Formal Mechanised Semantics of CHERI C: Capabilities, Undefined Behaviour, and Provenance

Vadim Zaliva
University of Cambridge
Cambridge, UK
Vadim.Zaliva@cl.cam.ac.uk

Kayvan Memarian
University of Cambridge
Cambridge, UK
Kayvan.Memarian@cl.cam.ac.uk

Ricardo Almeida
University of Edinburgh
Edinburgh, UK
Ricardo.Almeida@ed.ac.uk

Jessica Clarke
University of Cambridge
Cambridge, UK
Jessica.Clarke@cl.cam.ac.uk

Brooks Davis
SRI International
Menlo Park, CA, USA
Brooks.Davis@sri.com

Alexander Richardson
University of Cambridge
Cambridge, UK
Alexander.Richardson@cl.cam.ac.uk

David Chisnall
Microsoft
Cambridge, UK
David.Chisnall@microsoft.com

Brian Campbell
University of Edinburgh
Edinburgh, UK
Brian.Campbell@ed.ac.uk

Ian Stark
University of Edinburgh
Edinburgh, UK
Ian.Stark@ed.ac.uk

Robert N. M. Watson
University of Cambridge
Cambridge, UK
Robert.Watson@cl.cam.ac.uk

Peter Sewell
University of Cambridge
Cambridge, UK
Peter.Sewell@cl.cam.ac.uk

Abstract

Memory safety issues are a persistent source of security vulnerabilities, with conventional architectures and the C codebase chronically prone to exploitable errors. The CHERI research project has shown how one can provide radically improved security for that existing codebase with minimal modification, using unforgeable hardware capabilities in place of machine-word pointers in CHERI dialects of C, implemented as adaptions of Clang/LLVM and GCC. CHERI was first prototyped as extensions of MIPS and RISC-V; it is currently being evaluated by Arm and others with the Arm Morello experimental architecture, processor, and platform, to explore its potential for mass-market adoption, and by Microsoft in their CHERIoT design for embedded cores.

There is thus considerable practical experience with CHERI C implementation and use, but exactly what CHERI C's semantics is (or should be) remains an open question. In this paper, we present the first attempt to rigorously and comprehensively define CHERI C semantics, discuss key semantics design questions relating to capabilities, provenance, and undefined behaviour, and clarify them with semantics in multiple complementary forms: in prose, as an executable semantics adapting the Cerberus C semantics, and mechanised in Coq.

This establishes a solid foundation for CHERI C, for those porting code to it, for compiler implementers, and for future semantics and verification.

ACM Reference Format:
Vadim Zaliva, Kayvan Memarian, Ricardo Almeida, Jessica Clarke, Brooks Davis, Alexander Richardson, David Chisnall, Brian Campbell, Ian Stark, Robert N. M. Watson, and Peter Sewell. 2024. Formal Mechanised Semantics of CHERI C: Capabilities, Undefined Behaviour, and Provenance. In *28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1 (ASPLOS '24), April 27-May 1, 2024, La Jolla, CA, USA.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3617232.3624859

ASPLOS '24, April 27-May 1, 2024, La Jolla, CA, USA

ACM ISBN 979-8-4007-0372-0/24/04.
https://doi.org/10.1145/3617232.3624859

1 Introduction

Memory safety bugs continue to be a major source of security vulnerabilities, despite much research on software bug-finding and mitigation approaches. For example, they are responsible for most of those addressed by Microsoft security updates or impacting Chromium [19, 29]. They are a particular concern for the large codebases in C and C++ that comprise the infrastructure that we all depend on. Alternative memory-safe languages offer promise, but these C/C++ codebases will clearly be an ongoing challenge for the foreseeable future.

The CHERI project [44], developed by the University of Cambridge and SRI International since 2010, offers a promising hardware-based approach. CHERI extends conventional hardware Instruction-Set Architectures (ISAs) to enable support for fine-grained memory protection and for scalable software compartmentalisation, with hardware-supported *capabilities*. In a 64-bit CHERI ISA, instead of using simple 64-bit machine-word virtual-address pointer values to access memory, restricted only by the memory management unit (MMU), one can use 128+1-bit capabilities that encode a virtual address together with the bounds of the memory it can access. Encoding these within the capability enables a fast access-time check, faulting if there is a safety violation. The ISA design ensures that capabilities cannot be forged, i.e, that normal code execution can shrink capabilities but never grow them, and there are additional "sealed-capability" features for secure encapsulation.

The initial academic work developed CHERI-MIPS and CHERI-RISC-V architecture extensions, along with FPGA processor implementations and system software (including adaptions of Clang/LLVM, linkers, debuggers, FreeRTOS, FreeBSD, and WebKit). Some initial design work on potential CHERI-x86 designs is in progress [1]. Arm, partly supported by the £190m UKRI Digital Security by Design (DSbD) programme [41], have now developed the Morello architecture, processor, and development board, extending the Armv8-A architecture and high-performance Neoverse N1 processor, to enable industrial evaluation that may support mass-market adoption in mobile or server cores [4, 5]. Meanwhile, the Microsoft CHERIoT project has developed the eponymous architecture, reference hardware design, and RTOS and software stack for an extension of 32-bit RISC-V with CHERI-based protection for small embedded cores [3].

A key design goal for CHERI is to provide radically improved security for those critical existing C codebases with minimal modification. It does so with a dialect of C, implemented initially as modifications to Clang/LLVM, and now also as a GCC port by Arm. The CHERI architectural mechanisms can be used by language implementations and systems software in various ways to provide improved security, but the basic idea for fine-grained memory protection is to implement C pointer types with machine capabilities instead of machine words, so that pointer integrity and memory accesses are checked by the hardware. For simple code, recompilation of the unmodified existing code with the CHERI C compiler will do this, while more exotic code, for example code that manipulates the bit-representations of pointers, may need some source adaptation.

A 2019 analysis [29] suggested that 30–70% of the vulnerabilities reported to the Microsoft Security Response Center (MSRC) would have been deterministically mitigated by CHERI memory-safety, and porting the FreeBSD kernel and userspace to CHERI required changes only to 0.18% and 0.04% LoC respectively. Analysis of an open-source desktop stack [42] estimated a 73.8% vulnerability mitigation rate through a combination of memory protection and software compartmentalisation requiring a 0.026% LoC change.

All this raises the question that we address in this paper: **what is CHERI C, exactly?** This is important from several perspectives: those porting legacy C code to CHERI C, or writing new code, need to know what is permitted; those implementing CHERI C compilers (notably the Clang and GCC extensions) need a common understanding, lest those diverge from each other and from the programmer's model; all these need to understand what is common and what varies across CHERI C implementations above distinct CHERI hardware architectures, so that CHERI C code can be portable across architectures; future semantics and verification for CHERI C needs a basis for its work; and all involved need an understanding of what security properties CHERI C enforces, and what vulnerabilities it mitigates.

We make the following contributions:

- Discussion of the design issues that arise in the design of CHERI C and its semantics, including the subtle interactions between capabilities, undefined behaviour, and pointer provenance, illustrated with a test suite of examples (§3).
- An executable mechanised semantics of CHERI C, reifying the above as an extension of the Cerberus ISO C semantics [27, 28] and the PNVI-ae-udi memory object model supported by the ISO C standards committee [18, 28] (§4).
- The CHERI C memory object model is mechanised within Coq, with the extracted code used in the executable semantics (§4.3).
- A prose definition of CHERI C (published as a separate Technical Report [49]).
- Validation and experimental comparison (§5).

The several different versions of the semantics serve different purposes: the prose version should be widely accessible; the extension of Cerberus gives a semantics that is executable as a test oracle, to compute the allowed behaviour of small and modest-sized tests and programs, and helped us check that we have considered all interactions of CHERI and ISO C features; and the Coq formalisation of that helped nail down all the details and provides a basis for later mechanised proofs (§7).

Without all this, CHERI C would remain merely "defined" by its implementations, leaving many important aspects unclear, and with no solid basis for future discussion.

We begin with background on CHERI hardware capabilities, C undefined behaviour, and pointer provenance (§2) and conclude with discussion of related and future work (§6,7). Our semantics and examples will be available open-source.

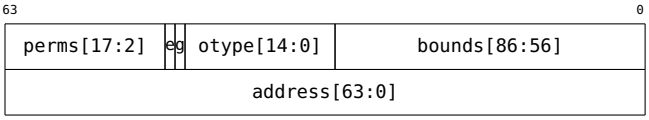

Figure 1. Bit-field layout of Morello capability

2 Background

2.1 CHERI Architecture

CHERI-enabled architectures add new hardware support for *capabilities* in registers and in memory. Capabilities have larger bit-width than architecture addresses, and have an additional out-of-band tag bit, which is not independently addressable. A capability includes an address together with bounds, permissions, and other metadata. The details vary between architectures, and CHERI C has to accommodate this variation, but for concreteness we briefly describe the Morello version (CHERI 64-bit Arm-A) [4, 5]. Here capabilities are 128+1 bits. The lower 64 bits in most cases represent a virtual address, while the upper 64 bits encode bounds, permissions, and other metadata.

A sophisticated compression scheme allows a capability to include 64-bit lower and upper bounds, encoded into 87 bits in total, with 56 of those shared with the address field [5, §2.5.1],[47]. Small regions can be described precisely, with an arbitrary size in bytes, while for larger regions, only certain combinations of bounds and size are representable (though all addresses are representable for *some* base and size). The one-bit tag provides integrity protection: it is preserved only by legitimate operations on capabilities and cleared by any others (e.g. by overwriting individual bytes). A capability can only be used as such, e.g. for a dereference, if its tag is set. The permission bits control whether a capability can be used for loading or storing non-capability data, loading or storing capabilities, and fetching instructions, among other things. Capabilities can also be *sealed*, making them immutable and unusable for anything but branching to them; this allows controlled transitions between different security domains. Sealing (or unsealing) a capability requires an authority capability with the `Seal` (or `Unseal`) permission. Some variations of this are indexed by an *object type* `otype`. *Global* `g` and *executive* `e` bits restrict the locations where a capability can be stored and the banking of certain system registers.

