CHERI: Hardware-Enabled C/C++ Memory Protection at Scale

Robert N.M. Watson, *University of Cambridge*
robert.watson@cl.cam.ac.uk

David Chisnall, *SCI Semiconductor Ltd*

Jessica Clarke, *University of Cambridge*

Brooks Davis, *SRI International*

Nathaniel Wesley Filardo, *Microsoft*

Ben Laurie, *Google, Inc*

Simon W. Moore, *University of Cambridge*

Peter G. Neumann, *SRI International*

Alexander Richardson, *Google, Inc*

Peter Sewell, *University of Cambridge*

Konrad Witaszczyk, *University of Cambridge*

Jonathan Woodruff, *University of Cambridge*

Abstract—The memory-safe CHERI C and C++ languages build on architectural capabilities in the CHERI protection model. With the development of two industrial CHERI-enabled processors over the past two years, Arm's Morello and Microsoft's CHERIoT, CHERI may offer the fastest path to widely deployed memory safety.

Keywords—Memory safety, C, C++, memory protection, CHERI

The lack of memory safety in current software implementations has led to a long and catastrophic history of software vulnerabilities – from enabling the spread of the Morris Internet Worm in 1988 to making up the majority of critical security vulnerabilities in Android, iOS, Windows, and numerous other contemporary software systems. Many attempts have been made to replace unsafe C and C++ with memory-safe and type-safe languages, but these have made only limited inroads in the most critical software Trusted Computing Bases (TCBs), due to the implied need for total software-stack rewrites. Widely deployed exploit-mitigation mechanisms – especially those based on random secrets – have simply engaged in an expensive arms race with attackers, leading to successively more sophisticated attack approaches that bypass those new defenses within a couple of years of their deployment.[1] This has left C and C++ OS kernels, language runtimes, web browsers, and server components in a nearly continuous state of vulnerability – subject to unwinnable "patch and pray" races with highly capable adversaries.

CHERI, a hardware-software co-design project started in 2010 by the University of Cambridge and SRI International, has pursued an alternative strategy: Use adapted hardware, enable memory-safe variants of the C and C++ programming languages themselves.[2] CHERI extends conventional instruction-set architectures (ISAs) with *architectural capability-system features* – an idea that arose in computer science in the 1970s, but has seen little impact since the 1990s.[3] As with research into type-safe languages, past capability systems have frequently been premised on clean-slate programming languages and/or software stacks. With CHERI, our focus has instead been on clean composition of the *architectural capabilities* with conventional ISAs, microarchitectures, the C and C++ programming languages, and de facto standard OS

XXXX-XXX © 2024 IEEE
Digital Object Identifier 10.1109/XXX.0000.0000000

designs, enabling memory-safe use of these widely used languages and their voluminous software stacks. Our iterative co-design cycle has been driven by (sometimes competing) demands for architectural and microarchitectural viability and performance, software vulnerability mitigation, and minimising potential adoption friction in both hardware and software stacks.

In 2011, our early “soft” CHERI MIPS cores were implemented on FPGA and able to run a bespoke assembly-language microkernel demonstrating rudimentary domain switching without compiler support. Today, in 2024, CHERI-enabled processors have been developed by Arm, Microsoft, and other companies and universities, and are able to run rich software stacks consisting of tens of millions of lines of open-source, memory-safe C and C++ including complete UNIX operating systems, windowing systems, and complex desktop and server applications. Two systems of particular note are Arm’s Morello and Microsoft’s CHERIoT, spanning a broad range of microarchitectural scales. Morello is a 7nm, superscalar, multi-core System-on-Chip (SoC) in which CHERI extensions have been added to the Neoverse N1 processor widely used in cloud computing environments.[4] Microsoft’s first CHERIoT implementation is a microcontroller based on the open-source Ibex core developed at ETH Zurich and now used in lowRISC’s OpenTitan hardware root of trust.[5] The first CHERI-based commercial products, rather than research prototypes, are expected to ship within 18 months, with initial impacts on embedded and Internet of Things (IoT) systems, and in roots of trust embedded within higher-end SoCs.

The CHERI proposition is a simple one: In return for modest changes to shipping computer architecture and microarchitecture, total system memory safety can be achieved through modest source-code change and recompilation. CHERI also opens the door for further forms of safety, including fine-grained scalable software compartmentalization, which builds on CHERI memory safety. The successful application of CHERI to multiple architectural and microarchitectural approaches has validated a key early hypothesis in our work: As with virtual memory, CHERI is a *model* and not just implementation, with its concepts portable across a broad range of scales and use cases. With approaching 100MLoC of CHERI-adapted open-source C and C++ code, including complete operating systems, desktop environments, and server applications, another key hypothesis has been validated: That there is low friction in adapting many critical pieces of C/C++-language software to the CHERI model.

In this article, we introduce the CHERI model, implementations, and early results. We also explore the current status of the research and productization. With increasing experience with large-scale hardware implementations and software adaptations, we are able to draw strong conclusions about the potential for adoptability of the CHERI technology.

BACKGROUND: MEMORY [UN]SAFETY

The rich history of memory unsafety could fill tomes. It is noteworthy, first, that the idea of unsafety is very old, dating from the 1970s, and also remarkably portable across languages and architectures. While C and C++ make such unsafety a natural programming style, a lack of memory safety can arise even in type-safe memory-safe language as a results of compiler bugs (e.g., as seen recently with Rust) or a language runtime written in an unsafe language (e.g., as seen with most Javascript runtimes).[6] CHERI is hardly the first attempt to introduce notions of memory safety into the C and C++ languages – earlier efforts fall into four (inevitably overlapping) categories:

- Debugging and mitigation techniques that either detect memory-safety violations or exploitation attempts enabled by memory-safety vulnerabilities (e.g., LLVM AddressSanitizer).
- Static analysis and proof techniques intended to statically detect and help eliminate unsafe programs, limiting them to a subset of behaviors that are safe (e.g., Coverity or the seL4 microkernel).
- Revised C and C++-like languages – either subsets, perhaps mechanically checked, of the existing languages, or new languages that resemble C and C++ (e.g., Cyclone).
- Longer-term improvements in C/C++ that introduce both primitives for memory safety and ownership, or simply stylizations, encouraging memory-safe programming within the language (e.g., MISRA and recent work on C++ memory safety).

