The CHERI CPU
RISC in the age of risk

David Chisnall
University of Cambridge

Approved for public release; distribution is unlimited. This research is sponsored by the Defense Advanced Research Projects Agency (DARPA) and the Air Force Research Laboratory (AFRL), under contracts FA8750-10-C-0237 and FA8750-11-C-0249. The views, opinions, and/or findings contained in this article/presentation are those of the author(s)/presenter(s) and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government.
Memory: You’re doing it wrong!

~82% of exploited vulnerabilities in 2012
— Software Vulnerability Exploitation Trends, Microsoft
Low-level languages rule

C: 8,323,328,367 lines of code
C++: 2,966,693,989 lines of code
Java: 2,861,498,030 lines of code
Scala: 13,780,744 lines of code

Source: openhub.net
Security is important again

Multi-user systems

Disconnected single-user systems

Single-user, multi-attacker systems
RISC is for compilers

• Nothing that can be done fast in software should be done in hardware.

• Everything that can only be done well in hardware should be in hardware.
The CHERI model

• Memory protection as a first-class part of the ISA

• A single abstraction for bounds checking and sandboxing

• Mechanism in the hardware, policy in software
Pointers should be capabilities

- Smalltalk (Java, etc) pointers confer the rights to access an object.
- C pointers can (in practice) be constructed from arbitrary integers.
- Capabilities are unforgeable tokens of authority.
CHERI capabilities

ISA Operations

<table>
<thead>
<tr>
<th>Field</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Permissions</td>
<td>Bitwise and</td>
</tr>
<tr>
<td>Base</td>
<td>Increment (and decrease length)</td>
</tr>
<tr>
<td>Length</td>
<td>Decrease</td>
</tr>
<tr>
<td>Offset</td>
<td>Arbitrary manipulation</td>
</tr>
</tbody>
</table>

32 capability registers

- base [64]
- length [64]
- virtual address [64] (exposed as offset)
Tags to Protect Capabilities in Memory

Capabilities on the stack and in data structures
Address Calculation

Instruction Fetch
- $PC
- $PCC

Legacy Data Access
- $Rn
- $C0

Capability Data Access
- $Cn

Offset
Virtual Address
Physical Address

TLB
Physical Memory
Tag Table in Commodity DRAM

- TAGS <0.5%
- Tag Lookup (with cache)
- DATA
- L2 Cache
- 128 tag bits per 4KB page
- Tags on physical memory
- Cache line is tag(s) + data
Paged Memory

- OS managed
- Enables swapping
- Centralised
- Allows revocation

Address validation
Capabilities

- Compiler managed
- Precise
- Can be delegated
- Many domains

Pointer safety
Paged Memory + Capabilities

- OS managed
- Enables swapping
- Centralised
- Allows revocation

Compiler managed
- Precise
- Can be delegated
- Many domains

Address validation

Pointer safety
Memory safety in hardware

- All memory accesses must be via a valid capability
- Instructions only allow restricting range / permissions of capabilities
- Now all we need is software…
Building on open source

• A full open source stack:
  • LLVM/Clang-based compiler.
  • Modified FreeBSD.
  • Extended BERI processor.
  • Real software from the FreeBSD ports collection.
Process start

- $c0$ and $pcc$ cover the entire address space.
- Unmodified code is completely oblivious.
- CHERI-aware code can derive restricted capabilities from either.
- Compartments can be created by discarding/subsetting $c0$ in some threads.
Don’t break the world!

- Code that doesn’t contain memory safety errors should work!
- Even if it does slightly (or very) evil things with pointers!
- Ideally only code with memory safety errors should break.
C is weird

• Long standard describes allowed behaviour.

• Lots of things are *implementation defined* or *undefined*.

• **All** nontrivial programs depend on implementation-defined behaviour.

• Breaking this makes programmers cranky!

• We discovered most of these things when we broke them and tried to compile real programs (e.g. tcpdump)
7.20.1.4 Integer types capable of holding object pointers

The following type designates a signed integer type with the property that any valid pointer to `void` can be converted to this type, then converted back to pointer to `void`, and the result will compare equal to the original pointer:

```
intptr_t
```

The following type designates an unsigned integer type with the property that any valid pointer to `void` can be converted to this type, then converted back to pointer to `void`, and the result will compare equal to the original pointer:

```
uintptr_t
```

These types are optional.

7.20.1.5 Greatest-width integer types

The following type designates a signed integer type capable of representing any value of any signed integer type:

```
intmax_t
```

The following type designates an unsigned integer type capable of representing any value of any unsigned integer type:

```
uintmax_t
```

These types are required.

