PARTS: Towards pointer integrity using ARM pointer authentication

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

PARTS, Pointer Authentication for Run-time Type Safety

- Approximates *pointer integrity* using ARMv8.3-A pointer authentication
- Protects *return addresses, code pointers, and data pointers*
- Prevents *pointer reuse attacks* by enforcing run-time type safety

**ARMv8.3-A Pointer Authentication**

- Embeds and verifies embedded Pointer Verification Codes (PACs):
  - Embedded in unused bits of a pointer
  - Keyed, tweakable MAC based on address and given modifier
- User-space support in Linux 5.0 with kernel-managed keys
- Used via new PA-specific `pac` and `aut` instructions

**PACing it up with PARTS**

- Uses *run-time type safety* to generate modifiers:
  - Secure modifiers created based on read-only code section
  - Pointer type known both at creation and use
  - Modifier not affected by memory copy
- LLVM 6.0 based implementation + Linux RFC patches for PA

**Return address protection**

`(pacib(ret, funcID║SP))`

- Unique function specific identifiers generated at compile-time
- Different function activations distinguished using stack pointer

**Code pointer protection**

`(pacia(ptr, type))`

- Modifier from pointer type (`LLVMElementType`)
- Pointers signed on memory write and verified on memory load
  - Allows efficient register use

**Data pointer protection**

`(pacda(ret, type))`

- Modifier from pointer type (`LLVMElementType`)
- Pointers signed on memory write and verified on memory load
  - Allows efficient register use

**Implementation challenges**

- LLVM backend looses high-level semantics, including pointer type
  - Solution: define new intrinsics for `pac` / `aut` operations
- Register spilling looses retains no semantics
  - Solution: must analyze spills to protect data-pointer spills
- Interoperability with non-instrumented libraries
  - Verify and sign pointers to / from non-instrumented code

**PARTS performance evaluation**

- Based on estimated overhead of 4-cycles per PA instruction

<table>
<thead>
<tr>
<th>Return address</th>
<th>Code pointer</th>
<th>Data pointer</th>
<th>All</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.5% overhead</td>
<td>19.5% overhead</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Challenge:** to prevent modifier must be sufficiently unique to value and:

- **Secure:** Cannot be stored in memory, hence cannot be random e.g., if securely stored, then why not just store pointer itself?
- **Available:** Known at both creation and use of pointer e.g., these could be spatially and temporally disjoint events
- **Location independent:** Storage location cannot be tied to mod e.g., must allow `memcpy` and embedding in other data structures

---

[Image of a diagram showing structure of an authenticated pointer]

- Modifier can be used to define context for pointer
- Vulnerable to *pointer reuse* when modifiers coincide!
  e.g., stack pointer as modifier in GCC / LLVM –msign-return-address

**Return address protection**

```
main {
  func1();
  func2();
}
```

**Code pointer protection**

```
func1:
  autia LR, mod
  ret
reuse possible if mod1 = mod2

func2:
  autia LR, mod
  ret
```

**Data pointer protection**

```
main {
  func1();
  func2();
}
```

---

**Intel Research Institute for Collaborative Autonomous and Resilient Systems**

https://github.com/pointer-authentication