Morello extends the Armv8-A general-purpose integer register file, and some control and status registers, from 64 bits to 128+1 bits. Memory is extended with a tag bit for each 128-bit sized and aligned unit of DRAM. The Program Counter (PC) is extended to become a *Program-Counter Capability* (PCC), constraining instruction fetch as well as PC-relative loads (e.g., of global variables). A new *Default Data Capability* (DDC) register controls memory accesses by legacy (non-capability) instructions, for legacy code using integer pointers. Morello extends Armv8-A with new instructions and modifies existing instructions to use and respect capabilities.

CHERI architectures introduce new detected errors, e.g. when an instruction tries to access memory outside the bounds of the capability used for the access, or with an untagged capability; the exact handling of these is ISA-specific. For example, in Morello such an access triggers a *synchronous data abort exception*. In other cases, e.g. when an instruction attempts to construct a non-representable capability, hardware will clear the tag of the resulting capability, to protect integrity.

2.2 C Undefined Behaviour

ISO C relies crucially on the notion of *undefined behaviour* (UB), to make it possible to define the semantics of a memory-unsafe language (in which the possible effects of a wild write are hard to bound without massively over-constraining implementations), and to enable high-performance optimising implementations on diverse platforms. In the ISO C abstract machine, out-of-bounds memory accesses have undefined behaviour, as do signed integer overflows, data races, and many other things. Any program for which there exists an abstract machine execution which has undefined behaviour is deemed to have undefined behaviour as a whole: programmers are required to avoid this, and compiler implementations are free to behave in any way for such programs. Importantly, UB of programs is not a temporal notion: while it is identified in the ISO abstract machine at specific execution points, it is not the case that a correct compiler is guaranteed to behave according to the abstract machine *until* such points, but rather that the compiled whole program can behave arbitrarily. This is forced by the desire to allow optimisations that move code around without a compile-time check or proof that it is UB-free: the compiler is allowed to assume that the preconditions for such optimisations hold, and the programmer is obliged to ensure that they do, otherwise the standard gives no guarantees about the whole-program behaviour.

2.3 C Memory Object Models and Pointer Provenance

In conventional C implementations, pointers are represented at run-time with simple machine-word integers, and the language exposes this representation to programmers (e.g. via pointer/integer casts, representation-byte accesses, and type punning); this expressiveness is important for low-level systems code. However, compile-time optimisations rely on alias analysis, e.g. to determine that two pointers cannot alias and hence accesses via them can be reordered. In particular, compile-time analysis commonly tracks the original allocation, or *provenance*, of pointer values, and two pointer values that can be statically determine to have different provenance are assumed to not alias. This has been discussed in the ISO

WG14 C standard since 2004 [46], and recent work has proposed a provenance semantics "PNVI-ae-udi" that is the basis for an in-progress ISO Technical Specification [18, 28]. In that semantics, the C abstract machine associates a provenance, which is either an allocation unique ID or `empty`, with every pointer value. Normal operations on pointers simply propagate the provenance, and memory accesses via a pointer check that the address is within the bounds of the original allocation identified by the provenance, with undefined behaviour otherwise. Of course, while this C abstract machine uses provenance at runtime, conventional implementations still do not and should not. Instead, this licences various existing compiler optimisations that rely on compile-time provenance analysis, by deeming certain programs UB. In PNVI-ae-udi, pointers carry provenance, but integers do not. If a pointer is cast to an integer (or its representation is otherwise examined), the allocation identified by its provenance is marked *exposed*. On a cast from integer to pointer, if the integer is within the range of some exposed allocation, the appropriate provenance is (in the semantics) attached to the resulting pointer value.

3 CHERI C Semantics Design Questions

CHERI C has been implemented as an adaption of the Clang/LLVM C compiler [23, 36, 45], and, in progress, as an adaption of GCC [6]. The details of these implementations are beyond the scope of this paper, but the basic idea is to represent all C source-language pointers with machine capabilities, instead of machine words. Pointer arithmetic is implemented as arithmetic over these capabilities, and thus the hardware checks that all accesses are within their bounds. For allocations of local variables whose address is taken, the compiler introduces code to construct a capability with the correctly narrowed bounds (derived from the stack-pointer capability), and for globals, thread-local variables, function pointers, and `malloc`'d allocations, the runtime linker and the allocator similarly construct capabilities with the appropriate bounds. The language provides new intrinsics for explicit manipulation of capabilities, e.g. to inspect their fields or further narrow their bounds, but these are not needed for porting straightforward C code. In addition to source-language pointers, CHERI C implementations also use capabilities to represent internal runtime pointers: the program counter, jump addresses, stack pointer, return addresses, and global-offset-table (GOT) machinery.

The design of CHERI C has to reconcile three major and at times conflicting objectives:

1. Existing C programmers should be able to port existing C codebases to CHERI C with little effort.
2. Existing compiler infrastructure and optimisations should require only limited changes, to maintain performance of the generated code and to make the required compiler engineering effort feasible.
3. Memory-safety errors which would lead to exploitable vulnerabilities should be deterministically mitigated wherever possible. In particular, CHERI C aims to provide a substantial level of *spatial safety*, ensuring that "pointers may be used only to access memory within bounds of their associated allocation" [45]. *Temporal safety* is not universally guaranteed by default across all existing CHERI architectures. Some already provide temporal safety guarantees [3] while for others it is a topic of active research [17].

Moreover, CHERI C needs a well-defined and comprehensible semantics, for all involved. To understand what this should be, we need to weigh considerations from the CHERI C implementation design, CHERI C code-porting experience, ISO specification, and advances in C semantics. Ideally we would have a semantics that makes precise the security guarantees that CHERI C provides. Unfortunately, as we shall see in the next subsections, the interactions with C optimisations and undefined behaviour appear to make it impossible to define those with a source-language semantics. Instead, we give what we call the *positive* semantics, to clearly define what programmers can rely on, and what they are obliged to ensure, for well-defined CHERI C code. This is useful in itself, and the discussion and examples also serve to highlight the need for further work on what security guarantees CHERI C provides.

3.1 Out-of-bounds memory access and undefined behaviour

```
1 void f(int *p, int i) {
2   int *q = p + i;
3   *q = 42;
4 }
5 int main(void) {
6   int x=0, y=0;
7   f(&x, 1);
8   return y;
9 }
```

Consider first the buggy C program on the right, which passes the address of local variable `x` into `f`, constructs a "one-past" out-of-bounds pointer at line 2 and writes to it in line 3. In ISO C the pointer construction is legal, but the access is not, and the program has undefined behaviour. Conventional C implementations will not flag this error, either at compile time (it is obviously undecidable to statically detect all such errors, although compilers can warn in some simple cases), or at run time. They will represent pointers with machine-word virtual addresses, and generate code that computes the address of the stack location of `x` plus `sizeof(int)` and write to whatever is there – which in some cases will be a security problem. Sanitisers can detect some (though not all) such errors at run time, by inserting software instrumentation, but at a substantial performance cost. In CHERI C, on the other hand, C source-language pointers are represented with hardware capabilities. The (non-optimised) generated code for `&x` constructs a capability with bounds spanning exactly the footprint of the stack slot used for `x`,

and the address of this slot. At Line 2 the capability `q` has that address plus **`sizeof(int)`**, but unchanged bounds. Then at the Line 3 access, the hardware check that the virtual address footprint of the access is within that of the bounds fails, and a hardware exception and then a signal are raised – the program will fail-stop safely, preventing exploitation of the bug.

This suggests what would be a straightforward and desirable semantics for CHERI C: that any such memory-access UB is instead guaranteed to trap. With some attention to other UB cases, that could provide strong general security guarantees for race-free programs. Unfortunately, the reality is more complex. Standard C-compiler optimisations can eliminate the call to `f` entirely – at `-O2` the current Clang/LLVM-based CHERI C compiler compiles this code to just return zero – and whether they do or not can depend in subtle and hard-to-predict ways on the rest of the code. For example, if `&x` is assigned to a global, then at `-O2` the inlined `f` survives and performs the doomed write (and, in a larger example, who-knows-what after that), while at `-O3` the doomed write is again eliminated.

Another instructive example is `g` on the right. Here, the compiler will assume the absence of UB, reason that the access `a[i]` must be in-bounds, and compile it to `a[0]`, removing the potential capability exception for `a[1]`. It is then hard for a source-language semantics to bound the behaviour of the rest of the program, as it does not correspond to any execution path of the (CHERI) C abstract machine.

```
1 char g(int i) {
2   char a[1];
3   h(a);
4   return a[i];//UB if i!=0
5 }
```

To make this range of implementation behaviour legal, our formal semantics has to retain the ISO C notion of undefined behaviour, leave implementations unconstrained for programs that are UB, and deem the first program to be UB, and likewise for any program that calls `g(1)` (with a terminating `h`). Such a semantics cannot capture the intended security properties that CHERI C aims to provide. CHERI C clearly deterministically mitigates many otherwise-exploitable security flaws, but undefined behaviour and compiler optimisations make it unclear what precise security properties it provides in general. Further work is needed to see whether the effects of those optimisations and undefined behaviours can be bounded more tightly at an acceptable performance cost.

3.2 Out-of-bounds pointer construction and representability

In ISO C, it is undefined behaviour to use pointer arithmetic to construct a pointer value that is either below or more than one byte past the footprint of the object [21, 6.5.6p8]. The one-past case has to be allowed to support the standard C idiom of iterating across an array, but in real-world C it is not uncommon for code to construct pointer values that are below or more than one-past the object [12, 28], e.g. for decreasing loops, or transiently in more complex arithmetic, as on the right. In CHERI architectures, the capability compression schemes [5, 47] cannot express arbitrary combinations of address, size, and bounds, but, to support porting such software to CHERI, they have been designed to allow combinations for which the address is somewhat outside the bounds. Exactly what combinations are representable is a complex property of the encoding scheme, but they have been designed to allow at least some ranges below and above the object. If a capability arithmetic operation would construct a non-representable value, the resulting address will be as expected, but the tag will be cleared and the bounds may have been changed (another possibility explored earlier in CHERI was to have the hardware trap on the attempt to construct such a capability, but that turns out to be less useful).

```
1 int main(void) {
2   int x[2];
3   int *p = &x[0];
4   int *q = p + 100001;
5   q = q - 100000;
6   *q = 1;
7 }
```

For CHERI C, we have to decide whether (a) to follow ISO, with UB for any pointer construction beyond one-past, (b) allow arbitrary virtual address values within the ranges allowed by all CHERI architectures (or some safe approximation thereto), or (c) allow whatever the underlying architecture makes representable. Moreover, for (b) and (c), we would have to decide whether to deem it UB to go outside those, or merely to make the resulting bounds to be unspecified and the resulting tag be unspecified or cleared.

