

Beyond the PDP-11: Architectural support for a memory-safe C abstract machine

David Chisnall Colin Rothwell
Robert N. M. Watson
Jonathan Woodruff Munraj Vadera
Simon W. Moore Michael Roe

University of Cambridge
firstname.lastname@cl.cam.ac.uk

Brooks Davis Peter G. Neumann
SRI International
{brooks,neumann}@csl.sri.com

Abstract

We propose a new memory-safe interpretation of the C abstract machine that provides stronger protection to benefit security and debugging. Despite ambiguities in the specification intended to provide implementation flexibility, contemporary implementations of C have converged on a memory model similar to the PDP-11, the original target for C. This model lacks support for *memory safety* despite well-documented impacts on security and reliability.

Attempts to change this model are often hampered by assumptions embedded in a large body of existing C code, dating back to the memory model exposed by the original C compiler for the PDP-11. Our experience with attempting to implement a memory-safe variant of C on the CHERI experimental microprocessor led us to identify a number of problematic idioms. We describe these as well as their interaction with existing memory safety schemes and the assumptions that they make beyond the requirements of the C specification. Finally, we refine the CHERI ISA and abstract model for C, by combining elements of the CHERI capability model and fat pointers, and present a softcore CPU that implements a C abstract machine that can run legacy C code with strong memory protection guarantees.

1. Introduction

A programming language defines both a concrete syntax and an abstract machine. The abstract machine describes the environment that a programmer should expect, such as the

range of sizes, behaviors, and representations of primitive types. A fully specified abstract machine for a language encapsulates all assumptions that are safe for a programmer to make about *every* possible implementation of the language.

Concrete implementations may perform native compilation to machine code, provide a virtual machine with a complete environment, or choose an intermediate step with complex parts of the system implemented as operating-system or library routines. The last approach is the most common for the C language, with memory allocation and certain complex floating-point operations implemented in shared libraries, but most language constructs compiled directly to short sequences of machine instructions. A key aspect of mapping the abstract model onto a target platform is the *memory model*, which defines (among other things) how abstract ideas of pointers and objects (ranges of memory with an associated type) are represented on the target—for example, as words and ranges of bytes in a flat address space.

The C abstract machine was originally designed to be sufficiently close to the PDP-11 that a simple compiler could achieve good performance translating C statements into short sequences of PDP-11 instructions. The abstract machine was subsequently extended to allow very different implementations. Early targets included the Honeywell 6000, IBM System/370, and Interdata 8/32 [25]. In spite of this generalization, most contemporary computers designed to support C programs utilize instruction sets that are reminiscent of the PDP-11, particularly in terms of memory model. The PDP-11 model is appealing due to its simplicity and its portability; however, lack of memory-safety properties leaves code compiled to that model subject to memory corruption in the face of bugs. More concerningly, these bugs frequently prove exploitable by attackers with broad scope for malicious data and control-flow modification [29]; for example, CWE/SANS lists “Classic Buffer Overflow”, possible due to lack of memory safety, as the third most dangerous software error [21].

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

ASPLOS '15, March 14–18, 2015, Istanbul, Turkey.

Copyright is held by the owner/author(s). Publication rights licensed to ACM.

ACM 978-1-4503-2835-7/15/03...\$15.00.

<http://dx.doi.org/10.1145/2694344.2694367>

Bugs with respect to the abstract language model may be vulnerabilities, but the degree to which they are exploitable by attackers depends on concrete implementation choices such as stack layout [2]. Refinements to language implementations (such as *memory protection*) therefore offer opportunities to mitigate vulnerabilities as well as make debugging easier [26]. The potential benefits of bounds checking and other integrity techniques have been well documented by work such as Cyclone [17], CCured [23], SoftBound [22], HardBound [12], and CHERI [35]. Deployment of these techniques is hampered not just by performance concerns but also by whether current software relies on specific implementation choices. Evaluation cases have frequently been limited to benchmarks and application software that might be termed “extremely well-behaved C” rather than the low-level software that would most benefit from their support (e.g., buffer-centered packet parsing).

Understanding and addressing the practical effects of stronger memory models is critical in deploying them to mitigate vulnerabilities. We therefore propose a new interpretation of the C abstract machine that affords stronger memory-protection properties. We evaluate this approach against existing C-language corpora and a practical implementation via our open-source Capability Hardware Enhanced RISC Instructions (CHERI) softcore processor.

We first investigate how commonly held assumptions about pointer behavior go beyond the rules mandated by the C abstract machine, at times incorporating implicit assumptions regarding the PDP-11 memory model. We do this by surveying a variety of pointer-centered C idioms using an LLVM-based static analyzer over a large open-source application corpus. This allows us to identify a tradeoff space between protection and compatibility. Next, we link these ideas to portions of the C abstract machine—as described in the C11 specification [16]—that are implementation defined, investigating the possible implementation space.

We discuss the problems with supporting these idioms on existing models, including our CHERI ISA (a RISC instruction-set architecture that implements fine-grained memory protection). We propose a refinement to the CHERI ISA, *CHERIV3* [34], informed by state-of-the-art hardware-assisted fat-pointer schemes, to improve software compatibility. We evaluate performance and compatibility with respect to both hardware and software by implementing two capability-based memory models and exploring the impact (including lines-of-code changes) on a set of C-language applications, relative to the PDP-11 memory model.

We demonstrate that it is possible for a memory-safe implementation of C to support not just the C abstract machine as specified, but a broader interpretation that is still compatible with existing code. By enforcing the model in hardware, our implementation provides memory safety that can be used to provide high-level security properties for C TCBS, just as

memory safety is a building block for higher-level security abstractions in Java.

2. Common idioms

The CHERI ISA [33] introduces memory safety to a RISC ISA by supplementing general-purpose registers with *capability registers* and *tagged memory*, drawn from the capability-system model, providing strong integrity protection and bounds checking for pointers. In attempting to implement a C compiler for early versions of the CHERI ISA, we found that it was very easy to support well-behaved C code. On attempting to compile nontrivial C programs, we discovered a number of programs failed to either compile or run correctly, due to different interpretations of the C abstract machine. We collected examples of these failures and produced the following taxonomy of common C idioms that are difficult for memory-safe implementations to support.

To explore the extent to which existing programs depend on a traditional view of memory, we modified the Clang compiler to identify potentially problematic pointer operations, as well as code that removes the `const` qualifier. LLVM intermediate representation (IR) is typed, with pointers and integers being distinct. Type-safe pointer arithmetic is performed by the `GetElementPtr` instruction. The PDP-11 model allows arbitrary arithmetic to construct valid addresses in a flat address space, but is incompatible with memory safety that requires a substrate to be aware of objects and enforce bounds checking. We can detect assumptions of the PDP-11 model (and violations of the type system), as LLVM must generate `ptrtoint` and `inttoptr` instruction pairs that turn a pointer into an integer, perform arithmetic, and then convert it back.

Our modified LLVM identified all instances of pointer arithmetic that survive optimization and performed some simple categorization. We ignore those that do not survive optimization because they will have no effect on run-time enforcement. We compiled a sample corpus of around 2M lines of popular C code with this modified compiler, categorized violations by inspecting the code, and extracted test cases demonstrating the common patterns.

Table 1 shows a summary of the total number of occurrences of each of the idioms that we have identified. Some of these are entirely valid within the C memory model, some depend on implementation-defined behavior, and some depend on undefined behavior that happens to be implemented in a specific way on common compilers.

Note that these values are a result of machine-assisted human categorization, and are intended to be indicative of the frequency of these idioms rather than accurate measures. Breaking these idioms will break existing code, and our analysis gives a rough idea of the scope of such breakage.

