CTSRD CRASH-WORTHY TRUSTWORTHY SYSTEMS RESEARCH AND DEVELOPMENT

Protecting C++ Programs with CHERI

Khilan Gudka, Alexander Richardson, Robert N. M. Watson
University of Cambridge

PriSC 2019
13 January 2019

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 contract FA8750-10-C-0237. 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

The need for C++ memory safety

- Many widely used applications are written in C++: web browsers, mail readers, office suites, etc.
- These applications handle untrusted data and are thus quite susceptible to spatial and temporal security vulnerabilities.
- This can lead to information leaks, privilege escalation, arbitrary code execution.

CHERI protection model

- **RISC hybrid-capability architecture** supporting fine-grained, **pointer-based memory protection**:

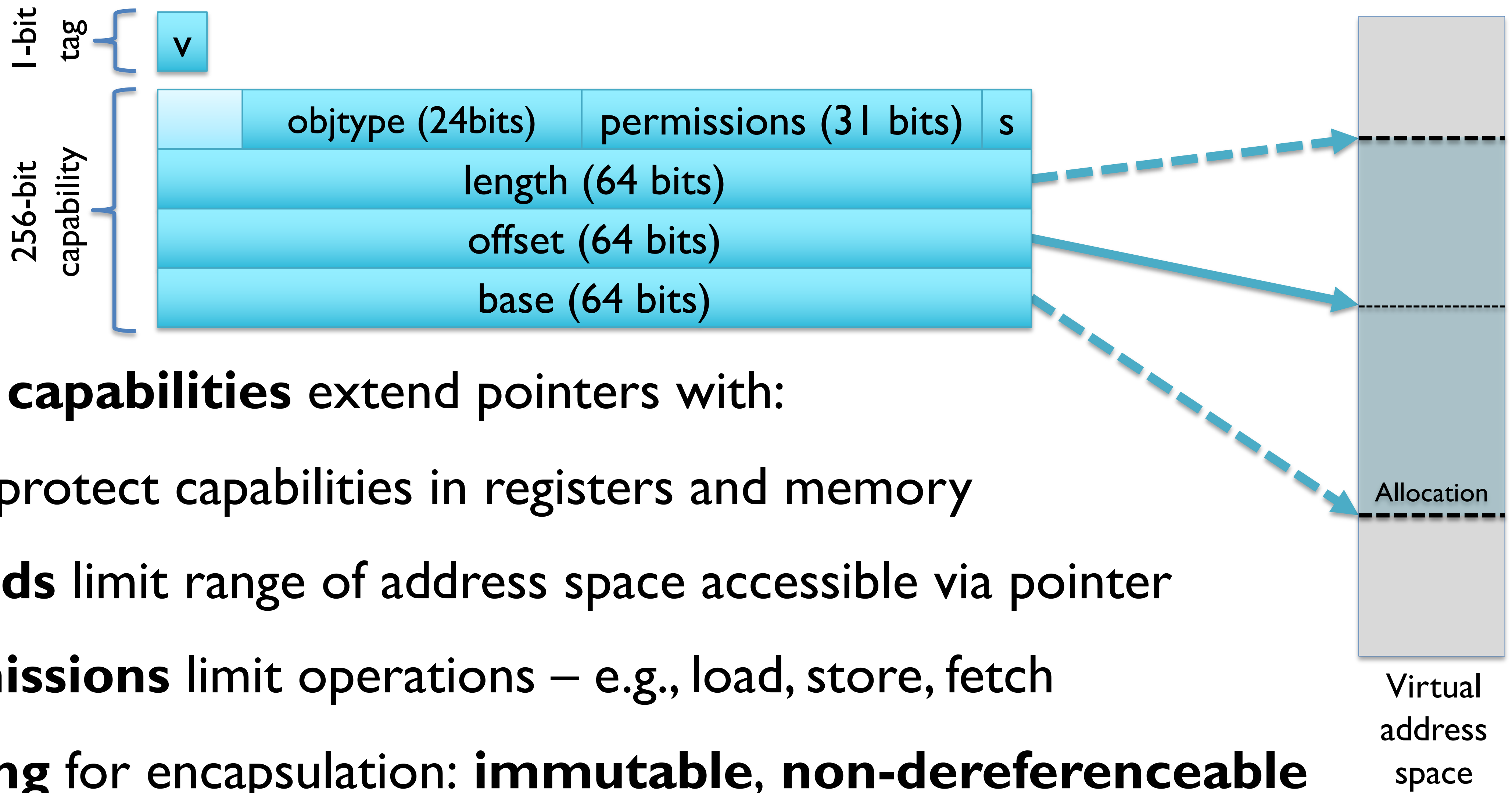
Protect **pointer**

  - **pointer integrity** (e.g., no pointer corruption)
  - **pointer provenance validity** (e.g., no pointer injection)

Protect **pointee**

  - **bounds checking** (e.g., no buffer overflows)
  - **permission checking** (e.g., W^X for pointers)
  - **monotonicity** (e.g., no privilege escalation / improper re-use)
  - **encapsulation** (e.g., protect software objects)

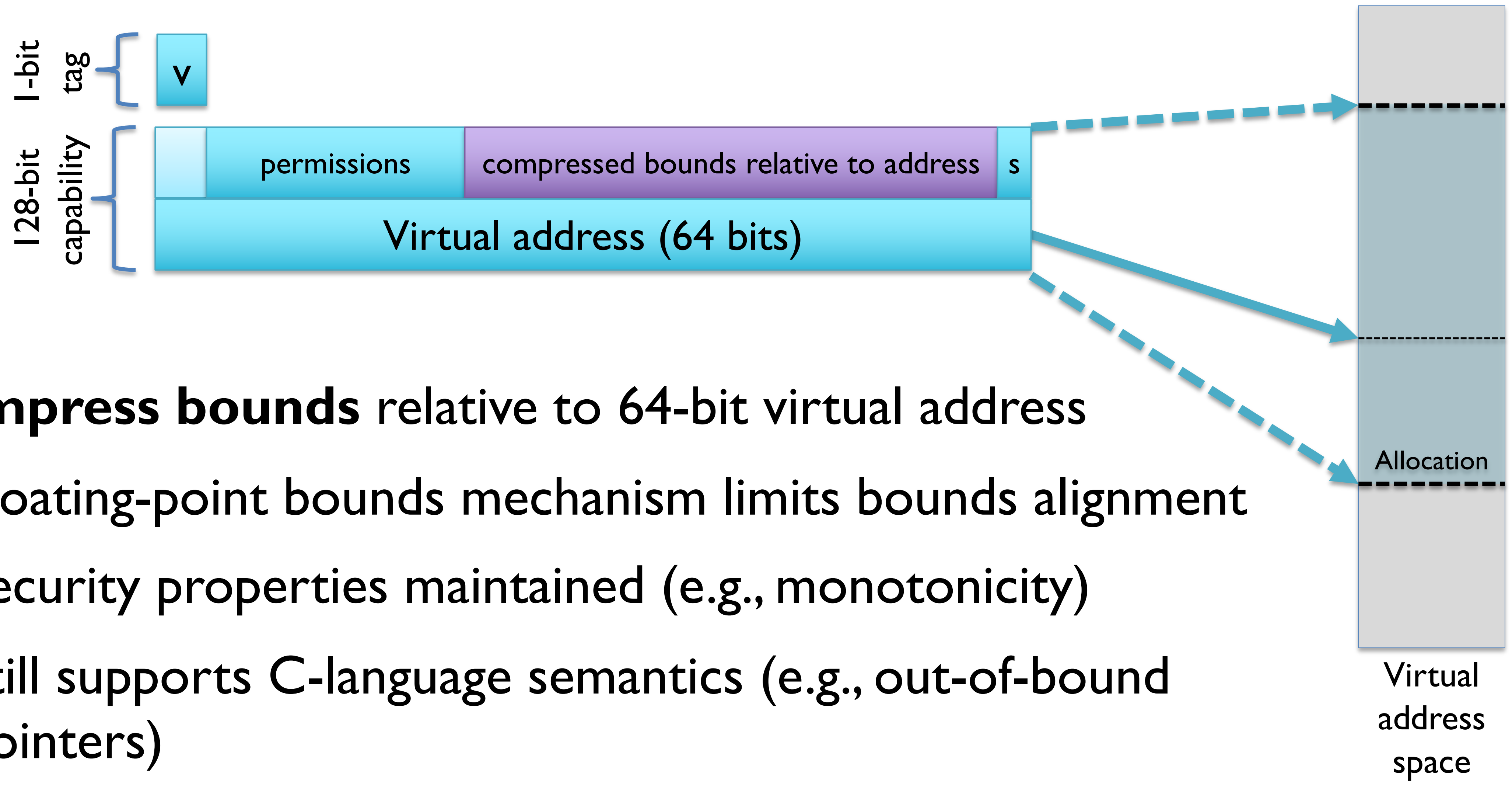
CHERI: 256-bit architectural capabilities



