CHERI
C/C++-language and compiler support


University of Cambridge, SRI International

CHERI Microkernel Workshop – 23 April 2016
Compatibility vs protection

Unmodified binaries (Legacy ABI)

Annotation code (Hybrid ABI)

Pure-capability ABI

More secure

More compatible

Pointer representation:

Integers
Manipulated as data
No tie to objects

Capabilities
Bounds checked
Permissions enforced
Protected by tag
Why capabilities for pointers?

```c
int foo[32];
union
{
    int *a;
    int b;
} un;

foo[32] = 12; // Bound violation, run-time trap
un.b = 12;
un.a[0]; // Tag violation, run-time trap
```

- Tags allow pointers to be identified for accurate garbage collection
- Memory protection is a foundation for compartmentalisation
Pointer provenance matters!

- CHERI C is a single-provenance model
- Every valid pointer is derived from precisely one object (e.g. `malloc()` or stack allocation)
- Pointer arithmetic moves the offset
- Bounds are never implicitly changed
Provenance-carrying integers

`intptr_t(__intcap_t)` carries provenance

\[
\begin{align*}
\text{int *cap} & = \ldots; \\
\text{intptr_t iptr} & = (\text{intptr_t})\text{cap}; \\
\text{cap} & = (\text{int *})\text{iptr}; \\
\text{long offset} & = \text{iptr};
\end{align*}
\]
Non-provenance-carrying integers

Other integer types do not carry provenance

```c
int *cap = ...;
long iptr = (long)cap;

Invalid pointer
Traps on dereference

cap = (int *)iptr;

Value stored in offset

intptr_t addr = iptr;
```
Memory-safe variadics

• `va_list` is a capability

• Caller passes the on-stack arguments in register

• Callee increments offset for next argument

// Ooops: Stack corruption
scanf("%ld %ld", &someDouble);

// Deep in scanf:
va_list ap;
// Length violation with CHERI:
long x = va_arg(ap, long);
Stack Protection

Legacy
- Return Address
- Stack Pointer
- Saved registers
- On-stack buffer…

Hybrid
- Return Address
- Stack Pointer
- Return Capability
- Saved registers
- On-stack buffer…

Pure-Capability
- Return Capability
- Stack Pointer
- Saved registers
- On-stack buffer…

\[ ld \] $ra, \$sp
\[ jr \] $ra
\[ cgetpccsetoffset \] $c17, $ra
\[ csc \] $c17, $sp($c11)
\[ csc \] $c17, $sp($c11)
\[ cjr \] $c17
\[ csc \] $c17, $sp($c11)
\[ cjr \] $c17
\[ cjr \] $c17
C-like languages

- C++
  - Adds vtables to C structs
  - Multiple inheritance
- Objective-C
  - Adds Smalltalk-like object model, closures

Object pointers should be capabilities
C++ Code-Reuse Attack

• Example initial gadget:

```cpp
virtual ~Course() {
    for (size_t i = 0; i < nStudents; i++)
        students[i]->decCourseCount();
    delete students;
}
```

• Overlapping objects for dataflow

```cpp
virtual calculateSum() {
    sum = scoreA + scoreB + scoreC;
}
```

The computed `sum` field becomes the `buffer` pointer for the next gadget
Possible approach

- Capabilities for vtable pointers, ensuring they always point to the start of a valid vtable
- Capabilities for object integrity
  - Read-only access to vptr
  - Write access only to member fields
- Have to consider CHERI-aware adversary
BACKUP SLIDES
Pure-capability Objective-C

- GNUstep Objective-C runtime
  - Used by WinObjC, CrystalX Android SDK, etc.
  - Complete modern Objective-C implementation
  - 11,533 lines of code, including 839 of assembly
  - 8 lines of intptr_t changes
  - 10 lines of changes for a bitfield encoded in a pointer-sized value
  - 163 lines of assembly for CHERI message send function (183 for MIPS, 114 for ARM, 79 for AArch64)
Incremental adoption

• Annotated pointers are capabilities
• Unannotated pointers are integers
• Compiler may use capabilities for non-ABI addresses (e.g., return address)
• Can protect high-value code
• Mostly useful for legacy interfaces to fully memory-safe libraries
Pointer annotation

```c
int foo[32];
__capability int *bar = (__capability int*)foo;
```

- Only specially annotated pointers are capabilities
- Compiler attempts to infer bounds
int foo(char *);
int bar(void) {
    char buffer[128];
    return foo(buffer);
}

Function Prolog

MIPS

```
bar:
    daddiu $sp, $sp, -160
    sd $ra, 152($sp)
    sd $fp, 144($sp)
    sd $gp, 136($sp)
    move $fp, $sp
```

CHERI

```
bar:
    daddiu $sp, $sp, -192
    csd $fp, $sp, 184($c11)
    csd $gp, $sp, 176($c11)
    csc $c17, $sp, 128($c11)
    move $fp, $sp
```

Save return address
```c
int foo(char *);
int bar(void) {
    char buffer[128];
    return foo(buffer);
}
```

**GOT address setup**

**MIPS**

```assembly
lui $1, %hi(\text{neg}(\%gp\_rel(bar)))
daddu $1, $1, $25
daddiu \$gp, $1, %lo(\text{neg}(\%gp\_rel(bar)))
```

**CHERI**

```assembly
cgetoffset \$25, \$c12
lui \$1, %hi(\text{neg}(\%gp\_rel(bar)))
daddu \$1, \$1, \$25
daddiu \$gp, \$1, %lo(\text{neg}(\%gp\_rel(bar)))
```

Get PCC-relative offset of function
(Will be obsoleted by a CHERI linker)
int foo(char *);
int bar(void) {
    char buffer[128];
    return foo(buffer);
}

Set base and bounds for buffer

MIPS

\[\text{daddiu } $4, $fp, 8\]

CHERI

\[
\begin{align*}
\text{daddiu } & $1, $fp, 0 \\
\text{csetoffset } & $c1, $c11, $1 \\
\text{daddiu } & $1, $zero, 128 \\
\text{csetbounds } & $c3, $c1, $1
\end{align*}
\]

Hope that foo doesn't overflow the buffer!
int foo(char *);
int bar(void) {
    char buffer[128];
    return foo(buffer);
}

Get address of foo

MIPS

```assembly
ld  $25, %call16(foo)($gp)
```

CHERI

```assembly
daddiu  $1, $gp, %call16(foo)
cfromptr $c1, $c0, $1
cld   $1, $zero, 0($c1)
cgetpccsetoffset  $c12, $1
```

Longer sequence on CHERI because we use MIPS relocations with CHERI instructions
(Will be fixed with a CHERI linker)
int foo(char *);
int bar(void) {
    char buffer[128];
    return foo(buffer);
}

Call foo

MIPS
jalr $25, $ra

CHERI
cjalr $c17, $c12
```c
int foo(char *);
int bar(void) {
    char buffer[128];
    return foo(buffer);
}
```

**Function Epilog**

**MIPS**

```
move $sp, $fp
ld $gp, 136($sp)
ld $fp, 144($sp)
ld $ra, 152($sp)
jr $ra
daddiu $sp, $sp, 160
```

**CHERI**

```
move $sp, $fp
clc $c17, $sp, 128($c11)
cld $gp, $sp, 176($c11)
cld $fp, $sp, 184($c11)
cjr $c17
daddiu $sp, $sp, 192
```

Reload return address