Unlike prior work, such as STACK [32], we are not looking purely for cases where programs rely on *undefined behavior* and therefore work only coincidentally. We are also

PROGRAM	DECONST	CONTAINER	SUB	II	INT	IA	MASK	WIDE	LOC
ffmpeg	150	0	800	4	0	0	4	0	693,010
libX11	117	0	19	9	1	0	0	5	120,386
FreeBSD libc	288	0	216	2	13	50	184	17	136,717
bash	43	0	207	11	0	0	15	4	109,250
libpng	20	0	175	1	0	0	0	0	50,071
tcpdump	579	0	9	1299	0	0	0	0	66,555
perf	575	151	46	0	53	151	31	4	52,033
pmc	2	0	0	0	18	0	0	0	8,886
pcre	98	0	52	0	0	0	0	0	70,447
python	494	0	358	1	109	0	131	8	383,813
wget	55	0	61	0	3	0	1	10	91,710
zlib	4	0	24	0	0	0	0	0	21,090
zsh	29	0	267	0	0	0	5	5	98,664
TOTAL	2491	151	2236	1557	197	201	371	53	1,902,632

Table 1. Summary of difficult idioms in popular C packages.

looking for cases where programs rely on *implementation-defined behavior* related to a specific understanding of the machine’s memory model, and thus would be fragile in the presence of different interpretations. When a compiler encounters undefined behavior, it is free to do whatever it wishes. For example, if the value of `a` is the result of undefined behavior, then it is acceptable for the compiler to optimize `a == b` and `a != b` to the same value. In contrast, implementation-defined behavior must be self-consistent. For example, the value of `sizeof(int)` is implementation defined and may be 4, 8, or some other value, but must be consistent within a program.

We assume that all of the code that we inspected works correctly, that all of the idioms that we find are intentional, and that the code depends on them working as expected. We identified the following idioms:

Deconst refers to programs that remove the `const` qualifier from a pointer. This will break with any implementation that enforces the `const` at run time. §6.7.3.4 [16] states:

If an attempt is made to modify an object defined with a const-qualified type through use of an lvalue with non-const-qualified type, the behavior is undefined.

This means that such removal is permitted unless the object identified by the pointer is declared `const`, but this guarantee is very hard to make statically and the removal can violate programmer intent.

We would like to be able to make a `const` pointer a guarantee that nothing that receives the pointer may write to the resulting memory. This allows `const` pointers to be passed across security-domain boundaries.

Container describes behavior in a macro common in the Linux, BSD, and Windows kernels that, given a pointer to a structure member, returns a pointer to the enclosing

structure [20]. This may or may not be permitted behavior according to the standard, due to the ambiguous definition of ‘object’. Use of this idiom would break with any implementation that associates strict bounds checking with a pointer based on its static type, but not one that employs the bounds of the original object.

This is a special case of pointer subtraction, but its use breaks assumptions that we would like to make for pointer bounds: A compiler can statically insert bounds information on a pointer to a structure field, which can be enforced by the underlying substrate—preventing some categories of pointer error with a fine granularity.

Sub refers to any arbitrary pointer subtraction. High-level languages commonly lack pointer subtraction, preferring a model where pointers are always references to objects (a base and a bounds), and accesses to object fields or array elements require that the programmer have a pointer to the object. With pointer subtraction, bounds checking requires an offset as well as the base and bounds and so bounds-checked pointers are larger.

II refers to computation of *invalid intermediate* results. The C specification allows pointers to point one element after the end of an array, but not be dereferenced when pointing outside of a valid object. This case refers to pointer arithmetic where the end result is within the bounds of an object, but intermediate results are not. This is undefined behavior according to the C specification and makes even conservative garbage collection impossible unless a valid pointer to the object is guaranteed to also exist.

Without this being expected to work, we could trap as soon as a pointer went out of range, rather than waiting until it is dereferenced. This would be useful mostly as a

debugging aid, as there are no extra security or reliability guarantees from preventing these operations.

Int refers to storing a pointer in an integer variable in memory—implementation-defined behavior in C. We ignore variables where the store-load sequence is optimized away in calculating these. Disallowing this behavior makes accurate garbage collection possible, as the compiler can statically track every pointer use. It also eliminates a category of error where a value is manipulated as an integer and later interpreted as a pointer.

IA refers to performing integer arithmetic on pointers—such as storing a pointer in an integer value and then performing arbitrary arithmetic on it. This is a more general case of the **Int** idiom and relies on the same implementation-defined behavior. It also makes conservative garbage collection impossible, as the garbage collector can assume that every integer may be a pointer and still do a reasonable job, but cannot do any collection if pointers can be easily hidden.

Mask refers to simple masking of pointers. For example, to store some other data in the low bits. This relies on pointers having a known representation. Breaking this allows more efficient pointer representations, for example the “low-fat pointers” [18] representation for fat pointers.

Wide refers to storing a pointer in an integer variable of a smaller size. This is undefined according to the C specification, but may work if you are able to guarantee that pointers are within a certain range, for example by allocating memory with `malloc` and the `MAP_32BIT` flag.

Code using this idiom is broken by existing implementations, and most likely reflects bugs in the code. We were surprised to see examples of this in programs that we inspected, but fortunately it is sufficiently rare that fixing all of the cases would be easy in these codebases. This idiom is the result of assuming that `sizeof(int) == sizeof(void*)` or `sizeof(int32_t) == sizeof(void*)`. This assumption was true for desktop computers for a long time, and mobile devices until recently. Had we run the same experiment 15 years ago, we expect we would have seen many more instances of this assumption. We conclude that C codebases adapt (over time) to changes in pointer behavior, and that additional small changes are not impossible to support.

Last Word refers to accessing an object as aligned words without regard for the fact that the object’s extent may not include all of the last word. This is used as an optimization for `strlen()` in FreeBSD `libc`. While this is undefined behavior in C, it works in systems with page-based memory protection mechanisms, but not in CHERI where objects have byte granularity. We have found this idiom only in FreeBSD’s `libc`, as reported by valgrind.

The relatively large number of instances of pointer subtraction in C code that we observed (Table 1) supports our anecdotal observation that the lack of this support in the original C implementation for CHERI (described in detail in the next section) would be problematic. We have not been able to find the Last Word pattern by static analysis and thus have not included it. Note that most of the cases of invalid intermediates also involve subtraction; we have predominantly classified instances as subtraction if the pointers are dereferenced immediately after the subtraction, indicating either that the code is buggy or the pointer is valid.

Arbitrary arithmetic on integers that are then cast to pointers is rare. The majority of the occurrences that we have seen are in the `malloc()` implementation in FreeBSD `libc` (`jemalloc` [13]). This is difficult to avoid, as `malloc()` is outside of the C abstract machine. The C specification indicates that each block of memory returned by `malloc()` is an object and that it is undefined behavior to use it after calling `free()`. This means that, with a strict interpretation of the specification, it is impossible for the code inside `free()` to recycle the memory. Similarly, the memory that has not yet been returned by `malloc()` is not yet part of the C abstract machine. In real implementations, the compiler makes sufficient allowances to permit these functions to be implemented in C atop some more primitive functionality (`mmap()` or `brk()`), which deals with pages of memory.

We investigate `tcpdump` more closely in §5.2, where we discuss porting it to two CHERI variants. It is worth noting that numerous cases of invalid intermediates involve bounds-checking code: These are not required at all if the underlying implementation supports bounds checking. We therefore observe that, for at least some code, modifications to allow compatibility with a restricted memory-safe implementation of C would simplify the code.

3. The C abstract machine

Having identified a set of idioms that an implementation must support, we now look at the requirements imposed by the language. It is possible to implement the C abstract machine for any Turing-complete target, but that does not mean it is easy or efficient to do so. When considering a low-level language like C, there are three important requirements on a compilation target:

Expressiveness: It must be sufficiently expressive to capture the semantics of the abstract machine.