For the purposes of our work, we have considered C/C++ memory safety to have the following language-level aspects:

1) *Referential safety* – a superset of *pointer integrity* that protects the integrity and provenance validity of pointers against various forms of corruption (e.g., partial memory overwrite) or misuse (e.g., inappropriate integer arithmetic on a pointer).
2) *Spatial safety*, which prevents a pointer intended to be used with one current in-memory object from being used to access another current object

– e.g., as a result of buggy pointer arithmetic.

3) *Temporal safety*, which prevents a pointer intended to be used with one current in-memory object from being used to access past (or future) objects that have used (or will use) the same storage – e.g., a memory store via a pointer passed to free() that, as a result of memory reuse, now accesses a new object returned to another context by malloc().

These concepts apply to a range of object types in C and C++: OS memory mappings, local and global variables, heap memory allocations, thread-local storage, and so on. The literature is rife with examples of violations of these properties leading to information leakage or data corruption, and frequently also enabling arbitrary code execution by an attacker.

Note that, amongst these properties, there are some important omissions that appear amongst key memory-safety vulnerabilities. Perhaps most important is that of uninitialized values, in which a failure to initialize a value before using it can allow visibility of earlier values present in prior allocations. This is omitted from the list of CHERI properties because reasonable solutions to the problem are well known within current software practice – zeroing memory mappings before first use, heap allocators returning zeroed memory, and compiler analysis passes that zero local variables before first use if improperly initialized. In security analyses of CHERI (e.g., by MSRC), it has been taken for granted that those initialization mitigations will continue to be used, and in general they complement the protections provided by CHERI.

Another key consideration in memory safety is not just protection of programmer-visible constructs, such as memory allocations, but also those constructs that are invisible to the programmer and yet are essential to the operation of the program. These *sub-language pointers*, such as return addresses, the frame pointer, GOT pointers, and other aspects of the sub-language that so greatly enable exploitation, have been the target for numerous mitigation techniques from stack canaries onwards.

Another concern in C memory safety is the boundary between memory safety and type safety. We seek to address only the former, although some protection of types is required to enable memory safety. The idea of general, dynamically enforced type safety, rather than memory safety, is challenging not least because it is not clear what can be achieved alongside for existing deployed C-language programs. For example, object oriented programming styles within C frequently involve casts between pointers types that are, from a language perspective, unrelated. However, clear differentiation of pointer and integer types, and between executable and non-executable pointers, is an essential part of memory safety. Overall, we feel that limiting our scope to memory safety has yielded significant security wins while also avoiding substantial compatibility issues – and yet certain vulnerabilities (e.g., programmer confusion regarding function pointer signatures) cannot be addressed by CHERI as a result.

CHERI C and C++ exist within a large space of works on C and C++ memory safety, but particularly focus in the following areas, that in combination set them apart from prior work:[7]

- A focus on *deterministic closing of vulnerabilities* through dynamic memory safety – without basing the work on secrets (that may be leaked or brute forced) or probabilistic techniques. Most currently deployed memory-safety mitigations are secrets-based and/or probabilistic.
- A focus on *minimizing disruption* to the vast majority of the C/C++ corpus, requiring recompilation but little or no change. Studies we have performed have found %LoC change counts as low as 0.02%LoC, seen in a sizeable open-source desktop software stack.[8] While simple mitigations such as stack canaries or ASLR have little impact on the programming model, more historic work on memory safety (such as Cyclone or even MISRA) can be very disruptive to current software.
- Willingness to depend on *modest changes to hardware architecture and microarchitecture* to enable certain properties (e.g., strong atomicity and software non-bypassability of provenance validity). There has been an important thread of new architecture exploit mitigations, such as pointer hashing and memory version tagging, that reflect minor and incrementally adoptable changes. CHERI, however, proposes a more significant revision of the underlying architecture to enable strong memory safety – not least, that recompilation is required.

CHERI: A CONTEMPORARY CAPABILITY MACHINE

CHERI learns from a long history of capability architectures, and also other tagged memory systems such as Lisp machines, in introducing a dedicated hardware type to represent rights: The architectural capability. Capabilities double the natural pointer size (typically 32- or 64-bit) to add additional metadata (compressed bounds, permissions, and sealing information), and carry a 1-bit tag that tracks the valid provenance of a value from one or more initial capability roots created at processor reset (see Figure 1). Capability values may be held in registers – typically extended general-purpose integer registers – or in memory. Capability values and their accompanying tags are carried atomically through the system. In memory, tags are associated with capability-sized, capability-aligned blocks of memory. Permissions on capabilities enable various architectural operations such as load, store, instruction fetch, and loading/storing tagged values.

New instructions are used to load, store, inspect, manipulate, and use capability values (e.g., for loading and storing via), which enforce *guarded manipulation*. In particular, the ISA ensures that properties such as monotonic non-increase of rights – e.g., that the bounds on a capability cannot be made more broad. Operations that might violate guarded manipulation clear a capability value's tag, which will prevent further use of the capability – preventing both violations of capability rules in register-to-register operations and via memory operations.

As implemented in our original MIPS prototype, CHERI duplicates the suite of load, store, and jump instructions to introduce capability-relative variants; e.g., 'load 32-bit integer via capability' to complement 'load 32-bit integer via 64-bit integer'. However, the load-store opcode footprint within most ISAs is a significant fraction of opcode space that should not (and in some cases cannot) be doubled. Instead, Morello and CHERI-RISC-V introduce a new 'capability mode' in which most integer-relative opcodes are reused for capability-relative accesses, with mode transitions during jumps. This reduces the opcode footprint for adding CHERI to an ISA down to a much smaller number of register-to-register capability instructions, as well as a small number of new instructions to load, store, and jump to capabilities.

Compressed bounds reflect a practical tradeoff: Notionally, *fat pointers* require a lower bound, an address, and an upper bound, each of which takes the space of a full address in the architecture. Doubling the size of the pointer in CHERI already risks significant performance overhead in software with high pointer-size dynamic access rates; quadrupling is simply non-viable for our target use cases. Bounds compression is a long researched technique that exploits redundancy between the address and its bounds – assuming that the value is within bounds. CHERI extends existing practice to improve microarchitectural efficiency, to adopt a floating-point style approach, and also support (to some extent) out-of-bounds pointers that occur in C code. Bounds compression does require increased alignment of bounds, with implications for allocation alignment and padding.

