# Mon CHÉRI: Mitigating Uninitialized Memory Access with Conditional Capabilities

Merve Gülmez[*,†], Håkan Englund[*], Jan Tobias Mühlberg[‡], Thomas Nyman[§]
[*]*Ericsson Security Research,* [†]*DistriNet, KU Leuven,* [‡]*Université Libre de Bruxelles,* [§]*Ericsson Product Security*
*{merve.gulmez,hakan.englund,thomas.nyman}@ericsson.com, jan.tobias.muehlberg@ulb.be*

***Abstract*—Up to 10% of memory-safety vulnerabilities in languages like C and C++ stem from uninitialized variables. This work addresses the prevalence and lack of adequate software mitigations for uninitialized memory issues, proposing architectural protections in hardware. Capability-based addressing, such as the University of Cambridge's CHERI, mitigates many memory defects, including spatial and temporal safety violations at an architectural level. CHERI, however, does not handle undefined behavior from uninitialized variables. We extend the CHERI capability model to include "*conditional capabilities*", enabling memory-access policies based on prior operations. This allows enforcement of policies that satisfy memory-safety objectives such as "*no reads to memory without at least one prior write*" (`Write-before-Read`). We present our architecture extension, compiler support, and detailed evaluation of our approach on the QEMU full-system simulator and a modified FPGA-based CHERI-RISCV softcore. Our evaluation shows conditional capabilities are practical, with high detection accuracy while adding a small (≈ 3.5%) overhead which is comparable to the cost of baseline CHERI capabilities.**

## 1. Introduction

*Uninitialized variables*, variables that are declared but not assigned a value, are a well-known source of software defects in the C-family of programming languages. In general, all run-time allocated memory in C, C++, and even in modern languages like Rust, starts out as *uninitialized*. In this state, the value of the memory is indeterminate and may not reflect a valid state for the variable type. Attempting to interpret uninitialized memory results in *undefined behavior*, which can cause security vulnerabilities. In particular, uninitialized memory may expose residual data from previously deallocated data structures. This may inadvertently result in information disclosure, ranging from leaked pointer values [1] to the exposure of cryptographic keys [2]. Leaked pointer values can be exploited by attackers to bypass *address space layout randomization (ASLR)* [1], while the use of uninitialized variables can enable arbitrary code execution attacks [3]. Following recent surveys, use-before-initialized conditions account for a sizable 10% of memory-safety vulnerabilities in the wild [4]–[6].

Hardware-assisted defenses against memory-safety issues [7] are motivated by the need to reduce the overhead of run-time defenses through hardware/software co-designs and willingness by processor manufacturers to incorporate mechanisms for software security into their designs [8]–[10]. A prominent example is *Capability Hardware Enhanced RISC Instructions (CHERI)* [11], a joint research project of SRI International and the University of Cambridge. CHERI extends instruction-set architectures (ISAs) with fine-grained memory protection and software compartmentalization.

In its default configuration, CHERI provides spatial safety through capability-based addressing. While this allows software to preclude many classes of memory-safety defects, CHERI does not address uninitialized variables. Microsoft Security Response Center (MSRC) conducted a comprehensive analysis of vulnerabilities reported in 2019 to assess the CHERI ISAv7 [4]. The findings indicate that only 31% of the reported vulnerabilities could have been mitigated through the default configuration of CHERI. An additional 24% could be mitigated by CHERI when configured to provide partial temporal safety under the Cornucopia mechanism [12]. The analysis further highlights that at least 12% of the assessed vulnerabilities could have been mitigated if CHERI would protect against uninitialized access. This work explores and evaluates extensions for CHERI to mitigate those 12%.

**This paper and contributions.** In this paper, we extend the CHERI capability model to express memory-access policies that eliminate undefined behavior associated with uninitialized memory. We introduce *conditional permissions (CPs)* to capability-based addressing. CPs can express memory-access policies that take previous operations on memory into account. This enables *conditional capabilities* with fine-grained policies that satisfy different instances of memory-safety objectives at different granularities, such as *"no reads of memory which has not been the subject of at least one write"* (`Write-before-Read`) or *"this memory can be written to only once"* (`Write-Once`). We describe these CPs in § 3. CPs are enforced by introducing the notion of *operation-specific bounds* to capability-based addressing:

- We introduce *conditional permissions* and *conditional capabilities* for capability-based addressing (§ 3).
- We integrate conditional capabilities to CHERI-RISC-V in prototypes based on the QEMU full-system emulator and an FPGA softcore based on Flute64Cheri IP (§ 4).
- We add support for `Write-before-Read` conditional permissions to the CHERI-enabled Clang/LLVM compiler (§ 4.3) and embedded memory allocators (§ 4.4).
- We evaluate CPs using > 1000 NIST Juliet test suite

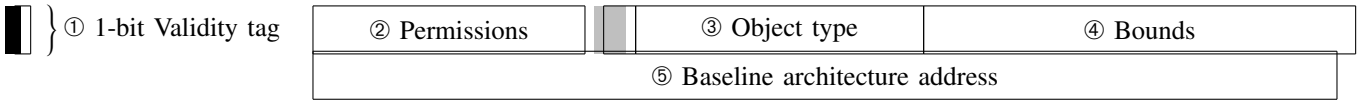



Figure 1: In-memory representation of CHERI capabilities adapted from Watson et al. [13]

test cases for uninitialized variables and using EEMBC CoreMark performance benchmarks (§ 5).

Our results show that CPs achieve 100% detection with only six false positives (≈ 1%) for the Juliet test suite, where the false positives do exhibit uninitialized (but non-vulnerable) accesses. We further show that CPs on CHERI-RISC-V impose only ≈ 3.5% performance overhead in addition to that added by CHERI, compared to benchmarks on an unmodified RISC-V softcore (7% combined). Consequently, CP overhead is comparable to that of CHERI. In summary, our work effectively and efficiently addresses uninitialized memory vulnerabilities, combining very high detection accuracy with performance penalties that are substantially lower than other detection mechanisms in hardware or software. Our conditional capability-enhanced QEMU and toolchain prototypes, and evaluation artifacts are available at https://github.com/conditionalcapabilities.

## 2. Background

**How prevalent are uninitialized memory vulnerabilities?** The prevalence of uninitialized memory as a source of vulnerabilities has been highlighted in statistics based on the Common Vulnerability Enumeration (CVE) program. MSRC reports that, between 2017 and 2019, uninitialized memory vulnerabilities accounted for ≈ 5–10% of the Common Vulnerability Enumerations (CVEs) issued by Microsoft [4], [5]. In 2019, uninitialized memory accesses were the fourth largest class of memory defects (10%), after spatial (44%), temporal (29%), and type confusion (14%). Similarly, a more recent survey of memory-safety CVEs between 2015 – 2022 by Sutter [6] indicates that use-before-initialized conditions account for ≈ 9% of all reported memory-safety vulnerabilities, after lifetime safety (49%), bounds safety (18%), and type confusion (11%), indicating that uninitialized memory accesses remain an issue in industrial code bases that impacts system security.

### 2.1. Capability-Based Addressing

*Capability-based addressing* is a memory access-control paradigm originating from mainframe computers of the late 1950s and 60s [14]. In capability-based addressing, conventional references to locations in computer memory, i.e. *pointers*, are replaced by protected objects called *capabilities* [15]. Capabilities carry, in addition to the referenced memory address, additional permission information that is used by the processor or memory-management subsystem to determine whether the accesses performed through a capability are allowed. While the exact composition of a capability can vary between different hardware instantiations, virtually all capability-based addressing schemes express allowed operations through at least *Read* (`R`), *Write* (`W`), and *Execute* (`X`) permissions that control whether references through a capability are permitted for load and store instructions or instruction fetches respectively. Capabilities also include *bounds information* that limit the range of memory that can be referenced via a particular capability.

Interest in capability-based addressing diminished with the introduction of memory management units (MMUs) that, in addition to performing address translation between virtual and physical memory addresses, also manage access control to virtual memory. However, capabilities differ fundamentally from the access control in MMUs: whereas MMUs associate permissions to *individual memory pages*, capabilities associate permissions to the *references used to address memory*.

### 2.2. The CHERI Capability Architecture

CHERI, which stands for Capability Hardware Enhanced RISC Instructions, is an ISA extension for a capability-based architectural protection model and hardware-software co-design. The CHERI architecture extends an underlying conventional ISA with hardware-supported capabilities that are used to protect virtual addresses used as code or data pointers. The CHERI ISA specification [16] defines the representation of capabilities held in registers and memory, as well as capability-aware instructions to manipulate them. Currently, implementations of CHERI exist for MIPS, RISC-V, and Armv8-A instruction sets. CHERI-enabled processors have been developed by Arm [17], Microsoft [18], as well as in the RISC-V ecosystem.

Figure 1 shows the in-memory representation of a CHERI capability. Each capability is double the width of the native integer pointer type of the baseline architecture: 128 bits on 64-bit platforms and 64 bits on 32-bit platforms. One additional bit, the *validity tag* ①, is stored separate from the capability and is protects its integrity: any manipulation of the capability in-memory by non-capability-aware instructions invalidates its tag. Capability-aware instructions maintain the tag as long as certain architectural invariants are met. This prevents direct in-memory manipulations and injection of arbitrary data as capabilities. The *permissions* ② control how the capability can be used and consists of the permissions described in § 2.1. The *object type* ③ allows capabilities to be temporarily "sealed", which renders the capability unusable until it is "unsealed" by a special instruction. Sealing is used by CHERI to implement opaque pointer types and fine-grained in-process isolation. The *bounds* ④ describe a lower and upper bound relative to the baseline

architecture address ⑤, which limits the portion of address space the capability is able to access. To reduce the in-memory footprint of capabilities, the bounds are stored in a compressed format [19] with both bounds in 28 bits (for a 64-bit address), loosing precision as the object size increases.

New capabilities in the CHERI architecture are always derived from an existing capability. The heritage of all capabilities can thus be traced back to the initial capabilities made available to firmware at boot time. CHERI enforces *monotonicity* on newly created capabilities, ensuring that capabilities constructed by a capability-aware instruction cannot possess permissions or bounds that exceed those of the original capability. The only exceptions to capability monotonicity are facilities for exception handling and compartmentalization using sealed capabilities, which allow non-monotonicity in a controlled manner to enable software to gain access to additional data capabilities.

The bounds information stored together with the virtual address forms the basis for the memory-safety properties provided by CHERI. Each allocation made by a program running on a CHERI-capable processor is associated with a capability that describes, in addition to the address, the valid bounds of the object (or sub-object) in memory. This allows CHERI to provide inherent spatial memory-safety properties. Extensions to the CHERI software stack have explored adding temporal-safety properties to heap-based allocations [12], [20], [21] and sandboxing [22].

CHERI does not enforce type safety for capabilities, nor does it prevent software from accessing uninitialized memory using a capability it possesses. Consequently, CHERI must be complemented with compiler-based type-safety analysis, and instrumentation passes that zero local variables before first use [5], [23] as well as heap allocators returning zeroed memory. Security analyses of CHERI (e.g., by MSRC [4]) generally take such mitigations for granted.

## 3. Conditional Capability Design

Consider the uninitialized pointer-dereference vulnerability in the Linux kernel shown in Listing 1. This vulnerability allows control-flow hijacking due to the uninitialized `backlog` pointer ❶ being dereferenced at ❹ when `cpg->eng_st != ENGINE_IDLE` ❷. An attacker could exploit this vulnerability to achieve arbitrary code execution by spraying the stack to take control of the value of `backlog` and make it point to attacker-controlled code [3].

Our goal is to detect the first use of the uninitialized pointer in the `if`-clause at ❸. Unlike methods that automatically zero out memory [5], [23], conditional permissions (CPs) offer the same protection against uninitialized memory use but with smaller and more predictable run-time overhead.

### 3.1. Challenges

Integrating CPs into capability-based addressing presents several challenges. A key issue is managing the tracking of uninitialized data. Previous approaches [25]–[27] assume that data between the lower bound and address pointed to

```
1  static int queue_manag(void *data){
2      cpg->eng_st = ENGINE_IDLE;
3      do {
4          struct crypto_async_request *async_req = NULL;
5        ❶ struct crypto_async_request *backlog;
6
7       /* ... cpg->eng_state may change state here ... */
8        ❷ if (cpg->eng_st == ENGINE_IDLE) {
9              backlog = crypto_get_backlog(&cpg->queue);
10             /* ... */
11         }
12
13       ❸ if (backlog) {
14       ❹     backlog->complete(backlog, -EINPROGRESS);
15         }
16         /* ... */
17     } while (!kthread_should_stop());
18     return 0;
19 }
```

Listing 1: Excerpt from Linux' `queue_manag()` function defined in `drivers/crypto/mv_cesa.c` showing an uninitialized pointer dereference patched in April 2015 [24].

```
1  /* Example function-level annotation */
2  __attribute__(("writebeforeread"))
3  void function_with_potentially_uninitialized_variables() {
4      /* ... */
5  }
6
7  /* Example variable-level annotation */
8  void function_with_potentially_uninitialized_variables() {
9      int __writebeforeread uninitialized_variable;
10     /* ... */
11 }
```

Listing 2: Function- and variable- level annotations exposing `Write-before-Read` conditional capabilities to developers.

