CHERI: Capability Hardware Enhanced
RISC Instructions

Dr. Nathaniel Wesley Filardo

April 10, 2019
Outline

- Pointers Are Not What We Meant
- Capability Machines
- Capabilities Capture What We Meant
- The CHERI Project’s Status
Pointers Are A Forgetful Encoding

Let’s think about a very simple program:

```c
int a_sub_3(const int *a) {
    return a[3];
}

int main(void) {
    int arr[4] = { 1, 1, 15, 410 };  
    printf("%d\n", a_sub_3(arr));
    return 0;
}
```
Pointers Are A Forgetful Encoding

a_sub_3 compiles to MIPS (different from x86!):

► Prologue (stack frame, frame pointer):

1. `daddiu $sp, $sp, -16`
2. `sd $fp, 8($sp)`
3. `move $fp, $sp`

► The actual load:

4. `lw $2, 12($4)`

► Epilogue (undo prologue, return):

5. `move $sp, $fp`
6. `ld $fp, 8($sp)`
7. `jr $ra`
8. `daddiu $sp, $sp, 16`
Pointers Are A Forgetful Encoding

a_sub_3 compiles to MIPS (different from x86!):

- **Prologue (stack frame, frame pointer):**
  1. `daddiu $sp, $sp, -16`
  2. `sd $fp, 8($sp)`
  3. `move $fp, $sp`

- **The actual load:**
  4. `lw $2, 12($4)`

- **Epilogue (undo prologue, return):**
  5. `move $sp, $fp`
  6. `ld $fp, 8($sp)`
  7. `jr $ra`
  8. `daddiu $sp, $sp, 16`
Pointers Are A Forgetful Encoding

\[ a_{\text{sub}}\_3 \text{ compiles to MIPS (different from x86!):} \]

\begin{itemize}
  \item Prologue (stack frame, frame pointer):
    \begin{enumerate}
      \item \texttt{daddiu $sp, $sp, -16}
      \item \texttt{sd $fp, 8($sp)}
      \item \texttt{move $fp, $sp}
    \end{enumerate}

  \item The actual load:
    \begin{enumerate}
      \item \texttt{lw $2, 12($4)}
    \end{enumerate}

  \item Epilogue (undo prologue, return):
    \begin{enumerate}
      \item \texttt{move $sp, $fp}
      \item \texttt{ld $fp, 8($sp)}
      \item \texttt{jr $ra}
      \item \texttt{daddiu $sp, $sp, 16}
    \end{enumerate}
\end{itemize}
Pointers Are A Forgetful Encoding

`a_sub_3` compiles to MIPS (different from x86!):

- **Prologue (stack frame, frame pointer):**
  
  1. `daddiu $sp, $sp, -16`
  2. `sd $fp, 8($sp)`
  3. `move $fp, $sp`

- **The actual load:**
  
  4. `lw $2, 12($4)`

- **Epilogue (undo prologue, return):**
  
  5. `move $sp, $fp`
  6. `ld $fp, 8($sp)`
  7. `jr $ra`
  8. `daddiu $sp, $sp, 16`
Pointers Are A Forgetful Encoding

a_sub_3 compiles to MIPS (different from x86!):

- **Prologue (stack frame, frame pointer):**
  1. `daddiu $sp, $sp, -16`  
  2. `sd $fp, 8($sp)`
  3. `move $fp, $sp`  

- **The actual load:** first argument
  4. `lw $2, 12($4)`

- **Epilogue (undo prologue, return):**
  5. `move $sp, $fp`
  6. `ld $fp, 8($sp)`
  7. `jr $ra`
  8. `daddiu $sp, $sp, 16`
Pointers Are A Forgetful Encoding

a_sub_3 compiles to MIPS (different from x86!):