Conventional C compiler optimisations impact this in two ways. First, despite common coding practices, C compilers do sometimes reason from the fact that in ISO C array indices must be in bounds, as in function `g` in the earlier example. It would be hard and probably performance-reducing to remove that from implementations, and problematic to bound the resulting behaviour for programs that trigger it if they were not UB. Second, optimisation can remove transient out-of-bounds construction, e.g. by collapsing `(p+100001)-100000` above to just the ISO-legal `p+1`, so one could not leave that as defined behaviour and deterministically clear the tag in the semantics. Moreover, importing the complexities of the architectural compression schemes into the language pointer arithmetic semantics is unappealing. These lead us to keep the stricter ISO rule also for CHERI C, option (a), even though that leaves code that exploits the architectural guarantee as UB (we would urge compiler developers to not treat that UB aggressively).

Another potential issue is, in the other direction, that a compiler might implement an ISO source-semantics-legal `p+(100001-100000)` as `(p+100001)-100000`, potentially leading to run-time non-representability. Further work is needed

to check or ensure that this does not occur, but it has not been observed in running the large corpus of code ported to CHERI. To summarise: compilers can optimise away, but not introduce, code that creates non-representability.

The architectural limits on representability also mean that in some (relatively uncommon) circumstances allocators need to use additional padding and/or alignment to ensure that the required capability is representable and does not overlap other allocations. This has a small cost in wasted memory usage, but does not impact the semantics.

3.3 Pointer/Integer conversions and `(u)intptr_t`

Systems C code often requires bitwise or integer operations on pointers, e.g. to examine or enforce particular alignments, or to exploit the fact that some pointers are known to be aligned or bounded, to store metadata in low-order or high-order bits. C pointer types only directly support addition and subtraction of a pointer and an integer value, so this requires casting a pointer to an integer type, doing whatever arithmetic is required, and, if using the result as a pointer, casting back.

In ISO C, the types `uintptr_t` and `intptr_t`, when available, are guaranteed to support identity round-trips; in de facto C, round-trips involving limited arithmetic are widely relied on, and older code often uses `(unsigned) long` for the same purposes. In CHERI C, if one only needs the integer result, one should cast to the new `ptraddr_t` and do conventional integer computation. If one ultimately needs a pointer value, casts to normal integer types will lose the tag and other metadata, so this would not work. Instead, in CHERI C `(u)intptr_t` are implemented with capabilities, casts between these and pointer types are no-ops (in both directions), and arithmetic operations on them are implemented with the corresponding capability operations (which have the expected effects on the address part of the capability). This minimises porting effort for such code.

However, this forces us to return to representability, e.g. for `(u)intptr_t` arithmetic that in the abstract machine becomes transiently non-representable and then comes back into representability, as in the example on the right, and in similar examples using bitwise operations instead of `+`/`-`.

```
1  #include <stdint.h>
2  void f(int a, int b) {
3    int x[2];
4    int *p = &x[0];
5    uintptr_t i = (uintptr_t)p;
6    uintptr_t j = i + a;
7    uintptr_t k = j - b;
8    int *q = (int*)k;
9    *q = 1;
10 }
11 int main(void) {
12   f(100001*sizeof(int),
13     100000*sizeof(int));
14 }
```

There are a number of semantic issues and options to consider, balancing usability (giving more code better-defined results), optimisation at `(u)intptr_t` types (with performance and compiler-modification costs), portability (among CHERI architectures that may differ in capability encoding details), and complexity.

(1) The simplest option would be to follow the semantics of pointers, declaring any `(u)intptr_t` arithmetic resulting in values outside one-past the original allocation bounds to be UB – but that would break many common C idioms, both where one eventually casts back to a pointer and uses that for an access, and where one just uses the integer value.

(2) Alternatively, we could allow `(u)intptr_t` arithmetic within some larger region of representability, with UB if one goes outside. That would also invalidate some reasonable idioms, e.g. using `(u)intptr_t` values as indices in a hash table (though in CHERI C one should ideally use `ptraddr_t` there).

(3) Finally, we could allow `(u)intptr_t` arithmetic within some region of representability, but keep defined behaviour and the integer (address-part) value of the result defined if one goes outside.

We choose (3), but have to consider the results of casting back to a pointer, and of inspecting the tag, bounds, and permissions. If hardware capability arithmetic in compiled code goes outside the architecturally representable region, then the tag will be cleared and any access via it will trap. However, in general, optimisation of `(u)intptr_t` arithmetic could either introduce or eliminate a CHERI C abstract-machine construction of a non-architecturally-representable capability, e.g. rewriting `i+(100001-100000)` to `(i+100001)-100000`, or `(i+100001)-100000` to `i+1`. The GCC *section anchor optimisation* with a negative offset could also introduce non-representability.

(a) At one extreme, one might allow any arithmetic transformations on `(u)intptr_t` where the integer (address) value of the transformed expression is the same as that of the original, but that would mean any `(u)intptr_t` expression could result in unspecified tags and bounds, which is not acceptable.

(b) At the other extreme, one could require that optimisation never introduces or eliminates any non-representability, requiring that the hardware execution matches some straightforward abstract-machine capability computation. This is attractively simple to specify and to use, but has some runtime cost and (perhaps more important) compiler-modification cost, to ensure that such optimisations are not done at these types. More data on these costs would be desirable, but for the time being we reject this option.

(c) The intermediate position we choose is to limit optimisations to those that do not introduce new non-representability, but allow them to eliminate excursions into non-representability. We express this precise-but-loose specification in the semantics with a ghost-state bit per capability value, recording whether abstract-machine `(u)intptr_t` arithmetic ever made it non-representable in abstract-machine execution. We permit casts to pointer types of `(u)intptr_t`

capabilities with this bit set, and loads and stores of them (otherwise `memcpy` of such values would become UB), but make it UB to access memory via them.

Then one has to consider the semantics of inspecting the bounds or tag (using intrinsics) and representation bytes (using `unsigned char*` pointers) of such values. We deem all these to give unspecified values (not UB).

Finally, we have to consider what the above "region of representability" should be (for (1) this would be moot, as pointers within or one-past the original allocation are always representable). CHERI capabilities are encoded in architecture-specific sophisticated ways. For CHERI C, we could:

(i) Fix a relatively simple and portable definition expressing some conservative extent supported across different CHERI architectures and values. For 64-bit CHERI architectures, [45, §4.3.5] says pointers are guaranteed representable if within the greater of 1KiB and $\frac{1}{8}$ of the object size below the lower bound, and the greater of 2KiB and $\frac{1}{4}$ of the object size above the upper bound. This is reasonably simple – but not portable to *all* CHERI architectures, in particular, CHERIoT [3], which, while based on 32-bit RISC-V, uses a different capability encoding scheme from 32-bit CHERI-RISC-V and provides byte-granularity bounds for any object up to 511 bytes.

(ii) Alternatively, we can make this implementation-defined, letting implementations choose either the above or the specific underlying architectural notion of representability. This option is attractive because it allows the use of the full range of representable addresses, and is thus "future-proof". The disadvantage is that it makes it difficult to write portable CHERI C code. For the time being we choose this option.

3.4 Pointer/Integer type punning

An additional benefit of keeping the pointer and `(u)intptr_t` representations identical is that it preserves the C possibility of type punning between them via a union, as shown below.

```
1 #include <stdint.h>
2 #include <inttypes.h>
3 #include <assert.h>
4 union ptr {
5     int *ptr;
6     uintptr_t iptr;
7 };
8 int main(void) {
9     int arr[] = {42,43};
10    union ptr x;
11    x.ptr = arr;
12    x.iptr += sizeof(int);
13    assert (*x.ptr == 43);
14 }
```

3.5 Accesses to capability representations

An essential aspect of CHERI architecture design is that capabilities are unforgeable: attempts to manipulate their representations, e.g. writing their bytes directly rather than with a capability instruction, are guaranteed to clear the tag. ISO C permits bytewise access to pointer-containing data, e.g. to support a bytewise `memcpy`, so we have to consider the extent to which this CHERI architectural property should be reflected in the CHERI C semantics.

We want to guarantee that the tag will be cleared in the case when its representation is modified directly. Again, optimisations make this challenging.

In this example, CHERI hardware execution of an unoptimised compilation will clear the tag of `*px` on the byte-write of Line 6, leading to a capability access fault at Line 7, but an optimising compiler may remove the identity byte-write entirely.

```
1 int main(void) {
2   int x = 0;
3   int *px = &x;
4   unsigned char *p
5    = (unsigned char *)&px;
6   p[0] = p[0];
7   *px = 1;
8   return x;
9 }
```

To allow optimisations which preserve the address, but do not necessarily preserve tag clearing, we use the ghost state, similarly to how we did in Section 3.3, to enforce that following any non-capability write to a capability, it is UB to use it for an access. (A more extreme semantics would be to deem any non-capability write to a capability to be UB, but that would prevent one `memzero` or `memcpy`'ing over some struct in a malloc'd region to re-use it, which should be permitted.)

Another example in which optimisations can remove tag clearing is below, in which the `for` loop may be optimised (e.g. by GCC's *tree-loop-distribute-patterns*) to a call to `memcpy (p1,p0,sizeof(int *))`. In CHERI C, `memcpy` must be implemented with capability-sized and aligned accesses where possible, to preserve pointers, so this optimised code would then preserve the capability and its tag.

```
1 int main(void) {
2   int x = 0;
3   int *px0 = &x;
4   int *px1;
5   unsigned char *p0 = (unsigned char *)&px0;
6   unsigned char *p1 = (unsigned char *)&px1;
7   for (int i=0; i<sizeof(int*); i++)
8     p1[i] = p0[i];
9   *px1 = 1;
10  return x;
11 }
```

Our semantics makes such optimisations sound using an additional per-capability-value ghost state bit to mark the capability, after its representation was modified directly, as no longer suitable for memory access, resulting in UB in Line 9.