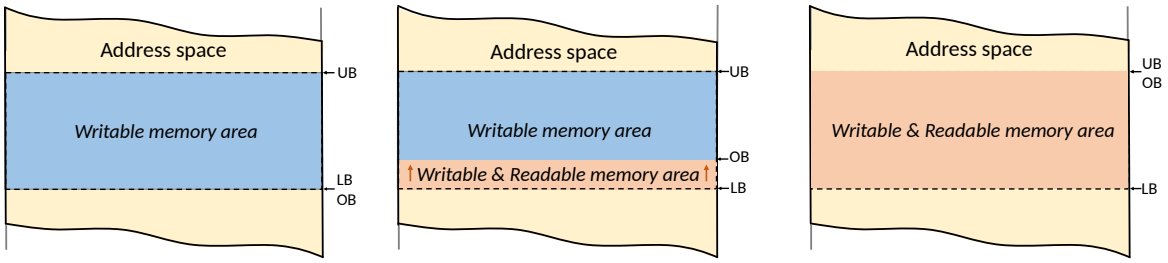
by the capability is initialized, while uninitialized data falls between the address and the upper bound. However, this approach has limitations, as discussed in § 6. To address this, we propose *operation-specific bounds* (OBs), which precisely track writes. OBs must be updated when a program writes to a region of memory to reflect the new bounds. Otherwise, stale or misaligned bounds can lead to false positives.

**Storing and updating operation bounds.** To store and update OBs efficiently, we propose leveraging unused bits in the baseline architecture address (Figure 1). This minimizes changes to the underlying hardware but makes significant compression of the OB necessary. Further details on the hardware modifications are provided in § 3.3 and § 4.

**Integration with existing architectures.** To be practical, CPs must integrate seamlessly with existing capability architectures, such as CHERI, that are gaining industry adoption. As a case study, we integrate CPs to CHERI while ensuring full compatibility with the underlying RISC-V ISA. In § 4, we describe the integration process and our hardware extension, "*Mon CHÉRI*", which introduces only minor modifications to the CHERI capability representation (§ 4.2).

**Maintaining capability monotonicity.** CPs add, similar to capability sealing (§ 2.2), controlled non-monotonicity to the CHERI design. They allow temporary suspension of permissions, which are regained once the associated condition is met. Our design (§ 3.2) guarantees that CPs do not to elevate privileges beyond the original capabilities.

**Maintaining capability linearity.** *Capability linearity* refers to the consistency and integrity of OBs across a program's




(a) Initial state of memory area. (b) State after store between LB and OB (c) Final state when OB reaches UB.

Figure 2: `Write-before-Read` conditional capability state transitions: *(a)* newly allocated memory under the `Write-before-Read` CP is write-only between the lower bound (LB) and upper bound (UB) and becomes gradually readable *(b)* after store operations advance the operation bound (OB). In the final state *(c)*, the OB has reached the UB and the conditional capability behaves identical to a corresponding conventional capability.

TABLE 1: Conditional permissions types for conditional capabilities

| Conditional permissions | Description |
|---|---|
| `Write-before-Read` | Memory must be written to at least once before read-access is granted |
| `Write-before-Execute` | Memory must be written to at least once before execute access is granted |
| `Write-before-Read-Only` | Memory must be written to exactly once before read access is granted |
| `Write-before-Execute-Only` | Memory must be written to exactly once before execute access is granted |
| `Write-Once` | Memory can be written to exactly once |
| `Read-Once` | Memory can be read exactly once |
| `Execute-Once` | Memory can be executed exactly once |

execution. To maintain this, we propose a new compiler pass to ensure that capabilities are properly updated as the program executes. This pass is implemented in the LLVM-based Mon CHÉRI toolchain, as detailed in § 3.3 and § 4.3.

**Minimizing run-time overhead.** A concern with any memory access control mechanism is the potential for performance degradation. For Mon CHÉRI, we ensure that checks involved in monitoring memory accesses do not add significant latency or computational overhead during normal program execution. In § 4.1, we explain how the OB checks are carefully designed to minimize additional cycles per instruction.

## 3.2. Conditional Capabilities

Conditional capabilities are designed to complement the conventional capability permissions with conditional permissions, allowing permissions to be tailored to specific needs or requirements for different sections of data or operations. The conventional capability permissions and CPs, when enabled for a capability, are evaluated in parallel; both sets of permissions need to be valid for memory access to be allowed. Conditional capabilities consist of two building blocks: 1) *Conditional permissions* enable capabilities to trace whether the corresponding condition is fulfilled. 2) *operation-specific bounds* enable CPs to be enforced at a granularity that allows the differentiation between accessible and inaccessible memory ranges within the conventional bounds of the capability.

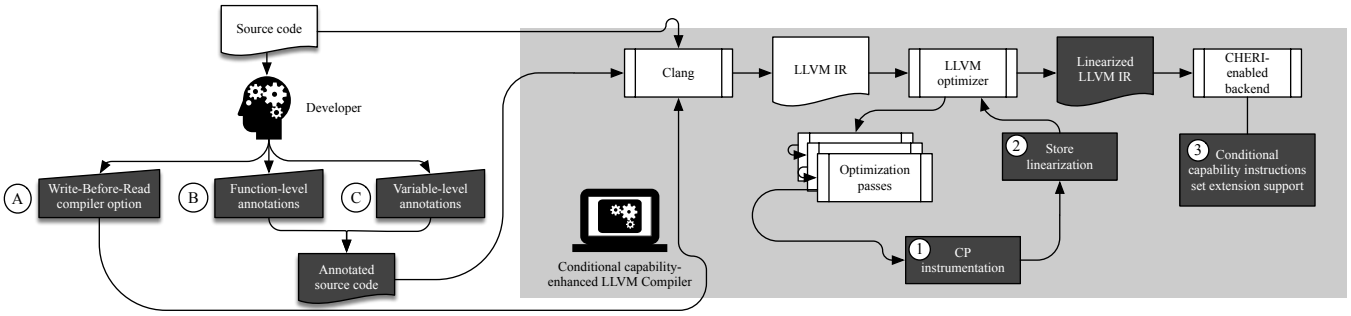
**Conditional permissions.** To describe conditional permissions, we denote conventional permissions that remain *unchanged* throughout the lifetime of a capability, as $\mathcal{P}^{\mathcal{U}}$, operations on memory as $\mathcal{X}$, and CPs as $\mathcal{P}^{\mathcal{C}}$. A $\mathcal{P}^{\mathcal{C}}$ is granted on the condition that $\mathcal{X}$ occurs: $\mathcal{X} \implies \mathcal{P}^{\mathcal{C}}$ (if $\mathcal{X}$, then $\mathcal{P}^{\mathcal{C}}$). If $\mathcal{X}$ does not occur, $\mathcal{P}^{\mathcal{C}}$ is not granted. On an architectural level, operations ($\mathcal{X}$) are limited to load, store and execute, while permissions ($\mathcal{P}^{\mathcal{U}}$ and $\mathcal{P}^{\mathcal{C}}$) can be read, write and execute. For simplicity, we will use the terms `Write`, `Read`, and `Execute` when referring to both $\mathcal{X}$ and $\mathcal{P}^{\mathcal{C}}$ in CP names. The effective permissions ($\mathcal{P}$) are a subset of the conventional and the conditional permissions:

$$\mathcal{P} = \begin{cases} \mathcal{P}^{\mathcal{U}} & if\ \mathcal{P}^{\mathcal{C}} = \emptyset \\ \mathcal{P}^{\mathcal{U}} \wedge \left(\mathcal{X} \implies \mathcal{P}^{\mathcal{C}}\right) & if\ \mathcal{P}^{\mathcal{C}} \neq \emptyset \end{cases}$$

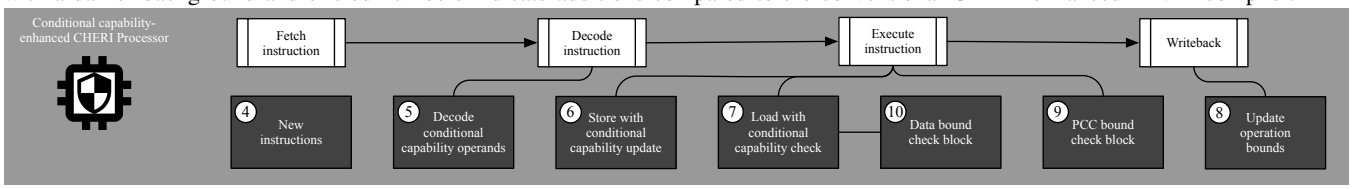
**Operation-specific bounds.** Tracking the memory range for which $\mathcal{P}^{\mathcal{C}} : \mathcal{X} \implies \mathcal{P}^{\mathcal{C}}$ requires *operation-specific* bounds. Each operation bound (OB) tracks the subset of memory within a capability's conventional bounds for which $\mathcal{X}$ has occurred. To accommodate multiple CPs at the same time, $\mathcal{X} \implies \mathcal{P}^{\mathcal{C}}$ and $\mathcal{Y} \implies \mathcal{Q}^{\mathcal{C}}$ where $\mathcal{X} \neq \mathcal{Y}$ requires two distinct OBs. Conversely, for $\mathcal{X} \implies \mathcal{P}^{\mathcal{C}}$ and $\mathcal{X} \implies \mathcal{Q}^{\mathcal{C}}$, where $\mathcal{P}^{\mathcal{C}} \neq \mathcal{Q}^{\mathcal{C}}$, one OB is sufficient.

Figure 2 illustrates an example use case for our conditional capabilities: `Write-before-Read`. In its initial state (Figure 2.a), the capability refers to a writable memory area with upper and lower bounds. If a store occurs in the memory area (Figure 2.b), the OB increases by the size of the store operand on the hardware level.

An advantage of our design is that it allows capability hardware to enforce a variety of novel access-control




(a) Overview of the conditional capability-enhanced LLVM compiler. Circled letters indicate the different usage models for conditional capabilities: (A) `Write-before-Read` compiler option, (B) function-level annotations, and (C) variable-level annotations. Components with a darker background and circled numbers indicate additions compared to the conventional CHERI-enhanced LLVM compiler.




(b) conditional capability-enhanced CHERI processor. Components with a darker background circled numbers indicate additions compared to the conventional CHERI processor.

Figure 3: High-level system architecture of the conditional capability-enhanced LLVM compiler and CHERI processor.

permissions beyond conventional `RWX`. We identify the CPs and corresponding use cases in Table 1. This set of CPs allows the definition of "single-use" code and data that enable policies that can be used to harden software against run-time attacks that aim to reuse existing program resource. Such attacks, e.g., code-reuse attacks such as return-oriented-programming [28], can use pre-existing sequences of instructions to craft malicious control flows that can be used to compromise program behavior. Code that a process executes only once, e.g., a fixed sequence of initialization instructions, could, using the `Execute-Once` CP be invoked once during program initialization and is subsequently rendered useless for code-reuse attacks later in the process' lifetime.

`Write-before-Execute` is useful for just-in-time (JIT) compilers that reuse memory buffers for generated code, making them targets for control-flow hijacking attacks using executable heap or stack buffers. `Write-before-Execute-Only` enables conditional capabilities to emulate eXecute-Only-Memory (XOM), typically limited to firmware, to protect secrets such as stack canary reference values or values used for control-flow enforcement [29] from being read by attackers.

### 3.3. High-Level System Architecture

Figure 3 depicts the high-level system architecture for our prototype conditional capability-enhanced LLVM compiler and CHERI processor. Inputs and process blocks in a darker color indicate the changes to a conventional CHERI-enabled compiler and processor architecture. For conciseness, in this description, we focus on `Write-before-Read` CPs.

In our prototype, conditional capabilities are exposed to developers through the three alternative usage models: (A) a `Write-before-Read` compiler option, (B) function-level annotations, and (C) variable-level annotations. Our current design of the compiler option (A) applies `Write-before-Read` to all stack variables, but we discuss possibilities for more intelligent heuristics in §7. Function- and variable-level annotations are implemented as Clang function and variable attributes, respectively (shown in Listing 2), which gives developers control over which variables `Write-before-Read` is applied to.

The conditional capability-enhanced LLVM compiler (Figure 3a) can operate either on original source code or source code annotated by the developer using the aforementioned attributes. This differs from the conventional CHERI-enabled LLVM in three ways: ① the *CP instrumentation* transform pass that marks variable for instrumentation based on the compiler option or developer annotations, and emits intrinsics that interface with the conditional capability hardware to initialize operation-specific bounds for newly created capabilities, ② a *store linearization* transform pass that ensures that accesses to uninitialized objects in memory are performed consistently through a capability that tracks the operation bound, and ③ support for the new conditional capability instructions in the CHERI RISC-V back end, allowing the compiler to interface with the conditional capability-enhanced CHERI processor.

The conditional capability-enhanced CHERI processor (Figure 3a) is modified to support the conditional capability instructions (④, § 4.1), and its instruction pipeline is augmented with logic to encode and decode capabilities containing an additional operation bound (⑤, § 3.2). Modified ⑥ store and ⑦ load logic updates and checks the operation bounds when executing store and load instructions,

respectively. Finally, ⑧ the pipeline performs writebacks of updated capabilities in register operands after the operations that update the operation bounds. Code capability checks for the Program Counter Capability (PCC) are done in a dedicated bounds check block ⑨ (needed for `Write-before-Execute`, `Write-before-Execute-Only`, and `Execute-Once`), while all data CPs use a general-purpose bounds check block ⑩.

# 4. Mon CHÉRI Implementation

In this section, we present Mon CHÉRI, our implementation of CPs for the CHERI-RISC-V architecture. We implemented two versions of Mon CHÉRI: 1) a Mon CHÉRI software model for the QEMU-system-CHERI128 full-system emulator, and 2) a Mon CHÉRI softcore based on the CHERI-Flute64 processor intellectual property (IP). The Mon CHÉRI ISA extension is described in § 4.1.