CHERI is designed to be sympathetic to contemporary microarchitecture, which is highly decentralized and highly concurrent. CHERI capabilities do not introduce new indirection, and similarly do not require the introduction of global tables, eschewing new types of stalls, dependencies, and synchronization – in contrast to page tables and TLB entries. This allows, for example, checking of capabilities during loads and stores to take place during address calculation, allowing only a calculated, authorized address to be carried forward through memory access. This design choice facilitates integration into a broad range of microarchitectures including complex out-of-order designs. However, while some types of protection arise naturally from this structure (such as spatial safety through bounds and permission checks), the lack of indirection and tables makes other types of protection more challenging (e.g., temporal safety). The design point selected for CHERI comes with many other trade-offs, which can be evaluated in terms of metrics reflecting microarchitectural and architectural impact, static and dynamic performance, vulnerability mitigation, and software impact. Not all CHERI implementations need pick exactly the same tradeoff point.

In addition to these largely processor-focused design choices, there are also concerns about where and how to store tags. In CHERI, the memory subsystem carries tags with the physical data cache lines they protect – e.g., across all levels of cache, coherency traffic, and so on. This requires modest changes to carry and appropriately clear tags during certain types of memory accesses. However, there is also the question of where tags are stored in memory themselves. For small designs with SRAM, storing tags inline is entirely viable. For larger memory systems, especially those using external DRAM, two options present themselves: Storing tag values in existing metadata storage in DRAM, such as in error-correcting coded bit storage; and introducing a 'tag controller' that stores tag data in lookaside tables in DRAM, to be 'glued' back together with data before entering the memory subsystem.

ARM MORELLO AND MICROSOFT CHERIOT CORES

Announced in 2019 and 2022 respectively, Arm's Morello and Microsoft's CHERIoT processors are the first publicly available, industrially developed CHERI-enabled processor cores. As case studies they represent two quite different points in the contemporary

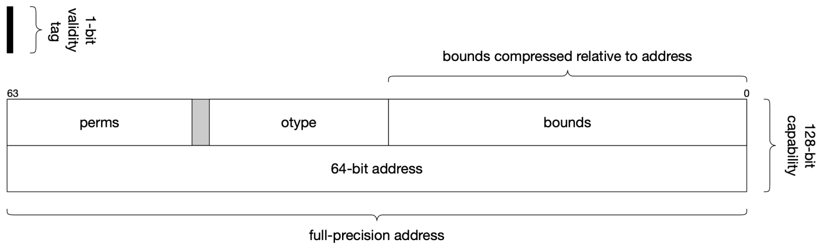


FIGURE 1. In-memory layout of 128-bit CHERI capability and its tag bit

microprocessor design space:

- **Arm's Morello** is a CHERI-extended version of the multi-core, superscalar 64-bit Neoverse N1 Armv8-A processor that is the foundation for Arm-based offerings in cloud environments such as AWS. Morello is fabricated using 7nm technology, and supports between 16GiB and 64GiB of DRAM. Morello is able to store CHERI tags either in ECC bits already present in DRAM, or using an integrated tag controller and cache. This and other design choices in Morello reflect its experimental goals, enabling cross-evaluation of multiple design choices.
- **Microsoft's CHERIoT Ibex**, in contrast, is a CHERI-extended version of the embedded 3-stage pipelined 32-bit Ibex RISC-V core originally developed by ETH Zurich and now maintained by lowRISC as part of the OpenTitan project. Initial expected use cases are in roots of trust, IoT devices, and low-power embedded control systems. SCI Semiconductor expects to ship embedded SoCs incorporating the CHERIoT Ibex using 22nm technology from late 2024. CHERIoT is able to support tags – both for CHERI and also optimized revocation (see below) inline within SRAM in the SoC design.

That the CHERI model is able to span such diverse platforms has definitively validated a key hypothesis of the CHERI project: That a single capability-based protection model can be used for fine-grained memory safety and scalable compartmentalization across a broad range of scales.

However, just as the baseline ISAs (Armv8-A and RISC-V) vary between such points in the CPU design space, so do the CHERI extensions. Whereas Morello must address design questions such as the composition of hardware virtualization extensions with CHERI, CHERIoT finds challenges in scaling down bounds and permissions to fit within smaller 64-bit capabilities.

At these two points in the spectrum, a key design difference lies in the implementation of temporal safety, despite both relying on quarantine-based batched sweeping revocation. CHERIoT is able to optimize for very small memory sizes that are more realistic to sweep frequently, and uses additional tags to track memory line generations. In contrast, Morello supports much larger memories with a multicore design, and makes use of virtual-memory extensions to track capability flow and virtual page generations to implement a load-barrier technique.[9]

Overall, and excitingly, CHERI memory protection for C and C++, as well as its enablement of compartmentalization, remains consistent across scales, in terms of reasonable microarchitectural implications and also a portable programmer model. Both implementations have supported the conclusion that CHERI is viable in production microarchitecture.

MEMORY-SAFE CHERI C AND C++ DIALECTS

Perhaps the most critical question to answer is: What does (or could) memory safety for C and C++ even mean?[10] Some aspects fall out naturally from an analysis of the most essential vulnerabilities:

- Accesses to all forms of allocated object – whether global variables, heap allocations, stack allocations, and so on – should conform to spatial and temporal safety.
- Pointers should be protected from corruption and mis-manipulation, including rudimentary type checking to ensure that code is not writable, pointers cannot be modified by integer arithmetic gadgets, and code pointers in particular are not subject to errant and unintended manipulation.

