Morello and the challenges of a capability-based ABI

Kevin Brodsky
24 August 2020
A world of capabilities
CHERI and Morello

- **CHERI**: a hardware architecture, part of a research project led by Cambridge University
  - Based on the concept of **hardware capabilities**, added on top of a conventional ISA
    - Aim: **spatial safety**, but also temporal safety
    - Has been implemented on top of MIPS and RISC-V
  - **Morello**: a research program, led by Arm, funded by the UK government
    - Also a **prototype** architecture, extending Armv8 with CHERI concepts
    - Also a quad-core prototype board implementing the Morello arch
      → Will allow for realistic **performance measurements**

🌟 Introduction to the Morello program: Richard Grisenthwaite on Digital Security by Design (slides)
Quick anatomy of a capability in Morello

128 bits of “regular” data
- 64-bit value (address)
- Bounds (compressed)
- Permissions (Load, Store, Execute, …)
- Object type

→ Can all be set directly

1 “magic” bit
- Unforgeable validity tag
  - In registers: next to 128-bit data
  - In memory: stored separately

→ Cannot be set by software*
→ Cleared by any invalid operation

Note: “the tag of capability C is set” == “C is valid”
The rules of the game

**Validation**  Dereferencing a capability pointer only succeeds if:
- Tag is set
- Access within bounds
- Permissions allow it
- Not sealed

**Provenance**  Only specific instructions may construct valid capabilities
- Arbitrary writes to memory zero the corresponding tag(s)

**Monotonicity**  Bounds and permissions can only be restricted, not extended

🌟 Excellent overview: *An Introduction to CHERI*
🌟 A proposal for a provenance-aware model in C2x: N2362
Capabilities in practice
## C language mappings

<table>
<thead>
<tr>
<th>Hybrid-capability</th>
<th>Pure-capability</th>
</tr>
</thead>
<tbody>
<tr>
<td>- Capabilities as a C extension</td>
<td>- Capabilities embedded in C</td>
</tr>
<tr>
<td>- <code>__capability</code> pointer annotation</td>
<td>- All pointers are capabilities</td>
</tr>
<tr>
<td>- For instance: <code>char* __capability</code></td>
<td>- Automatic instantiation of capabilities</td>
</tr>
<tr>
<td>- Explicit instantiation of capabilities</td>
<td>- Modified memory allocators</td>
</tr>
<tr>
<td>- Often derived from global capabilities</td>
<td>- Stack management</td>
</tr>
<tr>
<td>- Extended relocations</td>
<td>- Extended relocations</td>
</tr>
</tbody>
</table>

+ Compiler and runtime support for manipulating and preserving capabilities
Hybrid/Pure-cap: typical usage

- **Hybrid-cap:** mostly useful for *specialized code* (bits of kernel, libc, etc.)
  - Capabilities must be propagated explicitly (`__capability` everywhere 😞)
  - Library functions do not take capabilities!
  - But: less disruptive at runtime (contained capability checks)

- **Pure-cap:** *everything else* (all “normal” software)
  - Natural model for a capability architecture
  - All the benefits of capability checks (bounds, permissions, monotonicity, ...)
  - No or very few code changes required
  - But: (some) runtime cost, bugs to fix!

- In low-level software, hybrid-cap allows for controlled usage before switching to pure-cap
Lightweight compartmentalization

- Isolation of software components through capabilities
- Same address space, but access constrained by capabilities
  - By default: a compartment can only access its own memory
  - Can be extended by passing tightly bounded capabilities
- More lightweight and scalable than processes, cheaper IPC
  - Typical use-case: isolation between browser tabs
- Many possible implementations and usage models...
- Strong use-case for the pure-cap model

🌟 Much more about this: Hardware support for compartmentalisation
🌟 Also: CHERI: A Hybrid Capability-System Architecture for Scalable Software Compartmentalization
The pure-capability ABI
Holy pointers

- `sizeof(void*) == 16` and unforgeable tag attached
- Completely new ABI (think 32-bit → 64-bit transition)
- Transparent for most software
- Main exception: low-level software
  - C runtime
  - Memory allocators
  - JITs
  - In general: code making assumption about pointers
Pain points

Pointers must be handled with care

- Big enough + aligned enough storage
- Pointers cannot be stored in arbitrary integers: only (u)intptr_t is valid
- Bitwise operations can be tricky
- In general: address ≠ pointer
  - Still 64-bit addresses!
  - intptr_t has a 64-bit value range, but is 128-bit large
- Certain patterns around memory allocation can be problematic (especially realloc())

⭐ Everything about pure-cap: CHERI C/C++ Programming Guide
Here be dragons:
supporting the pure-cap ABI in userspace
The goal

Support userspace programs built in the pure-cap ABI

- Use the right types: all pointers at the kernel-user interface are capabilities
- Honor capability metadata: access memory “as if” dereferencing the capability
- Create capabilities for userspace with appropriate bounds and permissions
- Retain the base 64-bit ABI (32-bit not required)

→ Has been achieved on CheriBSD!