In addition, we extended the existing CHERI-LLVM toolchain [30] with support for the Mon CHÉRI ISA extension and `Write-before-Read` CP instrumentation for stack-allocated variables (§ 4.3). Finally, we added run-time `Write-before-Read` CP support for heap-allocated memory in the CheriFreeRTOS [31] and Two-Level Segregated Fit (TLSF) [32] memory allocators (§ 4.4).

## 4.1. Mon CHÉRI Extension for CHERI-RISC-V

**`CSetOpBounds` instructions.** The RISC-V ISA is extensible through portions of the instruction encoding space reserved for ISA extensions. We add a family of `CSetOpBounds` instructions to CHERI-RISC-V to the existing CHERIv9 [16] non-standard `Xcheri` extension in the *custom-2/rv128* opcode space to allow setting an initial, or updating an already set, value for a capability's operation bound. Our `CSetOpBounds` instructions require two operands: an existing capability and a length operand. When invoked, `CSetOpBounds` sets the operation bound to the range $[b, o]$ where $b$ is the "*base*" encoded as part of the conventional capability bound, and $o$ is the "*operation top*" encoded into the operation bound (see § 4.2). `CSetOpBounds` does not allow programs to increase the operation bounds, while they are allowed to invoke `CSetOpBounds` to decrease the operation bounds of a capability. The operational bounds set by `CSetOpBounds` are restricted to be within the original capability bounds.

Additionally, `CSetOpBounds` must identify the CPs which operation bound to set. Lacking free operands in the instruction encodings available in `Xcheri`, we chose to encode the CP information into the `CSetOpBounds` opcode. The `CSetOpBounds` instruction variants are shown in Table 2.

**Conditional permissions.** We assign new *conditional permissions control bits* from the CHERI capability representation ($p_{op}$ in Figure 4b). The CHERI-Flute64 IP reserves 12-bits for hardware-defined permissions, i.e., those authorizing load, store, etc., and 4-bits for software-defined permissions, which interpretation are left open by the CHERIv9 specification. For the purposes of prototyping CPs, we utilize the four available software-defined bits, treating them as a 4-bit integer that enumerates one of 16 possible, mutually exclusive permission states. Five of those states correspond to the CPs in table 2, ten modes are left unused, and one mode, the zero value, corresponds to the default state in which CP enforcement is disabled. In this default state, protections such as `Write-before-Read` are inactive, allowing uninitialized memory access within the capability's bounds. Invoking `csetwbropbound` on a capability enables `Write-before-Read` enforcement for that capability.

**Processor pipeline changes.** To avoid increasing processor latency, bounds checks are carefully structured within the execute instruction block of the pipeline (see Figure 3b), which has two stages. In the first stage, where single-cycle arithmetic logic unit (ALU) operations are performed, the CHERI implementation checks the conditional permissions and initiates the data capability bounds check, while the actual bounds check occurs in the second stage (used for longer-latency operations like memory access). OB checks run in parallel with the general-purpose bounds checking to minimize their latency. For stores and loads, the first stage checks if the conditional permission allows the operation, and if the target address results in an update of the current operation bound; if it does and is adjacent, the second stage extends the bound to cover it, with the update committed in the writeback stage. Table 2 shows how the OB checks and writeback are used by different CPs. Any violation raises a CHERI protection exception in the second stage.

**Avoiding data hazards.** Conditional Capabilities create dependencies between instructions that are typically independent. This is because stores using conditional capabilities update the OB in the capability operand, while stores on conventional capabilities do not. As a result, when a store using a conditional capability is immediately followed by a load or another store using the same capability, a *data hazard occurs*: the latter operation requires the updated OB at the ALU stage for the OB check but is not available in the operand register until the writeback stage completes.

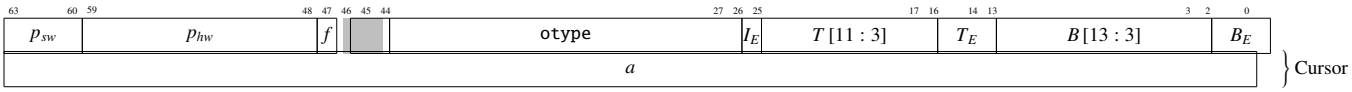
To solve this, we implemented a *bypass*—a new data path within the pipeline—that forwards the updated OB from the memory access or writeback stage to the ALU stage. This bypass activates when the processor detects that the next instruction in the pipeline operates on the same conditional capability. This avoids the need to stall the processor or reorder instructions during compilation and is compatible with conventional RISC-V instruction ordering. Appendix § A.1 shows an example of a data hazard that is avoided by the bypass.

## 4.2. Adding Operation Bounds to Capabilities

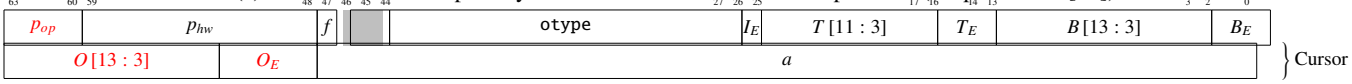
The CHERI ISA introduced capability compression in ISAv6 [33], with the current CHERI Concentrate [19] compression scheme introduced in ISAv7 [34]. Compression reduces the in-memory size of capabilities, which would otherwise occupy 256 bits, quadrupling the space needed for native 64-bit pointers, and increasing cache footprint and memory bandwidth requirements. Expanding capabilities

TABLE 2: Architectural changes to CHERI-RISC-V by conditional permission. The **CSetOpBounds variants** column shows the corresponding `CSetOpBounds` instruction mnemonic for each CP. The **Pipeline changes** column shows the impact on the *Load*, *Store*, and *Execute* logic inside the processor. A ○ indicates the operation is unaffected, a ◐ indicates the operation is augmented with an OB check, a ◑ indicates the operation is augmented with a writeback that, under certain conditions, updates the conditional capability OB. Finally, a ● indicates the operation is augmented with both.

| Conditional Permission | CSetOpBounds variants | Pipeline changes | | |
|---|---|---|---|---|
| | | *Load* | *Store* | *Execute* |
| `Write-before-Read` | `csetwbrbound` | ◐ | ◑ | ○ |
| `Write-before-Execute` | `csetwbxbound` | ○ | ◑ | ◐ |
| `Write-before-Read-Only` | `csetrobound` | ◐ | ● | ○ |
| `Write-before-Execute-Only` | `csetxobound` | ○ | ● | ◐ |
| `Write-Once` | `csetwtbound` | ○ | ● | ○ |
| `Read-Once` | `csetrtbound` | ● | ◑ | ○ |
| `Execute-Once` | `csetxtbound` | ○ | ○ | ● |




(a) 128-bit CHERI Capability with CHERI Concentrate compression (adapted from [16])




(b) 128-bit CHERI Capability with compressed operation bound

Figure 4: Layouts of 128-bit CHERI Capabilities without (a) and with (b) compressed operation bound. The labels indicate the fields for software-defined and hardware-defined permissions ($p_{sw}$ and $p_{hw}$ respectively), CP control bits ($p_{op}$), flag ($f$), object type (`otype`), internal exponent ($I_E$), compressed top ($T$) including high part of exponent ($T_E$), compressed base ($B$) including low part of exponent ($B_E$), and cursor consisting of the pointer address ($a$) in a, the address and compressed operation bound ($O$, $O_E$) in b.

further to include additional operation bounds is impractical. However, most modern 64-bit operating systems (OSs) use only part of the 64-bit virtual address space. For instance, RISC-V Linux uses 48-bit addresses [35], and 32-bit OSs like FreeRTOS use 32 bits even on 64-bit hardware. We leverage these unused bits to store an additional 16-bit OB.

**Conditional capability masking.** Masking a portion of the address in cases where OSs do not fully utilize 64-bit addresses has precedents in both conventional processors, such as in Arm's Top Byte Ignore (TBI) [36], Intel Linear Address Masking (LAM) [37], and AMD Upper Address Ignore (UAI) [38] features, as well in the CHERI design as alternative compression formats described in Appendix E of the CHERI ISAv9 [16]. To encode operation bounds, Mon CHÉRI applies an address mask when a register holds a conditional capability. Since CHERI-RISC-V uses a merged register file, general-purpose registers can hold either a 64-bit integer or a 128-bit capability. Unlike prior RISC-V pointer masking proposals [39], Mon CHÉRI's address mask is limited to conditional capabilities, leaving conventional CHERI capabilities and regular pointers unchanged.

**CHERI compressed capability representation.** A raw 256-bit capability is comprised of three virtual addresses: base ($b$), top ($t$), and address ($a$). The 128-bit representation of capabilities utilizes the redundancy between the three addresses and stronger alignment requirements (proportional to object size) for a more compact representation.

Figure 4a shows the capability format for CHERI Concentrate. $B$ and $T$ encode the $b$ and $t$ bounds in one of two formats depending on the internal exponent ($I_E$) bit: if $I_E = 1$ then an exponent ($E$) is stored in the lower three bits of $B$ and $T$ ($B_E$ and $T_E$) reducing their precision by three bits. $E$ determines the position at which $B$ and $T$ are inserted into $a$ to obtain $b$ and $t$. Otherwise ($I_E = 0$, $E = 0$) the full width of $b$ and $t$ are used. Their width is determined by an encoding parameter: *mantissa width* ($MW$) that determines the precision of the decoded bounds. The CHERI ISAv9 uses $MW = 14$ for 128-bit capabilities. However, $t$ is further compressed by two bits as the top two bits of $t$ can be derived from the equation $T = B + L$ where the most significant bit of $L$ ($L_{msb}$) is known from the values of $I_E$ and $E$ and a carry bit is implied if $T\,[11:0] < B\,[11:0]$ since $t$ is known to be larger than $b$.

When decoding the bounds, $b$ and $t$ are derived from $a$ by substituting $MW$ bits, $E$ to $E + MW$, with $B$ and $T$ and clearing the bottom $E$ bits. To allow $a$ to span a larger region

```
define dso_local void @function_with_potentially_uninitialized_variables() local_unnamed_addr addrspace(200) #11 !dbg
    !322 {
entry:
  %uninitialized_variable = alloca i32, align 4, addrspace(200), !clang.decl.ptr !325, !clang.var.writebeforeread !100
❶ %0 = call ptr addrspace(200) @llvm.cheri.bounded.stack.cap.i64(ptr addrspace(200) %uninitialized_variable, i64 4)
❷ %1 = call ptr addrspace(200) @llvm.cheri.cap.op.bounds.set.i64(ptr addrspace(200) %0, i64 0)
   /* ... */
}
```

Listing 3: LLVM intermediate representation (IR) of Listing 2 after CP LLVM Pass Instrumentation

while maintaining the original bounds, the most significant bits of $t$ and $b$ $a_{top} = a[63 : E + MW]$ can be adjusted up or down using corrections $c_t$ and $c_b$. The detailed description of the CHERI Concentrate compression can be found in Section 3.5.4 of the CHERI ISAv9 specification [16].

**Mon CHÉRI compressed capability representation.** Figure 4b shows the capability format for Mon CHÉRI. We encode the operation bound, operation top ($o$) in two fields in the most significant 16-bits of the cursor: $O[13 : 3]$ (11 bits) and $O_E$ (5 bits) which are freed by limiting $a_{top} = a[47 : E + MW]$. When $I_E = 0$, $o$ is stored identically to $b$ with its lowest three bits $O[2 : 0]$ derived from the most significant bits of $O_E$ ($O_E[4 : 2]$). When $I_E = 1$, up to five bits from $O_E[E + 2 : 0]$ are used to store the least significant bits of $o$. Our current implementation limits $O_E$ to five bits, limiting $o$ to an $I_E$ of at most 2. We discuss methods to alleviate this limitation in § 7. Expanding $o$ to full 48 bits happens similarly to $b$ with its own correction $c_o$. Figure 6 in § A.2 illustrates the changes relative to CHERI Concentrate compression.

## 4.3. Mon CHÉRI Support for CHERI-LLVM

**CP instrumentation.** Listing 3 shows an excerpt of the LLVM IR of one of the functions in Listing 2 after CP instrumentation (see ① in Figure 3). The CHERI-LLVM compiler adds capability bounds to stack variables in an IR-level compiler pass, `CheriBoundAllocas`. This pass replaces every IR stack allocation instruction (`alloca`, ❶ in Listing 3) with a `llvm .cheri .cap .bounds .set` intrinsic. These are then replaced with CHERI `csetbounds` instructions by the CHERI-RISC-V backend. We extended this pass to add `llvm .cheri .cap .op .bounds .set` intrinsics (❷ inListing 3)for variables that are either annotated with `__writebeforeread` (Ⓒ in fig. 3a), in functions annotated with `__attribute__(("writebeforeread"))` (Ⓑ), or all stack variables, when compiling with the `-cheri-write-before-read` option (Ⓐ). The `llvm .cheri .cap .op .bounds .set` intrinsic is replaced with `csetwbr-bound` instructions by the backend.

