pacia       cPtr , Xd            ; ❷ PAC
; no intermediate cPtr instrumentation
movTypeId Xd                     ; ❸ get id
blraa       cPtr , Xd            ; ❹ branch
```

Listing 3: The PARTS forward-edge code pointer signing uses the code pointer's `type-id` as the PA modifier (❶,❸). The 64-bit `type-id` is split into four 16-bit parts. The PAC is created only once when initially creating the code pointer (❷). Upon use, i.e., indirect call, the PAC is authenticated using the combined branch and authenticated instruction (❹). PARTS does not instrument intermediate store/load operations.

Instrumentation Contd.

- Data pointer signing



```
ldr     dPtr, [SP, #0]    ; load dPtr
movTypeId Xd, #type_id   ; ① get id
autda  dPtr, Xd          ; ② authenticate
; dPtr is directly usable
```

Listing 4: PARTS immediately authenticates data pointers loaded from writeable memory. This is done by first loading the type-id (①) and then verifying the PAC (②).

Figure 9: Data Pointer Signing

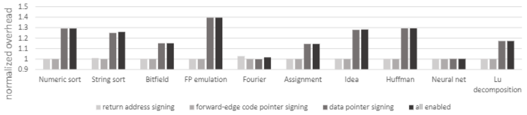
Evaluation

- Return Address Signing
  - Susceptible to pointer reuse between distinct invocations of the **same function** from call sites with same SP value
- Data Pointer Signing
  - ARMv8-A PA does not provide facilities to directly address:
    - spatial safety of pointer accesses to data objects
    - Spoof with Object of type `X`

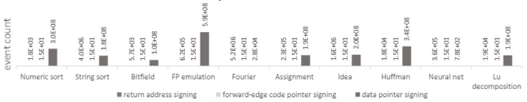
Evaluation Contd.

- PAC Entropy
  - To succeed with probability p, a PAC guessing attack requires $\frac{\log(1-p)}{\log(1-2^{-b})}$ guess on the assumption that a PAC comparison failure leads to program termination.
  - Simulator setup where b = 16, achieving a 50%-likelihood for a correct guess requires 45425 attempts; b == pac size

Evaluation Contd.



(a) Results of instrumented nbench-byte tests features, normalized to a non-instrumented baseline.



(b) Run-time count of executed locations instrumentable by PARTS. Because the program's memory profile affects performance the benchmark results clearly correlate with observed memory use (e.g., FP emulation has a large data pointer integrity overhead because it uses many data pointers)

Figure 5: nbench benchmark results

Figure 10: Performance Evaluation

- nBench Benchmark
  - Kirin 620 HiKey, ARMv8-A Cortex A53 Octa-core CPU (1.2GHz) / 2GB LPDDR3 SDRAM (800MHz) / 8GB eMMC, running the Linux kernel v4.18.0 and BusyBox v1.29.2
  - Return addr: 0.5% ~ 12 to 16 cycles
  - Forward code ptr signing: 6 to 8 cycles
  - Data pointer signing: 39.5%

Questions