However, other aspects are more nuanced. For example, in C, it is not just complete memory allocations that require spatial safety; sometimes, subobjects (such as arrays embedded within structures) require narrower bounds. This comes with greater

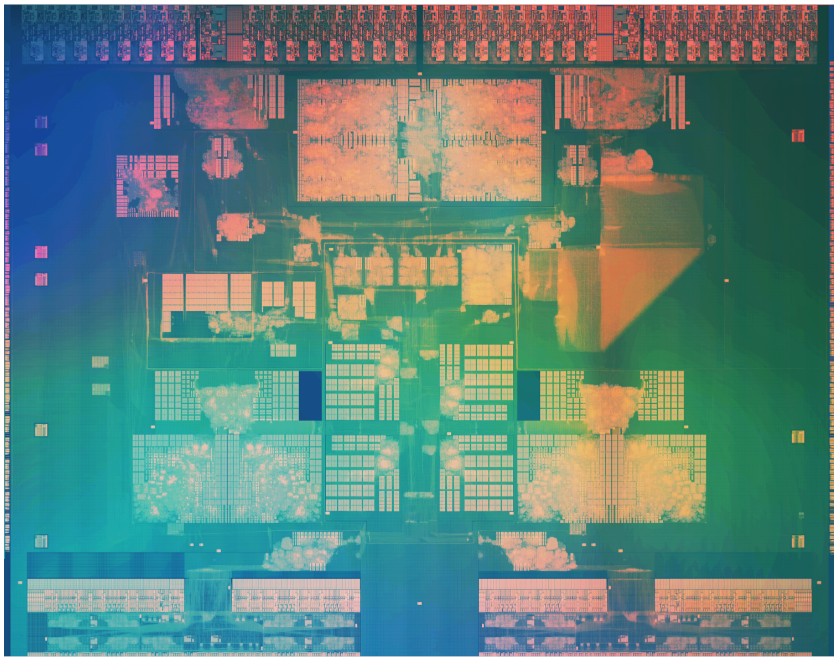


FIGURE 2. Die photo from the 7nm Arm Morello System-on-Chip (SoC), showing the four cores, GPU, and caches.

potential adoption friction, as the `containerof()` construct, which calculates a pointer to a container object from a pointer to its member, is frequently used in, for example, C intrusive linked-list macros.

CHERI C and C++ are not simply "memory-safe C" and "memory-safe C++": They are also implementations of C and C++ specifically intended to target a CHERI capability implementation of the pointer type. This also imposes changes visible in the language:

- Pointer types, as well as types able to store pointers (such as intptr_t) are now double the natural address size of the architecture. Tags require that they must also be naturally aligned. This changes the layout and alignment requirements of data structures, which can require changes to some programs – such as to offsets and assertions in JITs, and in custom memory allocators.
- Most existing integer types can no longer hold pointers while maintaining valid provenance. Types such as long and intmax_t are sufficient to hold the addresses from pointers, but not their metadata. For most code, this is undisruptive, but we have observed several language runtimes using types other than intptr_t for pointer storage.
- Memory-copy implementations intended to copy pointers must now propagate tags, requiring them to be implemented with a pointer-enabled type such as intptr_t. Surprising functions turn out to be memory-copy routines, such as sort functions used to sort C data structures containing pointers.

Some aspects of CHERI C and C++ memory safety map naturally into the direct use and manipulation of capabilities. For example, changes to the compiler to implement pointer types using architectural capabilities are relatively straightforward, although the compiler frontend can require some amount of "plumbing" to ensure that pointer types do not become integer types during optimization (for example). Memory allocators can easily use compiler builtins to set bounds on allocations they perform – and modified alignment and padding requirements are also adapted to as other similar alignment and padding requirements already exist.

Other aspects require considerably more work.[11] For example, the kernel virtual-memory subsystem must not only maintain tags (e.g., across copy-on-write and swapping), but now must also assist with efficient revocation by tracking capability flow. There are also interesting semantic challenges: For example, to what extent should debuggers using ptrace() be able to inject pointers into target processes?

CHERI C and C++ are not without limitations, and perhaps most critical of are the constraints on temporal memory safety imposed by our decision not to introduce indirection and tables. Quarantining and sweeping revocation is not without cost, and for some system designs, quarantined memory can also significantly impact free memory. Further, this technique, while suitable for memory mappings and heap allocations, is not applicable to stack allocations due to their frequency. While stack use-after-free is not all that common in C, use-after-scope and similar vulnerabilities occur more frequently in C++. And, unlike with uninitialized values, there are not obvious compiler-based techniques that complement CHERI.

On the whole, our experience is that software falls into one of three categories:

- OS kernels, language runtimes, or parts of the C/C++ runtime (such as the run-time linker) require modest but non-trivial adaptation to enable capability use in their own implementations and for the software that they host.
- 1980s C code that predates intptr_t and, for better or worse reasons, confuses pointer and integer types. This may require changes to use intptr_t – and in some cases can cause measurable friction, which arises through greater %LoC change, but is not usually technically challenging to resolve.
- Almost all other more contemporary C and C++ code requires little or no change. This has included quite significant pieces of software including the Wayland display server, git version control system, and significant portions of KDE.

In our 2021 desktop pilot study, we adapted roughly 6MLoC of appliction-level C and C++ code to CHERI memory safety in three staff months. This required an average of 0.026% LoC change across the full corpus. This rate of change is substantially lower than the 1.4% LoC change for the FreeBSD kernel, which required significantly greater capability awareness.

Changes of particular interest in lower-level aspects of the C runtime include adaptation of the run-time linker to initialize capabilities in the GOT and PLT, and to the heap allocator to set bounds, rederive pointers on free() rather than reach outside of bounds to find metadata, and interface with common temporal safety code for quarantining and revocation.

Through programs such as Innovate's Digital Security by Design (DSbD), which co-funded creation of Morello, over 40 companies have trialed CHERI C/C++. The results have been extremely positive, with low overheads to learn about CHERI C/C++ and adapt software to them, as well as multiple reports of discovered vulnerabilities when adapting code to memory safety. This latter effect was not expected: CHERI was designed to mitigate vulnerabilities, not to act as a debugging tool. However, it appears to prove quite effective at this; we hypothesize that this is because of its performance as compared to common memory-safety debugging tools such as ASAN, permitting larger-scale testing with memory safety.