**Optimizing `Write-before-Read`.** The `CheriBoundAllocas` pass checks if `alloca` instructions fall within the original capability bounds and omits `llvm .cheri .cap .bounds .set` in those cases. We disable this optimization for variables that receive `llvm .cheri .cap .op . bounds .set` since it does not guarantee stores to `Write-before-Read` variables fall within the OBs. However, not all variables require run-time CP checks. For `Write-before-Read`, we optimize arrays and scalar variables allocated in function entry blocks by using a simple, non-heuristic analysis that checks if they are fully initialized before being accessed. The analysis inspects each store in the function's first basic block and verifies whether they are preceded by loads to the same allocation. For arrays, we track initialization with a vector, checking store instructions for the base pointer or specific indices, ensuring each load is preceded by a store through dominance analysis [40]; if a load occurs before a store, the index is marked uninitialized in the vector. For scalars, we verify whether a store targets the variable, and each load is dominated by a corresponding store. Variables shown to be initialized do not receive the `Write-before-Read` CP and, therefore, can be checked by `CheriBoundAllocas` for capability-bound optimization. In § 7, we discuss the possibility of using heuristic static analysis to optimize `Write-before-Read` variables further.

**Store linearization.** LLVM IR uses static single-assignment (SSA) form, where each variable has exactly one assignment. The compiler creates new IR variables to maintain this rule when a variable has multiple assignments. If a variable is accessed through different control-flow paths, a phi (Φ) function is introduced to merge the values from these paths.

During register allocation, the compiler maps variables to processor registers. However, due to register pressure—the number of live variables exceeding the number of available registers—the compiler generates *spill code* that moves variable contents between memory and registers. Live-range splitting optimizes when variables are spilled and can make use of the fact that the same variable may, at times, be available in multiple registers simultaneously. However, when a conditional capability is stored in multiple registers, the state of its operation bound may become inconsistent across instances. This can occur due to: 1) register spilling, or 2) register forking, where the conditional capability state differs between registers. Inconsistencies arise when a duplicated capability is stored in memory, becomes stale as its copy's operation bound is updated, and is later restored.

To address this, we introduce a *store linearization pass*, which runs after SSA optimizations and ensures a single, canonical conditional capability instance is maintained across stores. Performing store linearization after optimizations guarantees robustness across optimization levels.

Store linearization inserts placeholder function calls for every store operand in the IR to prevent live-range splitting from spilling a canonical conditional capability, causing it to grow stale. This placeholder is an identity function—effectively a no-op—taking a conditional capability operand

```
#include <cheriintrin.h>

void *ptr malloc (size_t size) {
     /*Dynamic Memory Memory management */
   ① cheri_bounds_set(ptr, size);
   ② cheri_opbounds_set(ptr, 0, WriteBeforeRead);
     return ptr;
}
```

Listing 4: Example of CP-enhanced version of `malloc()`.

and returning it unchanged. This, however, creates a data dependency between the input and output operand, signaling to the live-range splitting algorithm that the conditional capability has changed since its use in the store.

Array indexing via the LLVM IR `GetElementPtr` (GEP) instruction requires special handling. GEP takes a pointer and an offset, returning a new pointer (in SSA-form) pointing at the specified offset in the array. Store linearization inserts a placeholder function for the GEP operand, not the indexing pointer. In practice, the CHERI-RISC-V backend generates code using integer-relative store instructions (`s[bhwd]`), where the GEP input is used as a capability operand with an offset. Therefore, the linearization targets the original operand to extend its liveness.

Finally, the pass recursively replaces any variables holding conditional capabilities used store operands with those returned by the corresponding placeholder functions.

**Escape value analysis.** When a value in SSA form escapes a code block, it must be updated when accessed along different control-flow paths. The `reg2mem` transformation [41] in LLVM handles escaped values by: 1) allocating stack memory for each escaped SSA register, 2) storing SSA values in this memory before exiting a block, and 3) reloading values upon entering a new block. This ensures consistency across code paths. Store linearization modifies `reg2mem` to update conditional capabilities across control-flow paths, similar to how phi functions merge SSA values. One optimization available to store linearization stems from the original `reg2mem` also tracking output operands from the GEP instruction as escaped values. The special handling of array indexing allows us to avoid storing output operands from GEP in memory.

Similarly, conditional capabilities stored explicitly in memory must be updated after being used for store operations, even in straight-line code. Store linearization recursively checks if the operand used in a store is itself stored in memory. If so, an additional store operation is inserted to update the capability stored in memory. Listings 7 and 8 in § A.3 illustrate this transformation.

### 4.4. Mon CHÉRI Support for Memory Allocators

We implemented the `cheri_opbounds_set()` application programming interface (API) to provide low-level support for conditional capabilities in system software, such as memory allocators. To enable `Write-before-Read` CPs for heap allocations, the allocator must initialize the capabilities to allocate memory with the appropriate CP.

Listing 4 shows how we modified the CheriFreeRTOS memory allocator, and the TLSF allocator [32], previously ported to CHERI [42] to enforce `Write-before-Read` on allocation made by `malloc()`. A CHERI-aware `malloc()` already uses the `cheri_bounds_set()` intrinsic to set bounds of the return pointer, `ptr`, based on the allocation size. To make it conditional capability-aware, we added a call to `cheri_opbounds_set(ptr, 0, WriteBeforeRead)` before returning from `malloc()`. This explicitly sets the operation bounds of `ptr` to zero and configures the CP control bits to indicate `ptr` should be treated as `Write-before-Read` by the hardware.

TABLE 3: Detection rate of Mon CHÉRI on uninitialized memory issues from Juliet Test Suite [43] CWE457 test cases. Green cells indicate the true positives and true negatives, while red cells indicate the false positives and false negatives.

| Ground truth | Mon CHÉRI **Positive** | Mon CHÉRI **Negative** |
|---|---|---|
| **Bad** | 560<br>100% | 0<br>0% |
| **Good** | 6*<br>1% | 554<br>99% |

*Following the "Good" and "Bad" classification in Juliet, Mon CHÉRI reports six false-positives. Yet, these cases do exhibit uninitialized memory access behavior (cf. § 5.1).

## 5. Evaluation

We evaluated the Mon CHÉRI prototypes for functionality, security, performance, and area cost. The functional and security evaluation (§ 5.1) was performed on a conditional capability-enhanced QEMU-system-CHERI128 full-system emulator, CheriFreeRTOS [31], which integrates CHERI-based compartmentalization, and a `Write-before-Read` memory allocator (§ 4.4). For performance and area cost, we extended the CHERI-RISC-V field-programmable gate array (FPGA) softcore [44] based on the open-source Bluespec [45] RISC-V processor IP with the CP ISA extension (§ 4.1). This processor family includes Piccolo, Flute, and Tooba IP cores. Although sharing significant portions of Bluespec System Verilog (BSV) code, our performance evaluation uses the 64-bit CHERI-RISC-V Flute (RV64ACDFIMSUxCHERI). This prototype, MonCHÉRI-Flute64, was validated against RISC-V specifications using 229 RISC-V ISA tests [46].

### 5.1. Functional and Security Evaluation on QEMU

We used the U.S. National Institute of Standards and Technology (NIST) Juliet Test Suite [43], which includes thousands C/C++ of test cases that demonstrate common programming defects that lead to memory vulnerabilities, to evaluate Mon CHÉRI. These tests are organized by CVE numbers, with "bad" versions exhibiting vulnerabilities and "good" versions showing patched code. We focused on the CWE-457 [47] (Use of Uninitialized Variable) C test cases, covering 560 bad and 560 corresponding good cases. These examples cover a range of realistic scenarios, such as conditional control flows where variables might

TABLE 4: Area cost on VCU118 @ 100MHz expressed in number of lookup tables (LUTs) and number of registers.

| | **LUTs** | | | | | | **registers** | | |
|---|---|---|---|---|---|---|---|---|---|
| | logic | Δ | | memory | Δ | | registers | Δ | |
| CHERI-Flute64 | 139109 | – | | 10649 | – | | 134427 | – | |
| MonCHÉRI-Flute64 | 142069 | 2960 | 2% | 10705 | 56 | 0.5% | 135114 | 687 | 0.5% |

TABLE 5: Performance cost on VCU118 @ 100MHz expressed as CoreMark test results. The CoreMark score for a processor is reported as CoreMark-iterations-per-second-per-core-MHz. The Δ is relative to CHERI-Flute64 nocap results.

| | | **CoreMark** | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | Binary size | Δ | | Total ticks | Δ | | Total time (sec) | Δ | | Iterations/sec | Δ | | Score |
| | CHERI-Flute64 | | | | | | | | | | | | | |
| | (baseline) nocap | 43728 | – | | 2704279286 | – | | 27 | – | | 370 | – | | 3.7 |
| | purecap | 49872 | 6144 | 14.05% | 2878393823 | 174114537 | 6.43% | 28 | 1 | 3.57% | 357 | 13 | 3,51% | 3.57 |
| | MonCHÉRI-Flute64 | | | | | | | | | | | | | |
| | nocap | 43728 | – | | 2704301802 | 22516 | 0.42% | 27 | 0 | 0% | 370 | 0 | 0% | 3.7 |
| | purecap | 49872 | 6144 | 14.05% | 2878516285 | 174236999 | 6.44% | 28 | 1 | 3.57% | 357 | 13 | 3.51% | 3.57 |
| ❶ | `Write-before-Read` + purecap | 50224 | 6496 | 14.85% | 2949321554 | 245019752 | 9.06% | 29 | 2 | 7.04% | 344 | 26 | 7.00% | 3.44 |
| ❷ | `Write-before-Read` + purecap excluding store linearization | 49232 | 5504 | 12.59% | 2882957831 | 178678545 | 6.61% | 28 | 1 | 3.57% | 357 | 13 | 3.51% | 3.57 |

remain uninitialized in certain branches (e.g., as illustrated in Listing 1), function calls that pass variables assumed to be initialized, and cases involving complex data types like arrays, pointers, and structures.

To extensively assess Mon CHERI's detection rate for uninitialized variable accesses, we made specific modifications to the Juliet test suite. Modern compilers like LLVM tend to optimize away uninitialized memory accesses or reject them outright due to their sophisticated static analysis capabilities. To prevent this behavior, we declared all variables as volatile, which stops the compiler from applying optimizations based on undefined behavior. This allows us to build the tests with optimization level `-O2` while ensuring all test bad cases trigger uninitialized memory accesses.

The tests were executed within CheriFreeRTOS, which uses an instrumented allocation API to track memory accesses. CheriFreeRTOS provides a "compartmentalize and return" mode that isolates each test case in a separate compartment, allowing test cases resulting in a CHERI protection fault to return control to the caller, which records the result and proceeds to the next test case. The test cases were compiled with the Mon CHÉRI-enhanced CHERI-LLVM at optimization level `-O2` and with `Write-before-Read` instrumentation and store linearization but without the `Write-before-Read` optimization described in § 4.3.

The results in Table 3 show that Mon CHÉRI detects all "bad" cases (100% true-positive rate) and reports only six false positives (1% false-positive rate). Upon closer inspection, these false positives stem from cases where uninitialized variables are copied, but never used. Although these cases are technically valid violations of the `Write-before-Read` policy, they do not pose a security risk, as data is immediately overwritten. Analysis tools relying on taint propagation [48] might not flag these cases, as the uninitialized memory does not propagate. As Mon CHÉRI enforces policies at an architectural level, distinguishing between these benign cases and actual vulnerabilities is not currently possible. All six false positives share this pattern, where the uninitialized memory is copied but later overwritten before being used. We provide the source code for one of these cases in § A.3, Listing 6. Further, all six cases have straightforward software workarounds that can be applied, once detected, to initialize variables early with a default zero value to avoid uninitialized access.

We evaluated the effectiveness of the store linearization pass by comparing the detection rate for the Juliet tests instrumented with and without store linearization. Without store linearization 119 out of 560 "good" test cases exhibit false positives (21% false positive rate). This demonstrates store linearization provides a significant improvement to the detection accuracy of Mon CHÉRI. We also verified the `Write-before-Read` optimization did not affect the detection accuracy as it only omits CP instrumentation for variables that are statically verified to be fully initialized.

For comparison, we compiled the Juliet test suite with all relevant warnings enabled in GCC and Clang/LLVM. GCC detected 170 out of 560 uninitialized cases (30%) at `-O0` and 173 cases (31%) at `-O2`. Clang/LLVM detected 117 cases (21%) regardless of optimization level. Detection rates for the Valgrind and Dr. Memory dynamic analysis tools range from below 10% to levels comparable to Mon CHÉRI, depending on the Juliet and compiler configurations. Under the same configuration used for Mon CHÉRI, (volatile variables and Clang with `-O2`), Valgrind and Dr. Memory detected 400 cases (71%) and 388 cases (69%), respectively.

As the Juliet CWE457 tests do not exhibit patterns that would require `Write-before-Execute`, `Write-before-Read-Only`, `Write-before-Execute-Only`, or `Write-Once` CPs we verified the functionality of these CPs in the conditional capability-enhanced QEMU-system-CHERI128 implementation using purpose-built synthetic test cases.

### 5.2. Performance and Area Evaluation on FPGA

**Area cost.** We synthesized MonCHÉRI-Flute64 at 100 MHz on an AMD Virtex UltraScale+ VCU-118 FPGA. Compared to the CHERI-Flute64 design, the area cost of MonCHÉRI-Flute64 increased by only 2%, a small cost considering the overhead of adding CHERI. The majority of this additional logic is shared across many CPs.

