Technical Report

UCAM-CL-TR-1010
ISSN 1476-2986

Number 1010



UNIVERSITY OF CAMBRIDGE

Computer Laboratory

Efficient linkage-based compartmentalization on CHERI

Dapeng Gao, John Baldwin, Jessica Clarke, Nicholas C. Connolly, Brooks Davis, Franz A. Fuchs, Alfredo Mazzinghi, Daniel Moghimi, Peter Rugg, Domagoj Stolfa, Konrad Witaszczyk, Simon W. Moore, Robert N. M. Watson

November 2026

15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom
phone +44 1223 763500

https://www.cl.cam.ac.uk/

© 2026 Dapeng Gao, John Baldwin, Jessica Clarke, Nicholas C. Connolly, Brooks Davis, Franz A. Fuchs, Alfredo Mazzinghi, Daniel Moghimi, Peter Rugg, Domagoj Stolfa, Konrad Witaszczyk, Simon W. Moore, Robert N. M. Watson

Technical reports published by the University of Cambridge Computer Laboratory are freely available via the Internet:

https://www.cl.cam.ac.uk/techreports/

ISSN 1476-2986
DOI *https://doi.org/10.48456/tr-1010*

Efficient Linkage-Based Compartmentalization on CHERI

DAPENG GAO, University of Cambridge, UK
JOHN BALDWIN, Ararat River Consulting, USA
JESSICA CLARKE, University of Cambridge, UK
NICHOLAS C. CONNOLLY, Arm Limited, UK
BROOKS DAVIS*, Capabilities Limited, UK
FRANZ A. FUCHS, University of Cambridge, UK
ALFREDO MAZZINGHI†, University of Cambridge, UK
DANIEL MOGHIMI, Google, USA
PETER RUGG†, University of Cambridge, UK
DOMAGOJ STOLFA‡, University of Cambridge, UK
KONRAD WITASZCZYK, University of Cambridge, UK
SIMON W. MOORE, University of Cambridge, UK
ROBERT N. M. WATSON†, University of Cambridge, UK

We present an efficient *linkage-based* model for in-process compartmentalization built on CHERI memory safety, which enables fine-grained compartmentalization of the entire UNIX user-space, scaling to 10K+ compartments on desktop systems. The model's "push-button" compartmentalization along existing library boundaries regularly hosts 500+ compartments per process for large applications such as Chromium, far exceeding the number of concurrently available protection domains supported by other mechanisms (e.g., up to 16 for Intel MPK).

*Also with University of Cambridge.
†Also with Capabilities Limited.
‡Also with MSB Associates.

Authors' Contact Information: Dapeng Gao, University of Cambridge, Cambridge, UK, dapeng.gao@cl.cam.ac.uk; John Baldwin, Ararat River Consulting, Ashland, VA, USA, john@araratriver.co; Jessica Clarke, University of Cambridge, Cambridge, UK, jessica.clarke@cl.cam.ac.uk; Nicholas C. Connolly, Arm Limited, Cambridge, UK, nick.connolly@arm.com; Brooks Davis, Capabilities Limited, Llanwrda, UK, brooks@capabilitieslimited.co.uk; Franz A. Fuchs, University of Cambridge, Cambridge, UK, franz.fuchs@cl.cam.ac.uk; Alfredo Mazzinghi, University of Cambridge, Cambridge, UK, alfredo.mazzinghi@cl.cam.ac.uk; Daniel Moghimi, Google, Mountain View, CA, USA, danielmm@google.com; Peter Rugg, University of Cambridge, Cambridge, UK, peter.rugg@cl.cam.ac.uk; Domagoj Stolfa, University of Cambridge, Cambridge, UK, domagoj.stolfa@cl.cam.ac.uk; Konrad Witaszczyk, University of Cambridge, Cambridge, UK, konrad.witaszczyk@cl.cam.ac.uk; Simon W. Moore, University of Cambridge, Cambridge, UK, simon.moore@cl.cam.ac.uk; Robert N. M. Watson, University of Cambridge, Cambridge, UK, robert.watson@cl.cam.ac.uk.



This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License.

Custom policies can further subdivide libraries. Of the thousands of C/C++ programs tested, only the V8 JavaScript engine required source-level adaptation (<300 lines of changed code concerning garbage collection and JIT compilation).

We implement the model for CHERI-extended versions of Armv8-A and RISC-V through support in the compiler toolchain and operating system. Case studies illustrate the smooth delegation of memory between compartments, compartment-aware debugging and visualization, as well as extensibility to a complex managed language runtime, demonstrating the benefits of our single-address-space model. We evaluate using multiple processors, including Arm's superscalar Morello and, notably, the first commercial CHERI-enabled RISC-V application core—Codasip's in-order dual-issue X730.

1 Introduction

Compartmentalization solutions are fundamentally constrained by the available architectural primitives. Techniques designed for commodity hardware often rely on features that were not originally intended for security, leading to programming models that are awkward to develop for, let alone to test and debug. Although innovations in programming languages have lessened these difficulties, they introduce new friction points, such as the high cost of retrofitting existing code, performance degradation due to stylized code generation, and vastly expanded toolchain dependencies.

Expanding the design space of compartmentalization through architectural extensions, which has often been considered prohibitively expensive, now seems increasingly attractive to system vendors. For example, Intel Memory Protection Keys (MPK) has enabled a family of efficient in-process compartmentalization systems by allowing user-space code to switch access permissions for groups of pages without kernel intervention, as discussed in Section 9. However, many designs based on such extensions require static assignment (or costly dynamic reassignment) of protection to memory regions. This makes them less suitable for *data-dependent* sharing, which can require a combinatorial number of configurations: in our case study in Section 7.1, tcpdump routes packets among 100+ dissectors based on their contents, yielding $\gg 2^{100}$ possible access-control configurations. To tackle this and other limitations, we set out to explore the alternative of building compartmentalization upon a *capability system*—CHERI [50, 52].

CHERI extends pointers in existing ISAs to rich *capabilities* that encode bounds and permissions in addition to integer memory addresses. Since its inception more than a decade ago, OS and hardware support for it have evolved significantly. On the CheriBSD operating system, CheriABI [10] elevates capabilities to first-class citizens in a UNIX environment, making it practical to run a substantial corpus of POSIX programs with fine-grained spatial memory safety. CHERI heap temporal safety [14, 51, 53] has also matured sufficiently to be enabled by default on CheriBSD. Arm developed the Morello prototype [19], featuring a CHERI-enabled 2.5 GHz quad-core processor based on the Neoverse N1 that enables large-scale experiments. Today, the CHERI ecosystem consists of a rich stack of memory-safe software, including a desktop environment, web servers, and more than 10K C/C++ packages [47].

Building upon these advances, this work provides an efficient mechanism for developers to *incrementally* enforce domain-specific compartmentalization policies. It also serves as a foundation for future policy construction and auditing techniques, as illustrated in Figure 1 and discussed in Section 8. In summary, we:

- Present a model for compartmentalizing dynamically linked programs within a UNIX process, comprising:
  a) Linkage-based policy representation designed for integration with existing compiler toolchains and build systems, and
  b) Efficient CHERI-based enforcement of security guarantees, integrated with existing OS components to preserve feature compatibility with minimal source-code disruption.
- Demonstrate that the model supports realistic software development workflows through compiler toolchain and debugger integration, and introduce CheriTree, a novel tool for tracing and visualizing run-time capability reachability.
- Implement the model for CHERI-extended versions of Armv8-A and RISC-V, and evaluate its performance across diverse applications, workloads, and processors.

In comparison to prior compartmentalization designs, including CHERI-based ones, our model contains the following key novelties:

- Dynamic linking, underpinned by CHERI's capability-based memory safety, enables almost transparent in-address-space compartmentalization.
- This technique applies to complex constructs such as callbacks, C++ exceptions, JIT compilation, and garbage collection.
- The model scales efficiently to hundreds of compartments on multi-million-line applications with minimal source-code changes.
- The model integrates compartment-aware debugging, tracing, and visualization into established development workflows.

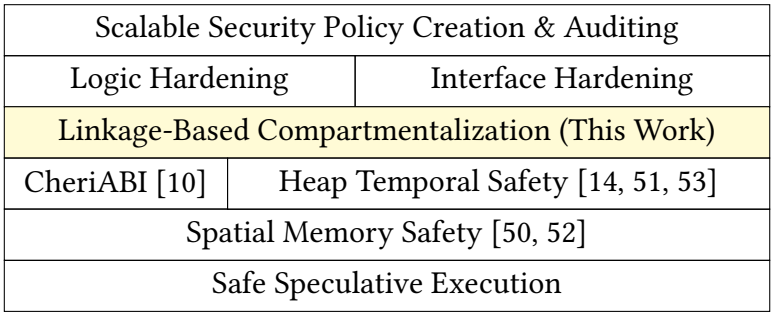


Fig. 1. Envisioned stack of protections enabled by CHERI.

2 A CHERI Primer

This section presents a simplified view of the CHERI ISA [49]. Figure 2 illustrates how CHERI represents a C pointer, which can only be used to access a bounded region of memory with restricted privileges. On a 64-bit implementation, a capability is represented as a 128-bit word in memory. Its lower 64 bits contain the address, and the upper bits encode metadata including bounds and permissions. The bounds are compressed like floating-point numbers, which leads to more rounding for wider bounds. Some bits are reserved. The register file is widened to hold capabilities, although it is only possible to perform arithmetic on the lower 64 bits of any register.

Valid capabilities are distinguished from plain data down to the hardware level. The hardware tracks whether each register and each 128-bit aligned word in memory contains a valid capability or not through a 1-bit tag associated with each location. Each tag bit is always propagated together with its associated capability throughout

the register file and memory hierarchy, providing atomicity. New capabilities can be produced by special capability manipulation instructions, which guarantee the following invariant:

Monotonicity No sequence of instructions can derive a capability that is not a subset[1] of an existing capability.

This implies the following property:

Unforgeability No valid capability can be created "out of thin air". Instead, it must be derived from other valid capabilities.

Consequently, compared to the linear, integer-addressable virtual address space on conventional systems, CHERI offers an alternative perspective of memory as a *graph* of bounded regions reachable from the register file via a chain of capability dereferences. This approach is orthogonal to existing MMU-based, page-granular memory management but is much more dynamic and finer-grained.

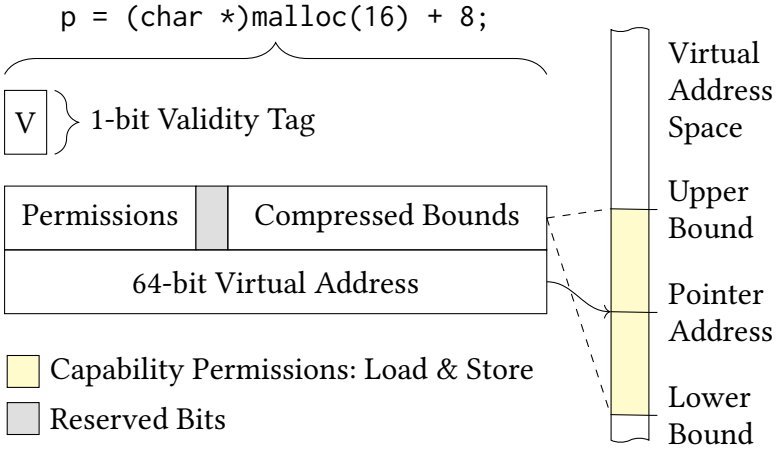


Fig. 2. `malloc` returns a capability authorizing read and write permissions to a 16-byte allocation. Incrementing the capability by 8 changes the pointed-to address while retaining the original bounds and permissions.

Sealed entry capabilities (also known as *sentries*) are a special type of immutable and non-dereferenceable capability. Only when installing it as the program counter via an indirect jump(-and-link) instruction can the processor *unseal* a sentry into a mutable and dereferenceable form. In CheriABI, function pointers are always sealed upon creation, and jump-and-link instructions always seal the return pointer. Indirect control transfers through function pointers therefore begin at their sentry-designated entry points. Sentries are also useful as opaque data pointers and unforgeable tokens.

3 Abstract Design

Threat Model. Linkage-based compartmentalization applies to dynamically linked user-space programs in the UNIX environment, where the OS kernel, dynamic linker, C/C++ runtime, threading library, and a few highly privileged standard library functions

[1]A valid capability c_1 is a subset of capability c_2 when c_2 is valid, and the bounds and permissions of c_1 are subsets of those of c_2. For the sake of this definition, invalid capabilities are viewed as "bottom" elements that are trivially a subset of any capability.

form the Trusted Computing Base (TCB).[2] Attackers can exploit existing vulnerabilities in application code and insert backdoors through supply-chain attacks. However, the compiler toolchain used to build all application code is trusted to produce ELF files that a) are well-formed and do not trigger parsing errors and b) faithfully apply the compartmentalization policy (see Section 3.1), effectively excluding supply-chain attacks on build systems. The hardware platforms are assumed to meet the CHERI ISA specification, whose soundness has been subject to formal verification [5, 34]. Micro-architectural side-channel attacks [24, 29] and physical attacks are out of scope.