A `memcpy` of part of a capability must behave semantically like any other non-capability-sized and aligned representation access, using that ghost state bit rather than deterministically clearing the tag (which also makes sound optimisations that combine `memcpy` calls for adjacent memory regions, that at the hardware level could introduce tag preservation).

This approach uses ghost state to record abstract-machine accesses to capability representations, to make subsequent accesses via such capabilities UB, but we also have to ask

to what extent such a capability can be examined – or, in other words, what does the ghost state "cover"? The example below shows some scenarios we need to consider.

```
1 int main(void) {
2   int x = 0;
3   int *px = &x;
4   size_t perms0 = cheri_perms_get(px);
5   unsigned char *p = (unsigned char *)&px;
6   p[0] = p[0];
7   int addr = (int)px;
8   bool tag = cheri_tag_get(px);
9   size_t perms = cheri_perms_get(px);
10  assert(perms == perms0);
11  return (*px);
12 }
```

Trying to access the memory using `px` in Line 11 should certainly be UB. For the other operations, we have to ask:

(1) Whether the pointer-to-integer cast in Line 7, to obtain the capability address, is UB or some *implementation defined* value? For example, in Morello, we know that the lower 64 bits of a capability contain its address and the compiler can reason how modifying the first byte will affect it. This knowledge is specific to a particular ISA and could be used only when targeting it.

(2) Whether the tag access in Line 8 is UB or returns an *unspecified* value?

(3) Whether the permission access in Line 9 is a UB or returns an *unspecified* or *implementation-defined* value, and, if the latter, what is guaranteed about it?

To summarize, our current proposed solution is for the abstract CHERI C machine to record any non-capability write to a capability (via representation pointers or using standard library functions) using ghost state. Using such a manipulated capability to access memory is UB. Comparing it to other capabilities using intrinsic `cheri_is_equal_exact`, or examining its tag via `cheri_tag_get`, will return an unspecified boolean value. This way we avoid declaring such checks to be UB. The effect of direct representation manipulation on other capability fields except the tag is *implementation defined*. That will permit ISA-specific optimisations where the compiler is aware of the capability encoding for a target ISA.

3.6 Pointer equality

There are several possible definitions of pointer equality (==). Ignoring pointer provenance for now, we could take either:

(1) bitwise equality of capability representations, with tags,

(2) the same but without tags, or

(3) equality just of their address fields, without all their capability metadata.

Intuitively the first definition may seem most natural, with equality implying interchangeability, and that was the choice for the early CHERI C implementation. However, pragmatically it seems that porting code is most straightforward with the third option, so that is what we adopt here. In fact, even in ISO C, equality of pointers does not guarantee interchangeability, due to pointer provenance [18], so this is perhaps less of a departure from standard practice than it may seem.

Additionally, CHERI C provides the intrinsic `cheri_is_equal_exact` which compares two capabilities (pointers or `(u)intptr_t`), comparing all fields, including meta-information such as a tag or permissions. If some of their fields, such as tag or bounds, are marked as unspecified in ghost state, its return value is unspecified as well.

3.7 Capability derivation in binary arithmetic

For binary arithmetic operations on two values of capability-carrying types, CHERI C has to define how the bounds and tag of the result are derived. Ignoring integer overflow and representability for the moment, in CHERI C the resulting capabilities `c0` and `c1` are derived from their left arguments. (This makes `(u)intptr_t` addition *non-commutative* with respect to inter-substitutability and to representation equality, while it remains commutative with respect to == equality.)

```
1 #include <stdint.h>
2 int main(void) {
3   int x=0, y=0;
4   intptr_t a=(intptr_t)&x;
5   intptr_t b=(intptr_t)&y;
6   intptr_t c0 = a + b;
7   intptr_t c1 = b + a;
8 }
```

The next example demonstrates a more interesting case of capability derivation. Here we implement array indexing via `intptr_t` arithmetic. In ISO C, the semantics of the addition will depend on the *integer conversion rank* of the `size_t` and `intptr_t` types. If `intptr_t` has a higher rank, then the first argument will be cast to `intptr_t` and then the addition of two `intptr_t` values will be performed. Otherwise, the second argument will be cast to `size_t`, the addition performed on `size_t` values, and the result will be cast back to `intptr_t` before assigning it to `ip1`.

```
1 #include <stdint.h>
2 int* array_shift(int *x, int n) {
3   intptr_t ip = (intptr_t)x;
4   intptr_t ip1 = sizeof(int)*n + ip;
5   int *p = (int*)ip1;
6   return p;
7 }
```

Using the latter strategy in CHERI C would result in `ip1` being derived from the *null capability*, and hence untagged. This means that converted back to a pointer, `p` would be non-dereferenceable. To avoid that, CHERI C semantics requires that no other *standard integer type* shall have a higher *integer conversion rank* than `intptr_t` and `uintptr_t`. Additionally, for binary operations, the capability derivation picks as a source for the resulting capability the argument which was

not a result of implicit or explicit conversion from a non-capability type.

3.8 Sub-object bounds

In C one routinely constructs pointers to a subobject of a data structure: to a member of a struct or an element of an array. For CHERI C, it is tempting to have the compiler automatically narrow the bounds of the corresponding capability to just that member or element, to implement the principle of least privilege, but it is also common in C to use pointer arithmetic and `offsetof` to move the resulting pointer to a different subobject, in array indexing and the "container-of" idiom.

Based on experience in porting code, the current default behaviour of CHERI C is to not enforce subobject bounds. The CHERI C semantics follows suit, though Clang/LLVM CHERI C provides options for stricter bounds enforcement, and the semantics should be revisited following further use.

3.9 Pointers to *const*-qualified types and permissions

In ISO C, objects created at const-qualified types are expected to be immutable, so it would be natural for a capability pointing to a const object to not have write permission, and CHERI C does this. Note that in ISO C it is allowed to cast a pointer to a non-const type to a pointer to the corresponding const type and later cast it back and modify the object; to allow this, in CHERI C those casts are no-ops on the underlying capability (in CHERI ISAs, clearing permission would be irreversible).

3.10 Abstracting capabilities across architectures

The main existing CHERI C implementations support Arm Morello (CHERI Arm-A) and CHERI RISC-V, both 64-bit, and there is another supporting the CHERIoT extension to RISC-V RV32E, the small RISC-V specification intended for embedded devices [3]. One should be able to write portable CHERI C programs across these architectures, or (sometimes) across just the first two. That means that the C semantics needs a common abstraction of hardware capabilities.

One part of this is the *abstract address* type denoted by the new `ptraddr_t` C type, an integer type with implementation-defined width and signedness. The list of permissions encoded in capability can vary between architectures, but there is a common basic set which is always present. The *object type* field width and values could vary. Finally, the *seal type* is also architecture-dependent. Abstracting these types and their properties allows us to talk in the CHERI C language semantics about portable capabilities.

All existing CHERI architectures use a capability encoding scheme to compress capability address and bounds, in 128 or 64 bits for 64- or 32-bit architectures, and there is a trade-off between compression and the set of representable addresses, as some combinations of fields may become non-representable. To abstract from this, we make several design choices. First, we restrict the abstract scope of compression to four capability fields: address, flags, and upper and lower bounds. Other fields, such as permissions, are always represented exactly. This allows us to describe most of CHERI C semantics in terms of abstract capabilities, making it portable across current and future implementations. In cases where architecture-specific details matter, the corresponding parts of CHERI C semantics are clearly designated as *implementation defined.*

3.11 Capabilities and provenance

As recalled in §2.3, the in-progress ISO definition of provenance tracks provenance in the C abstract machine, to define undefined-behaviour cases that are important to legitimise current compiler optimisations based on their static analysis of provenance – but provenance data is not carried at runtime in conventional implementations. Meanwhile, CHERI implementations carry capabilities at runtime, and the CHERI ISA specification [44] also speaks of "provenance". How do these relate to each other?

The ISO C + PNVI-ae-udi semantics rules call for several checks based on provenance:

(1) Checking whether a pointer is inside the bounds of the corresponding memory allocation's footprint.

(2) Checking whether two pointers possess the same provenance when subtracted or compared.

(3) Checking whether a pointer refers to a live allocation.

We believe (though have not formally proved) that the first check is redundant in the presence of capabilities. Provenance tracking enables the static overestimation of capability runtime bounds. Capability bounds are initially set to align with the allocation's footprint and can be narrowed down but not extended, thus resulting in an "overestimation".

The second check presents a challenge. It might be assumed that pointers with matching or at least overlapping bounds share the same provenance. However, this is not valid in two scenarios: 1) when bounds have been narrowed through intrinsic calls, resulting in non-intersecting regions, and 2) in the absence of a capability revocation mechanism, provenance is temporally unique, while capabilities are not. Consequently, one could have a pointer to a heap object that has been killed and another pointer to a newly allocated object at the same address.

The third check is impossible in general in the absence of some sort of capability revocation mechanism.[1]

In conclusion, the capability checks at runtime could not subsume provenance checks at compile time. The two are complementary.

[1]This is an area of ongoing research. For example, see [17]

4 CHERI C executable semantics

We codified CHERI C semantics, fleshing out many details associated with the high-level design choices described in the previous section, as an extension of Cerberus [27, 28], a well-validated semantic model for a substantial fragment of ISO C. The resulting CHERI C semantics is executable and permits running small C test programs to investigate semantic questions.

Cerberus is expressed as a translation, from C into a small Core language, combined with a *memory object model*. Most CHERI C-related changes relate to the latter. We defined the CHERI C memory object model in Coq and extracted it to OCaml to integrate into Cerberus. Previous Cerberus memory object models have been in OCaml; this Coq definition should support future proof about CHERI. The complete Coq definition could be found in Cerberus git repository at https://github.com/rems-project/cerberus; we explain the main features in non-mechanised mathematics here.

4.1 Abstract capabilities

We defined abstract capabilities as a Coq *module type* which defines an opaque capability type and operations on it. We chose Arm Morello [5] for the implementation-defined aspects, giving a concrete executable implementation of CHERI capabilities. We used the existing ISA model for Morello [8] mechanically extracted from the Arm ASL reference, from which we extracted low-level Coq implementations of the relevant functions using Sail's support for multiple backends [7].

