CTSRD

CRASH-WORTHY TRUSTWORTHY SYSTEMS RESEARCH AND DEVELOPMENT

CHERI

C/C++-language and compiler support

David Chisnall, **Khilan Gudka**, Robert N. M. Watson, Simon W. Moore, Peter G. Neumann, Jonathan Woodruff, Jonathan Anderson, Hadrien Barral, Ruslan Bukin, Nirav Dave, Brooks Davis, Lawrence Esswood, Alexandre Joannou, Chris Kitching, Ben Laurie, A. Theo Markettos, Alan Mujumdar, Steven J. Murdoch, Robert Norton, Philip Paeps, Alex Richardson, Michael Roe, Colin Rothwell, Hassen Saidi, Stacey Son, Munraj Vadera, Hongyan Xia, and Bjoern Zeeb

University of Cambridge, SRI International

CHERI Microkernel Workshop – 23 April 2016

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 ('CTSRD') and FA8750-11-C-0249 ('MRC2'). 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.

SRI International

UNIVERSITY OF CAMBRIDGE

Compatibility vs protection

More secure →

- Unmodified binaries (Legacy ABI)
- Annotated code (Hybrid ABI)
- Pure-capability ABI

← More compatible

Pointer representation:

Integers
Manipulated as data
No tie to objects

Capabilities
Bounds checked
Permissions enforced
Protected by tag

Why capabilities for pointers?

```
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

```
int *cap = ...;
```

No representation change

```
intptr_t iptr = (intptr_t)cap;
```

Safe round trip

```
cap = (int *)iptr;
```

Gets offset

```
long offset = iptr;
```

Non-provenance-carrying integers

Other integer types do not carry provenance

```
int *cap = ...;
```

Gets virtual address

```
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);
```

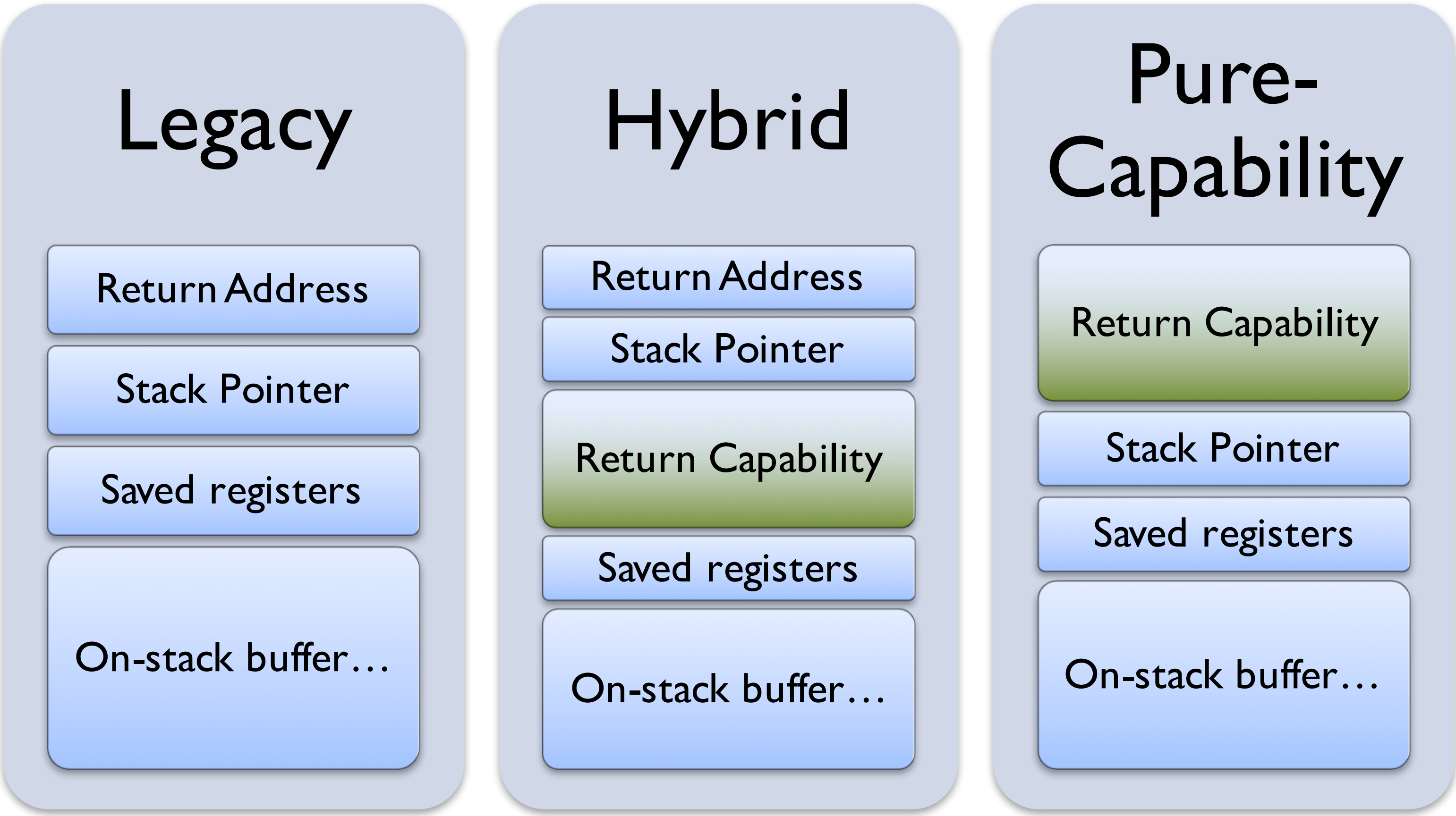
```
// Deep in scanf:
va_list ap;
// Length violation with CHERI:
long x = va_arg(ap, long);
```

Stack Protection