CHERI capabilities extend pointers with:

- **Tags** protect capabilities in registers and memory
- **Bounds** limit range of address space accessible via pointer
- **Permissions** limit operations – e.g., load, store, fetch
- **Sealing** for encapsulation: **immutable, non-dereferenceable**

CHERI: 128-bit micro-architectural capabilities



- **Compress bounds** relative to 64-bit virtual address
  - Floating-point bounds mechanism limits bounds alignment
  - Security properties maintained (e.g., monotonicity)
  - Still supports C-language semantics (e.g., out-of-bound pointers)

Compiling C++ to CHERI

- Two modes of compilation using Clang/LLVM:
  - *Hybrid* – annotate which pointers should become capabilities (like `const`, `volatile`)
  - *Pure* – all pointers are turned into capabilities

```
class A { public: int f; }

A* __capability a = new A;
a->f = 42;
```

new calls `malloc()` which sets bounds on the allocation

LLVM IR:

```
%call = tail call i8 addrspace(200)* @operator new(unsigned long)(i64 zeroext 4)
%f = bitcast i8 addrspace(200)* %call to i32 addrspace(200)*
store i32 42, i32 addrspace(200)* %f
```

Let’s start simple…

```
#include <iostream>
using namespace std;

cout << “Hello World!” << endl;
```

Look easy?

Challenges

- Almost all challenges have been in the compiler frontend
- Ensuring `__capability` is supported and propagated correctly
  - References, templates, function overloading
  - Initializer lists, static initialisation of structs
  - `nullptr`
- Memory alignment
  - std::align, `new`, `new[]`

Challenges

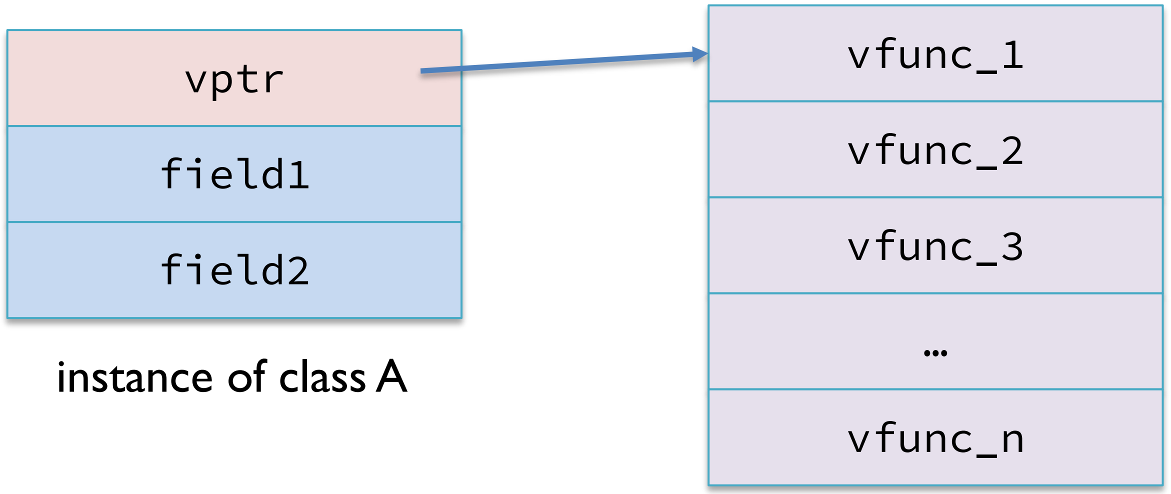
- Name mangling for capability-qualified types
  - `void foo(A* __capability)` → `_Z3fooU3capP1A`
- `<type_traits>` and `<hash>` specialisations for `__intcap_t`
- Pointer-to-members (last thing we had to fix for "Hello World!")
- **End result: can compile all of libc++ and all ~5K non-exception-non-rtti tests are passing.**
- Have implemented support for exceptions but not very well tested
  - Modifications to LLVM, libunwind, libcxxrt