4.2 Undefined behaviours

CHERI C adds the following new *undefined behaviours*:

`UB_CHERI_InvalidCap` is flagged when attempting to dereference a pointer with the capability tag cleared.

`UB_CHERI_UndefinedTag` is flagged on attempt to dereference a pointer with the capability tag marked as unspecified in the ghost state.

`UB_CHERI_InsufficientPermissions` is flagged on an attempt to perform memory access (e.g. read or write) via a capability which does not have the permission bit set for the given operation.

`UB_CHERI_BoundsViolation` is flagged on an attempt to dereference an *out of bounds pointer*.

The ISO C `UB012_lvalue_read_trap_representation` is flagged when an attempt to decode a stored representation of a capability object fails.

4.3 CHERI C memory object model, in Coq

The Cerberus memory object model encapsulates all memory-related logic, providing a clean abstract interface to the rest of the semantics. Key data types such as the *memory state*, and *pointer* and *integer* values are opaque in the module interface. All memory model operations accessing the memory state are implemented in a `memM` monad, which maintains the state and facilitates error handling.

The standard Cerberus memory model interface provides functions for dynamic memory management (allocation, release, `memcpy` and `memcmp`); reading and writing memory values; pointer arithmetic, comparison, and alignment; and pointer/integer conversion. Since memory values are abstract outside the module, it also provides functions for relational and arithmetic operations on integer, floating point, and pointer memory values. These operations do not depend on the access to the memory state and are not in the `memM` monad.

Our CHERI C memory object model fits nicely into this Cerberus memory model abstraction; its interface had to be extended only to add capability derivation (§4.4) and intrinsics type derivation (§4.5).

The memory state (see below) is a tuple (A, S, M) with information about *allocations* A, PNVI-ae-udi related data S, and the concrete representation of memory contents M. The CHERI memory model state is similar to the Cerberus *concrete* memory model, with the M component extended as follows. As before the memory content is stored in an integer-address-indexed dictionary B. Each byte consists of provenance (π), an optional 8-bit numeric value, and an optional integer index. Additionally, for each capability-size aligned memory location, we add metadata consisting of the capability tag and a two-bit *ghost state*, stored in the new C dictionary. The first bit of the ghost state for a given capability indicates whether the tag is unspecified, and the second bit indicates whether the address and bounds are unspecified.

$$
\begin{aligned}
\text{mem\_state} &\triangleq A \times S \times M \\
M &\triangleq B \times C \\
B &\triangleq \mathbb{Z} \rightharpoonup \text{AbsByte} \\
C &\triangleq \mathbb{Z} \rightharpoonup \mathbb{B} \times \text{ghost\_state} \\
\text{ghost\_state} &\triangleq \mathbb{B} \times \mathbb{B} \\
\text{AbsByte} &\triangleq \pi \times (\text{option byte}) \times (\text{option } \mathbb{N}) \\
\text{integer\_value} &\triangleq \mathbb{Z} \oplus (\mathbb{B} \times \text{Cap})
\end{aligned}
$$

Pointer values are capabilities, and tag, bounds, and permission checks are performed when they are used to access memory. When written to memory, a capability representation excluding the tag is written to B, and the tag is stored in C. Writing non-capabilities to memory marks all previously set tags for the corresponding address range as *unspecified* in the ghost state in C.

Integer values could be either pure numeric values for integer types, or capabilities (with signedness flag) for `(u)intptr_t` types. This representation allows us to preserve all capability fields when casting pointers to `(u)intptr_t` and back.

The memory monad used in Coq is a combination of *state* and *error* monads. The memory state is completely internal

to the memory model implementation. All monadic calls to the memory interface cross the OCaml to Coq language boundary only in one direction.

To give a flavour of the new checks require for CHERI C, we give a formal semantics rule for the *load* operation below, using similar notation to [27]. This corresponds to the function `load` in the full Coq definition (module `CheriMemory`), which involves considerably more detail. CHERI-specific changes are highlighted in blue and marked with †.

The auxiliary bounds-checking predicate now takes a capability instead of just an address$^{(1^\dagger_a)}$. The capability fields include the address a, the tag, permissions set, bounds $(base, limit)$, and the ghost state bits. The test succeeds if we have *read* permission$^{(1^\dagger_b)}$, the tag is known$^{(1^\dagger_c)}$ and set$^{(1^\dagger_d)}$, and the address is within the bounds$^{(1^\dagger_e)}$.

$$
\begin{array}{l}
\text{bounds\_check}_{\text{load}}(c, n, i, A) \triangleq \\
\quad {}^{(1^\dagger_a)}\, c = (a, tag, perm, (base, limit), (g_{tag}, g_{bounds}) \ \ldots) \wedge \\
\quad {}^{(1^\dagger_b)}\, \texttt{loadPerm} \in perm \wedge \\
\quad {}^{(1^\dagger_c)}\, g_{tag} = \texttt{false} \quad {}^{(1^\dagger_d)}\, tag = \texttt{true} \wedge \\
\quad {}^{(1^\dagger_e)}\, base \leq a \wedge a + n \leq limit \wedge \\
\quad {}^{(1_f)}\, A(i) = (n_i, \_, a_i, \texttt{alive}, \_, \_, \_) \wedge \\
\quad {}^{(1_g)}\, [a \mathinner{..} a + n - 1] \subseteq [a_i \mathinner{..} a_i + n_i - 1]
\end{array}
$$

The memory-object-model `load` operation takes a pointer$^{(2_a)}$, containing provenance information and a capability, and returns a *memory value* and a *footprint annotation*. The capability must not be a *null capability*$^{(2^\dagger_b)}$ and must pass the bounds check$^{(2_c)}$. The abstraction function$^{(2^\dagger_f)}$ must successfully interpret memory bytes$^{(2_d)}$ b and associated with them capability metadata$^{(2^\dagger_e)}$ m as a C value v of type τ. Finally, $^{(2_g)}$ prevents reading from uninitialized memory, which would result in unspecified values.

The helper function `expose`, which remains unchanged from PNVI-ae-udi, accepts the abstract state A and a set of tainted allocations I as input. It designates the specified allocation i as exposed if it was previously included in the set of allocations and marked as alive.

$$
\frac{
\begin{array}{ll}
[\textsc{label} : \mathbf{load}(\tau, p) = (v, fp)] & \\
{}^{(2_a)}\, p = (@i, c) & {}^{(2_c)}\, \text{bounds\_check}_{\text{load}}(c, \text{sizeof}(\tau), i, A) \\
c = (a, \ \ldots) & {}^{(2^\dagger_e)}\, m = C[a \mathinner{..} a + \text{sizeof}(\tau) - 1] \\
{}^{(2^\dagger_b)}\, \neg \text{cap\_is\_null}(c) & {}^{(2_d)}\, b = B[a \mathinner{..} a + \text{sizeof}(\tau) - 1] \\
{}^{(2_g)}\, v \neq \texttt{Unspecified} & {}^{(2^\dagger_f)}\, \texttt{Some}\ (v, I_{tainted}, S', []) = \text{abst}(A, S, \tau, b, m) \\
fp = \mathbf{R}(a, \text{sizeof}(\tau)) & A' = \begin{cases} \text{expose}(A, I_{tainted}) & \text{is\_integer}(\tau) \\ A & \text{otherwise} \end{cases}
\end{array}
}{(A, S, (B, C)) \rightarrow (A', S', (B, C))}
$$

4.4 Capability Derivation

For unary and binary operations on integer values involving at least one capability-carrying type, CHERI C needs to choose which will be used to derive the resulting capability. We made this derivation step explicit by elaborating it in the intermediate Core language.

4.5 Intrinsics

Many of the CHERI C intrinsics are polymorphic in the capability type they accept, and their return type may depend on it. This does not fit the standard C type system and to implement this in Cerberus we extended it with a special type derivation mechanism, implemented via an embedded DSL.

5 Validation and Experimental Comparison

We validate that our design decisions of §3 are appropriate for CHERI C by discussion with designers and implementers of the existing CHERI C implementations, and with developers who have ported large bodies of code to CHERI C; these discussions identified a number of previously unconsidered issues discussed there.

We validate experimentally that our executable mechanised semantics has the intended behaviour, and that this and the behaviour of the Clang/LLVM and GCC implementations are consistent. We developed a test suite of 94 tests exercising and demonstrating various aspects of CHERI C semantics, especially where they may be unclear or differ from ISO C. Table 1 summarizes the semantic categories along with the number of tests that cover each category.

We compiled and ran all our tests using three CHERI C implementations and compared the results. We found that existing implementations are mostly compatible with this standard, with some minor bugs but no principal disagreements. Our assessment of their compliance with CHERI C, as defined by this document, is summarised below. The complete results of our testing are available at https://www.cl.cam.ac.uk/~vz231/asplos24/test-results/.

The output from a single test is presented in Appendix A. This test evaluates how both signed and unsigned integer types manage bitwise operations with `intptr_t`. With Clang, non-representability issues arise for `cap&int` and `cap&uint` as the operation clears the upper bits of the address, leading to a value below the lower bound. In contrast, GCC does not exhibit this issue, likely because of its memory allocator's address ranges. Cerberus demonstrates non-representability in the ghost state for `cap&int`, where the value falls beneath the lower bound.

Several interesting aspects of the semantics are related to potential compiler optimisations, and writing tests to exercise those well is (as usual) a challenging problem, which we leave for future work. Our focus here is on exercising the main semantic choices.