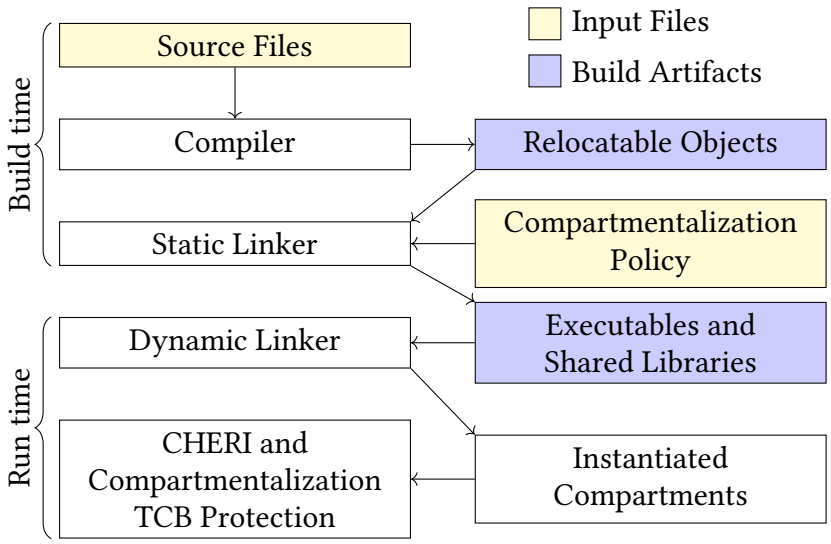


Fig. 3. Pipeline of linkage-based compartmentalization.

Synopsis. Our model operates in two phases as depicted by Figure 3. During *build time*, source files are compiled into relocatable objects, which are linked into executables or shared libraries. Developers can supply the static linker with a *compartmentalization policy* that dictates compartment composition and symbol accessibility in the output binaries (see Section 3.1). During *run time*, the dynamic linker loads these binaries into memory and relocates them according to their policy-approved ELF section layouts and dynamic relocations. Programs then execute under both standard CHERI memory safety protections and additional security guarantees provided by the compartmentalization TCB (see Section 3.2).

3.1 Compartmentalization Policy

We design our model to support effortless adoption and incremental refinement. With no explicit policy, the executable and shared libraries form separate compartments, requiring essentially no developer effort. With domain knowledge, developers can draw more effective security boundaries *incrementally* by (a) using *placement rules* to partition libraries into *sub-libraries* and (b) using an *access-control list* to authorize inter-compartment symbol references.

[2]The TCB can itself be further hardened through compartmentalization: C++ exception handling already occurs separately from the C++ standard library, and we have begun isolating privileged `libc` components such as `malloc` and `stdio`.

Placement Rules. Shared libraries can sometimes be divided into smaller isolation units. Suppose library A uses I/O only to log to one file. Rather than granting its compartment full I/O privileges, developers can move the file-specific logging functions to compartment B and expose only them to A. Compromising A then permits corruption of only the designated log file rather than arbitrary files.

Instead of manually splitting libraries, which would add major complications to build systems and package distribution, developers can refine the default compartments by supplying *placement rules* to the compiler toolchain to emit a single shared library file that contains multiple *sub-libraries*. These rules describe a) the name of each sub-library and b) the symbols that belong to each sub-library.

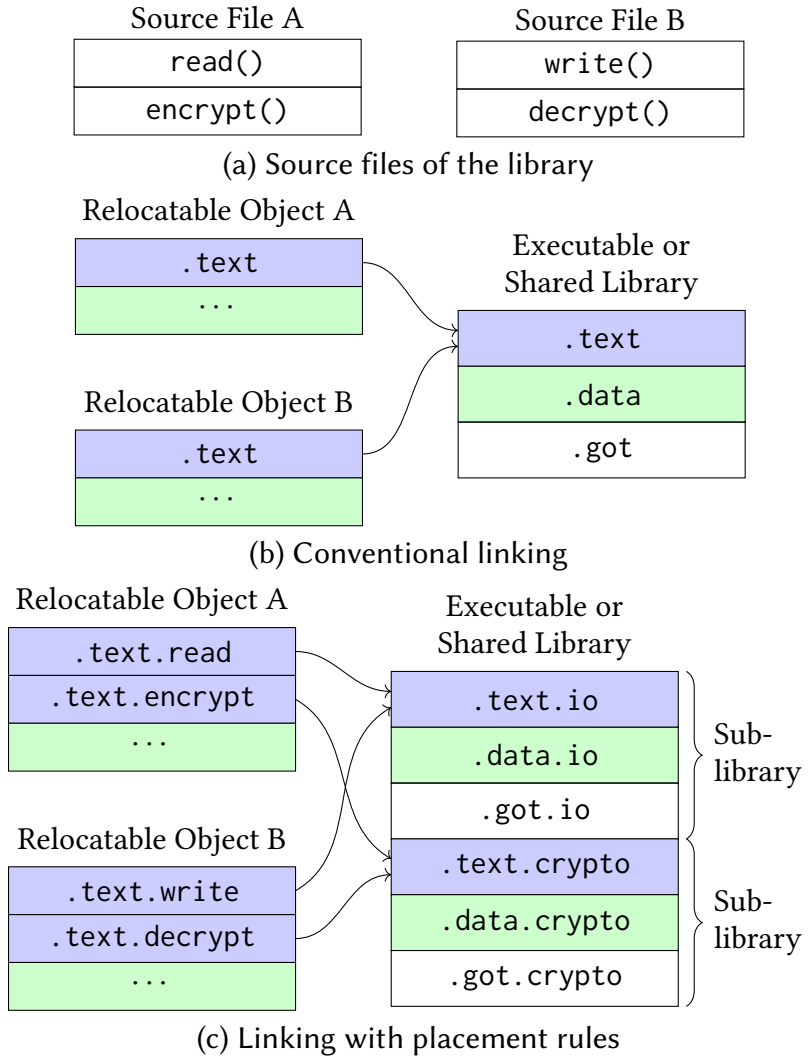


Fig. 4. Building a library with and without placement rules.

Figure 4 illustrates how to use placement rules to split a cryptographic library into multiple sub-libraries. The library consists of two source files defining various functions, as depicted in Figure 4a. The original build process involves compiling both source files into relocatable objects and linking them into a shared library, as illustrated in Figure 4b. The following placement rules are applied to put the I/O and cryptographic functions into distinct sub-libraries:

```
{"compartments": {
  "crypto": {"symbols": ["encrypt", "decrypt"]},
  "io": {"symbols": ["read", "write"]}}}
```

As shown in Figure 4c, the functions are now regrouped into two sub-libraries in the output binary, but the overall structure of the build process remains unchanged.

Access-Control List (ACL). The other component of the policy is an ACL that defines which symbols each sub-library may reference and with what permissions. Each ACL entry contains:

Subject A glob string identifying one or more sub-libraries that this ACL entry applies to.

Object A set of glob strings identifying symbols that the subject is allowed to reference.

Permissions A subset of {Read, Write, Execute} defining the rights that the subject has over the object.

3.2 Programming Model

We integrate most linkage-based compartmentalization features into existing OS components and expose few new primitives. In the general case, client code need not explicitly invoke any new APIs to be compartmentalized. We also make design choices such that:

- Programs built with a compartmentalization policy can still be run without compartmentalization.
- Programs built *without* a policy can still be run *with* compartmentalization using default per-library compartments.
- No compiler toolchain flag is required, apart from optionally specifying the policy, to build binaries for compartmentalization.

These strong compatibility properties yield the following benefits:

- Existing build systems can be reused to produce binaries that work with or without compartmentalization.
- Diagnosing compartmentalization issues is simplified because the same binaries also run without compartmentalization.
- The same shared libraries can be used by processes with or without compartmentalization, minimizing system bloat.

To support programming models with more complex features such as just-in-time compilation and garbage collection, the compartmentalization TCB does expose a few highly privileged APIs (see Section 4.7) for inspecting and manipulating its internal state. Section 7.2 showcases their use by the V8 garbage collector.

Security Guarantees. During compartment transitions, the following *ABI hygiene* properties are enforced:

Stack Safety No compartment can access another compartment's stack frames except through explicitly shared capabilities.

Register Integrity Callee-saved registers are restored to their pre-call values after a callee compartment returns.

Register Confidentiality Registers not used for holding arguments or return values are cleared.

Control Flow Integrity Control flow is "well-bracketed" so that it is impossible for a callee compartment to stash a return pointer and reuse it later to enter another compartment unexpectedly.

These properties ensure the integrity and confidentiality of inter-compartment control and data flow at the calling convention level, thereby establishing a limited form of *mutual distrust*.

Trusted Compartments. It is possible to opt out of ABI hygiene enforcement on certain compartment transitions for performance reasons. By declaring a compartment as "trusted" in the policy, all transitions to it are elided. Our implementation by default puts a number of stateless[3] standard library functions such as `memcpy` and `strcmp` into a trusted compartment. We have also prototyped a feature that allows a set of compartments to be marked as "mutually trusting", effectively coalescing them into the same compartment and eliding transitions among them when possible. Several benchmarks in Section 7 make use of this feature to improve performance.

POSIX Integration. Our model aims to be *orthogonal* to POSIX:
Threading Since compartments are associated with code alone, threads are not inherently associated with any one of them.
System Calls Only system calls made through wrapper functions provided by the C runtime are accepted by the kernel. Thus, access to system calls is controlled in the same way as access to regular functions by the ACL (see Section 4.8).
Signals A signal triggers a compartment transition from the interrupted compartment to the signal handler's (see Section 4.9).

4 Implementation

We extend the CHERI LLVM toolchain to enforce compartmentalization policies (Section 4.1) and introduce a compartmentalization runtime of fewer than 2500 lines of code to CheriBSD to implement compartment transitions, language runtime support, and OS integration (Sections 4.2 to 4.9). This section extensively uses terminology related to linking. See the appendix for a primer.

4.1 Compartmentalization Policy Enforcement

4.1.1 Applying Placement Rules. To keep the placement rules brief and aid the incremental partitioning of libraries into sub-libraries, not all functions need to be explicitly assigned to a sub-library by the placement rules. When assembling the sub-libraries, the static linker first processes those explicitly assigned functions and then implicitly assigns the rest. If the placement rules assign several functions from one relocatable object to different sub-libraries, then these functions must be in *different* input sections. However, the compiler places all functions in the same input section by default. We therefore force each function to be in its own input section by always enabling the "`-ffunction-sections`" compiler flag, allowing the static linker to satisfy arbitrary placement rules.

Creating Sub-libraries. The static linker first assigns the input sections that are explicitly mentioned by the placement rules to their respective sub-libraries. It also generates per-sub-library synthetic output sections containing the GOT, PLT, etc. Only input sections assigned to the same sub-library can be merged together, resulting in a set of output sections *per sub-library*. Each set is then assembled into a sub-library that contains its own code and data sections. Some output sections such as symbol tables

[3] Although statelessness is a desirable property for trusted compartments to have, it is neither a sufficient nor a necessary condition. The code must be audited to ensure that it cannot corrupt any caller's state.

and non-PLT relocation tables are shared by all sub-libraries, which minimizes changes required in the dynamic linker and other tools such as debuggers.

Assigning Anonymous Objects. Read-only data sections containing constants, such as anonymous strings or jump tables, must be placed in the same sub-library as the code that uses them because they are accessed through PC-relative addressing and hence must reside within the bounds of the PC (see Section 4.1.3). The static linker therefore implicitly assigns them to the relevant sub-libraries, which is not only required for correctness but also avoids the need for placement rules to name these anonymous objects.

Implicitly Assigning Sections. The static linker then attempts to use a graph-based algorithm to assign the remaining sections. Each static relocation creates an edge between the input section containing the instruction or data patched by the relocation and the input section containing the target symbol referenced by the relocation. These edges partition the input sections into disjoint connected components. If a connected component contains input sections already assigned to sub-libraries (whether explicitly or due to PC-relative accesses), and all such sections belong to the same sub-library, then its unassigned input sections are also assigned to that sub-library. Finally, any remaining unassigned input sections are assigned to a default catch-all sub-library.