Efficiency: The primitive operations in the abstract machine must map simply to primitive operations in the underlying system, allowing low-overhead implementation.

Exposure: The features of the underlying system must be exposed to the programmer. For example, the language should expose the register types of the underlying architecture as primitive types.

A language can be described as *low level* with regard to a particular platform only if it meets these requirements. It must be possible to compile the language to run on the target system, but also to do so without introducing significant abstraction layers that hinder performance or hide the details of the underlying architecture.

This section discusses the flexibility that the C abstract machine allows implementers, both with respect to the primitive types used for memory access (integers and pointers), and the layout and semantics of memory itself. We take particular note of changes that can be made to enforce memory safety without compromising the low-level nature of C. The C abstract machine does not include an idea of address translation—intentionally, as not all targets have MMUs—so, from the perspective of userspace C programs running on an operating system, we always mean virtual memory when we refer to memory as described by the abstract machine.

3.1 Primitive types

The C specification is intentionally vague on the representation of most primitive types, allowing significant variety.

3.1.1 Ranges and representations

Before we can consider modifying how pointers are implemented in C, we should examine the requirements that the specification places on the implementations of primitive types. §5.2.4.2.1 [16] is devoted to the ranges of integer types. It defines a minimum range that each can represent. Implementations may support larger ranges, with the exception of the `char` type. C11 requires it to be at least 8 bits, but POSIX requires that it is exactly 8 bits, providing a concrete practical limit for implementations wishing to run code designed for any UNIX-like system.

Within this size, the only constraint is that a signed `char` must be able to represent integers between -127 and 127. Most modern machines opt to use two's complement arithmetic, as it simplifies many operations. Two's complement representation allows values from -128 to 127 to be represented. This leaves one value that can be represented, but which is outside of the range required by the specification.

Most arithmetic operations can overflow. The §6.2.5 [16] defines that overflow of unsigned values should wrap. Signed overflow is undefined and may produce a *trap representation*: A value that may trap if used and whose use makes any subsequent behavior undefined.

These two facts lead to a potential implementation that allows cheap trapping on overflow in hardware. All signed arithmetic that overflowed would be set to this trap value, and all signed arithmetic operations that took this value and an input would raise an exception. This would necessitate different instructions for signed and unsigned addition, but there is some precedent for this. MIPS, for example, includes operations that differ only by whether they trap on overflow.

Trapping on signed overflow is not a new idea. Both clang and gcc provide a `-ftrapv` flag that causes every signed

arithmetic operation to be followed by a branch-on-carry with a trap instruction (`ud2` on x86) at the destination. The clang variant also permits a handler function to recover from the overflow. This was used to implement a prototype of the As-if Infinitely Ranged (AIR) proposal by CERT [11]. Similar work was done later by MIT's KINT [31] system, defining an integer equivalent of the not-a-number (NaN) values found in floating point systems. These efforts have all been purely focused on the compiler, and not on extending the underlying architecture.

3.1.2 Pointers

In BCPL [24], an ancestor of C, pointers and integers were both words: pointers were words that could be dereferenced. In contrast, C was intended to support minicomputer and mainframe architectures, including segmented memory models and microcontrollers with separate integer and address registers. The PDP-11 model of C follows closely from BCPL: pointers are integers, any pointer can be cast to any sufficiently large integer type, and results of casting can be used as pointers. Pointer arithmetic is just integer arithmetic.

This behavior is not mandated by the C standard. The abstract machine divides memory into objects: regions of memory with an associated type. Pointer arithmetic that ends outside of the original object is undefined, with the exception that pointers may point one element past the end of arrays, but such a pointer is valid only for comparison, not for dereferencing. §7.20.1.4 [16] defines an optional `intptr_t` as an integer type that can store a pointer and have the pointer value preserved. There is no guarantee that any arithmetic on `intptr_t` will result in a valid pointer and dereferencing any such pointer is implementation defined. Pointer comparisons between pointers to different objects are undefined, allowing the implementation to move objects as long as it does so atomically with respect to the running code.

3.1.3 The null pointer

As per §6.3.2.3 [16], the integer value 0 has a special meaning when cast to a pointer: It must be distinct from any valid object in the system. This value is relevant (according to the specification) only when it is an *integer constant expression* that evaluates to 0. This distinction is important. For example, the specification does not require that the following be a null pointer:

```
int i = 0; void *p = (void*)i;
```

This distinction is difficult to support in modern compilers, as discussed in a recent LLVM mailing list thread [1].

3.2 State of the unions

Footnote 95 in §6.5.2.3 [16] contains the exception to the normal aliasing rules:

If the member used to read the contents of a union object is not the same as the member last used to store a value in the object, the appropriate part of the object representation of

the value is reinterpreted as an object representation in the new type...This might be a trap representation.

This requirement is one of the things that makes C useful in low-level contexts: It is possible to subvert the type system and interpret memory as different forms. Supporting unions can be difficult in environments with guarantees about type safety but is important for our *exposure requirement*.

3.3 Code and data memory

C is intended to be usable on microcontrollers with separate address spaces for code and data. A numerical pointer value may be different when interpreted as referring to data and code. In particular, a `void*` may refer to any kind of data, but is not guaranteed to be able to represent function pointers. POSIX breaks this separation by introducing the `void*dlsym(...)` function, used to look up a symbol in a shared library. This is beyond the scope of the C language specification, which has no notion of shared libraries, but is very important, e.g., for finding plugin interfaces. Unfortunately, looking up function pointers is a common use for `dlsym`, and is not defined behavior in C.

3.4 Const enforcement

The `const` qualifier indicates that a pointer should not be used to write memory. In typical C implementations, the immutability of `const`-qualified objects varies. Literal strings are mapped read-only into the running process. Attempting to modify one will cause a segmentation fault. It is therefore safe to remove a `const` qualifier only if you know exactly where it was inserted. Unfortunately, the C specification contains functions such as `memchr`, which takes a `const`-qualified pointer as the first argument, and returns a (non-`const`) pointer derived from it, stripping the `const` qualifier.

This function signature exists because the C type system cannot express the real requirement: that the function has a contract not to modify the buffer, and that the mutability associated with the returned pointer should be equal to the rights that the caller has to the first argument.

3.5 Pointer provenance

The C specification makes the ability to store pointers in integer variables and then recreate the original pointer implementation defined, but many software packages in practice depend on this functionality. In common implementations, as long as the integer is derived from the initial pointer, and the result of arithmetic is within the bounds of the object, then arbitrary arithmetic is permitted. The notion of an integer derived from a pointer is somewhat vague.

The xor linked list is a simple example where this is problematic. Each node has a pointer that is the address of the previous node xor'd with the address of the next node, allowing traversal in both directions. The pointer is derived from the addresses of both objects. This is particularly challenging when attempting to produce a formal semantics of

C. This idiom is now very uncommon, due to its poor interaction with pipelined processors.

Relying on the alignment to identify unused bits in a pointer is more common. On a 64-bit platform, most values are 8-byte aligned, and so the low 3 bits in a pointer are zero. It is therefore safe for a program to store information in these bits. The ARMv8 architecture [3] extends this with a mode guaranteeing that the MMU will not use the top 8 bits in a virtual address when performing address translation, allowing them to be used for storing program-specific data.

3.6 Garbage hoarding

To add full memory safety to C would require *temporal* as well as *spatial* safety. We can add spatial safety within the confines of the abstract machine by changing the representation of pointers to include bounds. Efficiently adding temporal safety is more difficult.

Existing conservative garbage collectors for C, such as the Boehm-Demers-Weiser collector [5], fail in cases outlined in §3.5 because they make assumptions about pointer representations. The ability to recover pointers from integers arithmetic makes accurate garbage collection impossible because any integer value may potentially be combined with others to form a valid address.