- **Prologue (stack frame, frame pointer):**
  
  1. `daddiu $sp, $sp, -16`
  2. `sd $fp, 8($sp)`
  3. `move $fp, $sp`

- **The actual load:** first argument
  
  4. `lw $2, 12($4)`

  `return value` \( \text{sizeof(int)} == 4 \)

- **Epilogue (undo prologue, return):**
  
  5. `move $sp, $fp`
  6. `ld $fp, 8($sp)`
  7. `jr $ra`
  8. `daddiu $sp, $sp, 16`
Pointers Are A Forgetful Encoding

`a_sub_3` compiles to MIPS (different from x86!):

- **Prologue (stack frame, frame pointer):**

  1. `daddiu $sp, $sp, -16`
  2. `sd $fp, 8($sp)`
  3. `move $fp, $sp`

- **The actual load:** first argument

  4. `lw $2, 12($4)`

- **Epilogue (undo prologue, return):**

  5. `move $sp, $fp`
  6. `ld $fp, 8($sp)`
  7. `jr $ra`
  8. `daddiu $sp, $sp, 16`
Pointers Are A Forgetful Encoding

`a_sub_3` compiles to MIPS (different from x86!):

- **Prologue (stack frame, frame pointer):**
  
  1. `daddiu $sp, $sp, -16`
  2. `sd $fp, 8($sp)`
  3. `move $fp, $sp`

- **The actual load:** first argument
  
  4. `lw $2, 12($4)`

  return value \[\text{sizeof(int)} == 4\]

- **Epilogue (undo prologue, return):**
  
  5. `move $sp, $fp`
  6. `ld $fp, 8($sp)`
  7. `jr $ra`
  8. `daddiu $sp, $sp, 16` return address

"branch delay slot"
main compiles to MIPS:

- **Prologue:**
  1. `daddiu $sp, $sp, -48`
  2. `sd $ra, 32($sp)`
  3. `sd $fp, 24($sp)`
  4. `sd $gp, 16($sp)`
  5. `move $fp, $sp`
main compiles to MIPS:

**Prologue:**
1. `daddiu $sp, $sp, -48`
2. `sd $ra, 32($sp)`
3. `sd $fp, 24($sp)`
4. `sd $gp, 16($sp)`
5. `move $fp, $sp`

**PC-relative addressing for a_sub_3:**
6. `lui $1, 1`
7. `daddu $1, $1, $25`
8. `daddiu $gp, $1, 31680`
9. `ld $25, -32536($gp)`

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>32($sp)</th>
<th>24($sp)</th>
<th>16($sp)</th>
<th>8($sp)</th>
<th>0($sp)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$ra</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$fp</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$gp</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>a[3], a[2]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>a[1], a[0]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
main compiles to MIPS:

- **Prologue:**
  
  1. `daddiu $sp, $sp, -48`
  2. `sd $ra, 32($sp)`
  3. `sd $fp, 24($sp)`
  4. `sd $gp, 16($sp)`
  5. `move $fp, $sp`

- **PC-relative addressing for `a_sub_3`:**

- **Construct array on the stack:**

  10. `lui $4, 15`
  11. `ori $4, $4, 410`
  12. `sd $4, 8($sp)`
  13. `...`
Pointers Are A Forgetful Encoding

main compiles to MIPS:

- **Prologue:**
  1. daddiu $sp, $sp, -48
  2. `sd $ra, 32($sp)`
  3. `sd $fp, 24($sp)`
  4. `sd $gp, 16($sp)`
  5. `move $fp, $sp`

- **PC-relative addressing for a_sub_3:**

- **Construct array on the stack:**
  10. `lui $4, 15`
  11. `ori $4, $4, 410`
  12. `sd $4, 8($sp)`
  13. ...