4.1.2 ACL Enforcement. For each static relocation, the static linker consults the ACL to determine if the requested reference is permitted. If the reference is prohibited, a link-time error is emitted. If the reference is permitted and the result of the relocation is a capability (e.g., a GOT entry), the static linker constrains the permissions of the resulting capability according to the ACL. For example, references from sub-library A to a *writable* data symbol defined in sub-library B will be authorized by a *read-only* capability if the ACL *only* permits read access to that data symbol from sub-library A.

4.1.3 Isolating Sub-libraries. CheriABI already isolates shared libraries to some extent: each code pointer is a capability that is initialized at run time by the dynamic linker, which restricts the pointer's bounds to the memory image of the shared library to which the pointed-to code belongs. This choice of bounds is sufficiently permissive to allow the use of direct branches for intra-library function calls but restrictive enough to prevent any code from using PC-relative addressing to call into another library.

Linkage-based compartmentalization brings the granularity of this isolation down to sub-libraries. *First*, the static linker inserts appropriate padding between sub-libraries to ensure that the rounded capability bounds for each sub-library do not overlap with other sub-libraries. *Second*, the dynamic linker now bounds all code pointers to the memory image of their containing sub-libraries rather than that of the entire shared library. *Third*, the static linker now emits PLT stubs and PLTGOT entries for inter-sub-library calls to non-preemptible functions in the same relocatable object, as opposed to the convention of using direct PC-relative branches. This indirection is necessary due to the narrower bounds of the program counter making target functions in other sub-libraries no longer reachable via direct branches. Note that these new PLTGOT entries do not use lazy binding and jump slot relocations like typical PLTGOT entries but are initialized using relative relocations.

4.2 Trampoline

Recall from Section 4.1 that a compartment can only call into another through function pointers initialized by the dynamic linker. Each such function pointer is "wrapped" in a *trampoline*: when the dynamic linker creates a pointer to function F when processing relocations or lazy binding requests, it generates a piece of TCB code T_F known as a trampoline and returns a sentry pointing to it instead of F.[4] This sentry is returned every subsequent time F is referenced so that all pointers to the same function compare equal.

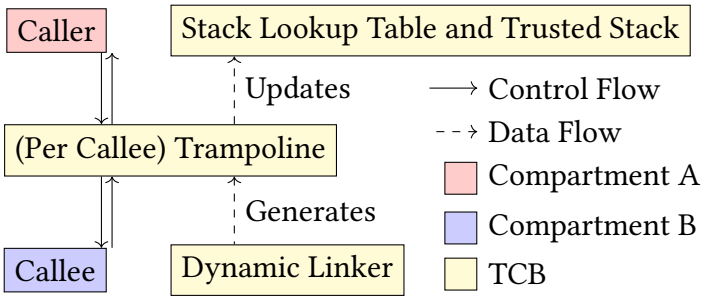


Fig. 5. Trampolines mediate inter-compartment calls.

Each trampoline has on the order of 100 instructions and is prefixed by a data header containing a pointer to the callee. These instructions can be divided into a *calling leg* and a *returning leg*. T_F uses the same calling convention as F, and its calling leg forwards control to F via a jump-and-link instruction, which points the link register to the returning leg through which F can return to the caller. As shown in Figure 5, the trampoline transparently transitions from compartment A to B and vice versa while updating the *Trusted Stack* and *Stack Lookup Table*, both crucial data structures for the compartmentalization TCB to be explained in more detail.

By default, the calling leg of a trampoline includes a *fast path* that elides transitions by directly tail-calling the callee if it belongs to the same compartment as the caller, which proves to be especially common in certain programs exemplified in Section 7.6.

As symbol resolution proceeds, trampolines are laid out contiguously, with suitable padding, on writable and executable pages mapped by the dynamic linker. While the use of such pages is normally frowned upon, there is little risk in doing so when leveraging CHERI's fine-grained memory protection in conjunction with existing Page Table Entry (PTE) permissions. Figure 6 shows a Morello register file whose Program Counter Capability[5] (PCC) covers both the PLTGOT and `.text`, as is the case for typical programs. The PCC grants Read and Execute permissions, but because the PLTGOT is mapped onto pages that lack the execute PTE permission, the MMU would still stop any attempt to execute it.

The figure then shows a PLTGOT entry pointing not to the callee but to a trampoline. This capability only has Read and Execute permissions and hence cannot be used to

[4]As an optimization, a function pointer need *not* be wrapped in a trampoline if it is known to be a) never used for inter-compartment calls and b) never compared against other values by client code. This happens when, for example, a compartment binds to a preemptible function defined by itself that is not preempted at run time. Here, the resulting function pointer is stored in the PLTGOT, hence impossible to be assigned to a C variable, and it is only used for *intra*-compartment calls.

[5]The program counter (PC) has been extended to a CHERI capability with bounds etc.

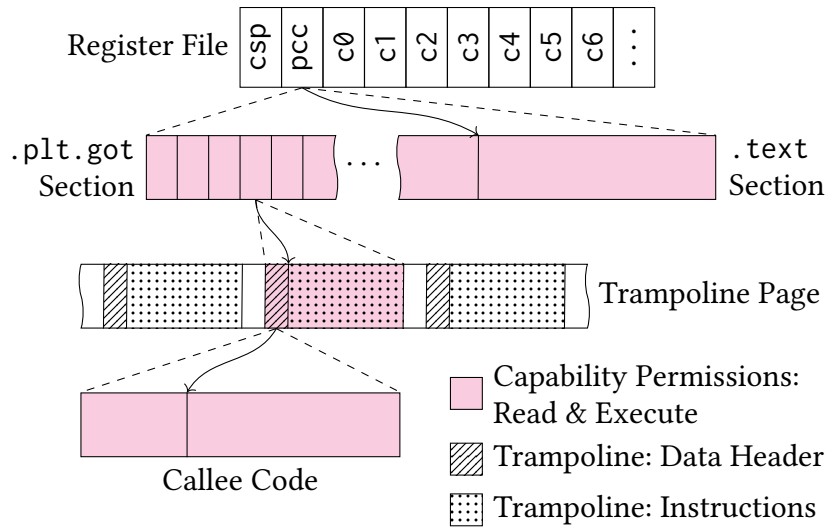


Fig. 6. The capability graph surrounding a trampoline.

modify the trampoline despite it being mapped onto a writable page. Note that unlike the previous case, no *architectural* feature prevents the data header of a trampoline from being executed. Instead, the trampoline is carefully designed such that under no circumstances should this happen as long as only sentries pointing to its entry point as well as return pointers to its returning leg are exposed to client code.

Compiler Toolchain Changes. When taking a reference to a non-preemptible function in the same compartment, the compiler may emit code to derive a function pointer directly from the PC. Such a function pointer would *not* be wrapped in a trampoline and would hence bypass compartment transitions if called from another compartment. To prevent this, the compiler is modified to never derive function pointers from the PC but always load them from the GOT.

4.3 Stack-Frame Isolation

To prevent any compartment from:

- Reading from the stack frames of other compartments, including deallocated frames that may have stale pointers, or
- Writing to the stack frames of other compartments

without explicit authorization by the programmer, our model assigns a dedicated stack to each compartment on each thread and switches the execution stack during compartment transitions. As usual, this should all be transparent to client code, which may continue to assume the stack model of the C abstract machine. But this implies that a *contiguous abstract stack* can be reconstructed from the split stacks. For this purpose, the compartmentalization TCB maintains two data structures per thread whose lifetimes are managed in conjunction with the threading library:

Stack Lookup Table For each compartment, record the top of its stack during the latest transition into or out of it.

Trusted Stack Track a sequence of *trusted frames* corresponding to the stack of inter-compartment calls so far. Each frame records the top of the calling compartment's stack during its *previous* transition.

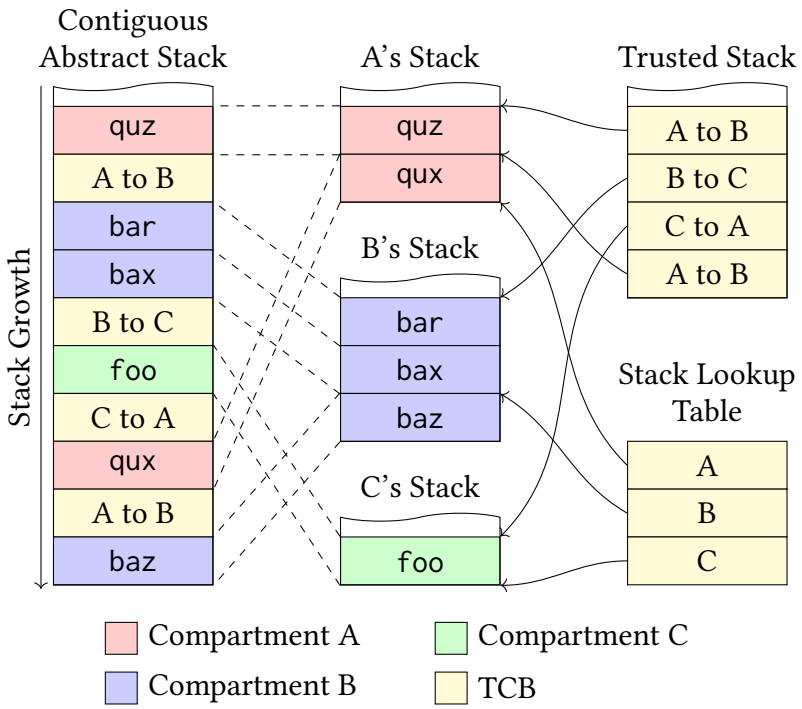


Fig. 7. Function quz initiates a chain of inter- and intra-compartment calls through bar, bax, foo, qux, and baz, causing the compartmentalization TCB to update the Trusted Stack and Stack Lookup Table.

During an inter-compartment call from compartment X to Y, the trampoline updates both data structures as follows:
(1) Push a new trusted frame for the "X to Y" transition.
(2) Copy X's entry in the Stack Lookup Table to that trusted frame.
(3) Update the entry to the current top of X's stack.
The trampoline can then install the value in Y's entry in the Stack Lookup Table as the execution stack before completing the transition. Later, upon returning from Y to X, the returning leg of the trampoline reverses the steps above, restoring the execution stack back to X's in the process. As an example, the right side of Figure 7 shows the configuration of several concrete stacks after a chain of calls. This can be mapped to the contiguous abstract stack on the left, which consists of call frames interspersed with TCB-managed trusted frames at compartment boundaries.

One implementation detail is worth highlighting: Morello and RISC-V use different architectural features to restrict access to the Trusted Stack and Stack Lookup Table to the compartmentalization TCB. The Morello implementation uses the "Executive" and "Restricted" modes that its ISA introduces: trampolines are run in Executive mode, granting access to additional stack and thread pointer registers, while compartment code is run in Restricted mode, forbidding access to Executive mode registers. RISC-V adds a new "UTIDC" control and status (CSR) register, readable by user-space code—both the compartmentalization TCB and compartments—but writable only by the kernel. By writing a sealed capability to this register on process and thread creation, and granting only the trampolines and compartmentalization TCB the ability to unseal this capability, the same protection can be provided.

ABI Changes. In the previous version of CheriABI, fixed arguments that do not fit into the argument registers are spilled onto the stack, and the callee reaches into this part of the *caller's* frame to access these arguments. This calling convention no longer works when stack safety is enforced. CheriABI is therefore modified such that the caller explicitly passes a bounded capability that covers the arguments which have been spilled into memory, whether fixed or variadic, and the callee can access these arguments relative to this capability rather than the stack pointer.

4.4 Register Integrity and Confidentiality

Integrity. To prevent callee-saved registers from being corrupted by a callee compartment, the calling leg of each trampoline saves all of them in the trusted frame that it pushes, while the returning leg of the trampoline pops the trusted frame and restores them.

Confidentiality. The compiler and static linker together emit a new ELF section that encodes the *signature* of each global function in terms of which registers are used for arguments and return values, respectively. When the dynamic linker generates the trampoline of any function, it refers to that function's signature and emits instructions to clear exactly those non-argument registers in the calling leg and non-return value registers in the returning leg.

It is important that a signature is emitted for each function referenced by the dynamic symbol table regardless of whether it is defined in that shared library so that any disagreement between the caller and callee on the signature of a function can be detected. Consider the following example where compartment A defines function `foo` and compartment B calls it with a different signature:
Compartment A Defines `int foo(int, void *);`
Compartment B Declares `int foo(int);`
If only compartment A's signature were taken into account, then `foo`'s trampoline would leave two argument registers uncleared. The caller in compartment B, believing `foo` to have only one argument, would then unknowingly leak the content of one register.