[CheriBSD: adaptation of FreeBSD for CHERI]

⭐ More on pure-cap in CheriBSD: CheriABI paper
Pointers in the kernel-user ABI

- Where they appear:
  - Syscalls (arguments, struct members)
  - A few other places (initial stack layout, signal handlers)

- How they are used:
  - Most common: user specifying where data should be read/written
    - Data accessed via user mappings using copy_to_user() and friends (e.g. read())
    - Data accessed via kernel mappings using get_user_pages() (e.g. readv())
  - Less common: kernel providing userspace with a pointer to some object
    - mmap() and friends
    - argv
  - Rare: arbitrary user data, stored by the kernel without processing
    - For instance epoll_ctl() and epoll_wait()
User pointers as capabilities

- Good: all user pointers are annotated with __user in Linux
- Hope for turning void __user * into a capability... with caveats
- Need a mechanism for enforcing capability bounds / permissions
A long issue

- `long` is everywhere in Linux
- Strong assumption that `long` is big enough to hold any scalar type...
- ...therefore can be used to represent a lot of things, in particular:
  - Addresses (fine) and/or pointers (not fine!)
  - Catch-all type (especially in syscalls)
- Really bad for multiplexed syscalls: `ptrace()`, `fcntl()`, `ioctl()`

Sounds overwhelming?
Avoiding dragons: a userspace shim
Userspace shim: principles

A stepping stone: userspace shim library

- Lives between libc and the kernel
- Checks input capabilities ("as-if" dereferenced by the kernel)
- Two-way ABI conversion
- Unmodified kernel-user ABI

```
getcwd(c_ptr, size)
```

```
check(c_ptr, size)
ptr = address(c_ptr)
```

```
sys_getcwd(ptr, size)
```

```
...  
```

```
copy_to_user(ptr, cwd, len)
```

Userspace shim: limitations

- Does not enforce any security boundary (raw 64-bit syscalls still available)
- Requires explicit checking of capabilities (extra cost)
  - Checking C-strings is inherently racy
- Needs to know whenever pointers are passed — not easy with multiplexed syscalls
  - ioctl almost impossible to handle reliably:
    $ git grep '\.unlocked_ioctl'| wc -l
    593
    + out-of-tree drivers!
- Complications whenever the kernel stores user pointers
Getting bolder: a kernel shim
An arch-specific in-kernel shim

- New kernel-user ABI: pure-cap
- Non-invasive shim in arch code
- Security boundary enforced
- Pure-cap as a secondary ABI (?)

→ Attempted on arm64
→ Experimental implementation in CheriBSD
Kernel shim: limitations

- Same as the userspace shim (security aside)
  - Somewhat easier to implement if making changes to generic code
- Only existing mechanism for a secondary ABI: COMPAT
  - Typical situation: `void __user *argp = compat_ptr(arg);`
  - Major obstacle: COMPAT pointers must fit in `void __user *`
- Would have to define a new mechanism...
Fighting dragons: propagate the capabilities
- All user pointers in the kernel become capabilities
- Capabilities propagated down to the point of use (typically uaccess routines)
- New integer type to represent user pointers: intuserptr_t?
  - Note: intmax_t not the right type: “ [...] integer type capable of representing any value of any signed integer type”
- long must be replaced whenever it may represent a user pointer
- Clearly an invasive approach
User ABIs

- Pure-cap must become the primary ABI
- 64-bit ABI becomes COMPAT
- Cleaner approach, especially for uaccess
  - User memory always accessed via capabilities, regardless of the task’s ABI
Option 1: hybrid-cap kernel

- Turn void __user * into a capability
  - __user on the wrong side of * 😞
  - void __capability * deprecated, but works in most cases
  - #define __user __capability worth a try!

- intcap_t available for storing capabilities

- Potential issue with uapi headers being built in different ABIs (hybrid-cap kernel, pure-cap userspace)
  - All pointers should already be annotated with __user, but may not be enough

→ Current approach used by CheriBSD
Option 2: pure-cap kernel

- The “proper” way
- Requires eradicating `long` everywhere it may represent any pointer
- Comes with all the benefits of pure-cap code...
- ...but also the usual difficulties of porting low-level code to CHERI
- Potential performance impact

→ Experimental implementation in CheriBSD
Common issues

- Explicit checking still needed for indirect accesses (`get_user_pages()`)  
- `memcpy()` should not always copy tags  
- `mmap()` interface unfriendly with capabilities  
  - `mprotect()` does not return a pointer
    → which capability permissions should `mmap()` return?

⭐ More on `mmap()`’s flaws: *Is it time to replace mmap? (slides)*
Looking forward
Overview

- 2 main approaches for supporting pure-cap in userspace:
  - **Shim** Wrapping around 64-bit syscalls: non-invasive, but fragile
  - **Propagate** Making all user pointers capabilities: “proper” approach but invasive

- Supporting pure-cap on Linux: painful in one way or another
- But: has been done on FreeBSD!
Morello project status

- First release in October, watch https://www.morello-project.org/
  - “Core” kernel support for Morello, unmodified ABI
  - Userspace shim library
  - Minimal Android with limited pure-cap support
- Morello support in CheriBSD to be published soon
- Starting now: new kernel-user ABI definition, investigation into the “propagate” approach
  - Many aspects of the ABI yet to be properly defined

☆ More info on the roadmap: Morello Software and Toolchain Work in Arm (slides)
Wider efforts in Linux that would be beneficial:

- Proper multi-ABI support
- Start the long war
- `void __user * → void * __user`
Resources

- CHERI landing page
- An Introduction to CHERI (technical report)
- CHERI C/C++ Programming Guide (technical report)
- CheriABI: Enforcing Valid Pointer Provenance... (paper)
- Compartmentalization:
  - Hardware support for compartmentalisation (technical report)
  - CHERI: A Hybrid Capability-System Architecture for Scalable Software Compartmentalization (paper)
- Morello program:
  - Richard Grisenthwaite’s talk at the Digital Security by Design workshop (slides)
  - Mark Nicholson’s talk “Morello Software and Toolchain Work in Arm” (slides)
- Brooks Davis’s talk “Is it time to replace mmap?” (slides)
Contacts

- CHERI community discussion mailing list (appropriate for generic Morello discussions as well)
- Or just drop me an email: `<first>.<last>@arm.com` 😊
Thank You
Danke
Merci
谢谢
ありがとう
Gracias
Kiitos
감사합니다
धन्यवाद
شكرًا
ধনযবাদ
תודה