- **And call with the array pointer:**
  16. `jalr $25`
  17. `move $4, $sp`
Alternate program #1:

```c
int a_sub_3(const int *a) {
    return a[3];
}

int main(void) {
    int arr[2] = { 15, 410 };
    printf("%d\n", a_sub_3(arr));
    return 0;
}
```
Pointers Are A Forgetful Encoding

Alternate program #2:

```c
int a_sub_3(const int *a) {
    return a[-1];
}

int main(void) {
    int arr[2] = { 15, 410 };
    printf("%d\n", a_sub_3(arr));
    return 0;
}
```
Pointers Are A Forgetful Encoding

Alternate (exciting!) program #3:

```c
int a_sub_3(const int *a) {
    ((int *)a)[-1] = 0x1337;
    return a[3];
}

int main(void) {
    int arr[2] = { 0 };
    printf("%d\n", a_sub_3(arr));
    return 0;
}
```
Pointers Are A Forgetful Encoding
What Did Pointers Forget?

When you think about these programs... 
  ▶ (I’d bet that) You think about *objects*.
  ▶ Objects have a type, an address in memory, a length, ...
  ▶ Procedures make certain accesses to some objects.

Are there languages that more closely follow your intuition?
Pointers Are A Forgetful Encoding
What Did Pointers Forget?

Are there languages that more closely follow your intuition?
Pointers Are A Forgetful Encoding
What Did Pointers Forget?

Are there languages that more closely follow your intuition?

```java
public class Foo {
    public int a[2];
    private int b[2];

    public int f() {
        return b[0];
    }
}

public class Bar {
    public static int g(Foo o) {
        return o.a[0];
    }
}
```
Are there languages that more closely follow your intuition?

```java
public class Foo {
    public int a[2];
    private int b[2];

    public int f() {
        return b[0];
    }
}

public class Bar {
    public static int g(Foo o) {
        return o.a[0];
    }
}
```
Pointers Are A Forgetful Encoding
What Did Pointers Forget?

OK, so why don’t we just write everything in Java?

- Some performance concerns, maybe.
- Correctness depends on extremely huge runtime system.
  - It’s written in C!
  - By humans!
- Correctness *can be subverted* using the FFI (“JNI”).
  - In fact, an earlier case for CHERI was to better enforce Java’s object model in native calls!
Pointers Are A Forgetful Encoding
What Did Pointers Forget?

Speaking of C,
- our array turned into... an address.
- we forgot the type, the length, the access rights...
What if we made a 

What if we made a *machine* where *every* memory access had its bounds and permissions checked?
What if...

- What if we made a *machine* where every memory access had its bounds and permissions checked?
- How?
What if... 

> What if we made a *machine* where every memory access had its bounds and permissions checked? 
> How? 
> Have pointers carry around more state!
What if we made a *machine* where *every* memory access had its bounds and permissions checked?

How?

Have pointers carry around more state!

Other people have tried this before (Intel iAPX 432).
  - It did not go well for them.
  - We think we are sufficiently different that things will go better this time!
A memory capability has...
▶ A current position, *cursor*, just like a pointer.
▶ Bounds: *base* and *limit*.
▶ A set of *rights* (R, W, X, LC, SC, ...) and some flags.
▶ A 128-bit format for being stored in memory.
A Systems Programmer’s CHERI
Architectural Capabilities

To use a capability, it has to be in a register (like pointer).

- load $target, $offset, offset($cap)
- store $source, $offset, offset($cap)
- Instructions fault if offsets take address out of bounds or $cap’s rights do not authorize the operation.

a_sub_3 is essentially unchanged: load via its capability parameter using the capability load instruction.
To *create* a capability, it needs a *progenitor* capability!

- It is *not possible* to create a capability *from scratch*!
- Can copy, though: move $dstcap, $srccap
To create a capability, it needs a progenitor capability!

- It is not possible to create a capability from scratch!
- Can copy, though: move $dstcap, $srccap
- Can raise base (but not past limit):*
  ```c
  csetbase $dstcap, $newbase, $srccap
  ```
- Can lower limit (but not past base):*
  ```c
  csetlen $dstcap, $size, $srccap
  ```

* marks some small fibs; ask me after class.
To create a capability, it needs a progenitor capability!