Virtual-table hijacking

- Common code re-use attack is to use the dynamic dispatch mechanism to invoke arbitrary C++ virtual functions.
- *Counterfeit Object-Oriented Programming* paper (IEEE S&P 2015) explains a version of this attack:
  1. Find a loop over a collection of objects that invokes a virtual function on each object. vtable index is fixed at the call site.
  2. Exploit a memory vulnerability and inject a collection of objects each with their `vptr` fields set such that when the vtable index is added, the desired virtual gadget will be called.
  3. Overlap instance fields of these injected objects to achieve passing values between gadgets.

Virtual-table hijacking

- CHERI already prevents injection of arbitrary objects.
- We can harden the virtual call mechanism by enforcing integrity of the `vptr` by using sealed capabilities.
- Integrity here also means that the `vptr` points to the right vtable.



instance of class A

vtable for class A

Capability sealing mechanism

- Capability sealing allows capabilities to be marked as *immutable* and *non-dereferenceable*.
- Hardware exceptions are thrown if attempts are made to modify or dereference them.
- Sealed capabilities contain an additional piece of metadata, an *object type*, set when a memory capability undergoes sealing.
- *Sealing* capability has the PERMIT_SEAL permission and the object types that it is allowed to seal for.
- Object types allow multiple sealed capabilities to be linked.

`vptr` sealing mechanism

- Idea: replace `vptr` with a sealed capability
- `vptr` capability is sealed with the `otype` set to the class's type. Call this new capability `sealed-vptr`
- When an object is created, the `vptr` field is initialized to the `sealed-vptr` capability
- At a virtual function call, `sealed-vptr` is unsealed to get back the `vptr` capability.
  - Unseal successful → correct vtable pointer, proceed to call virtual function

Hardening other C++ features

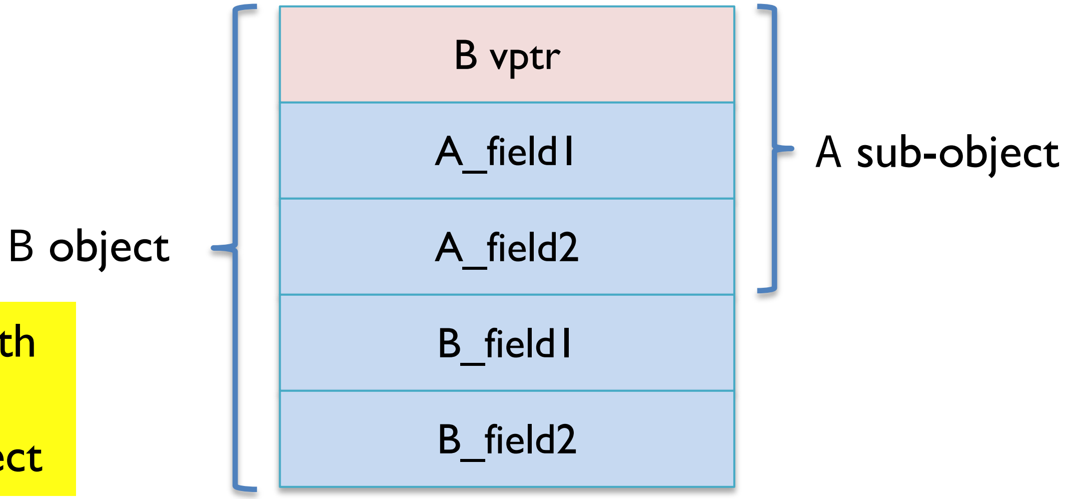
- **Tightening bounds for base-class sub objects.**
- When passing an object to a polymorphic call, which expects the base class type, set bounds to the base-class object.

```
class A { … }
class B : public A { … }

void foo(A* a) { … }

B* b = new B;
foo(b);
```