It is not possible to implement a copying or relocating garbage collector if it is possible for object addresses to escape from the collector. The conservative garbage collectors that are possible allow garbage objects to be accidentally hoarded by integers that contain valid addresses.

We believe that efficient implementations of full temporal safety will require C implementations that are valid within the requirements of the C abstract machine, but have unexpected behavior for much existing code. It is undefined behavior in C to compare two pointers to different addresses but in spite of this it is common to use addresses for comparison in trees or as input to hashes for hash tables. This works in current implementations but would break if a collector is allowed to modify addresses.

4. Refining the CHERI model

Our CHERI processor provides a fine-grained memory protection model via *memory capabilities*, which are hardware-enforced *unforgeable* references to regions of memory. In the CHERI world, all memory is accessed via a memory capability. The ISA is a superset of MIPS IV, an existing 64-bit ISA, and can run unmodified MIPS code.

Version 2 of the CHERI ISA [35] was created based on initial experiences attempting to build a capability system that was a usable compiler target for C. For example, this change propagated tag bits into capability tag bits, rather than preventing memory without tag bits from being loaded into capability registers. This was motivated by the need for `memcpy()` (which is called explicitly by the user and implicitly by the compiler) to be able to copy data without being

aware of whether it contained pointers. A similar requirement applies to unions.

CHERIV2 memory capabilities are represented as the triplet (*base*, *bound*, *permissions*), which is loosely packed into a 256-bit value. Here *base* provides an offset into a virtual address region, and *bound* limits the size of the region accessed; for CHERIV2 they are each 64-bits in length. For a discussion of *permissions*, see [33]. Capabilities reside either in a dedicated register file or can be spilled to memory, where their integrity is preserved by hardware-managed *tagged memory*. Capabilities must be naturally aligned and there is a single tag bit per 256 bits of memory. Conventional stores to an in-memory capability cause the tag bit to be cleared, invalidating the capability. Special capability load and store instructions allow capabilities to be spilled to the stack or stored in data structures, just like pointers. CHERIV2 supports use of capabilities as C pointers, with the caveat that pointer subtraction is not allowed.

Memory accesses are indirected via capabilities in three ways. Instruction fetches are relative to the *program counter capability* (PCC). Legacy MIPS loads and stores are relative to the *default data capability*. Finally, the CHERI ISA provides a set of load and store instructions that take an explicit capability register operand. The only instructions that modify a capability strictly reduce its permissions, either by increasing the base (and decreasing the length by the same amount), decreasing the length, or reducing the permissions.

When a process starts, it has a default data capability that covers the entire user address space. This can be partitioned and restricted, for example by limiting its use to the memory allocator, and referring to other memory purely via capabilities returned from the allocator, the linker, or from the stack capability. It is the responsibility of the allocator (or the compiler, for stack allocations) to correctly set the length on capabilities. Once set, it is impossible to use the resulting capability to gain access to memory outside the object.

Code compiled for our extension to the CHERI ISAv2 runs atop the FreeBSD operating system on a modified CHERI processor synthesized to run in FPGA at 100MHz. Our goal is to demonstrate that it is possible to meet all of our requirements for a low-level language compilation target from an instruction set that provides spatial memory safety. In particular, all of the mechanisms provided by the hardware can be exposed to a C programmer directly, and all of the aspects of memory behavior defined by the abstract machine are translated into simple machine instructions without any complex abstraction layers or support libraries.

4.1 Converging capabilities and fat pointers

CHERI ISAv2 capabilities provide C-like pointer semantics subject to non-increasing rights imposed by the capability model. The CHERI ISAv2 compiler allows code authors to qualify suitable pointers for compilation using capabilities, implicitly confirming that any manipulation of the pointer is compatible with those restricted rights; e.g., that subtrac-

tion of the *base* field is not required. Ideally, however, the compiler would be able to automatically employ capabilities for all pointers, subject to binary-compatibility concerns with capability-unaware code, which requires more broad support for arbitrary pointer arithmetic. We found, for example, that real-world code such as `tcpdump` (see §5.2) undertakes pointer arithmetic which, during intermediate steps, goes beyond the bounds of the structure the pointer is supposed to be referencing, but which (usually) ends up within the bounds during a memory access. To make CHERI-style capabilities better fit the C-based pointer idiom, and allow automated use with unqualified pointers, we have therefore extended the memory capabilities found in CHERIV2 with the attributes of *fat pointers*, to produce the CHERIV3 ISA. Our fat-capabilities add an *offset* to the CHERIV2 capability model to form: (*base*, *bound*, *offset*, *permissions*).

The general idea of a fat pointer is well explored and their benefits in providing spatial memory safety are well documented: pointers are supplemented by base and bounds values that, on dereference, throw a trap if an inappropriate access is requested. CHERIV3 fat-capabilities have two advantages over conventional fat pointers: (1) fat-capability integrity and non-decreasing rights are guaranteed by the hardware capability model while still supporting arbitrary pointer arithmetic; and (2) the permissions field permits additional hardware-checked constraints to be imposed, potentially for garbage collection or information-flow labeling. Integrity is especially important in the presence of concurrency: a thread-safe fat pointer implementation must guarantee that the entire fat pointer is updated atomically.

These distinctions allow memory capabilities to be used as building blocks for higher-level security features. The total memory that is reachable from a piece of code is the transitive closure of the memory capabilities reachable from its capability registers, making it possible for some threads (for example) to have limited rights to memory within a program. The addition of permissions allows capabilities to be tokens granting certain rights to the referenced memory. For example, a memory capability may have permissions to read data and capabilities, but not to write them (or just to write data but not capabilities). Attempting any of the operations that is not permitted will cause a trap.

This allows memory capabilities to enforce the `const` qualifier on pointers, by removing the store permission. We did implement support for this in the original C compiler for CHERIV2, but found that it broke a large amount of code. We subsequently added `__input` and `__output` qualifiers that make it possible to declaratively discard write and read permissions, respectively.

It is possible in the CHERIV2 model to implement fat pointers as a pair of a capability and an (integer) offset. This has several disadvantages. The first is the size. CHERIV2 capabilities are already 256 bits and must be aligned. The alignment requirement would mean that an array of fat point-

INSTRUCTION	USE
CIncOffset	Adds an integer to the offset
CSetOffset	Sets the offset
CGetOffset	Returns the current offset
CPtrCmp	Compares two capabilities
CFromPtr	Converts a MIPS pointer to a capability
CToPtr	Converts capability to a MIPS pointer

Table 2. New ChERI instructions to better support C

ers represented this way would use 64 *bytes* per pointer, although 24 of those would be padding. This approach would also make atomic updates to pointers difficult. Updating a pointer in memory requires writing the capability and the integer value atomically, which is not something that is easy to add with current DRAM memory widths.

To access this new *offset*, we added the instructions described in Table 2. In addition to these extra instructions, we modified CIncBase to update the pointer such that the offset remained constant and modified all of the load and store operations to use the pointer value. Our total changes amount to 496 lines of changes to the Bluespec source for the processor and 218 lines of changes to the test suite. These changes did not alter the critical path of the pipeline—the virtual address calculation for loads and stores—which is now done by adding the offset to the pointer, rather than to the base. It did add a small amount of complexity to the bounds checking logic. This now checks the resulting address against the base and top, rather than just against the length, so this pipeline stage (which happens in parallel with the cache fetch) is extended in length by one OR operation.

To avoid accidentally leaking virtual addresses into integer registers, we added a CPtrCmp instruction that compares the base plus offset of two capabilities as if they were pointers. This instruction orders all tagged capabilities after all untagged capabilities. Integer values stored in a capability are constructed by setting the offset of the canonical null capability and will never compare equal to any valid capability.