- It is not possible to create a capability from scratch!
- Can copy, though: move $dstcap, $srccap
- Can raise base (but not past limit):*
  csetbase $dstcap, $newbase, $srccap
- Can lower limit (but not past base):*
  csetlen $dstcap, $size, $srccap
- Can remove rights (logical AND):
  candperm $dstcap, $mask, $srccap

* marks some small fibs; ask me after class.
To create a capability, it needs a progenitor capability!

- It is not possible to create a capability from scratch!
- Can copy, though: move $dstcap, $srccap
- Can raise base (but not past limit):*
  \[
  \text{csetbase } \text{dstcap}, \text{newbase}, \text{srccap}
  \]
- Can lower limit (but not past base):*
  \[
  \text{csetlen } \text{dstcap}, \text{size}, \text{srccap}
  \]
- Can remove rights (logical AND):
  \[
  \text{candperm } \text{dstcap}, \text{mask}, \text{srccap}
  \]
- Invalid operations give rise to NULL (not faults).*

* marks some small fibs; ask me after class.
So what is main’s job, now?

- Allocate stack frame, spill callee saves
- Fill in array
- **Construct limited capability to array**
  - Use *the stack capability* as the progenitor
  - Set its base to the base of the array, and length 16
  - Remove its write permissions (maybe; breaks C)
- Call function* with *that capability*

* Might also have to guard *the stack capability* itself, if we don’t trust the caller to do the right thing. Ask me after class.
In terms of code, we

- use `move $4, $sp` to get a copy of the stack capability.
- Set its base to its current location:
  
  \[ \text{csetbase $4, $4, $4} \]
  
- Set its length to 16:
  
  \[ \text{ori $2, $zero, 16} \]
  \[ \text{csetlen $4, $2, $4} \]
  
- Remove its write permissions:
  
  \[ \text{ori $2, $zero, } \sim \text{CHERI_PERM_WRITE} \]
  \[ \text{candperm $4, $2, $4} \]
  
- Make call:
  
  \[ \text{jalr $25} \]
If `main` does that, then in `a_sub_3`,

- out of bounds accesses caught *where they happen*!
- writes will be caught!
Genre: A Systems Programmer’s CHERI
Title: Tag, You’re It!

- OK, but... capabilities are still just bits?
  - Just make them up whenever desired, yeah?

- Tags are 1-bit tags for each capability-sized and -aligned region of RAM.
- Tags are also present on cache lines and registers.
- Tags are set when capability is created from another.
- Tags are cleared by all byte-manipulating operations.
- (Tag memory is DRAM but not architecturally reachable.)
- Tagged capabilities to all memory present at power-on.
- Net result: cannot make these things up.
- Must have provenance chain for all capabilities.
A Systems Programmer’s CHERI
Tag, You’re It!

- OK, but... capabilities are still just bits?
  - Just make them up whenever desired, yeah?
- No! CHERI introduces 1-bit *tag*...
  - for each capability-sized and -aligned region of RAM.
  - Tags also present on cache lines and registers.
  - Tag is set when capability created from another.
  - Tag is cleared by all byte-manipulating operations.
  - *(Tag memory is DRAM but not architecturally reachable.)*
A Systems Programmer’s CHERI
Tag, You’re It!

- OK, but... capabilities are still just bits?
  - Just make them up whenever desired, yeah?
- No! CHERI introduces 1-bit tag...
  - for each capability-sized and -aligned region of RAM.
  - Tags also present on cache lines and registers.
  - Tag is set when capability created from another.
  - Tag is cleared by all byte-manipulating operations.
  - (Tag memory is DRAM but not architecturally reachable.)
- Tagged capabilities to all memory present *at power-on.*
A Systems Programmer’s CHERI
Tag, You’re It!