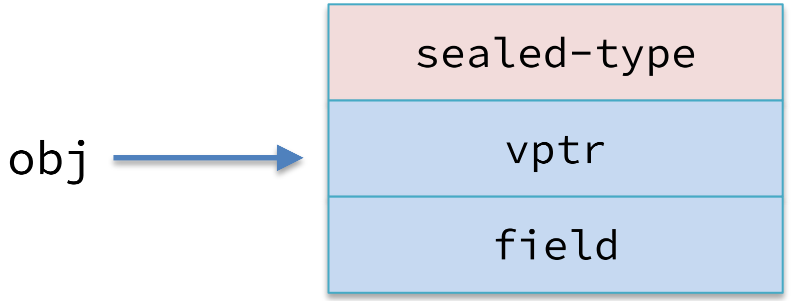
Pass capability with bounds tightened to the A sub-object



- Challenges: What if we later downcast? How common is this? Can the compiler identify these cases and not tighten bounds then?

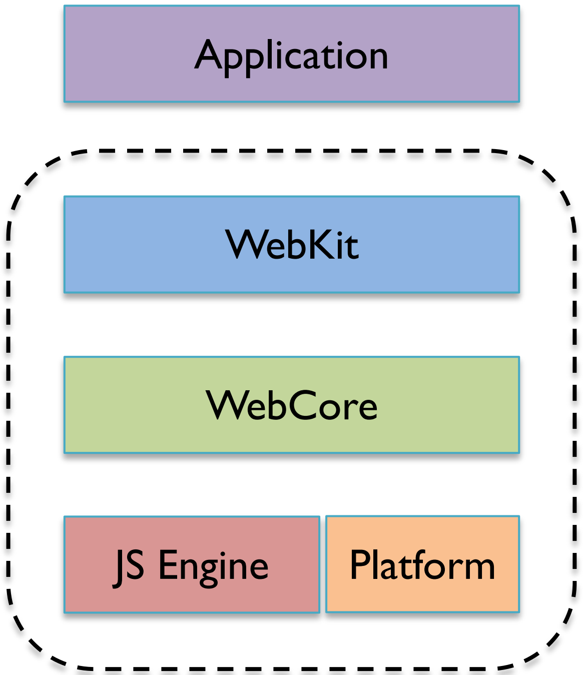
Hardening other C++ features

- **Type safety to prevent type confusion attacks.**
- A dangling pointer of class type A may now point to an object of some other class type B when the memory is re-allocated.
- Can lead to accessing sensitive data and executing arbitrary code.
- Idea: store a sealed type capability in each object and unseal it whenever deemed important.
  - Unseal successful → type matches, otherwise error.



WebKit case study

- Web rendering engine used in web browsers, such as Apple Safari.
- Very large C++ codebase: parsers, interpreters, untrusted data handling.



- **WebKit:** thin layer to link against from the applications
- **WebCore:** rendering, layout, network access, multimedia, accessibility support
- **JS Engine:** the JavaScript engine. JavaScriptCore by default, but can be replaced (e.g. V8 in Chromium)
- **Platform:** platform-specific hooks

JavaScriptCore case study

- JavaScriptCore Interpreter is written in a combination of C++ and typed target-independent assembly (?).
- The assembly is compiled (via ruby scripts!) to target-specific assembly (if supported) or C++.

```
subi 1, t3
loadp [protoCallFrame, t3, 16], extraTempReg
storep extraTempReg, CodeBlock[sp, t3, 16]
btinz t3, .copyHeaderLoop
```

Assuming JS pointers are 128-bit capabilities

Translation to C++ for CHERI

```
t3.i32 = t3.i32 - int32_t(0x1);
t3.clearHighWord();
t5.i = *CAST<intptr_t*>(t2.i8p + (t3.i << 4));
*CAST<intptr_t*>(sp.i8p + (t3.i << 4) + intptr_t(0x20)) = t5.i;
if (t3.i32 != 0)
   goto _offlineasm_doVMEntry__copyHeaderLoop;
```

