Hardware-assisted Run-time Protection

Thomas Nyman, Hans Liljestrand, Lachlan Gunn, N. Asokan
How to thwart run-time attacks?

Run-time attacks are now routine

Software defenses incur security vs. cost tradeoffs

Hardware-assisted defenses are attractive
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](image)

8 bits reserved bit 3 – 23 bits VA_SIZE bits

tag/PAC sign ext./PAC virtual address (A_p)

64-bit modifier (M)

H_K(A_p, M)

PA key (K)

general purpose registers

configuration register

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
}
```

Qualcomm “Pointer Authentication on ARMv8.3” (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!

Not necessarily unique!

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

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

Can we do more than PARTS?

PARTS narrows the scope of reuse attacks
• but cannot completely prevent them

How to optimally minimize scope for reuse attacks?
• Having unique modifiers often impossible
• Static approaches limited to large equivalence classes
Authenticated Call Stack: high-level idea

**Chained MAC** of authentication tokens cryptographically bound to return addresses

- Provides modifier \((\text{auth})\) bound to all previous return addresses on the call stack
- Statistically unique to control-flow path
  - prevents reuse
  - allows precise verification of returns

\[
\text{auth}_0 = H_K(\text{ret}_0, 0) \\
\text{auth}_1 = H_K(\text{ret}_1, \text{auth}_0) \\
\vdots \\
\text{auth}_i = H_K(\text{ret}_i, \text{auth}_{i-1}) \\
\vdots \\
\text{auth}_n = H_K(\text{ret}_n, \text{auth}_{n-1})
\]

\(\text{auth}_i, \ i \in [0, \ n - 1]\) bound to corresponding return addresses, \(\text{ret}_i, \ i \in [0, \ n]\), and \(\text{auth}_n\)

PACStack instrumentation

- Generate 16-bit auth with pacib instruction and embed in PAC-bits
- Topmost auth\(_n\) is always
  - Stored securely in dedicated CPU register (LR)
  - Passed to callees via the x28 register

```plaintext
prologue:
str X28, [SP] ; stack ← aret\(_{n-1}\)
pacib LR, X28 ; LR ← aret\(_n\)
function_body:
...
epilogue:
ldr X28, [SP] ; X28 ← aret\(_{n-1}\) ’from stack
autib LR, X28 ; LR ← (ret\(_n\) or ret\(_n^*\))
ret
```

In dedicated register
Mitigation of hash-collisions: PAC masking

- **Challenge**: PAC collisions occur on average after $1.253 \times 2^{b/2}$ return addresses
  - For $b=16$ this is only 321 addresses

- **Solution**: Prevent recognizing collisions by masking each auth
  - pseudo-random mask generated using $\text{paci}(0x0, auth_{i-1})$

<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
PARTS & PACStack performance

Functional evaluation
• On ARM Fast Models 11.4 FVP

Performance evaluation
• 96board Kirin 620 HiKey board
• PA-analog with overhead of 4-cycles
  • Based on QARMA overhead estimate
  • Uses XOR operations to “sign” pointer

PARTS on nbench-byte-2.2.3
• Return address protection <0.5%
• Code pointer integrity <0.5%
• Data pointer integrity ~20%

PACStack on SPEC CPU 2017
• Without masking ~0.4%
• With masking ~0.9%
  • Cf. LLVM ShadowCallStack ~0.5%

How does return-address protection using PA compare with other hardware-assisted approaches?
# Intel CET vs. ARMv8.3-A PA

<table>
<thead>
<tr>
<th></th>
<th>Intel CET</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>✓ (coarse-grained)</td>
<td>✓ (PARTS)</td>
</tr>
<tr>
<td>Data pointer protection</td>
<td>✗</td>
<td>✓ (PARTS)</td>
</tr>
<tr>
<td>Enforcement model</td>
<td>Deterministic</td>
<td>Probabilistic</td>
</tr>
<tr>
<td>Immune to pointer reuse</td>
<td>✓</td>
<td>✓ (PACStack)</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>


Other uses of PA

PA is a general-purpose primitive

PCan - using PA to generate stack-canaries
• Return address protection already functionally a canary:
  • Return address corruption due to overflow is detected
  • No reference canary needed
  • Canaries can differ from function to function
• Reuse still possible, but PCan can be anchored to other schemes
  • E.g., with PACStack statistically unique canaries for each function call

Other hardware primitives

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

• Can these strengthen each other?

• What becomes feasible by combining these primitives?

• How do different types of hardware-assistance compare?
  ➢ Is there an optimal set of hardware primitives for new platforms?
Optimal use of hardware primitives

PA is a powerful security primitive, but others are on the horizon

How to combine them for best trade-off in security, cost, and performance?

https://pacstack.github.io

github.com/pointer-authentication