- OK, but... capabilities are still just bits?
  - Just make them up whenever desired, yeah?
- No! CHERI introduces 1-bit *tag*...
  - for each capability-sized and -aligned region of RAM.
  - Tags also present on cache lines and registers.
  - Tag is set when capability created from another.
  - Tag is cleared by all byte-manipulating operations.
  - (Tag memory is DRAM but not architecturally reachable.)
- Tagged capabilities to all memory present *at power-on*.
- Net result: cannot make these things up.
  - Must have *provenance chain* for all capabilities.
Can’t make up a capability $\Rightarrow$ can use only what we have.

- Those in register files.
- Those reachable from other ones we have.

It’s possible to build confined pieces of software that clearly cannot interact with other parts of the system.

Sandboxing is easy!

CheriBSD demo running gzip in sandbox.

Too much for this talk, but: CHERI has fast mechanisms for domain transition between confined components.

Can transition to domain only via capability.

Possible to divide a program’s authority across mutually distrusting components that carefully validate requests and responses.
A Systems Programmer’s CHERI Confinement

- Can’t make up a capability ⇒ can use only what we have.
  - Those in register files.
  - Those reachable from other ones we have.
- It’s possible to build confined pieces of software that clearly cannot interact with other parts of the system.
  - Sandboxing is easy!
  - CheriBSD demo running gzip in sandbox.
A Systems Programmer’s CHERI Confinement

- Can’t make up a capability \( \Rightarrow \) can use only what we have.
  - Those in register files.
  - Those reachable from other ones we have.
- It’s possible to build *confined* pieces of software that clearly cannot interact with other parts of the system.
  - Sandboxing is easy!
  - CheriBSD demo running `gzip` in sandbox.
- Too much for this talk, but: CHERI has fast mechanisms for *domain transition* between confined components.
  - Can transition to domain only via capability.
  - Possible to divide a program’s authority across mutually distrusting components that carefully validate requests and responses.
The CHERI Project Status
At Large

- PostgreSQL
- QTWebKit
- libc
  (varargs, malloc, linker)
- Kernel (FreeBSD 12!)
  (ABI, VM, pager, execve)
- CHERI MIPS CPU
- Cache hierarchy
  Tag cache
- DRAM
- LLVM
  clang
  lld
- FPGA
The CHERI Project Status
At Large

- Large, real software stack.
  - FreeBSD fork ("CheriBSD") works great
  - LLVM cross-compiler works (CHERI C++ is still WIP)
  - Enterprise database works
  - JavaScript interpreter in C++ WIP, but coming
  - gdb works as well as it ever does

- FPGA implementation of CHERI MIPS
  - "Done" and entering maintenance stage?
  - Refocusing effort on CHERI RISC V.
  - (qemu implementation for us software types)

- Formal methods heavily used. Prove things about ISA!
The CHERI Project Status
In The Kernel

- Kernel is a *hybrid* executable: MIPS, but also caps.
- CheriBSD/CheriABI processes speak capabilities, not pointers, to the kernel.
The CHERI Project Status

In The Kernel

- Kernel is a *hybrid* executable: MIPS, but also caps.
- CheriBSD/CheriABI processes speak capabilities, not pointers, to the kernel.
- Kernel knows how to DTRT with caps and...
  - Context switch and signaling
  - Process construction (execve)
  - VM operations (mmap)
  - Swap (disks don’t support tags; reconstitute on page-in)
The CHERI Project Status
In The Kernel

- Kernel is a *hybrid* executable: MIPS, but also caps.
- CheriBSD/CheriABI processes speak capabilities, not pointers, to the kernel.
- Kernel knows how to DTRT with caps and...
  - Context switch and signaling
  - Process construction (execve)
  - VM operations (mmap)
  - Swap (disks don’t support tags; reconstitute on page-in)
- We are working to reduce the authority of pieces of the kernel itself.
  - No “omnipotent” capabilities after early load: separated kernel and user roots.
  - Very little of the kernel needs all the authority it has!
We want to *compartmentalize* the kernel:

- Interrupt handlers do not have access to user memory?
We want to *compartmentalize* the kernel:

- Interrupt handlers do not have access to user memory?
- Drivers only have access to driver state and granted capabilities?
The CHERI Project Status
In The Kernel

We want to *compartmentalize* the kernel:

- Interrupt handlers do not have access to user memory?
- Drivers only have access to driver state and granted capabilities?
- Only the page-in code and `mmap` have access to the user root capability?
We want to *compartamentalize* the kernel:

- Interrupt handlers do not have access to user memory?
- Drivers only have access to driver state and granted capabilities?
- Only the page-in code and `mmap` have access to the user root capability?
- Probably doable in months of engineering time.
The CHERI Project Status

Benchmarks

From our ASPLOS’19 submission:
The CHERI Project Status
Some Research Projects: *colocated processes*

- Pebbles and everyone else uses the MMU to isolate tasks. This stinks: TLB flushes, IPC slow (& SHM messy), . . .
The CHERI Project Status
Some Research Projects: *colocated processes*

- Pebbles and everyone else uses the MMU to isolate tasks. This stinks: TLB flushes, IPC slow (& SHM messy), . . .
- Recall: a CHERI process is *confined* by its registers.
The CHERI Project Status
Some Research Projects: *colocated processes*

- Pebbles and everyone else uses the MMU to isolate tasks. This stinks: TLB flushes, IPC slow (\& SHM messy), \ldots
- Recall: a CHERI process is *confined* by its registers.
- CHERI can run multiple, independent, isolated processes in one address space!
  - Multiprocessing as cheap as multithreading!
  - IPC is fast w/ just shared *capabilities*; much simpler.
The CHERI Project Status
Some Research Projects: *colocated processes*

- Pebbles and everyone else uses the MMU to isolate tasks. This stinks: TLB flushes, IPC slow (& SHM messy), . . .
- Recall: a CHERI process is *confined* by its registers.
- CHERI can run multiple, independent, isolated processes in one address space!
  - Multiprocessing as cheap as multithreading!
  - IPC is fast w/ just shared *capabilities*; much simpler.
- Requires extensive reworking of the kernel to separate “the current process” from “the current address space”.
The CHERI Project Status
Some Research Projects: *sweeping revocation*

- CHERI tags all pointers in the system.
- We can *enforce* that *free* means it!
  - Go and find and erase all references to *free’d* memory!
  - In batches, so we’re not excessively slow.
- Kernel now has “capability revoke” syscalls.
- Architectural changes to CHERI for performance.
I’m giving another, more detailed CHERI talk tomorrow:

- CIC Panther Hollow Conference Room 4101 @ 14:00
- Will be a “choose your own adventure” style, so come vote?

15-412 Fall ’19?

- Plan 9 + CHERI (MIPS or RISC V)?
- If you like FreeBSD, we can probably do that, too.
Advertising!

Reading List Suggestions

- Davis et al. *CheriABI: Enforcing Valid Pointer Provenance and Minimizing Pointer Privilege in the POSIX C Run-time Environment*. ASPLOS ’19 (next week!).
Questions?
A CHERI capability *cursors* are mostly for C compatibility.

- strcpy likes to use *p++, for example.
- If capabilities were just ranges, this would mean we’d need *two* registers to hold a C pointer: the capability and the current position within the range!
- Instead, we encode those together in the capability.

```c
void strcpy(char *dst, const char *src) {
    char c;
    while((c = *src++) = '\0')
        *dstrup++ = c;
}
```
Originally, CHERI capabilities were 256 bits for 64-bit virtual addresses.

- base + cursor + limit + permissions

But base, cursor, and limit share a lot of bits.

A floating-point like mechanism lets us compress capabilities to 128 bits.