4.2 Interoperability

We added two instructions, CToPtr and CFromPtr, to convert between a capability and a traditional pointer. These take three operands, the capability and integer registers as the input and output, and a base capability that indicates the capability to which linear pointers are relative.

The CFromPtr instruction allows a pointer to be derived from the default data capability. It has special case behavior of giving a canonical null capability (all zeros with the tag bit cleared) if the offset within the default capability is 0, to adhere to C’s null pointer semantics. CToPtr provides the address of the capability as an offset from a base capability (so that it can be used as a pointer relative to the default data capability) or 0 if this value would be out of range.

As bounds information is not carried by traditional pointers, these instructions must be used carefully and are relevant only in a hybrid environment, where the memory-safe capability view and the traditional view of pointers are mixed. As our canonical null capability is untagged, it can never become a valid capability. Arithmetic may alter the offset of the NULL capability allowing constructions such as returning -1 from mmap() to indicate failure.

As long as the appropriate modifications are made for each instance of its use, with ChERIV3, a `__capability`-qualified pointer can be used anywhere (library interfaces permitting) that a traditional pointer can. When a capability-qualified pointer is used in a union and then modified as a non-capability type, the result is an invalid capability that will trap when used for memory access. This is provided by ChERI’s tagged memory, which clears the is-a-capability tag bit associated with the memory when a non-capability operation is used to store to a given address.

ChERIV3 C supports storing data in unused bits of a pointer as the pointer can point anywhere in the virtual address space (but must be within the bounds before it can be used). These programmer optimizations rely on the knowledge of implementation-defined behavior (such as the alignment of types and representation of pointers) to be efficient. As such, we consider it sufficient that they work, without requiring that they be efficient.

With ChERIV3, it is possible to use a code capability for every function. Each function invocation becomes a capability jump-and-link-register (CJALR) instruction, which replaces the PCC with a new capability register, the PC with a new register, and saves the old ones. This means that when a function is executing, it is impossible to jump out of it without an explicit call. Unfortunately, implementing this in practice is somewhat difficult as compilers and linkers make assumptions about their ability to place constants close to functions and depend on the program counter address to locate globals. Resolving this would require a completely new ABI, which is the subject of ongoing work.

We have implemented a relocating generational garbage collector for ChERIV3 that uses the tagged memory to differentiate between capabilities and other data. Determining how much software will be broken by this is ongoing work.

5. Evaluation

To evaluate our approach, we consider two questions:

- To what extent does compilation of the current C-language corpus depend on C idioms specific to PDP-11-like concrete implementations?
- How do alternative memory models function with respect to correctness and performance of C applications?

We explored the first question earlier, discovering a set of common idioms, and extracting simple test cases for each of them. Having identified these cases, we explored which of

them will function with different interpretations of the standard. In addition to the x86 and MIPS baselines, the original CHERIv2 implementation, and our CHERIv3 variant, we implemented a translator for C code into a simple abstract machine interpreter. This runs very slowly but allows us to quickly modify the abstract machine and run the test cases extracted from the idioms to see which fail.

For the second question, we evaluated the two CHERI C implementations against the same code compiled using the MIPS ABI. For this part of the evaluation, we added `__capability` qualifiers to pointers in the application, but retained the non-memory-safe versions for external pointers – for example, for use with system library routines.

5.1 Comparing implementation models

These results are shown in Table 3. They contrast traditional PDP-11-like x86 and MIPS implementations with the following other possibilities:

HardBound refers to the model as described in [12].

Intel MPX refers to the bounds-checking extensions to x86 proposed by Intel.

Relaxed interpreter allows pointers to be constructed from integer values as long as the object is still valid. The top 32 bits of the pointer are used to look up a valid object in a map and the low 32 bits describe an offset.

Strict interpreter allows pointers to be reconstructed from integers if (and only if) they are not modified in their integer representation.

CHERIv2 describes the original C compiler for CHERI, without offset support, where pointer addition decreases the range of the underlying capability.

CHERIv3 describes a new version of the CHERI ISA that allows capabilities to act as fat pointers.

The MIPS and x86 abstract machines do not differ in ways relevant to our analysis.

Several of the results in this table require additional explanation. Both CHERI implementations allow storing capabilities in integers of type `intcap_t`, but not in normal C integers. The CHERIv3 implementation performs arithmetic on these using the offset, and so does permit arbitrary arithmetic. The original CHERI implementation permitted only storing and loading of these values. The Strict implementation has the same restriction: pointers can be stored in integers, but any arithmetic will invalidate the pointers.

These provide five points on a continuum trading safety for compatibility. CHERIv3 supports all of the idioms that we identified as being important to avoid breaking for compatibility with existing code. We address the `const` enforcement issue by making `const` advisory and providing a new qualifier (`__input`) that is hardware enforced. There is also the caveat that storing pointers in integers and performing

arbitrary arithmetic is allowed only in places where doing so would not damage the memory-safety model.

The Strict model is close to our ideal interpretation of the C standard, disallowing a large set of operations that are potentially dangerous. Unfortunately, Strict also breaks many assumptions in existing code. Given the amount of time taken to ensure that the majority of C code is 64-bit clean, it is important to provide incremental steps to improve C code. The CHERIv3 target is attainable in the short term with small amounts of work, from which it will be easier to refine code to operate in an environment where pointers are strictly distinct from integers.

Intel’s MPX and HardBound both rely on explicit instructions to tag pointer bounds. This approach relies on the compiler being able to determine that it is operating on a pointer. Casting a pointer to an integer and then performing arbitrary arithmetic prevents the compiler from asserting the bounds. These two models (neither is yet available as an implementation, although Intel provides an MPX simulator) provide subtly different tradeoffs. The Intel version leans more in the direction of compatibility. If a pointer is modified in such a way that the MPX extensions are not updated, then the value will fail its check against the copy of the pointer in the look-aside table that the CPU uses to maintain the pointer state. If this occurs, then the bounds checks succeed unconditionally. This preserves compatibility, but at the expense of safety. In contrast, HardBound does not store the pointer in the look-aside table and so will assume the old bounds, even if the pointer now refers to a different object and so will fail closed, preferring a detectable error to compatibility. Both have integer arithmetic on pointers and masking marked as not working, although this does depend on whether the compiler is able to correctly propagate the bounds.

In our simple test cases, MPX failed the Container check because the compiler associated bounds with the inner pointer and so hit a bounds check. Our test cases passed for MPX with arbitrary arithmetic and masking, but this depends on the compiler updating the bounds correctly. This is more difficult in programs where the compiler cannot see that a particular integer contains a pointer value.

MPX also suffers significant problems with atomicity. Its metadata is stored in look-aside tables, in separate pages to the relevant data. This leads to race conditions in multi-threaded environments, as both locations must be changed in response to pointer modifications. As MPX relies on explicitly updating pointer metadata, it is also difficult to implement `memcpy` and similar functions, which must copy memory without being aware of the types of its contents.

The “best effort” translation of integers to pointers in the Relaxed model allows accidental construction of valid, but incorrect, pointers. This cannot happen with the CHERI version, as unsafe arithmetic would invalidate the capability, and provides an interesting alternative similar to some attempts at adding bounds checking to C code; it gives little

PROGRAM	DECONST	CONTAINER	SUB	II	INT	IA	MASK	WIDE
x86/MIPS/PDP-11	yes	yes	yes	yes	yes	yes	yes	no
HardBound	yes	yes	yes	yes	(yes)	no	no	no
Intel MPX	yes	no	yes	yes	(yes)	(yes)	(yes)	no
Relaxed	yes	yes	yes	yes	yes	yes	yes	no
Strict	yes	yes	yes	yes	(yes)	no	no	no
CHERIv2	no	no	no	no	(yes)	no	no	no
CHERIv3	yes	yes	yes	yes	(yes)	yes	yes	no

Table 3. Summary of idioms supported by different interpretations of the C abstract machine.

benefit in terms of source compatibility, compared to the capability version, in exchange for a weaker memory model.