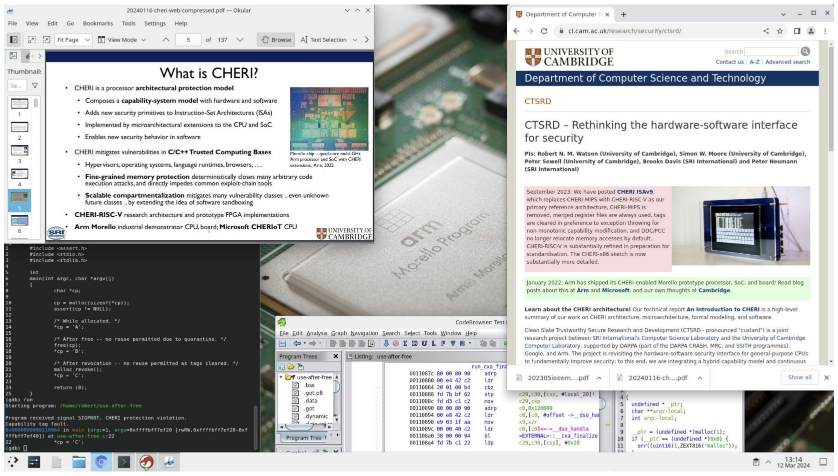
A RICH MEMORY-SAFE C/C++ ECOSYSTEM

Morello has provided a high-productivity environment for CHERI software experimentation at scale. The CheriBSD/Morello operating system (version 23.11) ships with:

- A referentially and spatially safe version of the FreeBSD OS kernel, as well as a referentially, spatially, and temporally safe version of the FreeBSD userspace.
- Roughly 10,000 memory-safe third-party software packages, which can be contrasted with roughly 24,000 "aarch64" (CHERI-unaware 64-bit Arm) software packages also supported on the platform. There is considerable variability in understanding regarding the quality of the 10,000 packages – some come with extensive test suites that pass with flying colors (e.g., Qt), and are used daily by our team, whereas others are known to compile but little more.
- Features such as userlevel library compartmentalization (built on memory safety) and CHERI-enabled guests virtual machines (using an adapted version of FreeBSD's bhyve hypervisor).

Adapted packages include software such as the KWin Wayland compositor, KDE Plasma desktop environment, nginx web server, Postgres database, git version control tool, and thousands of other open-source components (including all library dependencies for those we have named). Notable omissions include several essential language runtimes such as the OpenJDK and V8 Javascript runtime (although this is the target of ongoing development), the MySQL database, and web browsers such as Firefox and Chromium due to language runtime dependencies. These are, however, supported via CHERI's ability to support legacy – in this case 64-bit Arm – binaries, permitting a complete desktop or server environment even if not all components are yet adapted for memory safety.

We continue to work to expand the memory-safe


20240116-cheri-web-compressed.pdf — Okular
File View Edit Go Bookmarks Tools Settings Help
Fit Page
View Mode
5 of 137
Browse
Text Selection
Thumbnails
Se...

What is CHERI?

- CHERI is a processor **architectural protection model**
  - Composes a **capability-system model** with hardware and software
  - Adds new security primitives to Instruction-Set Architectures (ISAs)
  - Implemented by microarchitectural extensions to the CPU and SoC
  - Enables new security behavior in software
- CHERI mitigates vulnerabilities in **C/C++ Trusted Computing Bases**
  - Hypervisors, operating systems, language runtimes, browsers, ….
  - **Fine-grained memory protection** deterministically closes many arbitrary code execution attacks, and directly impedes common exploit-chain tools
  - **Scalable compartmentalization** mitigates many vulnerability classes .. even unknown future classes .. by extending the idea of software sandboxing
- **CHERI-RISC-V** research architecture and prototype FPGA implementations
- **Arm Morello** industrial demonstrator CPU, board; **Microsoft CHERIoT** CPU

Morello chip – quad-core multi-GHz Arm processor and SoC with CHERI extensions, Arm, 2022.

```
1       #include <assert.h>
2       #include <stdio.h>
3       #include <stdlib.h>
4
5       int
6       main(int argc, char *argv[])
7       {
8               char *cp;
9
10              cp = malloc(sizeof(*cp));
11              assert(cp != NULL);
12
13              /* While allocated. */
14              *cp = 'A';
15
16              /* After free -- no reuse permitted due to quarantine. */
17              free(cp);
18              *cp = 'B';
19
20              /* After revocation -- no reuse permitted as tags cleared. */
21              malloc_revoke();
22              *cp = 'C';
23
24              return (0);
25      }
(gdb) run
Starting program: /home/robert/use-after-free

Program received signal SIGPROT, CHERI protection violation.
Capability tag fault.
0x0000000000110964 in main (argc=1, argv=0xffffbff7ef20 [rwRW,0xffffbff7ef20-0xf
fffbff7ef40]) at use-after-free.c:22
22              *cp = 'C';
(gdb) 
```

```
                              run_cxa_fina
0011087c 80 00 80 90     adrp
00110880 00 e4 42 c2     ldr
00110884 20 01 00 b4     cbz
00110888 fd 7b bf 62     stp        c29,c30,[csp, #local_20]!
0011088c fd d3 c1 c2     mov        c29,csp
00110890 80 00 80 90     adrp       c0,0x120000
00110894 00 e8 42 c2     ldr        c0,[c0, #offset ->__dso_han
00110898 e9 03 1f aa     mov        x9,xzr
0011089c 00 00 40 c2     ldr        c0,[c0]=>->__dso_handle
001108a0 38 00 00 94     bl         <EXTERNAL>::__cxa_finalize
001108a4 fd 7b c1 22     ldp        c29,c30,[csp], #0x20
```

```
{
  undefined *__ptr;
  char **argv-local;
  int argc-local;

  __ptr = (undefined *)malloc(1);
  if (__ptr == (undefined *)0x0) {
    err((uint16)1,ZEXT816("malloc"));
```

Department of Computer Science and Technology

CTSRD

CTSRD – Rethinking the hardware-software interface for security

PIs: Robert N. M. Watson (University of Cambridge), Simon W. Moore (University of Cambridge), Peter Sewell (University of Cambridge), Brooks Davis (SRI International) and Peter Neumann (SRI International)

September 2023: We have posted **CHERI ISAv9**, which replaces CHERI-MIPS with CHERI-RISC-V as our primary reference architecture, CHERI-MIPS is removed, merged register files are always used, tags are cleared in preference to exception throwing for non-monotonic capability modification, and DDC/PCC no longer relocate memory accesses by default. CHERI-RISC-V is substantially refined in preparation for standardisation. The CHERI-x86 sketch is now substantially more detailed.

January 2022: Arm has shipped its CHERI-enabled Morello prototype processor, SoC, and board! Read blog posts about this at **Arm** and **Microsoft**, and our own thoughts at **Cambridge**.

Learn about the CHERI architecture! Our technical report **An Introduction to CHERI** is a high-level summary of our work on CHERI architecture, microarchitecture, formal modeling, and software.