```
sd $ra, $sp
```

```
cgetpccsetoffset  $c17, $ra
csc          $c17, $sp($c11)
```

```
csc  $c17, $sp($c11)
```




```
ld   $ra, $sp
jr   $ra
```

```
clc  $c17, $sp($c11)
cjr  $c17
```

```
clc   $c17, $sp($c11)
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

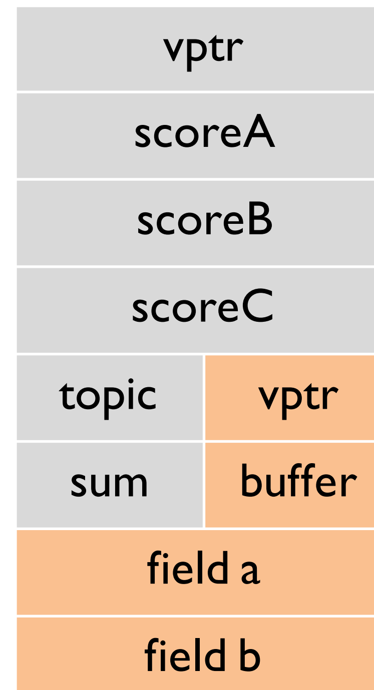
- Example initial gadget:

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

- Overlapping objects for dataflow

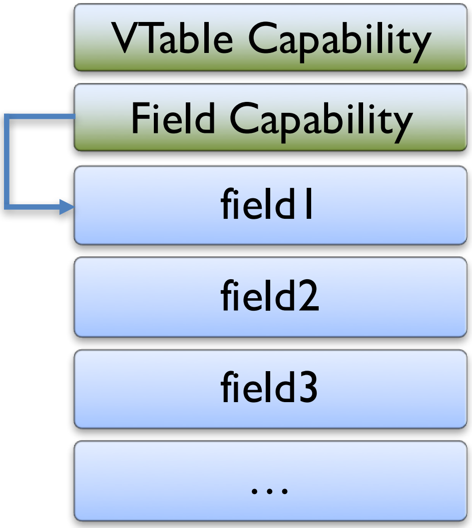
```
virtual calculateSum() {
    sum = scoreA + scoreB + scoreC;
}
```

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

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

- Only specially annotated pointers are capabilities
- Compiler attempts to infer bounds

```c
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

```
lui $1,
      %hi(%neg(%gp_rel(bar)))
daddu   $1, $1, $25
daddiu  $gp, $1,
      %lo(%neg(%gp_rel(bar)))
```

CHERI

```
cgetoffset  $25, $c12
lui $1,
   %hi(%neg(%gp_rel(bar)))
daddu   $1, $1, $25
daddiu  $gp, $1,
%lo(%neg(%gp_rel(bar)))
```

Get PCC-relative offset of function
(Will be obsoleted by a CHERI linker)

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

Set base and bounds for buffer

MIPS

```
daddiu  $4, $fp, 8
```

CHERI

```
daddiu     $1, $fp, 0
csetoffset $c1, $c11, $1
daddiu     $1, $zero, 128
csetbounds $c3, $c1, $1
```

Hope that foo doesn't overflow the buffer!

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

Get address of `foo`

MIPS

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

CHERI

```
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)

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

Call foo

MIPS	CHERI
`jalr $25, $ra`	`cjalr $c17, $c12`

```
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