All of our original set of requirements for an implementation of a low-level language, meet the expressiveness requirement, according to the specification—except for the original CHERI implementation. The lack of pointer subtraction means that the implementation did not meet the requirements of the abstract machine. More interesting is whether it captures the semantics expected by programmers. From Table 1, we observe that most C code can function with the restrictions imposed by CHERI. The relaxed restrictions in our modified CHERI will still break code that expects to be able to remove a `const` qualifier, but the only other modification required is changing the `intptr_t` typedef to refer to the `intcap_t` type and ensuring that this type is used whenever pointers are stored in integer values.

CHERIv2 and CHERIv3 score very highly on efficiency. Each C operation is a short sequence of machine instructions. Loads and stores are single instructions, and address calculations are either register or immediate offsets, or the result of a `CSetOffset` or `CIncOffset` instruction.

The exposure requirement is a bit more difficult to judge objectively. For the CHERI implementation, all of the functionality of the underlying substrate is exposed, and the properties of a capability (length, permissions, and so on) can be queried via intrinsics.

5.2 Whole program testing

Having demonstrated that our modified CHERI ought to be able to implement almost all of the requirements of C code, we next attempt to validate that assumption by porting a small selection of programs to run on it. We used the Olden benchmark suite (which is heavy in pointer use and so demonstrates a worst case for CHERI), Dhrystone (a less pointer-intensive benchmark), `tcpdump` (as an illustrative application), and `zlib` (as an illustrative library).

Table 4 shows the number of lines of code and percentage of the total changed to adapt each to run on CHERIv2 and CHERIv3. All of these results are for CHERI synthesized to run at 100MHz on a Stratix IV FPGA. The L1 data cache is 16KB and the L2 cache is 64KB. This is smaller than most contemporary processors, but the DDR DRAM is faster

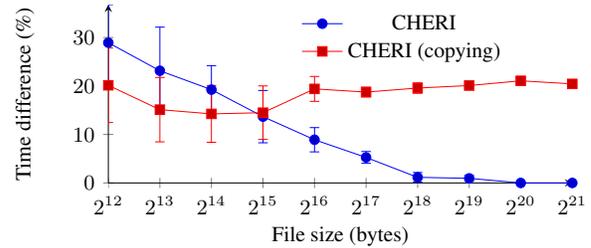


Figure 4. Overhead of CHERI-zlib normalized against zlib compiled for a conventional MIPS ISA.

relative to the CPU speed, so cache misses are more common but less costly than on most modern processors.

The first column shows the lines whose only changes are to mark pointers as capabilities. We made these changes manually to understand their placement, but the compiler can now use capabilities to represent all pointers. The important numbers are the semantic changes required because of the different memory models.

Having ported the code, we benchmarked it with unmodified MIPS code and CHERIv3. The Olden benchmarks were run with the parameters in the CHERI ISCA paper [35]. The Olden results are shown in Figure 1. The Dhrystone benchmark was run for 500,000 iterations on each run and the results are shown in Figure 2. Finally, the performance of `tcpdump` was measured by processing the first 100,000 packets from the `A.pcap061106170025.pcap` file from a network trace from OSDI 2006[6] is shown in Figure 3.

The slowdown for `tcpdump` (unmodified MIPS vs. CHERIv3) was $4\% \pm 3\%$. For comparison, small changes to optimizations in our LLVM gave a 15% speedup for `tcpdump`, so 4% is well within the margin for errors. Changes in cache and TLB pressure resulting from small changes to code layout can be larger. The Dhrystone results show the CHERI version to be around 2% faster than the MIPS code, but this is well within the margin of error simply for changes to instruction cache usage. These results lead us to believe that strong memory protection need not incur a statistically significant slowdown. The T-test places the difference in performance at 95% confidence below the experimental variation from small changes to compiler performance.

PROGRAM	Baseline LoC	CHERIv2			CHERIv3		
		Annotation	Semantic	Total	Annotation	Semantic	Total
Olden	1519	53 (3.5%)	0 (0%)	53 (3.5%)	53 (3.5%)	0 (0%)	53 (3.5%)
Dhrystone	448	11 (2.4%)	0 (0%)	11 (2.4%)	11 (2.4%)	0 (0%)	11 (2.4%)
tcpdump	66555	3006 (4.5%)	1,577 (2.4%)	4583 (6.9%)	3006 (4.5%)	2 (0.0%)	3008 (4.5%)

Table 4. Lines of code changed to port from MIPS to CHERIv2 and CHERIv3

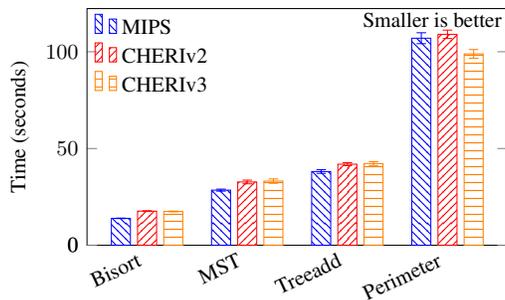


Figure 1. Olden results

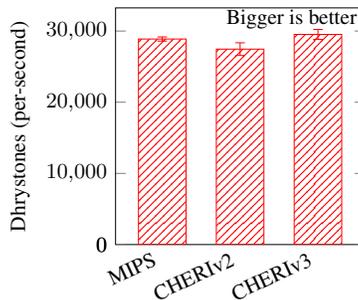


Figure 2. Dhrystone results

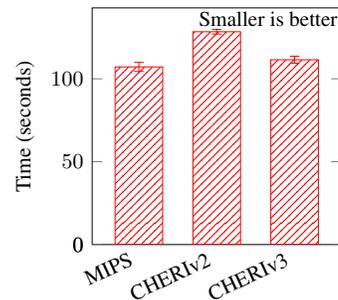


Figure 3. Tcpdump results

The effort required to port C code to a memory-safe environment depends significantly on the coding style employed. We have observed that the code that could most benefit from a memory-safe substrate (judging from published vulnerabilities related to memory errors) is also the code that is harder to port. This observation is one of the motivations for allowing unmodified MIPS code in our CHERI implementation: it allows us to apply coarse-grained sandboxing to such code. This means that even if we have to use non memory-safe code, the damage it can cause can be restricted.

The two benchmark suites that we modified are both conservative in their pointer use. They required changes only to annotate some values with `__capability`, and performed little or no unusual pointer arithmetic. In a pure capability environment, no annotation would be required. This is fairly common in benchmark code. The performance difference between them is primarily due to the larger pointers causing more cache misses. In spite of the relatively small cache on CHERI (16KB L1, 64KB L2), this overhead is significant only in the Olden benchmarks, which create and walk pointer-based data structures. Neither a CPU-focused benchmark nor a real application showed significant slowdown.

The tcpdump codebase is very different from the benchmarks in terms of porting effort. Packet dissection involves substantial pointer arithmetic—ironically, frequently in service of hand-crafted software bounds checking. Unfortunately, tcpdump typically runs as root (to access raw packet data from a network interface), and is often used for inspecting suspicious network traffic. This means that its packet parsers—written using extensive pointer arithmetic—are exposed to malicious data. Crafting malicious packets that

crash tcpdump (if not execute arbitrary code) to blind opponents is a common strategy in *capture the flag* competitions. The code would benefit substantially from strong memory safety, yet written in a style that makes this very difficult.