JavaScriptCore case study

- Interpreter has virtual registers, stack and heap.
- C++ version is a large switch statement with gotos and computed gotos:

  ```
  opcode = t0.opcode;
  goto *opcode;
  ```

- Each JavaScript expression is turned into an array of ‘instructions’.
- An instruction could be an opcode, operand or any of…

Caller Frame	0
Return PC	16
CodeBlock	32
Callee	48
Argument Count	64
this	80
First argument	96
…	
Last argument	
…	
…	

Stack Frame

JS pointers are 128-bit capabilities

JavaScriptCore case study

```
union {
   Opcode opcode;
   int operand;
   unsigned unsignedValue;
   WriteBarrierBase<Structure> structure;
   StructureID structureID;
   WriteBarrierBase<SymbolTable> symbolTable;
   WriteBarrierBase<StructureChain> structureChain;
   WriteBarrierBase<JSCell> jsCell;
   WriteBarrier<Unknown>* variablePointer;
   Special::Pointer specialPointer;
   PropertySlot::GetValueFunc getterFunc;
   LLIntCallLinkInfo* callLinkInfo;
   UniquedStringImpl* uid;
   ValueProfile* profile;
   ArrayProfile* arrayProfile;
   ArrayAllocationProfile* arrayAllocationProfile;
   ObjectAllocationProfile* objectAllocationProfile;
   WatchpointSet* watchpointSet;
   void* pointer;
   bool* predicatePointer;
   ToThisStatus toThisStatus;
   TypeLocation* location;
   BasicBlockLocation* basicBlockLocation;
   PutByIdFlags putByIdFlags;
} u;
```

Instruction

```
union {
   EncodedJSValue value;
   CallFrame* callFrame;
   CodeBlock* codeBlock;
   EncodedValueDescriptor encodedValue;
   double number;
   int64_t integer;
} u;
```

(Virtual) Register

JavaScriptCore case study

- 64-bit NaN-boxing encoding to identify types of value:
  - Integers (top 16-bits all set):
    e.g. `0xffff000000000003 → 3`
  - Double-precision (at least 1 of the top 16 bits is set but not all):
    e.g. `0x3ff4eb851eb851ec → 1.245`
    e.g. `0x7ff9000000000000 → NaN`
  - Pointer values (only use low 48 bits):
    e.g. `0x164066810`

JavaScriptCore case study

- Teaching WebKit/JavaScriptCore the following:
  - JS pointers and registers are 128-bit capabilities
  - Fixing mixing of pointer-typed and int64/int32-typed instructions on the same register values
  - Fixing constant offsets to reflect capability-sized fields
  - Using virtual addresses in cases when offset is not enough (e.g. bitwise ops, inequalities, subtracting entire capabilities)*