The signature adopted by the trampoline must, therefore, depend on both parties. The naive solution is to require that all callers and the callee agree on the number of argument and return value registers. But this is incompatible with existing practice. An example is the multiple function types permitted for POSIX signal handlers:
(1) `void (*)(int)`
(2) `void (*)(int, siginfo_t *, void *)`
Some OSes always invoke signal handlers as if they are of type 2, despite this being undefined behavior, since they assume this to work under any reasonable ABI. To support this type of usage, callers are allowed to declare more argument registers than does the callee, but they must not declare more return value registers.

4.5 Well-Bracketed Control Flow

The returning legs of all trampolines are interchangeable: they contain identical code and rely solely on protected data in the Stack Lookup Table and Trusted Stack to restore the caller's state. This design therefore enforces well-bracketed inter-compartment control flow regardless of which particular returning leg is entered.

4.6 Tail-Calls

Suppose function `foo` in compartment X calls into compartment Y, which in turn *tail-calls* function `bar` in compartment Z. A naive implementation would result in two back-to-back trusted frames, "X to Y" and "Y to Z", between `foo`'s and `bar`'s call frames on the contiguous abstract stack. But both trusted frames should ideally be collapsed into one "X to Z" frame so that it appears as if `foo` called `bar` directly. Otherwise, a program that continuously makes inter-compartment tail-calls would overflow the Trusted Stack with extraneous frames, even though it might never cause a stack overflow when run without compartmentalization.

The calling leg of each trampoline reliably detects inter-compartment tail-calls: when the return address points to the returning leg of the trampoline that pushed the topmost trusted frame. If this is the case, no new trusted frame is pushed so that stack space usage stays constant. Instead, the topmost trusted frame is modified in place to reflect the new identity of the callee compartment.

4.7 `setjmp`/`longjmp` and C++ Exceptions

Despite breaking the well-bracketedness of control flow, `setjmp` and `longjmp` must be supported by any realistic compartmentalization mechanism due to their wide presence in extant code. In our model, the definition of `longjmp` naturally extends to forcefully unwinding the contiguous abstract stack, destroying all intermediate call frames and trusted frames. To facilitate this, the compartmentalization TCB exposes two privileged APIs[6]:

GetTopTrustedFrame() Return a capability to the topmost trusted frame on the Trusted Stack.

UnwindTrustedStack(`tf`) Iteratively pop trusted frames until `tf` is reached, each time updating the Stack Lookup Table similarly to what the returning leg of a trampoline would do.

Initially, `setjmp` calls `GetTopTrustedFrame` and stores the return value in the jump buffer. Later, `longjmp` calls `UnwindTrustedStack` with this value to reverse through a series of compartment transitions until reaching the trusted frame that initially called `setjmp`.

The C++ runtime implements exceptions by walking through each call frame from where the exception is thrown to where it is caught. The compartmentalization TCB offers functional support via the following privileged APIs:

IsTrampoline(`pc`, `tf`) Return a Boolean indicating whether code pointer `pc` points to the returning leg of the trampoline that pushed trusted frame `tf`.

GetNextTrustedFrame(`tf`) Return the address and content of the parent trusted frame of `tf`.

Algorithm 1 illustrates this process with lines 4 and 5 representing newly added code.[7] Upon receiving an exception, the C++ runtime steps through the call frames, repeatedly calling `IsTrampoline` to determine whether the next call frame belongs to another compartment and is hence interposed by a trampoline. When this occurs, it calls `GetNextTrustedFrame` to obtain the address and machine state of that call

[6]The names of the APIs mentioned in this section are stylized for better readability and are not the names used in the actual implementation.

[7]Existing functions in the C++ runtime are italicized.

frame. Finally, the C++ runtime unwinds the contiguous abstract stack by calling `UnwindTrustedStack` and resumes execution by jumping to the exception handler.

Algorithm 1 Stack Walking for C++ Exception Handling

```
1: s ← GetMachineState()
2: repeat
3:     (s, end?) ← StepCallFrame(s)
4:     if ¬end? ∧ IsTrampoline(s.pc, s.trusted_frame) then
5:         s ← GetNextTrustedFrame(s.trusted_frame)
6: until end?
```

ABI Changes. The pointer to an exception handler should not be wrapped in a trampoline as it is a resumption point of a function. In previous versions of CheriABI, the dynamic linker could not distinguish such pointers from function pointers, and so could not omit the trampoline when creating them. This is solved by introducing distinct static relocations for "code pointers" (versus existing relocations for normal function pointers) emitted in the relocatable object files by the compiler and assembler. The distinction between relocations that should and should not result in trampolines is then propagated to the dynamic linker via differing dynamic relocations. This same mechanism is used to support GNU C's "computed goto".

4.8 System Calls

The vast majority of programs issue system calls by calling wrappers implemented by the C runtime, which in turn use a trap instruction to call the kernel. Our model makes this the *only* legal way for compartmentalized code to issue system calls: attempting to use the trap instruction outside a wrapper causes an error. To achieve this, the compartmentalization TCB tags all function pointers to these wrappers with an unforgeable *user-defined* permission bit. The kernel, upon receiving a system call, first checks this bit on the user-space PCC and proceeds only if it is set. The wrapper symbols remain subject to the ACL enforcement described in Section 4.1.

4.9 Signals

The kernel delivers a signal by calling a function pointer to the signal handler. Since function pointers are wrapped in trampolines, they would normally cause a compartment transition from the interrupted compartment to the signal handler's. However, this would not work if an interrupt occurs during a critical section of the compartmentalization TCB when the Trusted Stack and Stack Lookup Table are temporarily inconsistent with the machine state (e.g., in the middle of a trampoline). Therefore, the compartmentalization TCB registers a centralized handler for all signals and interposes all calls to `sigaction`, similarly to what `pthread` does. This centralized handler is responsible for completing the work of any critical sections before dispatching the signal to the actual handler. To make this possible, each critical section is carefully designed for reentrancy or, as a last resort, blocks all signals.

5 Limitations

System interfaces with powerful ambient authorities impose inherent limitations: `setjmp` exposes the machine state of the calling compartment; signal handlers receive

access to the machine context of interrupted code; integer file descriptors are process-granular; several APIs like `dlopen`, `ptrace`, `dl_iterate_phdr`, and `elf_aux_info` grant wide-reaching authorities to the process.

We expect the following engineering limitations to be resolved. Assembly-generated "wild" function pointers do not cause compartment transitions and function pointer callers currently do not check for this. Toolchain-emitted function signatures can be incorrect in the presence of undefined behavior (even if unreachable). Thread-local storage is not compartmentalized yet, which would be achieved by requiring the thread control block to expose *not* the direct backing storage of all thread-local variables, but a GOT-like structure where each entry contains a bounded pointer to an object in the backing storage. When compartmentalization is enabled, the dynamic linker would create a dedicated thread control block for each compartment and only populate those entries for thread-local variables that the compartment is permitted to access.

6 Security Analysis

The following properties are provided by baseline CheriABI:

Spatial Memory Safety Authority to memory objects, including global data, stack objects, executable mappings, and heap allocations, is conveyed through capabilities rather than a global mapping from compartments to address ranges (see Section 2). Attackers cannot gain access to objects merely by guessing their addresses.

Temporal Memory Safety CheriBSD's temporal-safety mechanisms revoke outstanding capabilities to regions that are freed or unmapped before their storage or virtual addresses are reused [14, 51, 53]. This prevents a stale capability from regaining authority when a heap allocation is recycled or a virtual-address range is remapped, even when the capability has been delegated across compartment boundaries. The same guarantee applies to code and global data: unloading a shared library or retiring a JIT-code mapping revokes capabilities into that region, preventing stale pointers from accessing or executing a later mapping at the same address.

Control-Flow Safety The dynamic linker bounds each code pointer to its target compartment's memory image (see Section 4.1.3). CHERI monotonicity prevents compromised code from widening these bounds to cover another compartment. Function and return pointers are sentries that can transfer control only to their designated addresses (see Section 2). A compromised compartment may therefore reuse jump-oriented programming gadgets within itself, but it cannot extend a gadget chain into another compartment by retargeting a sentry to enter the middle of a function.

The present work additionally enforces the following properties:

Compartmentalization Policy Enforcement The static linker rejects prohibited references and records ACL-derived permission constraints for permitted ones to code and global data (see Section 3.1), while the dynamic linker applies these constraints when constructing capabilities and bounds code capabilities to their containing sub-libraries (see Section 4.1). Capabilities to stack, heap, and JIT-compiled objects are constrained dynamically due to CheriABI. This permits data-dependent sharing without knowing every eventual recipient when the allocation is created, as illustrated by tcpdump's recursive packet decoding in Section 7.1.

Mediation of Compartment Transitions Inter-compartment function references are obtained through the GOT and wrapped in trampolines so that calls cannot bypass compartment transitions (see Section 4.2). The kernel accepts system calls only from approved C-runtime wrappers carrying an unforgeable permission, whose symbols

remain governed by the ACL (see Section 4.8). Signals are first delivered to a centralized handler that completes any interrupted TCB critical section before transitioning to the application handler (see Section 4.9). Together, these mechanisms prevent attacks based on bypassing a transition with a function pointer, issuing a raw system-call trap to evade policy, or interrupting a transition while its protected state is temporarily inconsistent.

Stack Safety Each thread uses a distinct execution stack for every compartment, and trampolines switch stacks during compartment transitions (see Section 4.3). Rather than receiving a capability spanning its caller's frames, a callee whose arguments reside in memory receives a separate capability bounded to only the argument area. This prevents a compromised callee from inspecting or corrupting unrelated caller frames while preserving explicit sharing of stack objects required by the C programming language.

Register Integrity and Confidentiality Trampolines save callee-saved registers in protected trusted frames and restore them on return, preserving the caller's register state across a mutually distrusting call. They also clear non-argument registers before entering the callee and non-result registers before returning, using function-signature metadata from both parties to determine the permitted interface (see Section 4.4). These measures prevent a malicious callee from corrupting preserved registers and prevent either party from recovering residual register contents, including leakage caused by mismatched function declarations.

Control Flow Integrity Inter-compartment calls and returns are tracked in the Trusted Stack and Stack Lookup Table (see Section 4.3), and returning trampoline legs use this state alone to identify and restore the caller (see Section 4.5). These invariants neutralize attempts by a malicious callee to reuse a retained return pointer, substitute another trampoline's returning leg, or otherwise return into a compartment under the wrong calling context.

7 Evaluation

We first present case studies that demonstrate particular aspects of our design, including sub-libraries, debugging, tracing, and managed language support. We then present benchmark results involving the platforms listed in Table 1, measured on non-debug builds of CheriBSD[8] with global heap temporal safety [14] enabled[9].

7.1 Case Study: tcpdump

Tcpdump is a network analysis tool that processes packets captured in real time or from a saved file. It uses a chain of dissectors, each printing a human-readable decoding of the header before passing the remaining data to the next relevant dissector. This processing of untrustworthy data is often performed in a single process as a highly privileged user, making tcpdump a natural candidate for compartmentalization.[10] The tcpdump build system already places dissectors in an internal library, `libnetdissect`, which we first extend to a shared library. We then use a mix of generated and hand-written policies (the latter shown below) to further split it into infrastructure and per-protocol sub-libraries following existing software engineering boundaries: each

[8]On the Morello platform, we use the Benchmark ABI [48] to work around micro-architectural quirks and simulate performance achievable by an ideal implementation.

[9]Global heap temporal safety is disabled for the gRPC benchmark. See Section 7.6.

[10]On some platforms, tcpdump supports privilege separation, but this is not universal.

Table 1. Evaluated CHERI-Extended Platforms

Processor	**Base ISA**	**μArchitecture**	**Remark**
Arm Morello (ASIC)	Armv8-A	Superscalar, Out-of-Order	Research design [19]
Cambridge Toooba (FPGA)	RISC-V	Superscalar, Out-of-Order	Research design [36]
Codasip X730 (FPGA)	RISC-V	Dual-issue, 9-stage	Industrial proprietary design
CapLtd CVA6-CHERI (FPGA)	RISC-V	Dual-issue, 6-stage	Industrial open-source design [28]

`print-<proto>.c` is placed in its own compartment; NFS and SMB protocol helper functions, as well as shared cryptographic signature validation routines, are placed in their own respective compartments; finally, utility functions such as `txtproto_print` end up in the `netdissect` compartment, a catch-all compartment described in Section 4.1.1. These simple and maintainable build system changes, made in a few hours, dramatically scale tcpdump from 11 compartments to 180.

```
{"compartments": {
  "print-nfs": {"files": ["parsenfsfh.*o"]},
  "print-smb": {"files": ["smbutil.*o"]},
  "signature": {"files": ["signature.*o"]}}}
```

Our model's minimal perturbation to the ABI allows debuggers to be extended with compartment awareness. Figure 8 shows a stack trace, generated by an extended version of GDB, of tcpdump when printing an HTTP packet. Similar to the contiguous abstract stack of Section 4.3, call frames shown in black are interspersed with colored trusted frames that record compartment transitions, which mirror the Ethernet-IP-TCP-HTTP packet hierarchy.[11]

The policies define compartment boundaries, but validating their security benefit requires understanding how each compartment's privileges evolve at run time as capabilities are propagated. As a first step in this direction, we present *CheriTree*, a tool that traverses the capability graph (see Section 2) at any given point in execution to map out reachable memory as intervals on the address-space "spectrum" colored by the union of the permissions of all capabilities covering each interval. This traversal is exact because all valid capabilities can be precisely identified in memory by their tags, and their bounds enable an exhaustive graph search.