Tests	Description
10	Checking capability alignment in the memory.
10	Memory allocator interface (locals, globals, and heap).
2	Capabilities produced by taking addresses of arrays and their elements.
3	Operations offseting pointers as in taking an address of array element at an index.
2	Assigning constants and values of capability-carrying types to capability-typed variables.
1	Issues related to calling convention: passing arguments, variable argument functions, etc.
5	Implicit/explicit casts between capability-carrying types.
5	C `const` modifier and its effects on capabilities.
10	Equality between capability-carrying types.
11	Pointers to functions.
6	Pointers to global vs. local variables.
4	Initialization of variables carrying capabilities.
19	Properties and definition of `(u)intptr_t` types.
9	Arithmetic operations on `(u)intptr_t` values.
3	Bitwise operations on `(u)intptr_t` values.
16	Semantics of CHERI C intrinsic functions (e.g, permission manipulation).
15	Unforgeability enforcement for capabilities.
6	Capabilities encoding for Arm Morello architecture.
6	*null pointers* and `NULL` constant as capabilities.
1	ISO-legal pointers *one-past* an object's footprint and their bounds.
5	Out-of-bounds memory-access handling.
10	Effects of compiler optimisations.
5	Capability permissions: setting and enforcement.
7	*pointer provenance* tracking per [18].
2	New `ptraddr_t` type definition and usage.
2	Implementation of pointer arithmetic on capabilities.
9	Conversion between pointer and integer types.
4	Relational comparison operators (e.g. <,>,<= and >=) for capabilities.
6	Issues related to potential non-representability of some combinations of capability fields.
9	Tests related to accessing capabilities in-memory representation.
5	Accessing memory via capabilities after the region has been deallocated.
5	Handling of (un)signed integer types in casts, accessing capability fields, and intrinsics.
6	Standard C library functions handling of capabilities.
3	Sub-objects bound enforcement via capabilities.

Table 1. Summary of the tests for which we compared the results on three CHERI C implementations.

5.1 Cerberus

This is our reference implementation of semantics and it passes all our tests with the results we expect, modulo one known bug relating to `const` behaviour. A further known shortcoming is that not all intrinsic functions were implemented.

5.2 Clang/LLVM

This was the first CHERI C compiler and is the most mature. It supports three CHERI backends: Morello (CHERI Arm-A), CHERI-RISC-V, and CHERI-MIPS. It has proven to be quite robust and used to port and compile CheriBSD and other software such as KDE. The CHERI C language was developed and refined using this compiler as a testbed to try various aspects of CHERI C semantics. The compiler supports several modes of sub-object bounds enforcement, but we only tested the "conservative" setting as it is the one closest to our semantic definition. We compiled for Morello and CHERI-RISC-V and tested compiled binaries under CheriBSD running under CHERI-QEMU.

Not surprisingly, we found it to be mostly compliant with our CHERI C semantics definition. Our test suite independently identified two known issues that had been previously reported and acknowledged by the compiler team. It also rediscovered an upstream bug present in the LLVM version CHERI LLVM is currently based on but already fixed in later versions. Additionally, our suite detected one spurious warning message and two bugs in the `realloc` function of the CheriBSD *jemalloc* library. Some warning messages emitted by the compiler use terminology that is different from the terminology used in this paper (e.g., "capability provenance" vs "capability derivation"), but they are not otherwise incorrect.

Under CHERI-RISC-V (version 8), some test results do not match our semantics. These failures are caused by: 1) an exception when attempting to change the bounds of an untagged capability, and 2) an exception when attempting to modify the sentry capability. However, the current draft of version 9 of the ISA specification [43] changes these behaviours to changing bounds and clearing the tag, respectively. With these changes, it should be compatible with our semantics.

5.3 GCC

CHERI GCC is a relatively new arrival. There have been two public releases, and we have seen significant progress in CHERI C support between them. We run compiled "bare metal" binaries under CHERI-QEMU. Our test suite identified five issues in the latest public release of the compiler and runtime, all of which were reported to the developers. One was confirmed as a bug in the compiler. Two issues related to a memory allocator were deferred to a different project (*newlib*, the libc implementation used in this baremetal environment. The two remaining issues have not yet been confirmed at the time of writing.

5.4 CHERIoT

CHERIoT utilises an LLVM-based compiler, specifically targeting embedded systems. As a result of its focus on embedded systems, executing our test suite would require extensive tooling and modifications to the tests, which we have

not done. Nevertheless, based on discussion with CHERIoT designers and their review of our semantics specification, our CHERI C semantics are applicable to CHERIoT. It is important to note that CHERIoT provides additional temporal guarantees and defines certain aspects that we regard as undefined behaviour.

6 Related work

Memory safety has been a pivotal area of research for several decades [13]. While there exists a vast body of work on this topic, in this section, we focus on the work most closely related to ours. These can be broadly categorized into four groups below, with examples for the last three being non-exhaustive.

6.1 CHERI C semantics and program analysis

The most closely related work is Park et al. [34], which presents a formalized CHERI-C memory object model in Isabelle/HOL [33]. That memory model, which is based on the CompCert block/offset model [24], is more abstract than ours, which utilises a finite flat address space that aligns more closely with hardware architectures. They do not address non-representability, pointer-to-integer conversion, or potential optimisations eliminating capability invalidation from direct byte manipulations. Handling of `(u)intptr_t` types and capability derivation in arithmetic operations are also absent. They provide proofs of essential properties (which we do not), and their model is combined with the Gillian program analysis framework [25], with a front-end based on Clang and ESBMC [9] to support execution of concrete examples.

Braußе et al. present a bounded model checker for CHERI-C programs [9]. This is a viable approach to find some potential code safety violations in CHERI C programs, but does not provide a complete formal semantics for the CHERI C language, or even for the CHERI C memory object model.

6.2 Architecture-level memory protection

This category comprises systems like Hardbound [14] and Softbound [30] which have similar goals to CHERI, using additional metadata associated with pointers to provide memory protection. Comparisons between these systems and CHERI are available in the existing CHERI literature. Specifically, one can refer to the ISCA 2014 paper by Woodruff et al. [48] and "Historical Context and Related Work" section of [44]. They tend not to consider the source-level C semantics beyond issues that arise during implementation. Providing such semantics presents similar challenges to CHERI C and while the details would differ considerably we expect that our approach would be equally applicable.

Softbound illustrates an interesting example of this: the protection checks are added after the main compiler optimisation passes, and so their safety proof does not apply to the original source program. In work on santizers, Isemann et al [20] demonstrate that this is a real problem because (for example) bad memory accesses can be optimised away and so the checks are never performed. This is exactly the difficulty that our semantics anticipates and allows for in Section 3.1.

6.3 C dialects with added memory safety

Without the hardware support of the CHERI ISA, these projects use a combination of compile-time and runtime checks. They employ static code analysis and often depend on type annotations.

There are several dialects of C, such as Checked-C [37] and CCured [31], that aim to provide memory safety. Broadly speaking, they are further away in their semantics and type system from ISO C than CHERI C is, and require additional type annotations and source code changes to ensure memory safety. In some cases, safety guarantees only apply to parts of the code (*checked regions* in Checked-C), and mixing checked and unchecked pointers is allowed. CCured relies on whole-program analysis, using different pointer representations for various pointer types. This poses some difficulties with separate compilation and the use of third-party libraries. At the same time, they provide a path for incremental migration to a type-safe language which does not require hardware support.

Castro et al. [10] employ static code analysis to enforce *data flow integrity*, preserving the standard ISO C semantics without modifications. However, this approach has notable drawbacks: it results in substantial memory and runtime overhead, requires major alterations to existing compilers, and demands extra efforts to instrument both the standard library and third-party binary code.

The Deputy [11] project augments the C language type system with dependent types. While it relies on type annotations provided by the programmer, in some cases types can be inferred. It was implemented through several compiler passes, including type inference, flow-insensitive type checking and instrumentation, and check optimisation. Beyond the added complexity of implementing these passes, dynamic assertions are generated for type constraints involving dynamic values, leading to additional runtime overhead. Porting existing code necessitates the addition of type annotations, and in some instances, requires the code to be rewritten to mark it as trusted.

Dynamic binary instrumentation frameworks, such as *Valgrind* [32], and tools based on them, like *memcheck* [40], are useful for debugging and testing to find memory safety problems. However, due to significant performance overhead and deployment complexity, they are not typically suitable for production use.

Backward-compatible bounds checking techniques, such as those in [2, 15, 16, 22, 38, 39], modify the compiler and utilise a runtime library to track pointers' bounds information in a separate data structure at runtime. Limitations

of these approaches, including interactions with uninstrumented libraries, the lack of support for integer-to-pointer casts ([15]), substitution of array bounds checks with coarser pool bounds checks ([2, 15]), and reliance on the undecidable *flow-insensitive points-to analysis* [35], do not ensure the complete elimination of classes of memory problems, a guarantee that CHERI C promises to offer. Lastly, the non-negligible performance costs render them unsuitable for production use.

6.4 Memory-safe languages

The final category includes memory-safe languages like Rust [26]. The main appeal of CHERI C and other memory-safe dialects of the C language lies in their ability to port existing legacy C code without the need for substantial rewriting. With this in mind, we do not compare CHERI C to other such languages here.

7 Conclusion

Our mechanised semantics for CHERI C should provide clarity of what is (and is not) guaranteed by the language, helping to avoid any divergence between implementations and promote portability of CHERI C code; it has already clarified a number of the issues we describe. It moreover enables a wide range of potential future work.

The discussion here of the interaction between CHERI hardware architectural guarantees and C compiler optimisations and undefined behaviour makes clear that further work is needed to understand what precise security properties CHERI C implementations could reasonably provide.

The fact that our semantics is executable means that it could be used as a test oracle for more aggressive compiler testing, letting one use randomly generated tests without manually curating their intended results.

The fact that the memory object model is mechanised in a theorem prover (Coq) makes it potentially usable for proof about the language, e.g. to make precise properties such as *provenance validity* and *capability integrity* that are informally described in the CHERI architecture specification [45].

The semantics would provide a solid basis for program analysis or model-checking of CHERI C.

Finally, this work can provide a basis for extensions to CHERI temporal safety [17] and subobject bounds [36].

Acknowledgements. This work was supported by the UK Industrial Strategy Challenge Fund (ISCF) under the Digital Security by Design (DSbD) Programme, to deliver a DSbDtech enabled digital platform (grant 105694). This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No 789108, ERC AdG ELVER). Distribution Statement A: Approved for public release; distribution is unlimited. This work was supported by the Defense Advanced Research Projects Agency (DARPA) and the Air Force Research Laboratory (AFRL), under contracts HR0011-22-C-0110 ("ETC") and HR0011-23-C-0031 ("MTSS"). The views, opinions, and/or findings contained in this report are those of the authors and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government.