**Performance cost.** We integrated the MonCHÉRI-Flute64 softcore into the BESSPIN-GFE security evaluation platform [49], which allows for a full-system evaluation of Mon CHÉRI performance. The GFE system includes the MonCHÉRI-Flute64 softcore, a BootROM, Soft Reset and JTAG, UART, Ethernet/DMA, DDR4, and Flash controllers. A host-based gdb debugger connects to the system over the USB/JTAG connector, and a host-based console connects over USB/UART. We measured performance using the EEMBC CoreMark [50] benchmark, running bare-metal on the GFE. Although MonCHÉRI-Flute64 supports all CPs in Table 2 we focus in these experiments on `Write-before-Read` as it is applicable to all variables in CoreMark. Consequently, applying it to all variables in the CoreMark benchmark code provides a worst-case estimate of Mon CHÉRI's performance impact. As we expect other CPs to only be applied to a subset of variables, their impact is a fraction of `Write-before-Read`'s.

Table 5 compares the performance results MonCHÉRI-Flute64 to the CHERI-Flute64 across different configurations: no capability enforcement (no-cap), pure-capability mode (purecap), and `Write-before-Read` enabled in pure-capability mode (`Write-before-Read` + purecap ❶). The results show that the `Write-before-Read` extension adds a modest ≈ 3.5% overhead over pure-capability mode, with minimal impact on baseline performance (≈ 0.4%) when capability enforcement is disabled. The combined performance impact of `Write-before-Read` and CHERI pure-capability mode over the baseline performance with no capability enforcement is 7%.

We also compared the performance of MonCHÉRI-Flute64 with and without store linearization (❷). Although store linearization is necessary, as explained § 5.1, for the correctness of `Write-before-Read` enforcement, disabling the hardware fault for this experiment allows us to compare the performance impact of the hardware changes with that of the store linearization program transformation. Enabling store linearization resulted in most of the performance degradation observed in earlier tests (≈ 3.5%), with the hardware changes contributing negligible additional overhead (≈ 0.2%). This suggests that performance can be improved further by optimizing the store linearization strategy.

**Microbenchmarks.** To assess the impact of our store and load pipeline changes we microbenchmark stores and loads between CHERI-Flute64 in purecap mode and MonCHÉRI-Flute64 in `Write-before-Execute` + purecap mode. The store microbenchmark writes a 256-element array, recording total ticks. The load microbenchmark, we measured read from an of equal size. Here we report the difference in mean times for the experiment over 10 repetitions. We observed a negligible difference between CHERI-Flute64 and MonCHÉRI-Flute64: ≈ 5 ticks for the load and ≈ 30 ticks for the store benchmark.

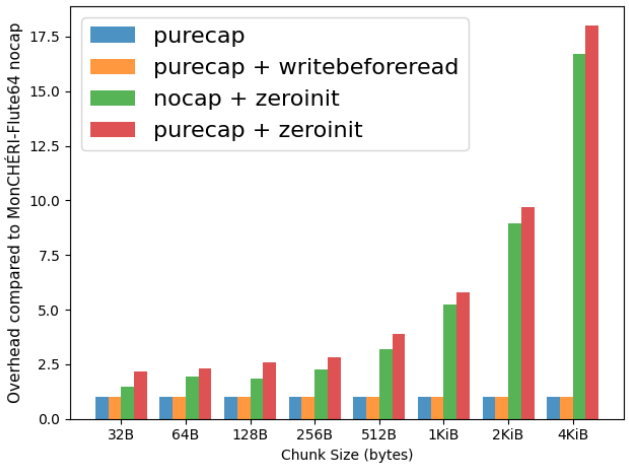



Figure 5: TLSF allocator microbenchmark. Bars show overhead relative to TLSF in MonCHÉRI-Flute64 nocap mode.

Finally, we evaluated the impact of Mon CHÉRI on the TLSF allocator through microbenchmarks that allocate and free 1 MB of memory in chunks ranging from 32 bytes to 4 KiB. The results, shown in Figure 5, indicate that the `Write-before-Read`-enhanced TLSF allocator introduces negligible performance overhead: ≈ 0.1% compared to purecap and ≈ 1.5% (g.m.) compared to no-cap. The overhead of `Write-before-Read` + purecap is constant regardless of allocation size, while the overhead of zero-initialization increases linearly with allocation size.

## 6. Related Work

Various techniques for detecting the use of uninitialized variables are routinely used in modern software development. We categorize the existing approaches into six categories, as shown in Table 6. In this section, we compare conditional capabilities and Mon CHÉRI to existing approaches.

**Static analysis.** Static analysis evaluates a program's code without executing it. By analyzing the code's structure and syntax, compilers, and dedicated static analysis tools can detect potential errors, security issues, and coding standard violations. Static analysis is performed early in development, allowing developers to address problems proactively.

Most major C/C++ compilers, including GCC [51], Clang [52], Intel Data Parallel C++ (DPC++), and Microsoft Visual C++ (MSVC) support compile-time checks for uninitialized variables. Static analysis tools such as Adlint [63], Clang-Check [64], Clang-Tidy [65], CodeSonar [66], Coverity Scan [67], CppCheck [68], Flawfinder [69], Frama-C [70], IKOS [71], Infer [72], and LCLint/Splint [73] can also check for uninitialized variables in C/C++ code. These tools, particularly commercial tools focused on secure software development, are commonly referred to and marketed as static application security testing (SAST).

TABLE 6: Related work

| | Support for stack allocations | Support for heap allocations | Usable with dynamic analysis | Unaffected by optimization level | Performance overhead | Memory overhead |
|---|---|---|---|---|---|---|
| **Static code analysis**[1] | | | | | | |
| `-Wuninitialized` (GCC [51] | ✓ | ✓ | ✓ | ✗ | – | – |
| `-Wmaybe-uninitialized` (GCC [51]) | ✓ | ✓ | ✓ | ✗ | – | – |
| `-Wuninitialized` (Clang [52]) | ✓ | ✓ | ✓ | ✓ | – | – |
| `-Wsometimes-uninitialized` (Clang [52]) | ✓ | ✓ | ✓ | ✓ | – | – |
| **Dynamic analysis** | | | | | | |
| Valgrind Memcheck [53] | ✓ | ✓ | ✓ | ✗ | 20× | yes |
| Dr. Memory [54] | ✓ | ✓ | ✓ | ✗ | 10× | yes |
| **Sanitizers** | | | | | | |
| Memory Sanitizer [55] | ✓ | ✓ | ✓ | ✓ | 2×–4× | yes |
| **Redundant execution** | | | | | | |
| DieHard [56] | ✓ | ✓ | ✓ | ✓ | ≈ 40% | yes |
| Differential Replay [57] | ✓ | ✓ | ? | ✓ | 22×–24× | yes |
| **Automatic initialization** | | | | | | |
| Secure deallocation [58] | ✓ | ✓ | ✓ | ✓ | <7% | – |
| UniSan [59] | ✓ | ✓ | ✗ | ✓ | ≈ 5% | – |
| SafeInit [23] | ✓ | ✓ | ✗ | ✓ | ≈ 5% | – |
| STACKLEAK [60] | *[2] | ✗ | ✗ | ✓ | ≈ 1% – 5% | – |
| initAll (MSVC) [5] | ✓ | ✗ | ✗ | ✓ | ≈ 10% | – |
| `-ftrivial-auto-var-init` (GCC [51], Clang) | ✓ | ✗ | ✗ | ✓ | ≈ 1% [61] – 35% [62] | – |
| **Hardware-based detection** | | | | | | |
| Uninitialized capabilities [25] | *[4] | ✗ | ✓ | ✓ | ? | *[3] |
| Capstone [27] | *[4] | ? | ✓ | ✓ | ≈ 50%[5] | *[3] |
| Mon CHÉRI | ✓ | ✓ | ✓ | ✓ | ≈ 3.5% / 7%[6] | *[3] |

[1] For conciseness, we include only compiler-based static analyzers focusing on uninitialized variable detection in Table 6. [2] STACKLEAK protects the Linux kernel call stack after system calls.
[3] Memory overhead due to replacement of pointers with CHERI / Capstone capabilities. [5] Overhead reported for the Capstone isolation model by Yu et al. [27].
[4] Uninitialized capabilities and Capstone protect stack frames at coarser granularity than what uninitialized variable detection for individual variables requires.
[6] Overhead for Mon CHÉRI given both excluding overhead for CHERI (purecap) and including overhead for CHERI, based on Table 5.

However, all static analysis is limited by Rice's Theorem [74], which states that analyzing non-trivial properties of program behavior is undecidable for Turing-complete languages. This implies that static detection of uninitialized variables is equivalent to solving the halting problem. As a result, static methods are approximations, balancing verbosity with false positives and analysis time.

**Dynamic analysis.** Dynamic analysis examines program behavior during execution. Unlike static analysis, which analyzes code without running it, dynamic analysis monitors run-time characteristics such as performance, memory usage, and interaction with system resources. It can, therefore, identify bugs, security vulnerabilities, and performance bottlenecks that may not be evident through static analysis. Tools like Valgrind Memcheck [53] and Dr. Memory [54] use dynamic instrumentation frameworks (e.g, DynamoRIO [75]) to track memory accesses and detect use of uninitialized variables. These frameworks act as process virtual machines, interposing and transforming original program instructions before they get executed by the hardware. This enables dynamic analysis to freely transform the target program and add extra instrumentation around program instructions to keep track of memory accesses

Dynamic analysis, however, incurs significant performance overhead, often degrading program speed by 10× to 20×, making it impractical for continuous use. These tools also struggle to accurately identify the origin of uninitialized memory. For example, Memcheck traces uninitialized variables to heap blocks or stack allocations that occur in a particular function, but may not always pinpoint the exact source. Dr. Memory, meanwhile, does not detect uninitialized variables smaller than a machine word.

**Sanitizers.** Sanitizers are compiler-based tools designed to detect memory-safety, concurrency, and undefined behavior issues in C and C++ programs. They intercept memory accesses via compile-time instrumentation, offering higher efficiency and accuracy than dynamic analysis tools. Sanitizers like MemorySanitizer [55] detect uninitialized stack- and heap-allocated memory at individual bit granularity, with less overhead (2× to 4× slowdown) compared to Valgrind's one-order-of-magnitude slower dynamic analysis [76].

However, like Valgrind Memcheck, MemorySanitizer only reports uninitialized values that affect control flow, which limits its effectiveness in identifying memory issues related to information disclosure to, e.g., uninitialized data that overlaps with previously allocated pointers and which can reveal information to bypass ASLR.

**Redundant execution techniques.** Redundant execution techniques, such as DieHard [56], enhance memory safety by using randomized memory allocation and replication. DieHard scatters memory allocations across a large heap, reducing the chances of uninitialized memory being adjacent to other active regions. By comparing execution across multiple program replicas, DieHard can detect discrepancies caused by uninitialized memory reads. Similarly, differential replay [57] captures execution traces and replays them with varied initial memory states. These techniques assume that correctly initialized variables will produce consistent

outputs across runs, while uninitialized variables will lead to variations due to differences in their initial memory state. By identifying discrepancies between execution instances, DieHard and differential replay can pinpoint instances where uninitialized variables are affecting the program's behavior.

Multi-variant execution [77]–[82] generalizes this concept, running multiple functional equivalent, but independently developed, programs in parallel. This principle has been applied to safety-critical software in various domains, such as train switching and flight control systems, electronic voting, and specialized software testing, such as detecting zero-day exploits and kernel information leaks [83]. However, replicating compute instances and I/O across each variant is resource-intensive and impractical for general-purpose use.

**Automatic initialization.** Automatic initialization approaches, such as UniSan [59] and SafeInit [23], automatically set variables to default values, mitigating uninitialized memory issues. These methods introduce performance overhead, particularly with large allocations [62], and may miss issues with non-stack variables. For example, Microsoft MSVC's initAll [5] and the `-ftrivial-auto-var-init` option in GCC and Clang focus on stack variables, but their their use is limited by their performance penalties [5], [61].

Automatic initialization can also interfere with dynamic analysis tools and sanitizers, masking issues with uninitialized variables that could be detected and fixed, making it less suitable for debugging and software testing [84]. Chow et al. [58] propose a secure deallocation technique to zero memory upon function exit, though it shares similar drawbacks, introducing overhead at the end of object lifetimes. The Linux Kernel implements a similar scheme, STACKLEAK [60], that clears the kernel stack at the end of system calls. This mitigates the impact of information leakage bugs, although it can impact system performance by up to 5%.

**Hardware-based detection.** Hardware-based detection methods, such as Georges et al.'s uninitialized capabilities [25] and Capstone [27], attempt to address uninitialized memory issues by simply introducing a new permission to CHERI. Unlike Mon CHÉRI, which uses an operation-specific bound to track written portions of the address space, uninitialized capabilities grant read permissions in the range $[a,t]$ and write permission in $[b,t]$ where $a$ is the baseline address, $b$ is the capability bound base, and $t$ is the capability bound top. However, this approach has significant limitations.

Firstly, the use of uninitialized capabilities requires software to derive new capability, with an adjusted $a$, with each write operation, introducing complexity in managing memory permissions. This limits the utility of uninitialized capabilities to a secure calling convention that enhances local stack frame encapsulation, first proposed by Skorstengaard et al. [85], by additionally protecting against uninitialized stack reads.

Secondly, uninitialized capabilities only support `Write-before-Read` semantics, meaning they do not provide protection against other forms of uninitialized memory access. This limited expressibility reduces the utility of the approach, particularly when compared to more flexible solutions like Mon CHÉRI, which supports a broader range of access control policies as well as protection for both stack, heap, and other types of memory allocations.

Capstone [27] is a redesign of the CHERI capability model that enables broader memory isolation and attempts to generalize uninitialized capabilities. It addresses the first drawback of Georges et al.'s method by introducing self-incrementing write semantics. However, like Georges et al.'s uninitialized capabilities, Capstone shares the limitation of only addressing `Write-before-Read` scenarios. Capstone also suffers from a major drawback of its own: it incurs a significant performance overhead of up to 50%.