We first take a snapshot of tcpdump compartmentalized with *no policy* in `http_print`. Reachable memory totals 8.37 MB, with 4.78 MB being read-writable. With the policy, the numbers drop to 814 KB and 408 KB, and the address space is visualized in the upper row of Figure 9, with over 100 unreachable compartments and memory mappings omitted. As expected, apart from `print-http` itself and its stack, the dynamic linker

[11]The transition through the TCP dissector compartment is absent because `tcp_print` tail-calls `http_print`. As explained in Section 4.6, this causes the topmost trusted frame to be overwritten with the HTTP dissector compartment as the callee.

```
#0  http_print
#1  <print-ip> -> <print-http>
#2  ip_print
#3  <print-ether> -> <print-ip>
#4  ethertype_print
#5  ether_common_print
#6  ether_print
#7  ether_if_print
#8  <netdissect> -> <print-ether>
#9  pretty_print_packet
#10 <tcpdump> -> <netdissect>
#11 print_packet
#12 <pcap> -> <tcpdump>
#13 pcapint_offline_read
#14 pcap_loop
#15 <tcpdump> -> <pcap>
#16 main
```

Fig. 8. tcpdump stack trace when printing an HTTP packet.

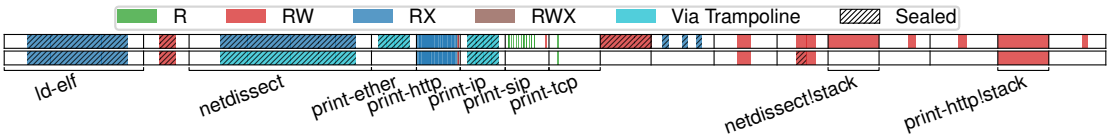


Fig. 9. Memory reachable from `http_print` before and after sealing access to `jmp_buf`. Labeled segments include sub-libraries and stacks, and unlabeled ones include heaps and other custom mappings. Hatched intervals are covered *only* by sealed capabilities, and cyan intervals are reachable *only* through calling or returning via trampolines. Small intervals are widened for visibility. `RWX` intervals displayed here hold PC-reachable mutable data, which are not actually mapped on executable pages.

(`ld-elf`) and the caller (`print-ip`) are reachable, gated by sentries and trampolines, respectively. *Unexpectedly*, `print-ether` and the entire stack of the catch-all `netdissect` compartment are also reachable. Examining the memory with GDB reveals this to be due to a `jmp_buf` reachable from a function argument. And because this argument is used as an opaque pointer, a simple refactoring can seal it into a sentry (see Section 2), which is non-dereferenceable. Indeed, this closes both gaps and a few more, as shown in the lower plot. Total reachable memory further falls to 527 KB, with 150 KB being read-writable.

To measure the performance, we first use tcpdump compartmentalized with *no policy* to print a capture file containing 100K packets of various protocols at maximum verbosity on Morello. Compartment transition tracing shows that there are on average 384 transitions per packet. Tcpdump compartmentalized with the policy is then run with the same input. Tracing shows that only 34 compartments out of 180 are entered at least once, and 659 compartment transitions occur per packet—a 71.6% increase. The CPU cycle count increases by 33.3% from 2.84B to 3.79B.

7.2 Case Study: Chromium and V8

Chromium is an open-source web browser underpinning widely used applications such as Google Chrome and VS Code. We take an existing research prototype port of Chromium to CheriABI for Morello, which consists of 47M LoC, and adapt it to work with linkage-based compartmentalization. The adaptation focuses on the bundled V8 JavaScript engine—a complex managed language runtime containing multiple JIT compilers and garbage collectors—and changes fewer than 300 lines out of V8's roughly 2M LoC (<0.015%). By successfully adapting V8 with limited effort, we demonstrate the generality of our approach in supporting diverse language features.

Garbage Collection. Some of V8's garbage collectors periodically scan the callee-saved registers and the stack to mark live objects, as illustrated in Algorithm 2. The core stack scanning loop is adapted for linkage-based compartmentalization using the same APIs introduced for `setjmp`/`longjmp` and C++ exceptions in Section 4.7 to traverse disjoint stacks belonging to different compartments, resulting in Algorithm 3.

Algorithm 2 Original V8 Core Stack Scanning Loop

```
1: for all r ∈ GetCalleeSaved() do
2:     Push r onto the stack
3: begin ← GetStackBegin()
4: end ← GetStackEnd()
5: for p ← begin to end step 16 do
6:     Visit(*p)
```

Algorithm 3 Adapted V8 Core Stack Scanning Loop

```
1: f ← GetTopTrustedFrame()
2: repeat
3:     for p ← f.stack_begin to f.stack_end step 16 do
4:         Visit(*p)
5:     for all r ∈ f.callee_saved do
6:         Visit(r)
7:     f ← GetNextTrustedFrame(f)
8: until f = null
```

Built-Ins. Through a complex process, V8's build system pre-compiles JavaScript's built-in functions from a DSL into optimized native code snippets that JIT-compiled code can invoke at run time. These code snippets are then packed into a blob of assembly instructions to be embedded in the executable section of the V8 binary, even though they are treated as mere data to perform the equivalent of dynamic linking. To prevent V8's pointer to this blob from being treated as a function pointer and wrapped in a trampoline, which would render it unusable as a data pointer, we relocate it using the "code pointer" relocation described in Section 4.7.

Outcome. With compartmentalization enabled, a *component* build of Chromium, which does not statically link most dependent libraries into the main executable, obtains 536 run-time compartments with no policy. V8's ~115K tests, which heavily

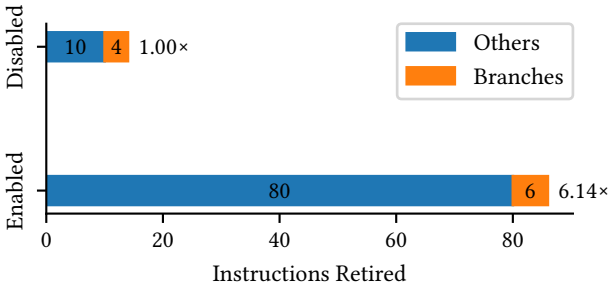
exercise garbage collection and JIT compilation, pass at the same rate of ~98% (±0.5% due to test non-determinism) as the baseline.

7.3 Micro-benchmark: Compartment Transition

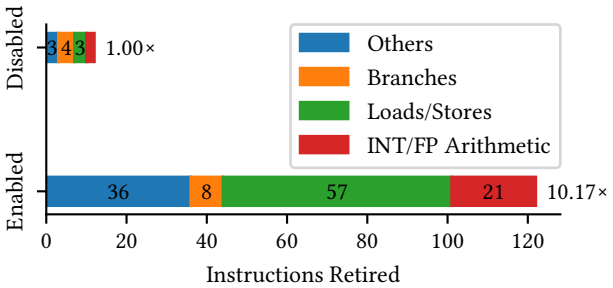
We use performance counters to analyze the best-case overhead of a compartment transition. The experiment makes a large number of calls to a function in another library that returns immediately. The call goes through both the PLT and a trampoline when compartmentalization is enabled and only the PLT when it is disabled. Counter values for the entire run are divided by the number of calls to compute the cost of a single call and amortize the probe effect.

Armv8-A's conditional instructions enable trampolines to use fewer branch instructions than on RISC-V, as shown in Figure 10.

Micro-architecture considerably affects the CPU cycle overhead introduced by trampolines. As shown in Table 2, out-of-order processors consume fewer cycles regardless



(a) Armv8-A with CHERI (Morello)



(b) RISC-V with CHERI (CVA6)

Fig. 10. Instruction type breakdown of an inter-library function call with and without compartmentalization, measured for Armv8-A and RISC-V. Only branch instruction retirement counters are available on Morello. Pointer arithmetic instructions are classified as "Others" on CVA6.

Table 2. Cycle Count of an Inter-library Function Call under Different Compartmentalization Settings

Processor	Base ISA	OoO	Disabled	Enabled	Overhead
Morello	Armv8-A	Yes	10	60	6.0×
Toooba	RISC-V	Yes	8	96	12.0×
X730	RISC-V	No	10	121	12.1×
CVA6	RISC-V	No	20	155	7.8×

of whether compartmentalization is enabled. Between in-order processors, the proprietary X730 dominates the less optimized CVA6 in absolute terms, but the poorer baseline of the latter results in superior relative overhead.

7.4 Macro-benchmark: FFmpeg Decoding

Media codecs are widely deployed but are highly susceptible to malicious input. We use performance counters to analyze the overhead of compartmentalizing `libdav1d`, a popular decoder for the mainstream AV1 format. The experiment on Morello uses FFmpeg to decode 10-second videos at three resolutions, five times each. As shown in Table 3, up to 1.15 million compartment transitions incur only small micro-architectural overheads.

Table 3. Relative Overhead of Decoding 10-Second Videos

Resolution	Transition Count	ΔCPU Cycles	ΔInst. Retired	ΔBranch Retired	ΔMem. Access
1280 × 720	751K	0.92%	0.53%	1.50%	0.71%
1920 × 1080	849K	0.68%	0.49%	1.29%	0.60%
3840 × 2160	1.15M	0.29%	0.30%	0.99%	0.40%

7.5 Macro-benchmark: SPDK

Storage Performance Development Kit (SPDK) [56] is a user-space storage stack widely deployed in enterprise storage products and cloud-native solutions. It depends on multiple independent projects such as Data Plane Development Kit (DPDK), totaling over 4M lines of code in over 200 libraries. The libraries, although intended for functional separation rather than security boundaries, provide a valuable platform for performance evaluation.

SPDK 22.11 is modified to remove the dependency on the DPDK contiguous memory kernel driver for ease of benchmarking. Performance is determined to be equivalent. The experiment on Morello uses `fio` 3.41 with the SPDK plugin to generate random

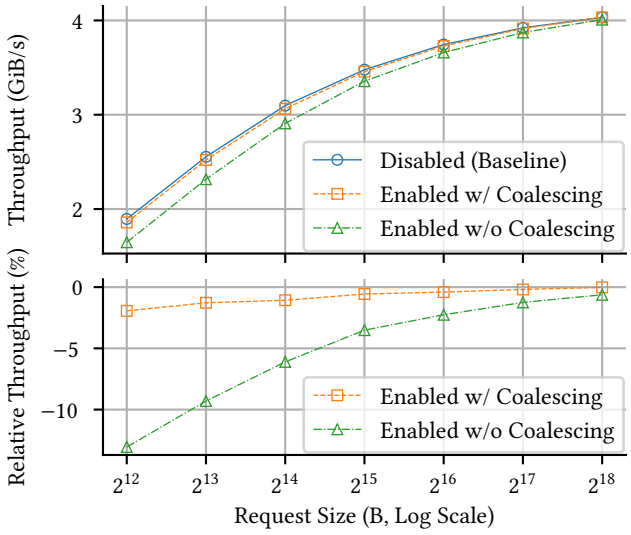


Fig. 11. SPDK throughput under different compartmentalization settings.

reads with various request sizes against a RAM disk. The baseline disables compartmentalization, and two other configurations are measured:
(1) SPDK and DPDK libraries are coalesced into 2 separate compartments, totaling 28 run-time compartments.
(2) All libraries are separate compartments, increasing the number of run-time compartments to 78.

As shown in Figure 11, throughput for 4 KB reqeusts decreases by 1.9% when coalescing compartments and by 13% without coalescing. The impact decreases as the request size increases. To model a worst-case scenario of throughput reduction, SPDK is then modified to transfer only a single byte at a time rather than entire data blocks. The baseline rate is 1209K IOPS, and compartmentalization reduces it by 4.4% and 26% with and without coalescing, respectively.