A Sample test suite output

```c
#include <stdint.h>
#include <stdio.h>
#include <limits.h>
#include "capprint.h"

int main(void) {
    int x[2]={42,43};
    intptr_t ip = (intptr_t)&x;
    fprintf(stderr,"cap_%" PTR_FMT "\n", sptr((void*)ip));
    intptr_t ip2 = ip & UINT_MAX;
    fprintf(stderr,"cap&uint_%" PTR_FMT "\n", sptr((void*)ip2));
    intptr_t ip3 = ip & INT_MAX;
    fprintf(stderr,"cap&int_%" PTR_FMT "\n", sptr((void*)ip3));
}
```

```
cerberus-cheri-coq:
cap (@86, 0xffffe6dc [rwRW,0xffffe6dc-0xffffe6e4])
cap&uint (@86, 0xffffe6dc [rwRW,0xffffe6dc-0xffffe6e4])
cap&int (@empty, 0x7fffe6dc [?-?] (notag))

clang-riscv-O3-bounds-subobject-safe:
cap 0x3fffdfff08 [rwRW,0x3fffdfff08-0x3fffdfff10]
cap&uint 0xffdfff08 [rwRW,0xffdfff08-0xffdfff10] (invalid)
cap&int 0x7fdfff08 [rwRW,0x7fdfff08-0x7fdfff10] (invalid)

clang-riscv-O3-bounds-conservative:
cap 0x3fffdffef8 [rwRW,0x3fffdffef8-0x3fffdfff00]
cap&uint 0xffdffef8 [rwRW,0xffdffef8-0xffdfff00] (invalid)
cap&int 0x7fdffef8 [rwRW,0x7fdffef8-0x7fdfff00] (invalid)

clang-riscv-O0-bounds-conservative,clang-riscv-O0-bounds-references-only,
clang-riscv-O0-bounds-subobject-safe,clang-riscv-O0-bounds-aggressive,
clang-riscv-O0-bounds-very-aggressive,
clang-riscv-O0-bounds-everywhere-unsafe:
cap 0x3fffdfff78 [rwRW,0x3fffdfff78-0x3fffdfff80]
cap&uint 0xffdfff78 [rwRW,0xffdfff78-0xffdfff80] (invalid)
cap&int 0x7fdfff78 [rwRW,0x7fdfff78-0x7fdfff80] (invalid)

clang-morello-O3-bounds-subobject-safe:
cap 0xffffffff7ff08 [rwRW,0xffffffff7ff08-0xffffffff7ff10]
cap&uint 0xfff7ff08 [rwRW,0xfff7ff08-0xfff7ff10] (invalid)
cap&int 0x7ff7ff08 [rwRW,0x7ff7ff08-0x7ff7ff10] (invalid)

clang-morello-O3-bounds-conservative:
cap 0xffffffff7ff28 [rwRW,0xffffffff7ff28-0xffffffff7ff30]
cap&uint 0xfff7ff28 [rwRW,0xfff7ff28-0xfff7ff30] (invalid)
cap&int 0x7ff7ff28 [rwRW,0x7ff7ff28-0x7ff7ff30] (invalid)

clang-morello-O0-bounds-conservative,clang-morello-O0-bounds-references-only,
clang-morello-O0-bounds-subobject-safe,  clang-morello-O0-bounds-aggressive,
clang-morello-O0-bounds-very-aggressive,
clang-morello-O0-bounds-everywhere-unsafe:
cap 0xffffffff7ff68 [rwRW,0xffffffff7ff68-0xffffffff7ff70]
cap&uint 0xfff7ff68 [rwRW,0xfff7ff68-0xfff7ff70] (invalid)
cap&int 0x7ff7ff68 [rwRW,0x7ff7ff68-0x7ff7ff70] (invalid)

gcc-morello-O3:
cap 0x7fffffc8 [rwRW,0x7fffffc8-0x7fffffd0]
cap&uint 0x7fffffc8 [rwRW,0x7fffffc8-0x7fffffd0]
cap&int 0x7fffffc8 [rwRW,0x7fffffc8-0x7fffffd0]

gcc-morello-O0:
cap 0x7fffff88 [rwRW,0x7fffff88-0x7fffff90]
cap&uint 0x7fffff88 [rwRW,0x7fffff88-0x7fffff90]
cap&int 0x7fffff88 [rwRW,0x7fffff88-0x7fffff90]
```

References

[1] CHERI x86-64 Sail model. https://github.com/CTSRD-CHERI/sail-cheri-x86. Accessed 2023-04-17.

[2] Periklis Akritidis, Manuel Costa, Miguel Castro, and Steven Hand. Baggy bounds checking: An efficient and backwards-compatible defense against out-of-bounds errors. In *USENIX Security Symposium*, volume 10, page 96, 2009.

[3] Saar Amar, Tony Chen, David Chisnall, Felix Domke, Nathaniel Filardo, Kunyan Liu, Robert Norton-Wright, Yucong Tao, Robert N. M. Watson, and Hongyan Xia. CHERIoT: Rethinking security for low-cost embedded systems. Technical Report MSR-TR-2023-6, Microsoft, February 2023. URL: https://www.microsoft.com/en-us/research/publication/cheriot-rethinking-security-for-low-cost-embedded-systems/.

[4] Arm. Arm Morello Program. https://developer.arm.com/architectures/cpu-architecture/a-profile/morello, 2022. Accessed 2021-06-29.

[5] Arm Ltd. Arm® architecture reference manual supplement Morello for A-profile architecture. https://developer.arm.com/documentation/ddi0606/latest, June 2021. DDI0606A.j. 1288pp. Accessed 2022-06-15.

[6] Arm Ltd. Arm Morello program, landing page for Morello open source software. https://www.morello-project.org/, November 2022.

[7] Alasdair Armstrong, Thomas Bauereiss, Brian Campbell, Alastair Reid, Kathryn E. Gray, Robert M. Norton, Prashanth Mundkur, Mark Wassell, Jon French, Christopher Pulte, Shaked Flur, Ian Stark, Neel Krishnaswami, and Peter Sewell. ISA semantics for ARMv8-A, RISC-V, and CHERI-MIPS. *Proc. ACM Program. Lang.*, 3(POPL):71:1–71:31, 2019. doi:10.1145/3290384.

[8] Thomas Bauereiss, Brian Campbell, Thomas Sewell, Alasdair Armstrong, Lawrence Esswood, Ian Stark, Graeme Barnes, Robert N. M. Watson, and Peter Sewell. Verified security for the Morello capability-enhanced prototype arm architecture. In Ilya Sergey, editor, *Programming Languages and Systems - 31st European Symposium on Programming, ESOP 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings*, volume 13240 of *Lecture Notes in Computer Science*, pages 174–203. Springer, 2022. http://www.cl.cam.ac.uk/~pes20/morello-proofs-esop2022.pdf. doi:10.1007/978-3-030-99336-8_7.

[9] Franz Brauße, Fedor Shmarov, Rafael Menezes, Mikhail R. Gadelha, Konstantin Korovin, Giles Reger, and Lucas C. Cordeiro. ESBMC-CHERI: Towards verification of C programs for CHERI platforms with ESBMC. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2022, page 773–776, New York, NY, USA, 2022. Association for Computing Machinery. doi:10.1145/3533767.3543289.

[10] Miguel Castro, Manuel Costa, and Tim Harris. Securing software by enforcing data-flow integrity. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, OSDI '06, page 147–160, USA, 2006. USENIX Association.

[11] Jeremy Condit, Matthew Harren, Zachary Anderson, David Gay, and George C. Necula. Dependent types for low-level programming. In *Proceedings of the 16th European Symposium on Programming*, ESOP'07, page 520–535, Berlin, Heidelberg, 2007. Springer-Verlag.

[12] Brooks Davis, Robert N. M. Watson, Alexander Richardson, Peter G. Neumann, Simon W. Moore, John Baldwin, David Chisnall, Jessica Clarke, Nathaniel Wesley Filardo, Khilan Gudka, Alexandre Joannou, Ben Laurie, A. Theodore Markettos, J. Edward Maste, Alfredo Mazzinghi, Edward Tomasz Napierala, Robert M. Norton, Michael Roe, Peter Sewell, Stacey Son, and Jonathan Woodruff. CheriABI: Enforcing Valid Pointer Provenance and Minimizing Pointer Privilege in the POSIX C Run-time Environment. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, pages 379–393. ACM, 2019. URL: https://www.cl.cam.ac.uk/research/security/ctsrd/pdfs/201904-asplos-cheriabi.pdf, doi:10.1145/3297858.3304042.

[13] Peter J. Denning. Virtual memory. *ACM Comput. Surv.*, 2(3):153–189, sep 1970. doi:10.1145/356571.356573.

[14] Joe Devietti, Colin Blundell, Milo M. K. Martin, and Steve Zdancewic. Hardbound: Architectural support for spatial safety of the C programming language. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XIII, page 103–114, New York, NY, USA, 2008. Association for Computing Machinery. doi:10.1145/1346281.1346295.

[15] Dinakar Dhurjati, Sumant Kowshik, and Vikram Adve. Safecode: Enforcing alias analysis for weakly typed languages. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '06, page 144–157, New York, NY, USA, 2006. Association for Computing Machinery. doi:10.1145/1133981.1133999.

[16] B. Ding, Y. He, Y. Wu, A. Miller, and J. Criswell. Baggy bounds with accurate checking. In *2012 IEEE 23rd International Symposium on Software Reliability Engineering Workshops (ISSREW)*, pages 195–200, Los Alamitos, CA, USA, nov 2012. IEEE Computer Society. URL: https://doi.ieeecomputersociety.org/10.1109/ISSREW.2012.24, doi:10.1109/ISSREW.2012.24.

[17] Wesley Nathaniel Filardo, Brett F. Gutstein, Jonathan Woodruff, Sam Ainsworth, Lucian Paul-Trifu, Brooks Davis, Hongyan Xia, Edward Tomasz Napierala, Alexander Richardson, John Baldwin, David Chisnall, Jessica Clarke, Khilan Gudka, Alexandre Joannou, A. Theodore Markettos, Alfredo Mazzinghi, Robert M. Norton, Michael Roe, Peter Sewell, Stacey Son, Timothy M. Jones, Simon W. Moore, Peter G. Neumann, and Robert N. M. Watson. Cornucopia: Temporal Safety for CHERI Heaps. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 608–625, 2020. doi:10.1109/SP40000.2020.00098.