Additionally, in Capstone, fully initialized capabilities must be explicitly promoted to regular capabilities. Either the developer or compiler must understand when a capability is expected to be fully initialized, in order to promote it. In contrast, due to the way conditional capability semantics are designed (§ 3.2), a fully initialized conditional capability is equivalent to the corresponding CHERI capability.

These prior hardware-based approaches also face integration challenges. Uninitialized capabilities have only been simulated on the obsolete CHERI-MIPS ISA [26]. Capstone, on the other hand, comes with invasive changes to the established CHERI architecture and high overhead, making it impractical for real-world applications, especially in performance-critical systems.

**Comparison with Mon CHÉRI.** Our evaluation demonstrates that CPs, particularly `Write-before-Read` capabilities in Mon CHÉRI, offer high detection accuracy with minimal false positives when used in isolation. While static analysis is valuable for early detection, tools like dynamic analysis, sanitizers, and CPs require runtime errors to be exercised. We believe that `Write-before-Read` CPs can complement static analysis by improving the detection of uninitialized memory issues. In § 7, we discuss how static analysis can be used with CP instrumentation to further optimize CP performance.

Our assessment of Mon CHÉRI is based on extensive evaluation using standard public sector and industry benchmarks on prototypes based on the QEMU full-system emulator and the MonCHÉRI-Flute64 softcore (§ 5). To our knowledge, no practical implementation nor compiler support for Georges et al.'s uninitialized capabilities is available to enable a fair comparison with Mon CHÉRI on FPGA. We argue that conditional capabilities, carefully designed to impose only minimal changes to the CHERI capability representation (§ 4.2), have a better chance of real-world adoption than more invasive proposals, such as Capstone.

Hardware-enforced CPs can improve detection performance for uninitialized memory issues similar to how memory tagging [9] has enhanced memory-safety sanitizers [86]. Moreover, CPs offer an alternative to automatic initialization (see § 7), emulating it without the associated drawbacks. Lastly, CPs enable novel memory access control policies, such as `Write-before-Execute-Only`, which provide similar benefits to memory with special-purpose features.

## 7. Discussion and Future Work

Here, we discuss limitations of the current Mon CHÉRI prototype, alternate designs, and suggest future research.

**Leveraging compiler-based static analysis.** In § 4.3 we showed how `Write-before-Read` CPs can be omitted for variables that are statically verified to be initialized. Existing compiler heuristics could be used to further optimize `Write-before-Read`, e.g., by omitting `Write-before-Read` CPs when a variable is determined initialized by GCC's `-Wuninitialized`, or instrument only variables identified as potentially uninitialized in certain code paths by GCC's `-Wmaybe-uninitialized`. We leave further optimizations as future work as Clang/LLVM, which our conditional capability-enhanced compiler prototype is based on, does not currently replicate GCC's `-Wmaybe-uninitialized` heuristic. Clang's alternative `-Wsometimes-uninitialized` is more conservative as it only issues warnings when the conditions under which a variable is left uninitialized are known. Developers can already correct such cases based on the emitted warnings. We believe the `Write-before-Read` CPs are useful in complementing compiler-based analysis, covering cases where the analysis is inconclusive.

**Inter-function store linearization.** The store linearization pass (§ 4.3) is currently limited to intra-function analysis, such as when `Write-before-Read` conditional capabilities are used within functions or passed to callees. However, currently propagating conditional capabilities from a callee back to the caller must be done explicitly to avoid the conditions explained in § 4.3 to occur across function boundaries. Solving inter-function store linearization is simpler in languages that enforce *borrowing*, where only one mutable reference to an allocation can exist at a time. Borrowing is prominent in Rust [87], but similar, compiler-enforced borrow checking has been proposed for C [88] and C++ [89]–[91] as well. Future work should explore borrow-checker-aided full program conditional capability linearization.

**Emulating automatic initialization.** In § 6, we suggest `Write-before-Read` CPs could emulate automatic initialization, avoiding its drawbacks. Instead of issuing a hardware protection fault, a load to an address outside the conditional capability's OB could set the destination register to zero (or a default pattern). This mimics automatic initialization without the overhead of pre-initializing memory.

Another drawback of automatic initialization, discussed in § 6, is interference with dynamic analysis and sanitizers. Initialization emulation could be controlled via a hardware configuration, allowing it to be toggled on or off. This allows the same CP-instrumented program binary to be used in production and for testing, depending on the configuration.

**Overlapping capability permission bits.** In § 4.1 we explain how Mon CHÉRI reuses software-defined permission bits in the CHERI capability format. A drawback of the enumerator-based $p_{op}$ representation is that it makes CPs, and possible software-defined permissions, mutually exclusive.

To avoid this drawback, we considered an alternative based on that CPs always describe a subset of the conventional CHERI permissions, e.g., `Write-before-Read` confers the same access as `R` when $o = t$. In this alternative, a *conditional control bit*, $c$ is assigned from the reserved, but unused, capability bits to indicate whether the capability is in *conditional* mode. In conditional mode, i.e., when $c = 1$, the $p_{hw}$ bits `R`, `W`, and `X` represent the corresponding CPs: `Write-before-Read`, `Write-Once`, and `Write-before-Read` respectively. This allows CP bits to overlay conventional CHERI permissions bits without losing expressivity. The `Write-before-Read-Only`, and `Write-before-Execute-Only` CPs could be overlayed similarly with an additional *exclusive permission* bit.

**Non-sequentially written memory.** In § 4.1, we assume that memory accessible via CPs is written sequentially. This holds for most data types and operations like `memcpy()` and `memset()`, but not for C structures that are initialized field-by-field or contain uninitialized padding. Software workarounds like `#pragma pack` with ordered field-initialization or an initial `memset()` before individual fields are set can prevent false positives from field-by-field access.

Alternatively, conditional capabilities can treat the $o$ value as a bitmap, where each bit describes the initialization state of a memory segment. Memory can be segmented into equal chunks or structured data fields. If the 16-bit $o$ field is insufficient, it could point to a larger bitmap in shadow memory, similar to how MemorySanitizer [55] tracks initialized memory. However, we believe the existence of straightforward software workarounds makes the overhead of complex tracking solutions unnecessary and thus leave exploring solutions for tracking non-sequentially initialized memory outside the scope of this work.

**Supporting $I_E > 2$.** In § 4.2, we explain that the current operation bounds encoding restricts conditional capabilities to $I_E \leq 2$. Similar to how the CHERI Concentrate encoding sacrifices *alignment* precision as allocation sizes increase, the operation top ($o$) can sacrifice *write* precision as $I_E$ increases. At an architectural level, this is achieved by zero-padding the least significant bits of $O_E$, akin to handling $B$ and $T$. When the precision of the least significant stored bit in $O_E$ exceeds the ability to express writes to individual bytes, halfwords, or words, corresponding store instructions (`s[bhw]`) are disabled for that conditional capability. Extending writes beyond doubleword precision requires a variant of `SetOpBounds`, which extends $o$ under the condition that the two preceding instructions have been writes reaching a target granularity.

## 8. Conclusion

This paper presents Mon CHÉRI, a novel extension to the CHERI architecture that addresses uninitialized memory errors that account for ≈ 10% of all memory vulnerabilities.

By introducing conditional capabilities, Mon CHÉRI enables precise run-time detection of uninitialized memory access at instruction-level, with minimal performance overhead. Our extensive evaluation on the Mon CHÉRI QEMU-system-CHERI128 emulator and FPGA-based MonCHÉRI-Flute64 prototype shows that Mon CHÉRI achieves a 100% true-positive rate while maintaining a low, 1%, false-positive rate

on the Juliet test suite, and incurs only a $\approx 3.5\%$ performance overhead for the `Write-before-Read` extension.

Our comparison with state-of-the-art solutions, including static analysis tools, sanitizers, and other hardware-based detection techniques demonstrates that Mon CHÉRI complements static analysis and provides additional coverage where static analysis alone falls short. Moreover, conditional permissions (CPs) enable novel memory access control policies, while being carefully designed to impose only minimal changes to the CHERI capability representation and thus have a better chance of real-world adoption than previous proposals that provide only `Write-before-Read` semantics, with significant limitations or invasive changes.

Looking ahead, we plan to extend Mon CHÉRI capabilities by integrating it with broader real-world use cases, emulation of automatic initialization, explore further optimization strategies, and reducing false positives in edge cases though compiler improvements.

## Acknowledgments

Firstly, we would like to extend our gratitude to Prof. N. Asokan and Hossam ElAtali at the University of Waterloo for allowing us to conduct part of our evaluation on the group's FPGA equipment.

We are equally grateful for the guidance provided by the University of Cambridge Computer Laboratory Security Group members, particularly Dr. Hesham Almatary, Dr. Jonathan Woodruff, Jessica Clarke, and Prof. Robert Watson. We also wish to thank our colleagues at Ericsson: Christoph Baumann, Patrik Ekdahl, Peter Svensson, and Santeri Paavolainen for supporting various aspects of our research. Lastly, we appreciate the valuable discussions we've had with regards to this work with Hans Liljestrand, Adriaan Jacobs, and Fatih Aşağıdağ.

This work has received funding under EU H2020 MSCA-ITN action 5GhOSTS, grant agreement no. 814035, by the Research Fund KU Leuven, by the Flemish Research Programme Cybersecurity, and by the CyberExcellence programme of the Walloon Region, Belgium.

## References

[1] H. Cho *et al.*, "Exploiting Uses of Uninitialized Stack Variables in Linux Kernels to Leak Kernel Pointers," in *14th USENIX Workshop on Offensive Technologies (WOOT 20)*, ser. WOOT '20. USENIX Association, Aug. 2020.

[2] N. Sullivan, "The Results of the CloudFlare Challenge," Nov. 2014, (accessed 2024-07-05). [Online]. Available: https://blog.cloudflare.com/the-results-of-the-cloudflare-challenge

[3] K. Lu *et al.*, "Unleashing Use-Before-Initialization Vulnerabilities in the Linux Kernel Using Targeted Stack Spraying," in *Proceedings 2017 Network and Distributed System Security Symposium*. San Diego, CA: Internet Society, 2017. [Online]. Available: https://www.ndss-symposium.org/ndss2017/ndss-2017-programme/unleashing-use-initialization-vulnerabilities-linux-kernel-using-targeted-stack-spraying/

[4] N. Joly, S. ElSherei, and S. Amar, "Security Analysis of CHERI ISA," Microsoft Security Response Center, Tech. Rep., Oct. 2020. [Online]. Available: https://msrc.microsoft.com/blog/2020/10/security-analysis-of-cheri-isa/

[5] J. Bialek, "Solving Uninitialized Stack Memory on Windows | MSRC Blog | Microsoft Security Response Center," Mar. 2020, (accessed 2024-06-27). [Online]. Available: https://msrc.microsoft.com/blog/2020/05/solving-uninitialized-stack-memory-on-windows/

[6] H. Sutter, "C++ safety, in context," Mar. 2024, (accessed 2024-03-14). [Online]. Available: https://herbsutter.com/2024/03/11/safety-in-context/

[7] L. Zhao *et al.*, "A Survey of Hardware Improvements to Secure Program Execution," *ACM Comput. Surv.*, p. 35, Jun. 2024. [Online]. Available: https://doi.org/10.1145/3672392

[8] Qualcomm, "Pointer Authentication on ARMv8.3: Design and Analysis of the New Software Security Instructions," Whitepaper, Jan. 2017. [Online]. Available: https://www.qualcomm.com/content/dam/qcomm-martech/dm-assets/documents/pointer-auth-v7.pdf

[9] Arm, "Armv8.5-A Memory Tagging Extension," Whitepaper, Aug. 2019. [Online]. Available: https://developer.arm.com/-/media/Arm%20Developer%20Community/PDF/Arm_Memory_Tagging_Extension_Whitepaper.pdf

[10] Intel, "A Technical Look at Intel's Control-flow Enforcement Technology," Jun. 2020, (accessed 2023-10-28). [Online]. Available: https://www.intel.com/content/www/us/en/developer/articles/technical/technical-look-control-flow-enforcement-technology.html

[11] J. Woodruff *et al.*, "The CHERI capability model: Revisiting RISC in an age of risk," in *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, Jun. 2014, pp. 457–468. [Online]. Available: https://ieeexplore.ieee.org/document/6853201

[12] N. Wesley Filardo *et al.*, "Cornucopia: Temporal Safety for CHERI Heaps," in *2020 IEEE Symposium on Security and Privacy (SP)*, May 2020, pp. 608–625. [Online]. Available: https://ieeexplore.ieee.org/document/9152640

[13] R. N. M. Watson *et al.*, "An Introduction to CHERI," Computer Laboratory, University of Cambridge, Technical Report UCAM-CL-TR-941, Sep. 2019. [Online]. Available: https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-941.pdf

[14] H. M. Levy, *Capability-Based Computer Systems*. USA: Butterworth-Heinemann, 1984. [Online]. Available: https://homes.cs.washington.edu/~levy/capabook/

[15] J. B. Dennis and E. C. Van Horn, "Programming semantics for multiprogrammed computations," *Communications of the ACM*, vol. 9, no. 3, pp. 143–155, Mar. 1966. [Online]. Available: https://dl.acm.org/doi/10.1145/365230.365252