Porting tcpdump to CHERIv2 required changes to around 1.6K lines to avoid pointer subtraction: around 2.5% of the total codebase. CHERIv3 is fully able to support the parts of the C standard expected by these codebases. Only, two lines of code had to be changed in just one file, to ensure that tcpdump has read-only access to the packet being parsed rather than the whole packet buffer. This change was not strictly required, but provided stronger and finer-grained protection than would otherwise be possible.

We then modified the compiler to support a new ABI in which all pointers are implemented as `__capability`, including references to on-stack objects, which are derived from a stack `__capability`. We then compiled two versions of zlib using this mode. The first is entirely unmodified internally, but requires some annotations in a header to allow it to be work with code using the MIPS ABI. This was limited to a single pragma at the start and end of the library header, and was required because zlib passes pointers across the library interface. This approach breaks binary compatibility for the library (though it retains source compatibility), so we also implemented a second version that preserves binary compatibility by copying structures whose layouts have changed whenever they are passed across the library boundary.

Figure 4 shows the results for compressing files of varying sizes with the two modified and one unmodified version of zlib, linked to the gzip program. Simply using `__capabilities` incurs no measurable overhead for large files and a small

overhead for small files (there is a small constant overhead for setting up the capability environment). Copying data at the library boundaries to maintain binary compatibility incurs around a 21% overhead, independent of file size.

6. Related work

Substantial prior work targets improving C memory safety. Early examples include Cyclone [17], which explicitly broke compatibility with C to define a safer C dialect. Cyclone’s abstraction is close to our model, but adds many static annotations. Although Cyclone was not widely adopted, it influenced pointer annotation in current C compilers.

The copious buffer overflow vulnerabilities in C codebases have spurred development of a number of other bounds-checking systems. Research tools HardBound [12] and SoftBound [22] add fine-grained bounds checking to C, followed by commercial work, such as Intel’s Memory Protection Extensions [15]. None of these approaches handles some of the complex cases (for example, xor linked lists) outlined here, the key difference being that the HardBound and SoftBound approaches will fail closed (preventing potentially dangerous memory accesses if the can’t track pointer provenance), whereas the Intel solution will fail open (allowing unsafe access if the bounds can’t be determined).

Memory-related exploit techniques have spawned substantial work in vulnerability scanning and dynamic mitigation [27]. Static analysis tools such as lint [14] and general bug pattern-matching tools [4] are effective for local analyses, but are limited by weak type safety, global program complexity, and difficult-to-analyze dynamic behaviors. Dynamic protections such as stack canaries and address-space layout randomization offer probabilistic protection for specific exploits [7], but suffer a constant escalation [29].

To run untrusted C code within a process that contains other data rights, SFI [30] and Google’s Native Client (NaCl) [36] provide coarse-grained isolation for native code. Similar techniques like Robusta [28] (and earlier work on process isolation [10]) isolate almost a million lines of C code libraries from JVM’s. Without isolation or a memory-safe substrate, a single pointer error in the native code can damage the entire VM, including the typesafe Java code.

SAFECode and SVA [8, 9] provide control flow integrity and memory safety in TCB code, specifically in kernels, via compiler transforms. We believe that the run-time overhead of these techniques would be lower with hardware assistance described in this paper, and would provide powerful tools for quickly moving existing code to such a substrate.

Emscripten [37] is an interesting example of an unusual deployment target for C, running C code in a JavaScript VM. The runtime violates our exposure requirement by creating the C heap in a single JavaScript object and not exposing the JavaScript memory model. Microsoft’s various approaches to running C++ code in the .NET CLR [19] also provide some inspiration. These introduce different types of pointer

to differentiate between fully GC’d memory and “pinned” pointers, which have stable integer values. With better hardware support for protection, we argue that it becomes much easier to implement systems like these, with low-level code running in the same environment to managed code.

7. Conclusion

Lack of memory safety in C leads to real and significant vulnerabilities, particularly as near-ubiquitous networking for conventional computers and mobile devices has exposed users to unprecedented malicious activity. This has reinvigorated interest in fine-grained memory protection to mitigate exploit techniques. However, widespread assumptions about pointer behavior, unwarranted by the C abstract machine but consistent with the PDP-11 memory model, produce substantial compatibility problems in real-world programs that might most benefit from additional protection.

To address these problems, we survey C-pointer idioms across a large open-source corpus to understand the effective consensus on pointer use that constrains memory-protection schemes. We are then able to describe a range of points in the protection and compatibility tradeoff space.

Observing potentially significant source-code compatibility, we propose a new model, fully described and implemented as the CHERIv3 ISA, that combines CHERI’s capability-system model with results from recent fat-pointer research. Whole-program testing against conventional RISC, CHERIv2, and CHERIv3 illustrates significantly improved source-code compatibility relative to a pure capability-system model, as well as modest performance improvement.

These results confirm that it is possible to retain the strong semantics of a capability-system memory model (which provides non-bypassable memory protection) without sacrificing the advantages of a low-level language.

Acknowledgments

We thank our colleagues – especially Ross Anderson, Gregory Chadwick, Nirav Dave, Khilan Gudka, Steve Hand, Stephen Kell, Ben Laurie, Ilias Marinos, A Theodore Markettos, Ed Maste, Andrew W. Moore, Prashanth Mundkur, Steven J. Murdoch, Robert Norton, Hassen Saidi, Peter Sewell, Stacey Son, and Bjoern Zeeb; we also thank our anonymous reviewers for their feedback.

This work is part of the CTSRD and MRC2 projects that are sponsored by the Defense Advanced Research Projects Agency (DARPA) and the Air Force Research Laboratory (AFRL), under contracts FA8750-10-C-0237 and FA8750-11-C-0249. The views, opinions, and/or findings contained in this paper are those of the authors and should not be interpreted as representing the official views or policies, either expressed or implied, of the Department of Defense or the U.S. Government. We gratefully acknowledge Google, Inc. for its sponsorship.