Clean Slate Trustworthy Secure Research and Development (CTSRD - pronounced "custard") is a joint research project between SRI International's Computer Science Laboratory and the University of Cambridge Computer Laboratory, supported by DARPA (part of the DARPA CRASH, MRC, and SSITH programmes), Google, and Arm. The project is revisiting the hardware-software security interface for general-purpose CPUs to fundamentally improve security; to this end, we are integrating a hybrid capability model and continuous

FIGURE 3. Memory-safe KDE Plasma desktop environment running on an Arm Morello board.

software corpus, both to experimentally validate our hypotheses about CHERI memory safety, and also to build develop an increasingly usable memory-safe software ecosystem.

EVALUATING SECURITY IMPACT

Evaluating security is a challenging problem that does not lend itself to simple metrics. To date, we have relied on four forms of security argument in our work on CHERI:

Analytic or experimental approaches that enumerate and consider potential vulnerabilities and attack vectors: A CWE-centric analysis is productive, but most such analysis is performed at the whiteboard; and there may be disconnects between a language-level view and that of compiler code generation, especially with respect to optimization.

Retrospective vulnerability analyses that explore how CHERI would have impacted past vulnerabilities: Not only have we performed our own studies (e.g., in the 2021 desktop study), but there have also been independent studies (e.g., as performed by Microsoft looking at all of their 2019 critical security vulnerabilities). The notion of mitigation is itself non-trivial; for example, with respect to determining threat models when composing libraries and applications, and also whether deterministically crashing is sufficient to consider a vulnerability mitigated. However, the results have been compelling: MSRC reported an over two-thirds deterministic mitigation rate for memory-safety vulnerabilities with the deployment of CHERI referential, spatial, and temporal memory safety.

Formal proof of architectural security properties: Formal modeling of the Morello and CHERI-MIPS ISAs has supported formal verification (machine-checked mathematical proof) that the ISAs enforce key properties, such as correctness of capability bounds comparison and isolation of arbitrary code by compartmentalization mechanisms[12], and formal semantics for CHERI C has clarified its security properties[13].

Penetration testing exercises, ideally performed with a strong attacker awareness of the CHERI model so that attack strategies can take this into account: These exercisers have primarily been performed externally, and include an activity by MSRC to consider the impact of CHERI on WebKit JSC with CHERI-aware attackers, as well as a DARPA-sponsored, crowd-sourced penetration activity.[14] The former lent considerable insight and informed our decision to use sealed capabilities for all control-flow pointers, and the latter identified two software TCB vulnerabilities that a non-prototype implementation will need to avoid.

EARLY RESULTS FROM ARM MORELLO

Assessing the performance impact of new architectural features is challenging for many reasons, including:

- Research tools such as microarchitectural simulators (e.g., Gem5) and simplified FPGA implementations have limited fidelity and may permit considerable indiscretion in terms of realism for prototype changes.
- Even complete microarchitectural prototypes developed at considerable expense, and fabricated as test chips (such as Morello) will be first-

generation implementations that lack the benefit of a multi-year optimization cycle. This is especially tricky where there may be disproportionate degrees of optimization within a design – e.g., if you extend the Neoverse N1, designed for 64-bit addresses, CHERI aspects of the design will be substantially less mature than the implementation of the 64-bit baseline.
- Compiler optimization for new architecture and microarchitecture can take years to complete, as running rich workloads requires high-performance implementations.
- Even if overheads are known and well characterized, the degree to which an overhead is *acceptable* is far more a marketing and financial consideration than a technical one. This is especially true as many dynamic overheads may be mitigated through additional area or power, and the acceptability of area and power changes is similarly fraught.

Despite these limitations, it is important to characterize overhead to the extent possible, and significant effort is being invested to assess the performance (and other) impacts of CHERI on a variety of microarchitectures. Arm Morello is of particular interest as it is able to, for the first time, allow us to run very large software stacks on a CHERI-extended system. The key conclusions to date have been:[15]

- The Armv8.2-A ISA was extended with CHERI support without difficulty, although there remains considerable room for quantitative optimization around instruction and opcode selection.
- Morello has achieved its goal in enabling a large software ecosystem and research community, including approaching 100MLoC of memory-safe C and C++ code, and also allowing dozens of universities and over 40 companies to engage with CHERI.
- The Morello design met or exceeded its goals for frequency optimization, permitting the creation of a 2.5GHz 7nm design and achieving a low area overhead (<6% in the core cluster of the SoC, consisting of CPU cores and caches). Key concerns lay in integration with the MMU, expanded bus widths, capability bounds compression, and tag-related changes, all of which were overcome.
- Design tradeoffs arising from an abbreviated prototyping period (one year) for Morello mean that performance results fall into two categories: (1) best available performance measured on the SoC or using revised RTL running on FPGA cluster; and (2) estimate performance for future microarchitectures taking into account optimization lessons:
  - With respect to (1), using an FPGA implementation with post-tapeout re-tuning (e.g., of store-queue sizes), and compensating through ABI change for a microarchitectural issue affecting capability bounds prediction, the best measured overhead for a CHERI-adapted, spatially protected subset of SPECint 2006 running on Morello was 5.7%.
  - With respect to (2), measured using a 128-bit pointer compilation to the 64-bit ISA on the Neoverse N1 microarchitecture, the estimated overhead range for a more mature implementation was between 1.82% and 2.98%.

It is worth noting that these performance results are also constrained by our limited work on compiler optimization to date; further improvements should be possible.

Detailed presentation and explanation of these results can be found in the report *Early performance results from the prototype Morello microarchitecture*, published by Arm and the University of Cambridge.[15]

FUTURE WORK

We have been working on CHERI for over 13 years, performing a long-term hardware-software-semantics co-design cycle that has led to a memory-safety model that is architecturally and micro-architecturally viable across a range of processor designs and ISAs, enables strong memory protection within the C and C++ programming languages, and achieves low performance overheads. There are significant efforts in flight to achieve initial commercial deployment of CHERI within “vertically integrated systems” – for example, in roots-of-trust within larger SoC designs. We hope to see CHERI-based commercial implementations in use in 2024-2025. However, much remains to be done both in terms of concluding aspects of the research and also achieving broader transition. Perhaps most critical are:

- Continuing to refine our understanding of memory safety for C and C++ – in terms of meaning and also evaluation at scale.
- Pushing beyond memory safety to CHERI-enabled software compartmentalization, which strengthens the adversary model from “imperfect programmer” to “malicious programmer”.