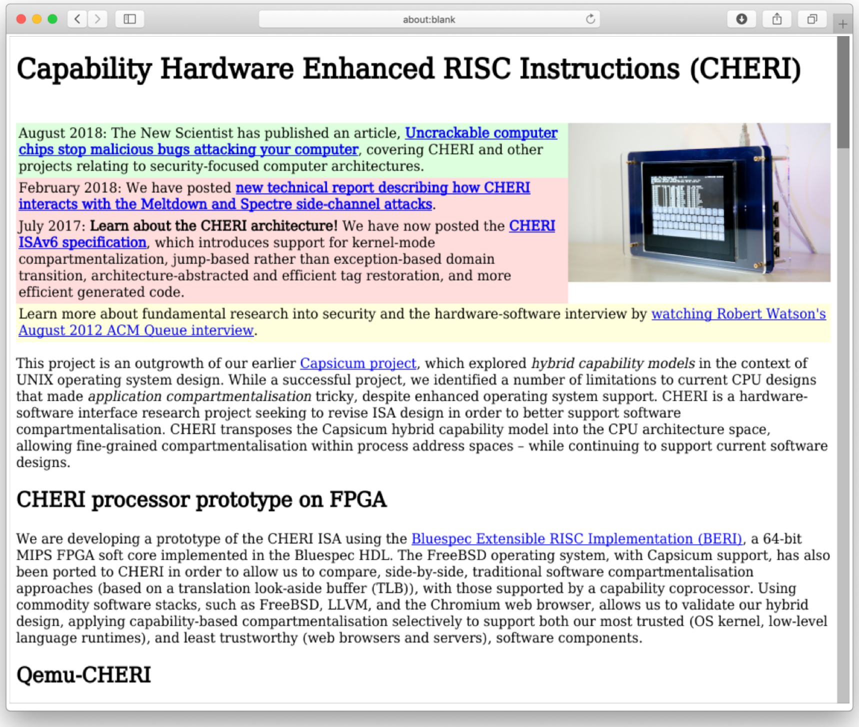
JavaScriptCore case study

- Other fixes:
  - Regular expressions
  - Exceptions, garbage collection
  - Reading and writing closure values
  - Various ops such as: op_inc, op_get_array_length, op_nstricteq, op_get_parent_scope, op_negate, op_to_number
  - Fix alignment when accessing and allocating multiple objects contiguously (in a single allocation)
  - Custom binary operations (e.g. concatenating an integer and a string)

JavaScriptCore case study

```
root@qemu-cheri128-kg365:~ # ./jsc
>>> var add3 = function(arg) { return arg + 3; }
>>> add3(5)
8
>>> add3(add3(5))
11
>>> 15.3 / 18 * 27.1 * (Math.ceil(1.3) * Math.exp(2.3) * Math.log (1.223) * Math.sin(32.22))
66.6192983328985
>>> print("hello" + ", " + "world!")
hello, world!
>>> var d = new Date()
>>> d.toDateString()
Sat May 12 2018
>>> parseInt('Infinity')
NaN
>>> new Date(0).toLocaleTimeString('zh-Hans-CN-u-nu-hanidec', { timeZone: 'Asia/Kolkata' } )
上午五:三〇:〇〇
```

WebKit



Capability Hardware Enhanced RISC Instructions (CHERI)

August 2018: The New Scientist has published an article, Uncrackable computer chips stop malicious bugs attacking your computer, covering CHERI and other projects relating to security-focused computer architectures.

February 2018: We have posted new technical report describing how CHERI interacts with the Meltdown and Spectre side-channel attacks.

July 2017: **Learn about the CHERI architecture!** We have now posted the CHERI ISAv6 specification, which introduces support for kernel-mode compartmentalization, jump-based rather than exception-based domain transition, architecture-abstracted and efficient tag restoration, and more efficient generated code.

Learn more about fundamental research into security and the hardware-software interview by watching Robert Watson's August 2012 ACM Queue interview.

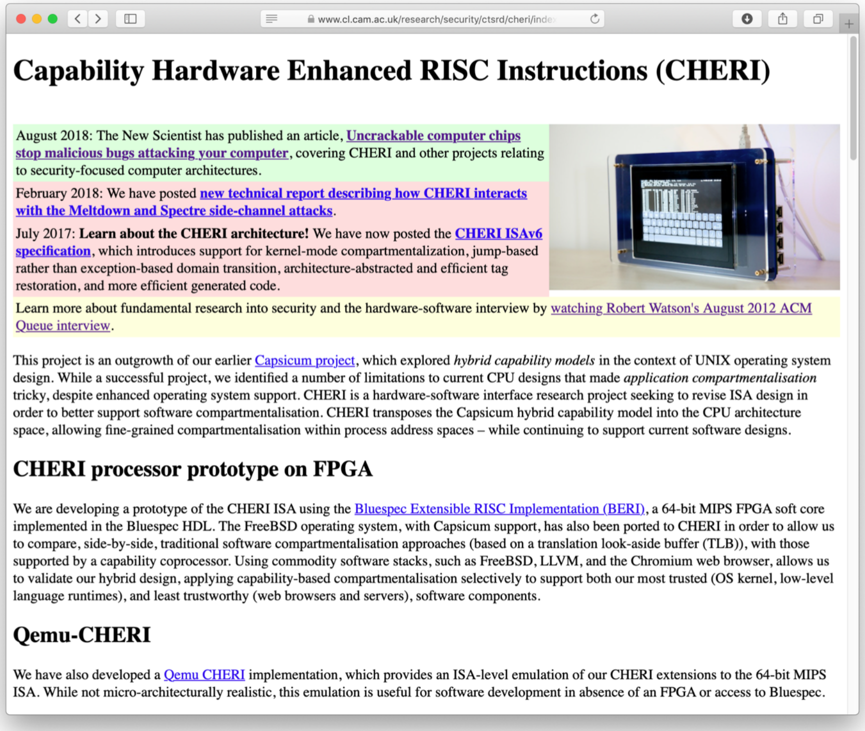
This project is an outgrowth of our earlier Capsicum project, which explored *hybrid capability models* in the context of UNIX operating system design. While a successful project, we identified a number of limitations to current CPU designs that made *application compartmentalisation* tricky, despite enhanced operating system support. CHERI is a hardware-software interface research project seeking to revise ISA design in order to better support software compartmentalisation. CHERI transposes the Capsicum hybrid capability model into the CPU architecture space, allowing fine-grained compartmentalisation within process address spaces – while continuing to support current software designs.