[16] R. N. M. Watson *et al.*, "Capability Hardware Enhanced RISC Instructions: CHERI Instruction-Set Architecture (Version 9)," Computer Laboratory, University of Cambridge, Technical Report UCAM-CL-TR-987, Sep. 2023. [Online]. Available: https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-987.html

[17] R. Grisenthwaite, "Arm Morello Evaluation Platform -Validating CHERI-based Security in a High-performance System," in *2022 IEEE Hot Chips 34 Symposium (HCS)*, Aug. 2022, pp. 1–22. [Online]. Available: https://ieeexplore.ieee.org/document/9895591

[18] S. Amar *et al.*, "CHERIoT: Complete Memory Safety for Embedded Devices," in *56th Annual IEEE/ACM International Symposium on Microarchitecture*. Toronto ON Canada: ACM, Oct. 2023, pp. 641–653. [Online]. Available: https://dl.acm.org/doi/10.1145/3613424.3614266

[19] J. Woodruff *et al.*, "CHERI Concentrate: Practical Compressed Capabilities," *IEEE Transactions on Computers*, vol. 68, no. 10, pp. 1455–1469, Oct. 2019. [Online]. Available: https://ieeexplore.ieee.org/document/8703061

[20] H. Xia *et al.*, "CHERIvoke: Characterising Pointer Revocation using CHERI Capabilities for Temporal Memory Safety," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '52. New York, NY, USA: Association for Computing Machinery, Oct. 2019, pp. 545–557. [Online]. Available: https://dl.acm.org/doi/10.1145/3352460.3358288

[21] N. W. Filardo *et al.*, "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: ACM, Apr. 2024, pp. 251–268. [Online]. Available: https://dl.acm.org/doi/10.1145/3620665.3640416

[22] D. Chisnall *et al.*, "CHERI JNI: Sinking the Java Security Model into the C," in *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '17. New York, NY, USA: Association for Computing Machinery, Apr. 2017, pp. 569–583. [Online]. Available: https://dl.acm.org/doi/10.1145/3037697.3037725

[23] A. Milburn, H. Bos, and C. Giuffrida, "SafeInit: Comprehensive and Practical Mitigation of Uninitialized Read Vulnerabilities," in *Proceedings 2017 Network and Distributed System Security Symposium*. San Diego, CA: Internet Society, 2017. [Online]. Available: https://www.ndss-symposium.org/ndss2017/ndss-2017-programme/safelnit-comprehensive-and-practical-mitigation-uninitialized-read-vulnerabilities/

[24] C. I. King, "Crypto: Mv_cesa - ensure backlog is initialised · torvalds/linux@1a92b2b · GitHub," Apr. 2015. [Online]. Available: https://github.com/torvalds/linux/commit/1a92b2ba339221a4afee43adf125fcc9a41353f7

[25] A. L. Georges *et al.*, "Efficient and provable local capability revocation using uninitialized capabilities," *Proceedings of the ACM on Programming Languages*, vol. 5, no. POPL, pp. 6:1–6:30, Jan. 2021. [Online]. Available: https://dl.acm.org/doi/10.1145/3434287

[26] S. Huyghebaert *et al.*, "Uninitialized Capabilities," Jun. 2020. [Online]. Available: http://arxiv.org/abs/2006.01608

[27] J. Z. Yu *et al.*, "Capstone: A Capability-based Foundation for Trustless Secure Memory Access," in *32nd USENIX Security Symposium (USENIX Security 23)*, 2023, pp. 787–804. [Online]. Available: https://www.usenix.org/conference/usenixsecurity23/presentation/yu-jason

[28] H. Shacham, "The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86)," in *Proceedings of the 14th ACM Conference on Computer and Communications Security*, ser. CCS '07. New York, NY, USA: Association for Computing Machinery, Oct. 2007, pp. 552–561. [Online]. Available: https://doi.org/10.1145/1315245.1315313

[29] R. Denis-Courmont *et al.*, "Camouflage: Hardware-assisted CFI for the ARM Linux kernel," in *2020 57th ACM/IEEE Design Automation Conference (DAC)*, Jul. 2020, pp. 1–6. [Online]. Available: https://ieeexplore.ieee.org/document/9218535

[30] CTSRD, "CTSRD-CHERI/llvm-project," Capability Hardware Enhanced RISC Instructions, Jul. 2024. [Online]. Available: https://github.com/CTSRD-CHERI/llvm-project

[31] H. Almatary *et al.*, "CompartOS: CHERI Compartmentalization for Embedded Systems," Jun. 2022. [Online]. Available: http://arxiv.org/abs/2206.02852

[32] M. Conte, "Mattconte/tlsf," Apr. 2016. [Online]. Available: https://github.com/mattconte/tlsf

[33] R. N. M. Watson *et al.*, "Capability Hardware Enhanced RISC Instructions: CHERI Instruction-Set Architecture (Version 6)," Computer Laboratory, University of Cambridge, Technical Report UCAM-CL-TR-907, Apr. 2017. [Online]. Available: http://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-907.pdf

[34] ——, "Capability Hardware Enhanced RISC Instructions: CHERI Instruction-Set Architecture (Version 7)," Computer Laboratory, University of Cambridge, Technical Report UCAM-CL-TR-927, Jun. 2019. [Online]. Available: https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-927.pdf

[35] A. Ghiti, "Virtual Memory Layout on RISC-V Linux," Feb. 2021, (accessed 2024-07-03). [Online]. Available: https://www.kernel.org/doc/html/latest/arch/riscv/vm-layout.html

[36] D. Spickett, "Top Byte Ignore For Fun and Memory Savings," Feb. 2023, (accessed 2024-07-08). [Online]. Available: https://www.linaro.org/blog/top-byte-ignore-for-fun-and-memory-savings/

[37] Intel, *Intel Architecture Instruction Set Extensions and Future Features - Programming Reference*, Mar. 2024. [Online]. Available: https://cdrdv2-public.intel.com/819680/architecture-instruction-set-extensions-programming-reference.pdf

[38] AMD, *AMD64 Architecture Programmer's Manual Volume 2: System Programming*, Mar. 2024. [Online]. Available: https://www.amd.com/content/dam/amd/en/documents/processor-tech-docs/programmer-references/24593.pdf

[39] M. Maas and A. Zabrocki, "Working Draft of the RISC-V J Extension Specification," Jun. 2024. [Online]. Available: https://github.com/riscv/riscv-j-extension

[40] E. S. Lowry and C. W. Medlock, "Object code optimization," *Commun. ACM*, vol. 12, no. 1, pp. 13–22, Jan. 1969. [Online]. Available: https://dl.acm.org/doi/10.1145/362835.362838

[41] LLVM team, "LLVM: Lib/Transforms/Scalar/Reg2Mem.cpp Source File," Jan. 2019. [Online]. Available: https://llvm.org/doxygen/Reg2Mem_8cpp_source.html

[42] S. Ruchlejmer, "Secure Rewind and Discard on ARM Morello," Jul. 2024. [Online]. Available: http://arxiv.org/abs/2407.04757

[43] NSA Center for Assured Software, "Juliet test suite," https://samate.nist.gov/SARD/test-suites/112, 2017, accessed: 2024-04-19.

[44] CTSRD, "CTSRD-CHERI/Flute," Capability Hardware Enhanced RISC Instructions, Jun. 2024. [Online]. Available: https://github.com/CTSRD-CHERI/Flute

[45] R. Nikhil, "Bluespec System Verilog: Efficient, correct RTL from high level specifications," in *Proceedings. Second ACM and IEEE International Conference on Formal Methods and Models for Co-Design, 2004. MEMOCODE '04.*, Jun. 2004, pp. 69–70. [Online]. Available: https://ieeexplore.ieee.org/document/1459818

[46] T. Newsome *et al.*, "Riscv-software-src/riscv-tests," RISC-V International, Jul. 2024. [Online]. Available: https://github.com/riscv-software-src/riscv-tests

[47] MITRE, "CWE-457: Use of Uninitialized Variable (4.14)," Jul. 2006, (accessed 2024-07-09). [Online]. Available: https://cwe.mitre.org/data/definitions/457.html

[48] E. J. Schwartz, T. Avgerinos, and D. Brumley, "All You Ever Wanted to Know about Dynamic Taint Analysis and Forward Symbolic Execution (but Might Have Been Afraid to Ask)," in *2010 IEEE Symposium on Security and Privacy*, May 2010, pp. 317–331. [Online]. Available: https://ieeexplore.ieee.org/document/5504796

[49] M. Podhradsky, R. Tadros, and A. Roach, "GaloisInc/BESSPIN-GFE," Galois, Inc., Oct. 2022. [Online]. Available: https://github.com/GaloisInc/BESSPIN-GFE

[50] EEMBC, "Eembc/coremark," Embedded Microprocessor Benchmark Consortium, Jul. 2024. [Online]. Available: https://github.com/eembc/coremark

[51] GCC developer community, *Using the GNU Compiler Collection For GCC Version 14.1.0*, May 2014. [Online]. Available: https://gcc.gnu.org/onlinedocs/gcc-14.1.0/gcc.pdf

[52] LLVM team, "Diagnostic flags in Clang," Jun. 2024, (accessed 2024-07-10). [Online]. Available: https://releases.llvm.org/18.1.8/tools/clang/docs/DiagnosticsReference.html

[53] J. Seward and N. Nethercote, "Using Valgrind to detect undefined value errors with bit-precision," in *2005 USENIX Annual Technical Conference (USENIX ATC 05)*. Anaheim, CA: USENIX Association, Apr. 2005. [Online]. Available: https://www.usenix.org/conference/2005-usenix-annual-technical-conference/using-valgrind-detect-undefined-value-errors-bit

[54] D. Bruening and Q. Zhao, "Practical memory checking with Dr. Memory," in *International Symposium on Code Generation and Optimization (CGO 2011)*, Apr. 2011, pp. 213–223. [Online]. Available: https://ieeexplore.ieee.org/document/5764689

[55] E. Stepanov and K. Serebryany, "MemorySanitizer: Fast detector of uninitialized memory use in C++," in *Proceedings of the 13th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, ser. CGO '15. USA: IEEE Computer Society, Feb. 2015, pp. 46–55. [Online]. Available: https://doi.org/10.1145/1250734.1250736

[56] E. D. Berger and B. G. Zorn, "DieHard: Probabilistic memory safety for unsafe languages," *ACM SIGPLAN Notices*, vol. 41, no. 6, pp. 158–168, Jun. 2006. [Online]. Available: https://doi.org/10.1145/1133255.1134000

[57] M. Cao *et al.*, "Different is Good: Detecting the Use of Uninitialized Variables through Differential Replay," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '19. New York, NY, USA: Association for Computing Machinery, Nov. 2019, pp. 1883–1897. [Online]. Available: https://doi.org/10.1145/3319535.3345654

[58] J. Chow *et al.*, "Shredding Your Garbage: Reducing Data Lifetime Through Secure Deallocation," 2005.

[59] K. Lu *et al.*, "UniSan: Proactive Kernel Memory Initialization to Eliminate Data Leakages," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '16. New York, NY, USA: Association for Computing Machinery, Oct. 2016, pp. 920–932. [Online]. Available: https://doi.org/10.1145/2976749.2978366

[60] A. Popov, "How STACKLEAK improves Linux kernel security," Nov. 2018, (accessed 2024-07-05). [Online]. Available: https://a13xp0p0v.github.io/2018/11/04/stackleak.html

[61] S. Guelton, "Trivial Auto Var Init Experiments," Dec. 2023, (accessed 2024-06-30). [Online]. Available: https://serge-sans-paille.github.io/pythran-stories/trivial-auto-var-init-experiments.html

[62] H. P. Nilsson, "Bug 111523 – Unexpected performance regression with -ftrivial-auto-var-init=zero for e.g. systemctl unmask," Sep. 2023, (accessed 2024-07-07). [Online]. Available: https://gcc.gnu.org/bugzilla/show_bug.cgi?id=111523

[63] Y. Yutaka, "AdLint," Aug. 2015, (accessed 2024-05-05). [Online]. Available: https://sourceforge.net/projects/adlint/

[64] LLVM team, "Clang-Check," 2024, (accessed 2024-06-30). [Online]. Available: https://clang.llvm.org/docs/ClangCheck.html

[65] ——, "Clang-Tidy," 2024, (accessed 2024-05-05). [Online]. Available: https://clang.llvm.org/extra/clang-tidy/

[66] CodeSecure, "CodeSonar," May 2023, (accessed 2024-05-05). [Online]. Available: https://codesecure.com/our-products/codesonar/

[67] Synopsys, "Coverity Scan," Nov. 2023, (accessed 2024-05-05). [Online]. Available: https://scan.coverity.com/

[68] D. Marjamäki, "Cppcheck," May 2024, (accessed 2024-05-05). [Online]. Available: https://www.cppcheck.com/

[69] D. A. Wheeler, "Flawfinder," Jan. 2007, (accessed 2024-05-05). [Online]. Available: https://dwheeler.com/flawfinder/

[70] F. Kirchner *et al.*, "Frama-C: A software analysis perspective," *Formal Aspects of Computing*, vol. 27, no. 3, pp. 573–609, May 2015. [Online]. Available: https://doi.org/10.1007/s00165-014-0326-7

[71] G. Brat *et al.*, "IKOS: A Framework for Static Analysis Based on Abstract Interpretation," in *Software Engineering and Formal Methods*, D. Giannakopoulou and G. Salaün, Eds. Cham: Springer International Publishing, 2014, pp. 271–277.