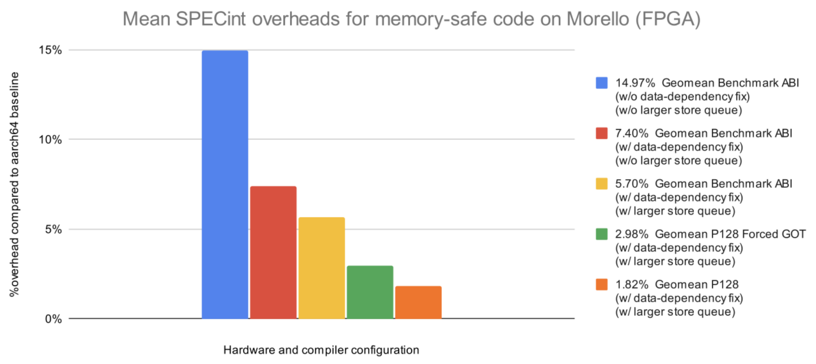


FIGURE 4. Early SPECint 2006 performance results running over Morello RTL on an FPGA cluster. Microarchitectural re-tuning has improved performance relative to the taped out Morello SoC, but estimated performance using 128-bit pointer compilation on the same microarchitecture suggests that substantial further optimization is possible.

- Continuing to pursue performance results and improvements as experience is gained with CHERI in industrial microarchitectures.
- Standardizing CHERI-RISC-V, our integration of CHERI with the open-source RISC-V ISA.
- Fleshing out the growing open-source software ecosystem to include further software, including firmware, additional OSes, software libraries, language runtimes, and applications.
- Exploring potential opportunities to compose CHERI with memory- and type-safe programming languages, such as Rust, which bring their own notions of memory safety, adversary models, strengths, and weaknesses.

CONCLUSION

After over a decade of research and the investment of hundreds of research and engineering staff years, CHERI is demonstrating strong viability as a technology to rapidly deploy memory safety in the large extant C/C++ software corpus – estimated at over 12bn lines of open-source code alone. With multiple commercial implementations in flight, dozens of companies and universities exploring trial deployments, and an increasingly rich software stack, there is growing evidence that the risky approach of hardware-software-semantics co-design and more disruptive architectural changes may be able to achieve transition. To learn more, we invite you to visit the CHERI project website: cheri-cpu.org.

ACKNOWLEDGMENTS

We gratefully acknowledge contributiond and feedback from the broader CHERI team, including Hesham Almatary, John Baldwin, Ruslan Bukin, Graeme Jenkinson, Alexandre Joannou, Mark Johnston, A. Theodore Markettos, Alfredo Mazzinghi, Edward Napierala, George Neville-Neil, and Peter Rugg.

Distribution Statement A: Approved for public release; distribution is unlimited. Sponsored in part by the Defense Advanced Research Projects Agency (DARPA) and the Air Force Research Laboratory (AFRL), under contract FA8750-10-C-0237 ("CTSRD"), with additional support from FA8750-11-C-0249 ("MRC2"), HR0011-18-C-0016 ("ECATS"), FA8650-18-C-7809 ("CIFV"), HR0011-22-C-0110 ("ETC"), HR0011-23-C-0031 ("MTSS"), and FA8750-24-C-B047 ("DEC") as part of the DARPA CRASH, MRC, SSITH, and CPM research programs. 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, either expressed or implied, of the Department of Defense or the U.S. Government.

This work was supported in part by the Innovate UK project Digital Security by Design (DSbD) Technology Platform Prototype, 105694, as will as Innovate UK projects 107145 and 10027440. 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). We further acknowledge the EPSRC REMS Programme Grant (EP/K008528/1) and the EPSRC CHaOS Grant (EP/V000292/1). Additional support was received from Arm, Google, and Microsoft.