CHERI processor prototype on FPGA

We are developing a prototype of the CHERI ISA using the Bluespec Extensible RISC Implementation (BERI), a 64-bit MIPS FPGA soft core implemented in the Bluespec HDL. The FreeBSD operating system, with Capsicum support, has also been ported to CHERI in order to allow us to compare, side-by-side, traditional software compartmentalisation approaches (based on a translation look-aside buffer (TLB)), with those supported by a capability coprocessor. Using commodity software stacks, such as FreeBSD, LLVM, and the Chromium web browser, allows us to validate our hybrid design, applying capability-based compartmentalisation selectively to support both our most trusted (OS kernel, low-level language runtimes), and least trustworthy (web browsers and servers), software components.

Qemu-CHERI

CHERI-WebKit



Capability Hardware Enhanced RISC Instructions (CHERI)

August 2018: The New Scientist has published an article, Uncrackable computer chips stop malicious bugs attacking your computer, covering CHERI and other projects relating to security-focused computer architectures.

February 2018: We have posted new technical report describing how CHERI interacts with the Meltdown and Spectre side-channel attacks.

July 2017: **Learn about the CHERI architecture!** We have now posted the CHERI ISAv6 specification, which introduces support for kernel-mode compartmentalization, jump-based rather than exception-based domain transition, architecture-abstracted and efficient tag restoration, and more efficient generated code.

Learn more about fundamental research into security and the hardware-software interview by watching Robert Watson's August 2012 ACM Queue interview.

This project is an outgrowth of our earlier Capsicum project, which explored *hybrid capability models* in the context of UNIX operating system design. While a successful project, we identified a number of limitations to current CPU designs that made *application compartmentalisation* tricky, despite enhanced operating system support. CHERI is a hardware-software interface research project seeking to revise ISA design in order to better support software compartmentalisation. CHERI transposes the Capsicum hybrid capability model into the CPU architecture space, allowing fine-grained compartmentalisation within process address spaces – while continuing to support current software designs.

CHERI processor prototype on FPGA

We are developing a prototype of the CHERI ISA using the Bluespec Extensible RISC Implementation (BERI), a 64-bit MIPS FPGA soft core implemented in the Bluespec HDL. The FreeBSD operating system, with Capsicum support, has also been ported to CHERI in order to allow us to compare, side-by-side, traditional software compartmentalisation approaches (based on a translation look-aside buffer (TLB)), with those supported by a capability coprocessor. Using commodity software stacks, such as FreeBSD, LLVM, and the Chromium web browser, allows us to validate our hybrid design, applying capability-based compartmentalisation selectively to support both our most trusted (OS kernel, low-level language runtimes), and least trustworthy (web browsers and servers), software components.

Qemu-CHERI

We have also developed a Qemu CHERI implementation, which provides an ISA-level emulation of our CHERI extensions to the 64-bit MIPS ISA. While not micro-architecturally realistic, this emulation is useful for software development in absence of an FPGA or access to Bluespec.

Safari

CHERI homepage

Conclusion

- C++ is widely used for important applications, such as web browsers, office suites, mail readers, etc.
- CHERI provides fine-grained memory protection providing bounds and permissions checking.
- We are looking at hardening C++ features using CHERI: vtable pointers, type safety, base class bounds
- Evaluating with the WebKit rendering engine because it is a substantial C++ codebase with complex behavior and large trade off space.
- Performance?

Questions?