Implementation defined!
Simple Problem: `memcpy()`

```c
struct foo {
    void *a;
    int   b;
};
struct foo new = old;
```

```c
memcpy(&new, &old, sizeof(struct foo);
```

The `memcpy()` function doesn’t know if it’s copying pointers or data!
Pointers as capabilities

C code used `__capability` qualifier to tag pointers to be represented as capabilities

```c
// 64-bit integer (address)
void *foo;

// 256-bit capability
__capability int *bar;

// Increment offset by sizeof(int)
bar++;

// Load 4 bytes at offset+sizeof(int)
bar[1];
```
Enabling pointer abuse

// The low bit of a sensibly aligned pointer is always 0, so we can hide a flag in it
__capability int *set_flag(__capability int *b) {
    return (__capability int*)((__intcap_t)b | 1);
}
Enabling pointer abuse

# Integer constant 1
daddiu $1, $zero, 1

# Derive a canonical null capability
cfromptr $c1, $c0, $zero

# Set intcap_t (tag not valid) to 1
csetoffset $c1, $c1, $1

# Get the integer values of both operands
cgetoffset $1, $c1
cgetoffset $2, $c3

# Perform the arithmetic
or $1, $1, $2

# Set the offset in the original capability
csetoffset $c3, $c3, $1
Legacy interoperability
(is hard)

```c
void *foo;
__capability char *bar;
// What does this do?
bar = (__capability char *)foo;
// Or this?
foo = (void *)bar;
```
First cut at Casts

# Cast from pointer ($1) to capability ($c1)
CIncBase $c1, $c0, $1

# Cast from capability ($c1) to pointer ($1)
CGetBase $c1, $1

• What happens if the pointer is null?

• What happens if the capability is outside the $c0 range or $c0 has a non-zero offset?
§6.3.2.3.3: An integer constant expression with the value 0, or such an expression cast to type `void *`, is called a null pointer constant. If a null pointer constant is converted to a pointer type, the resulting pointer, called a null pointer, is guaranteed to compare unequal to a pointer to any object or function.

```c
void *null = (void*)0;
int a = 0;
void *might_be_null = (void*)a;
```

No C programmer has ever paid attention to this!
Casts in CHERI

# Cast from pointer ($1) to capability ($c1)
CFromPtr $c1, $c0, $1

# Cast from capability ($c1) to pointer ($1)
CToPtr $1, $c0, $c1

- CFromPtr gives a null capability if the integer is 0
- CToPtr gives a 0 integer if the capability is null or outside of $c0
Where do bounds come from?

- An object in C is a single allocation.
- OpenSSL’s Heartbleed vulnerability was caused (partly) by splitting allocations.
- Some programmer policy is essential!
- Sizes of globals, stack allocations, `malloc()` calls are not enough (but they’re a good start!)
## Safer returning

Capabilities are for code, not just for Christmas data

<table>
<thead>
<tr>
<th></th>
<th>MIPS</th>
<th>CHERI</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Call</strong></td>
<td>jalr $t9, $ra</td>
<td>cjalr $c12, $c17</td>
</tr>
<tr>
<td><strong>Return</strong></td>
<td>jr $ra</td>
<td>cjr $c17</td>
</tr>
<tr>
<td><strong>Spill return address to stack</strong></td>
<td>sd $ra, 32($sp) csc $c17, $sp, 32($c11)</td>
<td>Trap</td>
</tr>
<tr>
<td><strong>Behaviour if spilled value is corrupted</strong></td>
<td><strong>Jump somewhere (attacker’s choice)</strong></td>
<td><strong>Trap</strong></td>
</tr>
</tbody>
</table>
Comparing pointers

• C says it’s undefined behaviour to compare pointers to different objects
• C programmers do it all the time
• CHERI adds pointer compare instructions
Some evil things people do to pointers

- Store them in integer variables (works if they’re [u]intcap_t)
- Do arbitrary arithmetic on them
- Let them go out of range in the middle of a calculation
- Compare pointers to different objects

_all of these need to work!_
A tale of 3 ABIs

- Incremental deployment is vital for testing
- Rewriting (or even recompiling) all code at once isn’t feasible

More compatible

n64
Pure MIPS

More safe

n64 + CHERI
Some pointers are capabilities

Pure-capability
All pointers are capabilities
The pure-capability ABI

• Code where all pointers are capabilities.
• May have a null $c0$.
• Can only see a subset of all memory.
• Incompatible with syscall ABI.
CHERI-friendly libraries

- Always use typedefs for pointer types.
- Don’t put struct definitions for opaque types in headers.
- Separate file-handling layers (that make syscalls) from buffer-handling layers.
- Write good code!
Memory Safety Overhead

Olden (pointer-chasing) benchmarks

tcpdump (real code!)

Dhrystone (CPU-intensive) benchmark
Building sandboxes

No access to any memory without valid capabilities
Process-based sandboxes?

- More expensive to create (new kernel process, virtual memory map)
- More expensive to share (one TLB entry per page)
- No fine-grained sharing (page granularity)
- Better separation of kernel rights (so far!)
Library Sandboxing

• Private heap per library instance (multiple isolated copies of the same library allowed)

• Shared code between all instances

• Calls to the library delegate access to shared buffers

• Maintaining ABIs can be a bit tricky!
Lessons Learned

Only testing with real code can tell you how useful your ISA really is. Convincing a compiler it’s useful is a lot harder than convincing yourself.

Software stack on GitHub now, hardware due new open-source release Real Soon Now™
Further Reading


http://chericpu.org
http://www.cl.cam.ac.uk/research/security/ctsrd/

Thanks to DARPA, AFRL, Google!