[72] Facebook, "Infer Static Analyzer," May 2016, (accessed 2024-05-05). [Online]. Available: https://fbinfer.com/

[73] D. Evans, "Splint," Jan. 2002, (accessed 2024-05-05). [Online]. Available: https://splint.org/

[74] H. G. Rice, "Classes of recursively enumerable sets and their decision problems," *Transactions of the American Mathematical Society*, vol. 74, no. 2, pp. 358–366, 1953. [Online]. Available: https://www.ams.org/tran/1953-074-02/S0002-9947-1953-0053041-6/

[75] D. L. Bruening, "Efficient, Transparent, and Comprehensive Runtime Code Manipulation," Ph.D. dissertation, Massachusetts Institute of Technology, Sep. 2004. [Online]. Available: https://www.burningcutlery.com/derek/phd.html

[76] N. Nethercote and J. Seward, "Valgrind: A framework for heavyweight dynamic binary instrumentation," in *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '07. New York, NY, USA: Association for Computing Machinery, Jun. 2007, pp. 89–100. [Online]. Available: https://doi.org/10.1145/1250734.1250746

[77] B. Cox *et al.*, "N-Variant Systems A Secretless Framework for Security through Diversity," in *15th USENIX Security Symposium (USENIX Security '06)*, ser. USENIX Security '06. Vancouver, B.C. Canada: USENIX Association, 2006, p. 16. [Online]. Available: https://www.usenix.org/conference/15th-usenix-security-symposium/n-variant-systems-secretless-framework-security-through

[78] B. Salamat *et al.*, "Orchestra: Intrusion detection using parallel execution and monitoring of program variants in user-space," in *Proceedings of the 4th ACM European Conference on Computer Systems*, ser. EuroSys '09. New York, NY, USA: Association for Computing Machinery, Apr. 2009, pp. 33–46. [Online]. Available: https://doi.org/10.1145/1519065.1519071

[79] T. Jackson, C. Wimmer, and M. Franz, "Multi-variant program execution for vulnerability detection and analysis," in *Proceedings of the Sixth Annual Workshop on Cyber Security and Information Intelligence Research*, ser. CSIIRW '10. New York, NY, USA: Association for Computing Machinery, Apr. 2010, pp. 1–4. [Online]. Available: https://doi.org/10.1145/1852666.1852708

[80] K. Koning, H. Bos, and C. Giuffrida, "Secure and Efficient Multi-Variant Execution Using Hardware-Assisted Process Virtualization," in *2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, Jun. 2016, pp. 431–442. [Online]. Available: https://ieeexplore.ieee.org/document/7579761

[81] S. Volckaert *et al.*, "Secure and efficient application monitoring and replication," in *Proceedings of the 2016 USENIX Conference on Usenix Annual Technical Conference*, ser. USENIX ATC '16. USA: USENIX Association, Jun. 2016, pp. 167–179.

[82] B. Coppens, B. De Sutter, and S. Volckaert, "Multi-variant execution environments," in *The Continuing Arms Race: Code-Reuse Attacks and Defenses*. Association for Computing Machinery and Morgan & Claypool, Mar. 2018, vol. 18, pp. 211–258. [Online]. Available: https://doi.org/10.1145/3129743.3129752

[83] S. Österlund *et al.*, "kMVX: Detecting Kernel Information Leaks with Multi-variant Execution," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '19. New York, NY, USA: Association for Computing Machinery, Apr. 2019, pp. 559–572. [Online]. Available: https://doi.org/10.1145/3297858.3304054

[84] OpenSSF contributors, "Compiler Options Hardening Guide for C and C++," Jun. 2024, (accessed 2024-06-30). [Online]. Available: https://best.openssf.org/Compiler-Hardening-Guides/Compiler-Options-Hardening-Guide-for-C-and-C++

[85] L. Skorstengaard, D. Devriese, and L. Birkedal, "Reasoning about a Machine with Local Capabilities: Provably Safe Stack and Return Pointer Management," *ACM Trans. Program. Lang. Syst.*, vol. 42, no. 1, pp. 5:1–5:53, Dec. 2019. [Online]. Available: https://doi.org/10.1145/3363519

[86] K. Serebryany *et al.*, "Memory Tagging and how it improves C/C++ memory safety," Feb. 2018. [Online]. Available: http://arxiv.org/abs/1802.09517

[87] S. Klabnik and C. Nichols, *The Rust Programming Language*. USA: No Starch Press, May 2018.

[88] T. Silva, J. Bispo, and T. Carvalho, "Foundations for a Rust-Like Borrow Checker for C," in *Proceedings of the 25th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems*. Copenhagen Denmark: ACM, Jun. 2024, pp. 155–165. [Online]. Available: https://dl.acm.org/doi/10.1145/3652032.3657579

[89] H. Sutter, “Lifetime safety: Preventing common dangling,” Microsoft, Technical Report P1179 R1 – version 1.1, Nov. 2019. [Online]. Available: https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p1179r1.pdf

[90] Lado, “Ladroid/CppBorrowChecker,” Jun. 2024. [Online]. Available: https://github.com/ladroid/CppBorrowChecker

[91] S. Baxter and C. Mazakas, “Safe C++,” Sep. 2024. [Online]. Available: https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2024/p3390r0.html

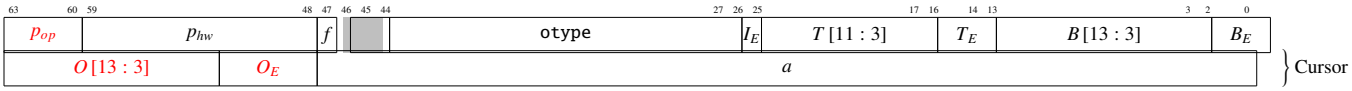
# Appendix A.
# Technical Supplements

## A.1. Example of Avoided Data Hazard

```
1 /* ... */
2   csetbounds      ca0, ca0, 4
3 ① csetwbrbound    ca0, ca0, zero
4   li              a1, 10
5 ② csw             a1, 0(ca0)
6 ③ clw             a0, 0(ca0)
7   clc             cra, 16(csp)
8 /* ... */
```

Listing 2: Example of instruction sequence causing data hazard as described in §4. At ①, the `csetwbrbound` instructions sets the `Write-before-Read` bound for the capability in register `ca0`, turning it into a conditional capability. At ②, the capability store word (`csw`) instructions performs a store operation on `ca0`, whereupon the operation bound of the capability in `ca0` is increased at Stage-2 of the pipeline. At the same time, the capability load word (`clw`) instruction at ③ has already entered the pipeline and prepares a read via the same capability in `ca0`. Without the the bypass allowing the updated operation bound to be forwarded from from Stage-2 to Stage-1, this instruction would fail due to the operation bound check on the out-of-date bound. With the bypass, this sequence of instructions is valid and does not incur additional latency.

## A.2. Changes to Capability Encoding




If $I_E = 0$:

$$E = 0$$
$$T[2:0] = T_E$$
$$B[2:0] = B_E$$
$$O[2:0] = O_E[4:2]$$
$$L_{carry\_out} = \begin{cases} 1, & \text{if } T[11:0] < B[11:0] \\ 0, & \text{otherwise} \end{cases}$$
$$L_{msb} = 0$$

If $I_E = 1$ and $E = 1, 2$ :

$$E = \{T_E, B_E\}$$
$$T[2:0] = 0$$
$$B[2:0] = 0$$
$$O[2:0] = O_E[E+2:E]$$
$$L_{carry\_out} = \begin{cases} 1, & \text{if } T[11:3] < B[13:3] \\ 0, & \text{otherwise} \end{cases}$$
$$L_{msb} = 1$$

Reconstituting the top two bits of $T$:

$$T[13:12] = B[13:12] + L_{carry\_out} + L_{msb}$$

Decoding the bounds:

| | | | |
|---|---|---|---|
| address, $a$ = | $a_{top} = a[47 : E+14]$ | $a_{mid} = a[E+13 : E]$ | $a_{low} = a[E-1 : 0]$ |
| top, $t$ = | $a_{top} + c_t$ | $T[13:0]$ | $0'E$ |
| bottom, $b$ = | $a_{top} + c_b$ | $B[13:0]$ | $0'E$ |
| | | | If $I_E = 0$ : $O'E$ |
| operation $o$ = | $a_{top} + c_o$ | $O[13:0]$ | If $I_E = 1$ and $E = 1, 2$ : $O_E[E-1:0]$ |

To calculate the corrections $c_t$, $c_b$ and $c_o$:

$$A_3 = a[E+13 : E+11] \quad (1)$$
$$B_3 = B[13:11] \quad (2)$$
$$T_3 = T[13:11] \quad (3)$$
$$O_3 = O[13:11] \quad (4)$$
$$R = B_3 - 1 \quad (5)$$

| $A_3 < R$ | $T_3 < R$ | $c_t$ |
|---|---|---|
| false | false | 0 |
| false | true | +1 |
| true | false | −1 |
| true | true | 0 |

| $A_3 < R$ | $B_3 < R$ | $c_b$ |
|---|---|---|
| false | false | 0 |
| false | true | +1 |
| true | false | −1 |
| true | true | 0 |

| $A_3 < R$ | $O_3 < R$ | $c_o$ |
|---|---|---|
| false | false | 0 |
| false | true | +1 |
| true | false | −1 |
| true | true | 0 |

Figure 6: Compressed 128-bit capability format and decoding (adapted from [16] with additions and changes marked in red).

## A.3. Examples from the Juliet Test Suite

```
void CWE457_Use_of_Uninitialized_Variable__double_64b_goodB2GSink(void * dataVoidPtr)
{
    /* cast void pointer to a pointer of the appropriate type */
    double * dataPtr = (double *)dataVoidPtr;
    /* dereference dataPtr into data */
 ③  volatile double data = (*dataPtr);
    /* FIX: Ensure data is initialized before use */
 ④  data = 5.0;
    printDoubleLine(data);
}

static void goodB2G()
{
 ①  volatile double data;
    /* POTENTIAL FLAW: Don't initialize data */
    ; /* empty statement needed for some flow variants */
 ②  CWE457_Use_of_Uninitialized_Variable__double_63b_goodB2GSink(&data);
}

void CWE457_Use_of_Uninitialized_Variable__double_63_good()
{
    goodG2B();
    goodB2G();
}
```

Listing 6: A "good" example from the Juliet Test Suite [43] where Mon CHÉRI detects an uninitialized variable use. A reference to an uninitialized `volatile double data` ① is passed in a function call ② and dereferenced and copied ③ in the callee where Mon CHÉRI detects an uninitialized load. The memory content of the copy is initialized before use in ④. The example is from `C/testcases/CWE457_Use_of_Uninitialized_Variable/s01/CWE457_Use_of_Uninitialized_Variable__double_63a.c` in the official test suite release.

```
__attribute__((writebeforeread, noinline))
void CWE457_Use_of_Uninitialized_Variable__int_array_alloca_partial_init_64_bad()
{
    volatile int *data;
 ❶  data = (int *)ALLOCA(10*sizeof(int));
    /* POTENTIAL FLAW: Partially initialize data */
    {
        int i;
 ❷      for(i=0; i<(10/2); i++)
        {
            data[i] = i;
        }
    }
 ❸  CWE457_Use_of_Uninitialized_Variable__int_array_alloca_partial_init_64b_badSink(&data);
}
```

Listing 7: An example code from the Juliet Test Suite, demonstrating the use of partially initialized arrays. The function `CWE457_Use_of_Uninitialized_Variable__int_array_alloca_partial_init_64_bad` allocates memory for an integer array using `ALLOCA` ❶ and partially initializes it ❷. The partially initialized array is then passed to `CWE457_Use_of_Uninitialized_Variable__int_array_alloca_partial_init_64b_badSink` ❸, and that function is reading the whole extent of the `data` array. That uninitialized memory access detected by Mon CHÉRI.

```
define dso_local void @CWE457_Use_of_Uninitialized_Variable__int_array_alloca_partial_init_64_bad() local_unnamed_addr
    addrspace(200) #0 !dbg !19 {
entry:
 ①  %data = alloca ptr addrspace(200), align 16, addrspace(200), !clang.decl.ptr !28
    %0 = call ptr addrspace(200) @llvm.cheri.bounded.stack.cap.i64(ptr addrspace(200) %data, i64 16)
    %1 = call ptr addrspace(200) @llvm.cheri.cap.op.bounds.set.i64(ptr addrspace(200) %0, i64 0)
 ②  %2 = alloca [40 x i8], align 16, addrspace(200), !dbg !30
    %3 = call ptr addrspace(200) @llvm.cheri.bounded.stack.cap.i64(ptr addrspace(200) %2, i64 40), !dbg !30
    %4 = call ptr addrspace(200) @llvm.cheri.cap.op.bounds.set.i64(ptr addrspace(200) %3, i64 0), !dbg !30
 ③  store ptr addrspace(200) %4, ptr addrspace(200) %1, align 16, !dbg !32, !tbaa !33
    ...
 ④  store volatile i32 0, ptr addrspace(200) %4, align 16, !dbg !38, !tbaa !42
 ⑤  store volatile ptr addrspace(200) %4, ptr addrspace(200) %1, align 16, !dbg !37
    ...
}
```

Listing 8: LLVM IR of Listing 7 after CP the store linearization. ① allocates memory on the stack for a pointer. ② allocates memory for an array of 40 bytes. ③ stores the capability `%4` at the memory location pointed to by `%1`. After ④, the operation bound is updated at the hardware level, so the capability stored in memory becomes invalid. Our instrumentation adds another store to update the capability in memory ⑤.