7.6 Macro-benchmark: gRPC

gRPC is a high-performance C/C++ remote procedure call framework deployed across data center services and edge devices. It has the Abseil C++ library as a notable dependency.

The experiment on Morello runs the queries-per-second (QPS) benchmark from gRPC 1.54.2. A server and client are pinned to different cores in the same cluster and communicate through the loopback interface. The workload exchanges unary RPC calls over a single channel in "synchronous" mode. The client maintains 8 outstanding RPC requests (one per thread) and disables TLS. For each payload size, 20 samples are collected, each lasting 60 s after a 15 s warm-up period. The baseline disables compartmentalization, and two compartmentalization settings are tested:
(1) Core gRPC, Abseil, and cryptographic libraries are coalesced into 3 compartments, totaling 18 run-time compartments.
(2) All libraries are separate compartments, increasing the number of run-time compartments to 82.

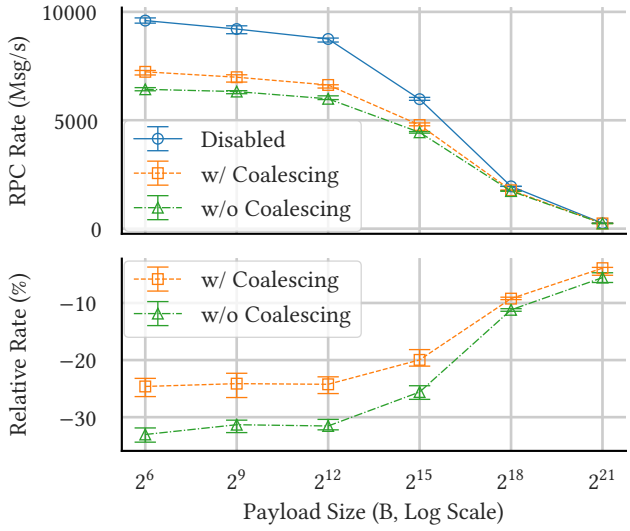


Fig. 12. gRPC median message rate under different compartmentalization settings with 95% confidence intervals.

For this benchmark, global heap temporal safety is disabled (cf. Section 7) due to unexpected, workload-specific performance penalties when it is enabled together with compartmentalization, an effect that appears to be addressable through further engineering.

This benchmark highlights the importance of the fast paths in trampolines described in Section 4.2. In a sample run with compartment coalescing and 64 B payloads before deploying fast paths, out of a total of 30.4M compartment transitions, 19.6M (64%) are compartment *self-transitions* almost entirely due to calling C++ virtual functions defined in the same compartment. Deploying fast paths dramatically cuts the number of self-transitions by 99.2%, reducing the total number of transitions to 10.5M.

As shown in Figure 12, without compartment coalescing, for a payload size of 64 B, the RPC rate decreases by 33%. The impact drops to below 5.6% as the payload size increases. Coalescing improves this figure to 25% and 3.9% for 64 B and 2 MB payloads.

8 Future Work

A recent SoK [27] argues that compartmentalization systems should be evaluated holistically across policy definition, abstractions, and enforcement mechanisms. Our main contribution lies in enforcement mechanisms, providing a novel level of *fine-grained* and *scalable* enforcement of compartmentalization guarantees such as ABI hygiene. A 2020 Microsoft study [23] estimated that a CHERI system would deterministically mitigate at least 67% of then-reported memory safety vulnerabilities, leaving substantial residual risks. JIT compilers also present a significant attack surface where code-generation flaws can produce unsafe code even when the compiler itself is memory safe. Further limiting attackers' abilities therefore requires techniques described in the present work and beyond:

Interface Safety Although ABI hygiene minimizes the unintentional sharing of objects between compartments, programmers remain responsible for safeguarding against compartment interface vulnerabilities [26] that arise from intentional sharing. Tools like RLBox [32] provide type-driven taint tracking and mandatory data validation to harden compartment interfaces. Our model could serve as an efficient backend for such tools by replacing many software-enforced properties with hardware-enforced guarantees.

Policy Automation Realizing the full benefit of scaling to thousands of compartments requires complementary research into automation, which has evolved from refactoring-oriented tools such as Privtrans [9], Wedge [8], and SOAAP [20] to more recent work with tunable security/performance objectives such as SCALPEL [35] and HAKC [31]. Preliminary results from joint work on these techniques indicate that automated compartmentalization is feasible for libraries at least the scale of `libpng`. Ideally, policy construction should also be paired with effective *auditing*. A useful example is CHERIoT's policy auditing tool [2], which allows users to write assertions that are checked against binary images.

Stack Temporal Safety Stack references passed across compartment boundaries, including both explicit references to local variables and implicitly allocated buffers for function arguments and return values, can be hoarded beyond their lifetimes [3]. Allocating such "unsafe" buffers on a shadow stack [6, 25] partially mitigates this, while proposed hardware-software extensions such as the "slinky stack" on CheriOS [13] and various new kinds of capabilities [17, 18, 40, 42] promise stronger guarantees.

Side-Channel Attacks Speculative execution attacks can leak compartment secrets [16]. Architectural speculation contracts and micro-architectural identification of compartments have been proposed for CHERI implementations to mitigate such risks [15]. However, these remain experimental solutions pending further research.

9 Related Work

`libcheri` A prior model [46] uses CHERI to construct sandboxes similar to classes in object-oriented programming. In contrast, our model a) is a refinement of existing dynamic linking rather than a standalone runtime, b) performs compartment transitions in user-space without trapping into the kernel, c) supports compartment transitions via function pointers, enabling callbacks and C++ virtual functions, d) handles complex language features such as JIT compilation and garbage collection, e) demonstrates integration with debugging and visualization tools, f) is accompanied by an open-source software evaluation corpus of over 50M LoC of C/C++, and g) works on multiple processors, including Arm's Morello.

Embedded CHERI CheriRTOS [54] introduces CHERI to embedded systems, and CompartOS [1] implements linkage-based compartmentalization with fault tolerance. CHERIoT [2] is a commercial hardware-software platform that additionally provides stack and heap temporal safety, a clean-slate OS practicing fine-grained privilege delegation and minimization, and powerful policy auditing facilities. Our work is inspired by these designs but the challenges of targeting a general-purpose system necessitate trade-offs:

- Embedded systems can afford to impose clean-slate resource allocation, privilege delegation, synchronization, and fault tolerance abstractions on applications, whereas our design currently aims not to cause such disruptions, limiting our support to POSIX.
- CHERIoT's stack temporal safety enforcement forbids global and heap memory from storing stack references, which enables the system to limit their lifetime simply through stack scanning. However, this restriction is incompatible with the C standard and indeed much existing code, while stack scanning is too costly on general-purpose systems with multi-megabyte stacks. Thus, we defer addressing stack temporal safety to future work.
- Compared to CHERIoT where the integrator is solely responsible for auditing the policy of a static system, our design faces multiple parties—developers, administrators, and users—who jointly and dynamically interact with, and even alter, the system. The desired *distributed* policy auditing remains an open problem.

Formal Foundations Formal modeling of software on capability systems complements our work by proving similar security properties under idealized models. CapablePtrs [12] proves that pointer-as-capabilities compilation does not increase the attack surface beyond a unit's source-level interface, assuming a TCB close to ours. Skorstengaard et al. [39] present an alternative capability-based calling convention with proven stack isolation and well-bracketed control-flow properties, followed by successive proposals with improved efficiency through architectural extensions [17, 18, 40].

Memory Tagging MTE [4] supports 16 colors at pointer-sized granularity, which helps protect against buffer overflows and use-after-free vulnerabilities. However, the small number of domains is insufficient to enable compartmentalization. LatticeBox [30] represents compartment relations as hierarchical labels, compactly encoding memory accessibility without access-control tables. However, this accessibility relation is still

pre-configured, so dynamic, data-dependent memory sharing remains a challenge. PUMP [11] and PIPE [41] are significantly more powerful tagging schemes that allow for not only larger tags but also more expressive rules that can automatically propagate tags. This enables much richer security policies to be implemented, such as control flow integrity (CFI) and fine-grained compartmentalization [35]. However, the centralized rules face a security-performance trade-off when modeling the complex delegation of pointers in modern applications. In contrast, CHERI's capability-based decentralized enforcement naturally scales for arbitrary pointer flows.

Page-Based Protection ERIM [43], Hodor [22], Cerberus [44], Jenny [37], and Endokernel [55] combine Intel MPK with software safeguards for efficient in-process isolation, but MPK's 16 hardware keys restrict them to a small number of concurrently available protection domains. Proposed architectural extensions relax this limit but retain other constraints: Donky [38] expands the key space to 1024 but makes only four keys usable at a time; SecureCells [7] supports many more domains, but its centralized permission table operates at VMA granularity and must be updated whenever access is delegated; HFI [33] supports arbitrarily many sandboxes by reconfiguring region registers on entry, but each active sandbox can access only ten regions. These architecture-driven constraints considerably restrict the applicability and scalability of such systems in general-purpose software, whose compartments exhibit diverse memory-access and communication patterns.

Software-Based Fault Isolation SFI [45] restricts memory access by untrusted code within the same process through a combination of compiler instrumentation, static analysis, and dynamic checks. Systems like Native Client [58] and WebAssembly [21] rely on software bounds checking with non-trivial performance overhead. Lightweight Fault Isolation (LFI) [57] reduces this cost by restricting AArch64 machine code to a stylized, efficiently verifiable form. These systems offer at most coarse-grained memory sharing and cannot readily share allocations such as stack objects, limiting their use for fine-grained compartmentalization of existing software.

10 Conclusion

Linkage-based compartmentalization leverages CHERI memory safety to enable scalable privilege separation within a UNIX process. With CheriBSD's system-wide compartmentalization enabled (see Section 4), a freshly booted Morello system logged into the KDE desktop environment already hosts more than 2000 compartments across all processes. Launching a moderate number of additional applications causes this number to exceed 10K. All this is achieved with little, if any, source-code disruption to the applications, and the impact on performance or behavior is imperceptible in our day-to-day software development and web-serving workloads.

Moreover, case studies demonstrate that the design integrates well with realistic software development workflows through its native compiler toolchain, debugger, and tracing support. Performance evaluations across diverse CHERI-extended processors also confirm the feasibility of this single-address-space approach.

Acknowledgments

We are grateful for assistance and feedback from David Chisnall, Alexandre Joannou, Timothy M. Jones, Marno van der Maas, Peter G. Neumann, George V. Neville-Neil, Murali Vijayaraghavan, and Jonathan Woodruff, as well as additional support from Arm, Codasip, and Google.

Distribution Statement A: Approved for public release; distribution is unlimited. This work is sponsored in part by the Defense Advanced Research Projects Agency (DARPA) and the Air Force Research Laboratory (AFRL) under contracts HR0011-22-C-0110 ("ETC"), HR0011-23-C-0031 ("MTSS"), and FA8750-24-C-B047 ("DEC") as part of the DARPA CPM research program, and by the Office of Naval Research (ONR) under contract N00014-22-1-2463 ("SWISH"). 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 is also sponsored in part by Innovate UK projects 105694 ("Digital Security by Design (DSbD) Technology Platform Prototype"), 10027440 ("Developing and Evaluating an Open-Source Desktop for Arm Morello"), and 10027332 ("MOJO – A Robust Java Virtual Machine for Morello"), as well as the EPSRC CHaOS grant (EP/V000292/1), the EPSRC "UKRI3001: CHERI Research Centre" grant, and a Croucher Scholarship from the Croucher Foundation.

References

[1] Hesham Almatary, Michael Dodson, Jessica Clarke, Peter Rugg, Ivan Gomes, Michal Podhradsky, Peter G. Neumann, Simon W. Moore, and Robert N. M. Watson. 2022. CompartOS: CHERI Compartmentalization for Embedded Systems. arXiv:2206.02852 [cs.CR] https://arxiv.org/abs/2206.02852

[2] Saar Amar, Tony Chen, David Chisnall, Nathaniel Wesley Filardo, Ben Laurie, Hugo Lefeuvre, Kunyan Liu, Simon W. Moore, Robert Norton-Wright, Margo Seltzer, Yucong Tao, Robert N. M. Watson, and Hongyan Xia. 2025. CHERIoT RTOS: An OS for Fine-Grained Memory-Safe Compartments on Low-Cost Embedded Devices. In *Proceedings of the ACM SIGOPS 31st Symposium on Operating Systems Principles* (Lotte Hotel World, Seoul, Republic of Korea) *(SOSP '25)*. Association for Computing Machinery, New York, NY, USA, 67–84. doi:10.1145/3731569.3764844

[3] Sean Noble Anderson, Roberto Blanco, Leonidas Lampropoulos, Benjamin C. Pierce, and Andrew Tolmach. 2023. Formalizing Stack Safety as a Security Property. In *2023 IEEE 36th Computer Security Foundations Symposium (CSF)*. 356–371. doi:10.1109/CSF57540.2023.00037

[4] Arm. 2019. Memory Tagging Extension: Enhancing memory safety through architecture. https://developer.arm.com/community/arm-community-blogs/b/architectures-and-processors-blog/posts/enhancing-memory-safety

[5] Thomas Bauereiss, Brian Campbell, Thomas Sewell, Alasdair Armstrong, Lawrence Esswood, Ian Stark, Graeme Barnes, Robert N. M. Watson, and Peter Sewell. 2022. Verified Security for the Morello Capability-enhanced Prototype Arm Architecture. In *Programming Languages and Systems*, Ilya Sergey (Ed.). Springer International Publishing, Cham, 174–203.