References

- [1] Is address space 1 reserved? URL <http://lists.cs.uiuc.edu/pipermail/11vmdev/2015-January/080288.html>.
- [2] Aleph One. Smashing the stack for fun and profit. *Phrack Magazine*, 7:14–16, 1996.
- [3] *ARM Architecture Reference Manual. ARMv8, for ARMv8-A architecture profile*. ARM Limited, 110 Fulbourn Road, Cambridge, England CB1 9NJ, 2013.
- [4] A. Bessey, K. Block, B. Chelf, A. Chou, B. Fulton, S. Hallem, C. Henri-Gros, A. Kamsky, S. McPeak, and D. Engler. A few billion lines of code later: Using static analysis to find bugs in the real world. *Commun. ACM*, 53(2):66–75, Feb. 2010. ISSN 0001-0782. URL <http://doi.acm.org/10.1145/1646353.1646374>.
- [5] H.-J. Boehm and M. Weiser. Garbage collection in an uncooperative environment. *Softw. Pract. Exper.*, 18(9):807–820, Sept. 1988. ISSN 0038-0644. . URL <http://dx.doi.org/10.1002/spe.4380180902>.
- [6] R. Chandra, V. Padmanabhan, and M. Zhang. CRAWDDAD data set microsoft/osdi2006 (v. 2007-05-23). Downloaded from <http://crawdad.org/microsoft/osdi2006/>, May 2007.
- [7] C. Cowan, P. Wagle, C. Pu, S. Beattie, and J. Walpole. Buffer overflows: attacks and defenses for the vulnerability of the decade. In *DARPA Information Survivability Conference and Exposition, 2000. DISCEX '00. Proceedings*, volume 2, pages 119–129 vol.2, 2000. .
- [8] J. Criswell, A. Lenharth, D. Dhurjati, and V. Adve. Secure virtual architecture: A safe execution environment for commodity operating systems. In *SOSP '07: Proceedings of the Twenty First ACM Symposium on Operating Systems Principles*, October 2007.
- [9] J. Criswell, N. Geoffray, and V. Adve. Memory safety for low-level software/hardware interactions. In *Proceedings of the Eighteenth Usenix Security Symposium*, August 2009.
- [10] G. Czajkowski, L. Daynes, and M. Wolczko. Automated and portable native code isolation. In *Software Reliability Engineering, 2001. ISSRE 2001. Proceedings. 12th International Symposium on*, pages 298–307, Nov 2001. .
- [11] R. Dannenberg, W. Dormann, D. Keaton, R. Seacord, D. Svoboda, A. Volkovitsky, T. Wilson, and T. Plum. As-if infinitely ranged integer model. In *Software Reliability Engineering (IS-SRE), 2010 IEEE 21st International Symposium on*, pages 91–100, Nov 2010. .
- [12] J. Devietti, C. Blundell, M. M. K. Martin, and S. Zdancewic. Hardbound: Architectural support for spatial safety of the C programming language. *SIGPLAN Not.*, 43(3):103–114, Mar. 2008. ISSN 0362-1340. . URL <http://doi.acm.org/10.1145/1353536.1346295>.
- [13] J. Evans. A scalable concurrent malloc(3) implementation for FreeBSD. In *BSDCan*, 2006.
- [14] Gimpel Software. FlexeLint for C/C++, August 2014. URL <http://www.gimpel.com/html/flex.htm>.
- [15] Intel Plc. Introduction to Intel® memory protection extensions. <http://software.intel.com/en-us/articles/introduction-to-intel-memory-protection-extensions>, July 2013.
- [16] ISO. *ISO/IEC 9899:2011 Information technology — Programming languages — C*. International Organization for Standardization, Geneva, Switzerland, Dec. 2011. URL http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=57853.
- [17] T. Jim, J. G. Morrisett, D. Grossman, M. W. Hicks, J. Cheney, and Y. Wang. Cyclone: A safe dialect of C. In *Proceedings of the General Track of the Annual Conference on USENIX Annual Technical Conference, ATEC '02*, pages 275–288, Berkeley, CA, USA, 2002. USENIX Association. ISBN 1-880446-00-6. URL <http://dl.acm.org/citation.cfm?id=647057.713871>.
- [18] A. Kwon, U. Dhawan, J. M. Smith, T. F. Knight, Jr., and A. DeHon. Low-fat pointers: Compact encoding and efficient gate-level implementation of fat pointers for spatial safety and capability-based security. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer and Communications Security, CCS '13*, pages 721–732, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2477-9. . URL <http://doi.acm.org/10.1145/2508859.2516713>.
- [19] Managed C++. Managed extensions for C++ specification. <http://msdn.microsoft.com/en-us/library/Aa712867> (accessed 2014/07/14).
- [20] Microsoft Corporation. CONTAINING_RECORD macro. URL <http://msdn.microsoft.com/en-us/library/windows/hardware/ff542043%28v=vs.85%29.aspx>.
- [21] Mitre. CWE/SANS top 25 most dangerous software errors, 2011. URL <http://cwe.mitre.org/top25>.
- [22] S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic. Softbound: Highly compatible and complete spatial memory safety for C. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '09*, pages 245–258, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-392-1. . URL <http://doi.acm.org/10.1145/1542476.1542504>.
- [23] G. C. Necula, S. McPeak, and W. Weimer. Cured: Type-safe retrofitting of legacy code. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '02*, pages 128–139, New York, NY, USA, 2002. ACM. ISBN 1-58113-450-9. . URL <http://doi.acm.org/10.1145/503272.503286>.
- [24] M. Richards. BCPL: A Tool for Compiler Writing and System Programming. In *Proceedings of the May 14-16, 1969, Spring Joint Computer Conference, AFIPS '69 (Spring)*, pages 557–566, New York, NY, USA, 1969. ACM. . URL <http://doi.acm.org/10.1145/1476793.1476880>.
- [25] D. Ritchie, S. Johnson, M. Lesk, and B. Kernighan. UNIX time-sharing system: The C programming language. *Bell System Technical Journal*, 57(6):1991–2019, July-Aug 1978.
- [26] J. Saltzer and M. Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9):1278–1308, September 1975. URL <http://www.multicians.org>.

- [27] H. Shahriar and M. Zulkernine. Mitigating program security vulnerabilities: Approaches and challenges. *ACM Comput. Surv.*, 44(3):11:1–11:46, June 2012. ISSN 0360-0300. URL <http://doi.acm.org/10.1145/2187671.2187673>.
- [28] M. Sun, G. Tan, J. Siefers, B. Zeng, and G. Morrisett. Bringing Java’s wild native world under control. *ACM Trans. Inf. Syst. Secur.*, 16(3):9:1–9:28, Dec. 2013. ISSN 1094-9224. . URL <http://doi.acm.org/10.1145/2535505>.
- [29] L. Szekeres, M. Payer, T. Wei, and D. Song. Eternal war in memory. In *IEEE Symposium on Security and Privacy*, 2013.
- [30] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham. Efficient software-based fault isolation. In *SOSP ’93: Proceedings of the fourteenth ACM Symposium on Operating Systems Principles*, pages 203–216, New York, NY, USA, 1993. ACM. ISBN 0-89791-632-8.
- [31] X. Wang, H. Chen, Z. Jia, N. Zeldovich, and M. F. Kaashoek. Improving integer security for systems with KINT. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation, OSDI’12*, pages 163–177, Berkeley, CA, USA, 2012. USENIX Association. ISBN 978-1-931971-96-6. URL <http://dl.acm.org/citation.cfm?id=2387880.2387897>.
- [32] X. Wang, N. Zeldovich, M. F. Kaashoek, and A. Solar-Lezama. Towards optimization-safe systems: Analyzing the impact of undefined behavior. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles, SOSP ’13*, pages 260–275, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2388-8. . URL <http://doi.acm.org/10.1145/2517349.2522728>.
- [33] R. N. Watson, P. G. Neumann, J. Woodruff, J. Anderson, D. Chisnall, B. Davis, B. Laurie, S. W. Moore, S. J. Murdoch, and M. Roe. Capability Hardware Enhanced RISC Instructions: CHERI Instruction-set architecture. Technical Report UCAM-CL-TR-850, University of Cambridge, Computer Laboratory, Apr. 2014. URL <http://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-850.pdf>.
- [34] R. N. M. Watson, P. G. Neumann, J. Woodruff, J. Anderson, D. Chisnall, B. Davis, B. Laurie, S. W. Moore, S. J. Murdoch, and M. Roe. Capability Hardware Enhanced RISC Instructions: CHERI Instruction-set architecture. Technical Report UCAM-CL-TR-864, University of Cambridge, Computer Laboratory, 15 JJ Thomson Avenue, Cambridge CB3 0FD, United Kingdom, phone +44 1223 763500, Dec. 2014. URL <http://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-864.pdf>.
- [35] J. Woodruff, R. N. M. Watson, D. Chisnall, S. W. Moore, J. Anderson, B. Davis, B. Laurie, P. G. Neumann, R. Norton, and M. Roe. The CHERI capability model: Revisiting RISC in an age of risk. In *Proceedings of the 41st International Symposium on Computer Architecture (ISCA 2014)*, June 2014.
- [36] B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar. Native client: A sandbox for portable, untrusted x86 native code. *Commun. ACM*, 53(1):91–99, Jan. 2010. ISSN 0001-0782. . URL <http://doi.acm.org/10.1145/1629175.1629203>.
- [37] A. Zakai. Emscripten: An LLVM-to-JavaScript Compiler. In *Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion, SPLASH ’11*, pages 301–312, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0942-4. . URL <http://doi.acm.org/10.1145/2048147.2048224>.