[18] Jens Gustedt, Peter Sewell, Kayvan Memarian, Victor BF Gomes, and Martin Uecker. A Provenance-aware Memory Object Model for C, 2022. Working draft ISO Technical Specification TS6010.

[19] Ben Hawkes. 0day in the wild. 2019. Project Zero team blog, Google. https://googleprojectzero.blogspot.com/p/0day.html. Accessed 2023-04-19.

[20] Raphael Isemann, Cristiano Giuffrida, Herbert Bos, Erik van der Kouwe, and Klaus von Gleissenthall. Don't look UB: Exposing sanitizer-eliding compiler optimizations. *Proc. ACM Program. Lang.*, 7(PLDI), jun 2023. doi:10.1145/3591257.

[21] ISO WG14. *Programming languages – C*, ISO/IEC 9899:2018 edition, July 2018.

[22] Richard WM Jones and Paul HJ Kelly. Backwards-compatible bounds checking for arrays and pointers in C programs. In *AADEBUG*, volume 97, pages 13–26, 1997.

[23] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar 2004.

[24] Xavier Leroy, Andrew W Appel, Sandrine Blazy, and Gordon Stewart. *The CompCert memory model, version 2.* PhD thesis, Inria, 2012.

[25] Petar Maksimovic, Sacha-Élie Ayoun, José Fragoso Santos, and Philippa Gardner. Gillian, part II: real-world verification for JavaScript and C. In Alexandra Silva and K. Rustan M. Leino, editors, *Proceedings of the 33rd Computer Aided Verification International Conference, CAV 2021, Virtual Event, July 20-23, 2021, Part II*, volume 12760 of *Lecture Notes in Computer Science*, pages 827–850. Springer, 2021. doi:10.1007/978-3-030-81688-9_38.

[26] Nicholas D. Matsakis and Felix S. Klock. The Rust language. In *Proceedings of the 2014 ACM SIGAda Annual Conference on High Integrity Language Technology*, HILT '14, page 103–104, New York, NY, USA, 2014. Association for Computing Machinery. doi:10.1145/2663171.2663188.

[27] Kayvan Memarian. The Cerberus C semantics. Technical Report UCAM-CL-TR-981, University of Cambridge, Computer Laboratory,

May 2023. URL: https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-981.pdf, doi:10.48456/tr-981.

[28] Kayvan Memarian, Victor B. F. Gomes, Brooks Davis, Stephen Kell, Alexander Richardson, Robert N. M. Watson, and Peter Sewell. Exploring C Semantics and Pointer Provenance. *Proc. ACM Program. Lang.*, 3(POPL), January 2019. doi:10.1145/3290380.

[29] Matt Miller. Trends, challenge, and shifts in software vulnerability mitigation. https://github.com/Microsoft/MSRC-Security-Research/tree/master/presentations/2019_02_BlueHatIL, February 2019. Microsoft Security Response Center.

[30] Santosh Nagarakatte, Jianzhou Zhao, Milo M.K. Martin, and Steve Zdancewic. Softbound: Highly compatible and complete spatial memory safety for c. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, page 245–258, New York, NY, USA, 2009. Association for Computing Machinery. doi:10.1145/1542476.1542504.

[31] George C. Necula, Scott McPeak, and Westley Weimer. CCured: type-safe retrofitting of legacy code. In John Launchbury and John C. Mitchell, editors, *Conference Record of POPL 2002: The 29th SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Portland, OR, USA, January 16-18, 2002*, pages 128–139. ACM, 2002. doi:10.1145/503272.503286.

[32] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. *ACM Sigplan notices*, 42(6):89–100, 2007.

[33] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. Isabelle/HOL: A proof assistant for higher-order logic. In *TPHOLs*, pages 1–18. Springer, 2002.

[34] Seung Hoon Park, Rekha Pai, and Tom Melham. A formal CHERI-C semantics for verification, 2023. Accepted to appear in TACAS 2023. https://arxiv.org/abs/2211.07511. arXiv:2211.07511.

[35] Thomas Reps. Undecidability of context-sensitive data-dependence analysis. *ACM Trans. Program. Lang. Syst.*, 22(1):162–186, jan 2000. doi:10.1145/345099.345137.

[36] Alexander Richardson. Complete spatial safety for C and C++ using CHERI capabilities. Technical Report UCAM-CL-TR-949, University of Cambridge, Computer Laboratory, June 2020. URL: https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-949.pdf, doi:10.48456/tr-949.

[37] Andrew Ruef, Leonidas Lampropoulos, Ian Sweet, David Tarditi, and Michael Hicks. Achieving safety incrementally with Checked C. In Flemming Nielson and David Sands, editors, *Principles of Security and Trust - 8th International Conference, POST 2019, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2019, Prague, Czech Republic, April 6-11, 2019, Proceedings*, volume 11426 of *Lecture Notes in Computer Science*, pages 76–98. Springer, 2019. doi:10.1007/978-3-030-17138-4_4.

[38] Olatunji Ruwase and Monica S Lam. A practical dynamic buffer overflow detector. In *NDSS*, volume 2004, pages 159–169, 2004.

[39] Konstantin Serebryany and Timur Iskhodzhanov. AddressSanitizer: A fast address sanity checker. In *Presented as part of the 2012 USENIX Annual Technical Conference (ATC 12)*, pages 309–318. USENIX, 2012.

[40] Julian Seward and Nicholas Nethercote. Using Valgrind to detect undefined value errors with Bit-Precision. In *2005 USENIX Annual Technical Conference (USENIX ATC 05)*, Anaheim, CA, April 2005. USENIX Association. URL: https://www.usenix.org/conference/2005-usenix-annual-technical-conference/using-valgrind-detect-undefined-value-errors-bit.

[41] UKRI. Digital security by design. https://www.dsbd.tech/ and https://www.ukri.org/our-work/our-main-funds/industrial-strategy-challenge-fund/artificial-intelligence-and-data-economy/digital-security-by-design-challenge/, 2022. Accessed 2021-06-29.

[42] Robert N. M. Watson, Ben Laurie, and Alexander Richardson. Assessing the Viability of an Open- Source CHERI Desktop Software Ecosystem. http://www.capabilitieslimited.co.uk/pdfs/20210917-capltd-cheri-desktop-report-version1-FINAL.pdf, September 2021.

[43] Robert N. M. Watson, Peter G. Neumann, Jonathan Woodruff, Michael Roe, Hesham Almatary, Jonathan Anderson, John Baldwin, Graeme Barnes, David Chisnall, Jessica Clarke, Brooks Davis, Lee Eisen, Nathaniel Wesley Filardo, Richard Grisenthwaite, Alexandre Joannou, Ben Laurie, A. Theodore Markettos, Simon W. Moore, Steven J. Murdoch, Kyndylan Nienhuis, Robert Norton, Alexander Richardson, Peter Rugg, Peter Sewell, Stacey Son, and Hongyan Xia. Capability Hardware Enhanced RISC Instructions: CHERI Instruction-Set Architecture (Version 9 - DRAFT). Accessed 2023-04-12. URL: https://github.com/CTSRD-CHERI/cheri-specification.

[44] Robert N. M. Watson, Peter G. Neumann, Jonathan Woodruff, Michael Roe, Hesham Almatary, Jonathan Anderson, John Baldwin, Graeme Barnes, David Chisnall, Jessica Clarke, Brooks Davis, Lee Eisen, Nathaniel Wesley Filardo, Richard Grisenthwaite, Alexandre Joannou, Ben Laurie, A. Theodore Markettos, Simon W. Moore, Steven J. Murdoch, Kyndylan Nienhuis, Robert Norton, Alexander Richardson, Peter Rugg, Peter Sewell, Stacey Son, and Hongyan Xia. Capability Hardware Enhanced RISC Instructions: CHERI Instruction-Set Architecture (Version 8). Technical Report UCAM-CL-TR-951, University of Cambridge, Computer Laboratory, October 2020. URL: https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-951.pdf, doi:10.48456/tr-951.

[45] Robert N. M. Watson, Alexander Richardson, Brooks Davis, John Baldwin, David Chisnall, Jessica Clarke, Nathaniel Filardo, Simon W. Moore, Edward Napierala, Peter Sewell, and Peter G. Neumann. CHERI C/C++ Programming Guide. Technical Report UCAM-CL-TR-947, University of Cambridge, Computer Laboratory, June 2020. URL: https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-947.pdf, doi:10.48456/tr-947.

[46] WG14. Defect report 260, September 2004. http://www.open-std.org/jtc1/sc22/wg14/www/docs/dr_260.htm.

[47] Jonathan Woodruff, Alexandre Joannou, Hongyan Xia, Anthony Fox, Robert Norton, Thomas Baureiss, David Chisnall, Brooks Davis, Khilan Gudka, Nathaniel Wesley Filardo, A. Theodore Markettos, Michael Roe, Peter G. Neumann, Robert N. M. Watson, and Simon W. Moore. CHERI Concentrate: Practical Compressed Capabilities. *IEEE Transactions on Computers*, 68(10):1455–1469, October 2019. URL: https://www.cl.cam.ac.uk/research/security/ctsrd/pdfs/2019tc-cheri-concentrate.pdf, doi:10.1109/TC.2019.2914037.

[48] Jonathan Woodruff, Robert N. M. Watson, David Chisnall, Simon W. Moore, Jonathan Anderson, Brooks Davis, Ben Laurie, Peter G. Neumann, Robert Norton, and Michael Roe. The CHERI capability model: Revisiting RISC in an age of risk. In *Proc. ISCA*, 2014.

[49] Vadim Zaliva, Kayvan Memarian, Ricardo Almeida, Jessica Clarke, Brooks Davis, Alex Richardson, David Chisnall, Brian Campbell, Ian Stark, Robert N. M. Watson, and Peter Sewell. CHERI C semantics as an extension of the ISO C17 standard. Technical Report UCAM-CL-TR-988, University of Cambridge, Computer Laboratory, 15 JJ Thomson Avenue, Cambridge CB3 0FD, United Kingdom, phone +44 1223 763500. URL: https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-988.html.