[6] Sandeep Bhatkar and Daniel C. DuVarney. 2005. Efficient Techniques for Comprehensive Protection from Memory Error Exploits. In *14th USENIX Security Symposium (USENIX Security 05)*. USENIX Association, Baltimore, MD. https://www.usenix.org/conference/14th-usenix-security-symposium/efficient-techniques-comprehensive-protection-memory-error

[7] Atri Bhattacharyya, Florian Hofhammer, Yuanlong Li, Siddharth Gupta, Andres Sanchez, Babak Falsafi, and Mathias Payer. 2023. SecureCells: A Secure Compartmentalized Architecture . In *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, Los Alamitos, CA, USA, 2921–2939. doi:10.1109/SP46215.2023.10179472

[8] Andrea Bittau, Petr Marchenko, Mark Handley, and Brad Karp. 2008. Wedge: Splitting Applications into Reduced-Privilege Compartments. In *5th USENIX Symposium on Networked Systems Design and Implementation (NSDI 08)*. USENIX Association, San Francisco, CA. https://www.usenix.org/conference/nsdi-08/wedge-splitting-applications-reduced-privilege-compartments

[9] David Brumley and Dawn Song. 2004. Privtrans: Automatically Partitioning Programs for Privilege Separation. In *13th USENIX Security Symposium (USENIX Security 04)*. USENIX Association, San Diego, CA. https://www.usenix.org/conference/13th-usenix-security-symposium/privtrans-automatically-partitioning-programs-privilege

[10] Brooks Davis, Robert N. M. Watson, Alexander Richardson, Peter G. Neumann, Simon W. Moore, John Baldwin, David Chisnall, Jessica Clarke, Nathaniel Wesley Filardo, Khilan Gudka, Alexandre Joannou, Ben Laurie, A. Theodore Markettos, J. Edward Maste, Alfredo Mazzinghi, Edward Tomasz Napierala, Robert M. Norton, Michael Roe, Peter Sewell, Stacey Son, and Jonathan Woodruff. 2019. CheriABI: Enforcing Valid Pointer Provenance and Minimizing Pointer Privilege in the POSIX C Run-time Environment. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems* (Providence, RI, USA) *(ASPLOS '19)*. Association for Computing Machinery, New York, NY, USA, 379–393. doi:10.1145/3297858.3304042

[11] Udit Dhawan, Catalin Hritcu, Raphael Rubin, Nikos Vasilakis, Silviu Chiricescu, Jonathan M. Smith, Thomas F. Knight, Benjamin C. Pierce, and Andre DeHon. 2015. Architectural Support for Software-Defined Metadata Processing. *SIGARCH Comput. Archit. News* 43, 1 (March 2015), 487–502. doi:10.1145/2786763.2694383

[12] Akram El-Korashy, Stelios Tsampas, Marco Patrignani, Dominique Devriese, Deepak Garg, and Frank Piessens. 2021. CapablePtrs: Securely Compiling Partial Programs Using the Pointers-as-Capabilities Principle. In *2021 IEEE 34th Computer Security Foundations Symposium (CSF)*. 1–16. doi:10.1109/CSF51468.2021.00036

[13] Lawrence G. Esswood. 2021. *CheriOS: designing an untrusted single-address-space capability operating system utilising capability hardware and a minimal hypervisor.* Technical Report UCAM-CL-TR-961. University of Cambridge, Computer Laboratory. doi:10.48456/tr-961

[14] Nathaniel Wesley Filardo, Brett F. Gutstein, Jonathan Woodruff, Jessica Clarke, Peter Rugg, Brooks Davis, Mark Johnston, Robert Norton, David Chisnall, Simon W. Moore, Peter G. Neumann, and Robert N. M. Watson. 2024. Cornucopia Reloaded: Load Barriers for CHERI Heap Temporal Safety. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2* (La Jolla, CA, USA) *(ASPLOS '24)*. Association for Computing Machinery, New York, NY, USA, 251–268. doi:10.1145/3620665.3640416

[15] Franz A. Fuchs. 2025. *Toward transient-execution attack mitigations on CHERI.* Technical Report UCAM-CL-TR-1001. University of Cambridge, Computer Laboratory. doi:10.48456/tr-1001

[16] Franz A. Fuchs, Jonathan Woodruff, Peter Rugg, Alexandre Joannou, Jessica Clarke, John Baldwin, Brooks Davis, Peter G. Neumann, Robert N. M. Watson, and Simon W. Moore. 2024. Safe Speculation for CHERI . In *2024 IEEE 42nd International Conference on Computer Design (ICCD)*. IEEE Computer Society, Los Alamitos, CA, USA, 364–372. doi:10.1109/ICCD63220.2024.00063

[17] Aïna Linn Georges, Armaël Guéneau, Thomas Van Strydonck, Amin Timany, Alix Trieu, Sander Huyghebaert, Dominique Devriese, and Lars Birkedal. 2021. Efficient and provable local capability revocation using uninitialized capabilities. *Proc. ACM Program. Lang.* 5, POPL, Article 6 (Jan. 2021), 30 pages. doi:10.1145/3434287

[18] Aïna Linn Georges, Alix Trieu, and Lars Birkedal. 2022. Le temps des cerises: efficient temporal stack safety on capability machines using directed capabilities. *Proc. ACM Program. Lang.* 6, OOPSLA1, Article 74 (April 2022), 30 pages. doi:10.1145/3527318

[19] Richard Grisenthwaite, Graeme Barnes, Robert N. M. Watson, Simon W. Moore, Peter Sewell, and Jonathan Woodruff. 2023. The Arm Morello Evaluation Platform—Validating CHERI-Based Security in a High-Performance System. *IEEE Micro* 43, 3 (May 2023), 50–57. doi:10.1109/MM.2023.3264676

[20] Khilan Gudka, Robert N.M. Watson, Jonathan Anderson, David Chisnall, Brooks Davis, Ben Laurie, Ilias Marinos, Peter G. Neumann, and Alex Richardson. 2015. Clean Application Compartmentalization with SOAAP. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security* (Denver, Colorado, USA) *(CCS '15)*. Association for Computing Machinery, New York, NY, USA, 1016–1031. doi:10.1145/2810103.2813611

[21] Andreas Haas, Andreas Rossberg, Derek L. Schuff, Ben L. Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. 2017. Bringing the web up to speed with WebAssembly. *SIGPLAN Not.* 52, 6, 185–200. doi:10.1145/3140587.3062363

[22] Mohammad Hedayati, Spyridoula Gravani, Ethan Johnson, John Criswell, Michael L. Scott, Kai Shen, and Mike Marty. 2019. Hodor: Intra-Process Isolation for High-Throughput Data Plane Libraries. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. USENIX Association, Renton, WA, 489–504. http://www.usenix.org/conference/atc19/presentation/hedayati-hodor

[23] Nicolas Joly, Saif ElSherei, and Saar Amar. 2020. Security analysis of CHERI ISA. https://github.com/microsoft/MSRC-Security-Research/blob/master/papers/2020/Security%20analysis%20of%20CHERI%20ISA.pdf

[24] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. 2019. Spectre Attacks: Exploiting Speculative Execution. In *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1–19. doi:10.1109/SP.2019.00002

[25] Volodymyr Kuznetsov, László Szekeres, Mathias Payer, George Candea, R. Sekar, and Dawn Song. 2014. Code-pointer integrity. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation* (Broomfield, CO) *(OSDI'14)*. USENIX Association, USA, 147–163.

[26] Hugo Lefeuvre, Vlad-Andrei Bădoiu, Yi Chen, Felipe Huici, Nathan Dautenhahn, and Pierre Olivier. 2023. Assessing the Impact of Interface Vulnerabilities in Compartmentalized Software. In *Proceedings 2023 Network and Distributed System Security Symposium*. Internet Society. doi:10.14722/ndss.2023.24117

[27] Hugo Lefeuvre, Nathan Dautenhahn, David Chisnall, and Pierre Olivier. 2025. SoK: Software Compartmentalization . In *2025 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, Los Alamitos, CA, USA, 3107–3126. doi:10.1109/SP61157.2025.00075

[28] Capabilities Limited. 2026. CVA6-CHERI. https://www.capabilitieslimited.co.uk/current-projects/cheri-cva6

[29] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. 2018. Meltdown: Reading Kernel Memory from User Space. In *27th USENIX Security Symposium (USENIX Security 18)*. USENIX Association, Baltimore, MD, 973–990. https://www.usenix.org/conference/usenixsecurity18/presentation/lipp

[30] Zhanpeng Liu, Chenyang Li, Wende Tan, Yuan Li, Xinhui Han, Xi Cao, Yong Xie, and Chao Zhang. 2026. LatticeBox: A Hardware-Software Co-Designed Framework for Scalable and Low-Latency Compartmentalization. In *33rd Annual Network and Distributed System Security Symposium, NDSS 2026, San Diego, California, USA, February 23-27, 2026*. The Internet Society. https://www.ndss-symposium.org/ndss-paper/latticebox-a-hardware-software-co-designed-framework-for-scalable-and-low-latency-compartmentalization/

[31] Derrick Paul McKee, Yianni Giannaris, Carolina Ortega, Howard E. Shrobe, Mathias Payer, Hamed Okhravi, and Nathan Burow. 2022. Preventing Kernel Hacks with HAKCs. In *29th Annual Network and Distributed System Security Symposium, NDSS 2022, San Diego, California, USA, April 24-28, 2022*. The Internet Society. https://www.ndss-symposium.org/ndss-paper/auto-draft-257/

[32] Shravan Narayan, Craig Disselkoen, Tal Garfinkel, Nathan Froyd, Eric Rahm, Sorin Lerner, Hovav Shacham, and Deian Stefan. 2020. Retrofitting Fine Grain Isolation in the Firefox Renderer. In *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, 699–716. https://www.usenix.org/conference/usenixsecurity20/presentation/narayan

[33] Shravan Narayan, Tal Garfinkel, Mohammadkazem Taram, Joey Rudek, Daniel Moghimi, Evan Johnson, Chris Fallin, Anjo Vahldiek-Oberwagner, Michael LeMay, Ravi Sahita, Dean Tullsen, and Deian Stefan. 2023. Going beyond the Limits of SFI: Flexible and Secure Hardware-Assisted In-Process Isolation with HFI. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3* (Vancouver, BC, Canada) *(ASPLOS 2023)*. Association for Computing Machinery, New York, NY, USA, 266–281. doi:10.1145/3582016.3582023

[34] Kyndylan Nienhuis, Alexandre Joannou, Thomas Bauereiss, Anthony Fox, Michael Roe, Brian Campbell, Matthew Naylor, Robert M. Norton, Simon W. Moore, Peter G. Neumann, Ian Stark, Robert N. M. Watson, and Peter Sewell. 2020. Rigorous engineering for hardware security: Formal modelling and proof in the CHERI design and implementation process. In *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1003–1020. doi:10.1109/SP40000.2020.00055

[35] Nick Roessler and André DeHon. 2021. SCALPEL: Exploring the Limits of Tag-enforced Compartmentalization. *J. Emerg. Technol. Comput. Syst.* 18, 1, Article 2 (Sept. 2021), 28 pages. doi:10.1145/3461673

[36] Peter Rugg, Jonathan Woodruff, Alexandre Joannou, and Simon W. Moore. 2024. A Suite of Processors to Explore CHERI-RISC-V Micro Architecture . In *2024 27th Euromicro Conference on Digital System Design (DSD)*. IEEE Computer Society, Los Alamitos, CA, USA, 351–360. doi:10.1109/DSD64264.2024.00054