REFERENCES

1. L. Szekeres, M. Payer, T. Wei, and D. Song. "SoK: Eternal War in Memory," Proceedings of the IEEE Symposium on Security and Privacy, Washington, DC, USA, May 19-22 2013, 10.1109/SP.2013.13.
2. R. N. M. Watson, P. G. Neumann, J. Woodruff, M. Roe, H. Almatary, J. Anderson, J. Baldwin, G. Barnes, D. Chisnall, J. Clarke, B. Davis, L. Eisen, N. W. Filardo, F. A. Fuchs, R. Grisenthwaite, A. Joannou, B. Laurie, A. T. Markettos, S. W. Moore, S. J. Murdoch, K. Nienhuis, R. Norton, A. Richardson, P. Rugg, P. Sewell, S. Son, and H. Xia. "Capability Hardware Enhanced RISC Instructions: CHERI Instruction-Set Architecture (Version 9)," Technical Report UCAM-CL-TR-987, Computer Laboratory, September 2023.
3. H. M. Levy, "Capability-Based Computer Systems," Butterworth-Heinemann, Newton, MA, USA, 1984.
4. R. Grisenthwaite, G. Barnes, R. N. M. Watson, S. W. Moore, P. Sewell, and J. Woodruff. "The Arm Morello Evaluation Platform—Validating CHERI-Based Security in a High-Performance System", IEEE Micro, vol. 43, no. 3, pp. 50-57, May-June 2023, doi: 10.1109/MM.2023.3264676.
5. S. Amar, D. Chisnall, T. Chen, N. W. Filardo, B. Laurie, K. Liu, R. Norton, S. W. Moore, Y. Tao, R. N. M. Watson, and H. Xia. "CHERIoT: Complete Memory Safety for Embedded Devices," Proceedings of the 56th IEEE/ACM International Symposium on Microarchitecture (MICRO 2023). Toronto, Canada, October 28-November 1 2023, doi: 10.1145/3613424.3614266.
6. Y. Bae, Y. Kim, A. Askar, J. Lim, and T. Kim. "Rudra: Finding Memory Safety Bugs in Rust at the Ecosystem Scale," Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles (SOSP 2021). October 26-29, 2021, doi: 10.1145/3477132.3483570.
7. R. N. M. Watson, A. Richardson, B. Davis, J. Baldwin, D. Chisnall, J. Clarke, N. F., S. W. Moore, E. Napierala, P. Sewell, and P. G. Neumann. "CHERI C/C++ Programming Guide," Technical Report UCAM-CL-TR-947, Computer Laboratory, June 2020.
8. R. N. M. Watson, B. Laurie, and A. Richardson. "Assessing the Viability of an Open-Source CHERI Desktop Software Ecosystem," Technical Report, Capabilities Limited, 17 September 2021.
9. N. W. Filardo, B. F. Gutstein, J. Woodruff, J. Clarke, P. Rugg, B. Davis, M. Johnston, R. Norton-Wright, D. Chisnall, S. W. Moore, P. G. Neumann, and R. N. M. Watson. "Load Barriers for CHERI Heap Temporal Safety," Proceedings of ACM International Conference on Architectural Support for Programming Languages and Operating Systems. San Diego, CA, USA, April 27 - May 1, 2024, doi: 10.1145/3620665.3640416.
10. D. Chisnall, C. Rothwell, R. N.M. Watson, J. Woodruff, M. Vadera, S. W. Moore, M. Roe, B. Davis, and P. G. Neumann. "Beyond the PDP-11: Architectural support for a memory-safe C abstract machine," Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2015), Istanbul, Turkey, March 2015, doi: 10.1145/2775054.2694367.
11. B. Davis, R. N. M. Watson, A. Richardson, P. G. Neumann, S. W. Moore, J. Baldwin, D. Chisnall, J. Clarke, N. W. Filardo, K. Gudka, A. Joannou, B. Laurie, A. T. Markettos, J. E. Maste, A. Mazzinghi, E. T. Napierala, R. M. Norton, M. Roe, P. Sewell, S. Son, and J. Woodruff. "CheriABI: Enforcing Valid Pointer Provenance and Minimizing Pointer Privilege in the POSIX C Run-time Environment," In Proceedings of 2019 Architectural Support for Programming Languages and Operating Systems (ASPLOS'19). Providence, RI, USA, April 13-17, 2019, doi: 10.1145/3297858.3304042.
12. T. Bauereiss, B. Campbell, T. Sewell, A. Armstrong, L. Esswood, I. Stark, G. Barnes, R. N. M. Watson, and P. Sewell. "Verified Security for the Morello Capability-enhanced Prototype Arm Architecture", 31st European Symposium on Programming (ESOP 2022), May 2022, doi: 10.1007/978-3-030-99336-8_7.
13. V. Zaliva, K. Memarian, R. Almeida, J. Clarke, B. Davis, A. Richardson, D. Chisnall, B. Campbell, I. Stark, R. N. M. Watson, P. Sewell. "Formal Mechanised Semantics of CHERI C: Capabilities, Provenance, and Undefined Behaviour", in Proceedings of 2024 Architectural Support for Programming Languages and Operating Systems (ASPLOS'24).
14. N. Joly, S. ElSherei, and S. Amar. "Security analysis of CHERI ISA," Microsoft Security Response Center (MSRC), October 2020.
15. R. N. M. Watson, J. Clarke, P. Sewell, J. Woodruff, S. W. Moore, G. Barnes, R. Grisenthwaite, K. Stacer, S. Baranga, A. Richardson. "Early performance results from the prototype Morello microarchitecture", Technical Report UCAM-CL-TR-986, Computer Laboratory, September 2023.

Robert N. M. Watson is Professor of Systems, Security, and Architecture at the Department of Computer Science and Technology at the University of Cambridge, Cambridge, UK. Watson received his PhD in Computer Science from the University of Cambridge. His research interests include security, operating systems, compilers, and computer architecture. Contact him at robert.watson@cl.cam.ac.uk.

David Chisnall is Director of System Architecture at SCI Semiconductor in Cambridge, UK. Chisnall received his PhD in Computer Science at Swansea University. His research interests include programming languages, compilers, security, and computer architecture. Contact him at david.chisnall@cl.cam.ac.uk.

Jessica Clarke is a PhD student in the Department of Computer Science and Technology at the University of Cambridge, Cambridge, UK. Clarke received her Masters in Computer Science from the University of Cambridge. Her research interests include compilers, architecture, and microarchiteture. Contact her at jessica.clarke@cl.cam.ac.uk

Brooks Davis is a Principal Computer Scientist in the Computer Science Laboratory at SRI International, Menlo Park, CA, USA. Davis received his BS in Computer Science from Harvey Mudd College. His research interests include operating systems and security. Contact him at brooks.davis@sri.com.

Nathaniel Wesley Filardo is a Senior Researcher at Microsoft Canada. He received his PhD in Computer Science from Johns Hopkins University. His research interests include computer architecture, programming languages, and operating systems. Contact him at nfilardo@microsoft.com.

Ben Laurie is a Principal Engineer at Google, London, UK, and is a visiting fellow at the University of Cambridge. His research interests include security, privacy, and hardware/software co-design. Contact him at benl@google.com.

Simon W. Moore is Professor of Computer Engineering at the Department of Computer Science and Technology at the University of Cambridge, Cambridge, UK. Moore received his PhD in Computer Science from the University of Cambridge. His research interests include computer design and security. Contact him at simon.moore@cl.cam.ac.uk.

Peter G. Neumann is Chief Scientist at the Computer Science Laboratory at SRI International, Menlo Park, CA, USA. Neumann received his PhD in Applied Mathematics from Harvard University and a Dr rerum naturalium in Math and Physics, Darmstadt, Germany. Contact him at neumann@csl.sri.com.

Alexander Richardson is a Senior Software Engineer at Google in Mountain View, CA, USA. Richardson received his PhD in Computer Science from the University of Cambridge. His research interests include compilers, operating systems, and security. Contact him at alexrichardson@google.com.

Peter Sewell is Professor of Computer Science at the University of Cambridge, Cambridge, UK. Sewell received his PhD in Computer Science from the University of Edinburgh. His research interests include formal modeling and verification for real-world systems, including operating systems and computer architecture. Contact him at peter.sewell@cl.cam.ac.uk.

Konrad Witaszczyk is a PhD student in the Department of Computer Science and Technology at the University of Cambridge in Cambridge, UK. His research interests include memory-safe software adaptability and compartmentalized operating-system kernels. Contact him at konrad.witaszczyk@cl.cam.ac.uk.

Jonathan Woodruff is a Senior Research Associate in the Department of Computer Science and Technology at the University of Cambridge in Cambridge, UK. His research interests include architectural and microarchitectural support for security. Contact him at jonathan.woodruff@cl.cam.ac.uk.