[37] David Schrammel, Samuel Weiser, Richard Sadek, and Stefan Mangard. 2022. Jenny: Securing Syscalls for PKU-based Memory Isolation Systems. In *31st USENIX Security Symposium (USENIX*

Security 22). USENIX Association, Boston, MA, 936–952. https://www.usenix.org/conference/usenixsecurity22/presentation/schrammel

[38] David Schrammel, Samuel Weiser, Stefan Steinegger, Martin Schwarzl, Michael Schwarz, Stefan Mangard, and Daniel Gruss. 2020. Donky: Domain Keys – Efficient In-Process Isolation for RISC-V and x86. In *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, 1677–1694. https://www.usenix.org/conference/usenixsecurity20/presentation/schrammel

[39] Lau Skorstengaard, Dominique Devriese, and Lars Birkedal. 2019. Reasoning about a Machine with Local Capabilities: Provably Safe Stack and Return Pointer Management. *ACM Trans. Program. Lang. Syst.* 42, 1, Article 5 (Dec. 2019), 53 pages. doi:10.1145/3363519

[40] Lau Skorstengaard, Dominique Devriese, and Lars Birkedal. 2019. StkTokens: enforcing well-bracketed control flow and stack encapsulation using linear capabilities. *Proc. ACM Program. Lang.* 3, POPL, Article 19 (Jan. 2019), 28 pages. doi:10.1145/3290332

[41] Gregory T. Sullivan, André DeHon, Steven Milburn, Eli Boling, Marco Ciaffi, Jothy Rosenberg, and Andrew Sutherland. 2017. The Dover inherently secure processor. In *2017 IEEE International Symposium on Technologies for Homeland Security (HST)*. 1–5. doi:10.1109/THS.2017.7943502

[42] Stelios Tsampas, Dominique Devriese, and Frank Piessens. 2019. Temporal Safety for Stack Allocated Memory on Capability Machines. In *2019 IEEE 32nd Computer Security Foundations Symposium (CSF)*. 243–24312. doi:10.1109/CSF.2019.00024

[43] Anjo Vahldiek-Oberwagner, Eslam Elnikety, Nuno O. Duarte, Michael Sammler, Peter Druschel, and Deepak Garg. 2019. ERIM: Secure, Efficient In-process Isolation with Protection Keys (MPK). In *28th USENIX Security Symposium (USENIX Security 19)*. USENIX Association, Santa Clara, CA, 1221–1238. https://www.usenix.org/conference/usenixsecurity19/presentation/vahldiek-oberwagner

[44] Alexios Voulimeneas, Jonas Vinck, Ruben Mechelinck, and Stijn Volckaert. 2022. You shall not (by)pass! practical, secure, and fast PKU-based sandboxing. In *Proceedings of the Seventeenth European Conference on Computer Systems* (Rennes, France) *(EuroSys '22)*. Association for Computing Machinery, New York, NY, USA, 266–282. doi:10.1145/3492321.3519560

[45] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. 1993. Efficient software-based fault isolation. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles* (Asheville, North Carolina, USA) *(SOSP '93)*. Association for Computing Machinery, New York, NY, USA, 203–216. doi:10.1145/168619.168635

[46] Robert N.M. Watson, Jonathan Woodruff, Peter G. Neumann, Simon W. Moore, Jonathan Anderson, David Chisnall, Nirav Dave, Brooks Davis, Khilan Gudka, Ben Laurie, Steven J. Murdoch, Robert Norton, Michael Roe, Stacey Son, and Munraj Vadera. 2015. CHERI: A Hybrid Capability-System Architecture for Scalable Software Compartmentalization . In *2015 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, Los Alamitos, CA, USA, 20–37. doi:10.1109/SP.2015.9

[47] Robert N. M. Watson, David Chisnall, Jessica Clarke, Brooks Davis, Nathaniel Wesley Filardo, Ben Laurie, Simon W. Moore, Peter G. Neumann, Alexander Richardson, Peter Sewell, Konrad Witaszczyk, and Jonathan Woodruff. 2024. CHERI: Hardware-Enabled C/C++ Memory Protection at Scale. *IEEE Security and Privacy* 22, 4 (July 2024), 50–61. doi:10.1109/MSEC.2024.3396701

[48] Robert N. M. Watson, Jessica Clarke, Peter Sewell, Jonathan Woodruff, Simon W. Moore, Graeme Barnes, Richard Grisenthwaite, Kathryn Stacer, Silviu Baranga, and Alexander Richardson. 2023. *Early performance results from the prototype Morello microarchitecture.* Technical Report UCAM-CL-TR-986. University of Cambridge, Computer Laboratory. doi:10.48456/tr-986

[49] Robert N. M. Watson, Peter G. Neumann, Jonathan Woodruff, Michael Roe, Hesham Almatary, Jonathan Anderson, John Baldwin, Graeme Barnes, David Chisnall, Jessica Clarke, Brooks Davis, Lee Eisen, Nathaniel Wesley Filardo, Franz A. Fuchs, Richard Grisenthwaite, Alexandre Joannou, Ben Laurie, A. Theodore Markettos, Simon W. Moore, Steven J. Murdoch, Kyndylan Nienhuis, Robert Norton, Alexander Richardson, Peter Rugg, Peter Sewell, Stacey Son, and Hongyan Xia. 2023. *Capability Hardware Enhanced RISC Instructions: CHERI Instruction-Set Architecture (Version 9).* Technical Report UCAM-CL-TR-987. University of Cambridge, Computer Laboratory. doi:10.48456/tr-987

[50] Robert N. M. Watson, Peter G. Neumann Jonathan Woodruff, Jonathan Anderson, Ross Anderson, Nirav Dave, Ben Laurie, Simon W. Moore, Steven J. Murdoch, Philip Paeps, Michael Roe, and Hassen Saidi. 2012. CHERI: a research platform deconflating hardware virtualization and protection. In *Runtime Environments, Systems, Layering and Virtualized Environments (RESoLVE 2012)*.

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

[52] Jonathan Woodruff, Robert N.M. Watson, David Chisnall, Simon W. Moore, Jonathan Anderson, Brooks Davis, Ben Laurie, Peter G. Neumann, Robert Norton, and Michael Roe. 2014. The CHERI capability model: revisiting RISC in an age of risk. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture* (Minneapolis, Minnesota, USA) *(ISCA '14)*. IEEE Press, 457–468.

[53] Hongyan Xia, Jonathan Woodruff, Sam Ainsworth, Nathaniel W. Filardo, Michael Roe, Alexander Richardson, Peter Rugg, Peter G. Neumann, Simon W. Moore, Robert N. M. Watson, and Timothy M. Jones. 2019. CHERIvoke: Characterising Pointer Revocation using CHERI Capabilities for Temporal Memory Safety. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture* (Columbus, OH, USA) *(MICRO-52)*. Association for Computing Machinery, New York, NY, USA, 545–557. doi:10.1145/3352460.3358288

[54] Hongyan Xia, Jonathan Woodruff, Hadrien Barral, Lawrence Esswood, Alexandre Joannou, Robert Kovacsics, David Chisnall, Michael Roe, Brooks Davis, Edward Napierala, John Baldwin, Khilan Gudka, Peter G. Neumann, Alexander Richardson, Simon W. Moore, and Robert N. M. Watson. 2018. CheriRTOS: A Capability Model for Embedded Devices. In *2018 IEEE 36th International Conference on Computer Design (ICCD)*. 92–99. doi:10.1109/ICCD.2018.00023

[55] Fangfei Yang, Bumjin Im, Weijie Huang, Kelly Kaoudis, Anjo Vahldiek-Oberwagner, Chia che Tsai, and Nathan Dautenhahn. 2024. Endokernel: A Thread Safe Monitor for Lightweight Subprocess Isolation. In *33rd USENIX Security Symposium (USENIX Security 24)*. USENIX Association, Philadelphia, PA, 145–162. https://www.usenix.org/conference/usenixsecurity24/presentation/yang-fangfei

[56] Ziye Yang, James R. Harris, Benjamin Walker, Daniel Verkamp, Changpeng Liu, Cunyin Chang, Gang Cao, Jonathan Stern, Vishal Verma, and Luse E. Paul. 2017. SPDK: A Development Kit to Build High Performance Storage Applications. In *2017 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*. 154–161. doi:10.1109/CloudCom.2017.14

[57] Zachary Yedidia. 2024. Lightweight Fault Isolation: Practical, Efficient, and Secure Software Sandboxing. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2* (La Jolla, CA, USA) *(ASPLOS '24)*. Association for Computing Machinery, New York, NY, USA, 649–665. doi:10.1145/3620665.3640408

[58] Bennet Yee, David Sehr, Gregory Dardyk, J. Bradley Chen, Robert Muth, Tavis Ormandy, Shiki Okasaka, Neha Narula, and Nicholas Fullagar. 2010. Native Client: a sandbox for portable, untrusted x86 native code. *Commun. ACM* 53, 1 (Jan. 2010), 91–99. doi:10.1145/1629175.1629203

A Static Linking Primer

The static linker constructs output files by merging several relocatable object files: *input sections* from object files are merged with other input sections of the same type to produce *output sections*. For example, all input sections containing machine code are merged into a single output section named "`.text`". The static linker uses *static relocations* from object files associated with each input section to patch code and data in the output sections. Some static relocations cause *dynamic relocations* to be generated in the output file, which are used by the dynamic linker to initialize GOT entries and global variables at run time. In addition, some static relocations add indirection. For example, a relocation may redirect a branch to a PLT stub, which loads the final branch target from the PLTGOT. The static linker is responsible for generating synthetic output sections that house the GOT, PLT, PLTGOT, and metadata needed for dynamic linking such as symbol tables and the `.dynamic` section.

B Artifacts

B.1 CheriBSD

The Armv8-A and RISC-V versions are hosted in separate branches:

ISA	Branch
Armv8-A	https://github.com/CTSRD-CHERI/cheribsd/tree/dev
RISC-V	https://github.com/CTSRD-CHERI/cheribsd/tree/cheri-rv64y-rebased-c18n

The following directories contain key parts of the implementation:

Component	Directory
Kernel	`sys/`
Dynamic linker	`libexec/rtld-elf/`
System libraries	`lib/`
C++ exception handling	`contrib/subrepo-cheri-libunwind/`

B.2 CHERI LLVM

The Armv8-A and RISC-V versions are hosted in separate repositories:

ISA	Branch
Armv8-A	https://github.com/CTSRD-CHERI/morello-llvm-project/tree/morello/dev
RISC-V	https://github.com/CTSRD-CHERI/llvm-project/tree/paper/2026ccs-c18n/codasip-rebased-cheribsd-c18n-acl

Each repository includes the following directories:

Component	Directory
Compiler front-end	`clang/`
Compiler back-end	`llvm/`
Static linker	`lld/`

B.3 GDB

The extended version of GDB demonstrated in Section 7.1 is hosted at https://github.com/CTSRD-CHERI/gdb/tree/cheri-14.

B.4 CheriTree

The CheriTree tool demonstrated in Section 7.1 is hosted at https://github.com/CTSRD-CHERI/cheritree.

B.5 Case Study: tcpdump

The compartmentalized tcpdump described in Section 7.1 is hosted at https://github.com/CTSRD-CHERI/cheribsd/tree/c18n-ccs. The benchmark driver is hosted at https://github.com/CTSRD-CHERI/c18n-paper-benchmarks/tree/main/tcpdump.

B.6 Case Study: Chromium and V8

The adapted V8 described in Section 7.2 is hosted at https://github.com/CTSRD-CHERI/v8. Commits `5eb7173` and `d723c72` contain the main adaptations.

B.7 Micro-benchmark: Compartment Transition

The micro-benchmark described in Section 7.3 is hosted at https://github.com/CTSRD-CHERI/c18n-microbenchmark, and the results are hosted at https://github.com/CTSRD-CHERI/c18n-paper-benchmarks/tree/main/microbench_results.

B.8 Macro-benchmark: FFmpeg

Section 7.4 uses the off-the-shelf FFmpeg package for CheriBSD. The benchmark driver is hosted at https://github.com/CTSRD-CHERI/c18n-paper-benchmarks/tree/main/ffmpeg.

B.9 Macro-benchmark: SPDK

Section 7.5 uses a modified SPDK hosted at https://github.com/CTSRD-CHERI/c18n-paper-benchmarks/tree/main/spdk. The benchmark driver is the `src/c18n-run` script in that repository.

B.10 Macro-benchmark: gRPC

Section 7.6 uses a research port of gRPC to CheriABI on Morello. The benchmark driver, which consists of a collection of program-generated scripts and JSON configuration files, is hosted at https://github.com/CTSRD-CHERI/c18n-paper-benchmarks/tree/main/grpc.