

USENIX

THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

# PICASSO: Scaling CHERI Use-After-Free Protection to Millions of Allocations using Colored Capabilities

Merve Gülmez, *Ericsson Security Research;* Ruben Sturm, *DistriNet, KU Leuven;* Hossam ElAtali, *University of Waterloo;* Håkan Englund, *Ericsson Security Research;* Jonathan Woodruff, *University of Cambridge;* N. Asokan, *University of Waterloo and KTH Royal Institute of Technology;* Thomas Nyman, *Ericsson Product Security*

https://www.usenix.org/conference/usenixsecurity26/presentation/guelmez

This paper is included in the Proceedings of the 35th USENIX Security Symposium.

August 12–14, 2026 • Baltimore, MD, USA

ISBN 978-1-939133-58-8

Open access to the Proceedings of the 35th USENIX Security Symposium is sponsored by



جامعة الملك عبدالله للعلوم والتقنية

King Abdullah University of Science and Technology

ARTIFACT EVALUATED
USENIX
AVAILABLE

# PICASSO: Scaling CHERI Use-After-Free Protection to Millions of Allocations using Colored Capabilities

*Merve Gülmez*[*], *Ruben Sturm*[†‡], *Hossam ElAtali*[§], *Håkan Englund*[*],
*Jonathan Woodruff*[‖], *N. Asokan*[§,¶], *Thomas Nyman*[#]
[*]*Ericsson Security Research,* [†]*DistriNet, KU Leuven,* [§]*University of Waterloo,*
[‖]*University of Cambridge,*[¶]*KTH Royal Institute of Technology,* [#]*Ericsson Product Security*
*{merve.gulmez,hakan.englund,thomas.nyman}@ericsson.com, ruben.sturm@kuleuven.be*
*hossam.elatali@uwaterloo.ca, jonathan.woodruff@cl.cam.ac.uk, asokan@acm.org*

## Abstract

While the CHERI instruction-set architecture extensions for capabilities enable strong spatial memory safety, CHERI lacks built-in *temporal safety*, particularly for heap allocations. Prior attempts to augment CHERI with temporal safety fall short in terms of scalability, memory overhead, and incomplete security guarantees due to periodical sweeps of the system's memory to individually revoke stale capabilities.

We address these limitations by introducing *colored capabilities* that add a controlled form of *indirection* to CHERI's capability model. This enables *provenance tracking* of capabilities to their respective allocations via a hardware-managed *provenance-validity table*, allowing bulk retraction of dangling pointers without needing to quarantine freed memory. Colored capabilities significantly reduce the frequency of capability revocation sweeps while improving security.

We realize colored capabilities in PICASSO, an extension of the CHERI-RISC-V architecture on a speculative out-of-order FPGA softcore (CHERI-Toooba). We also integrate colored-capability support into the CheriBSD OS and CHERI-enabled Clang/LLVM toolchain. Our evaluation shows effective mitigation of use-after-free and double-free bugs across all heap-based temporal memory-safety vulnerabilities in NIST Juliet test cases, real-world CVEs, only a small performance overhead on SPEC CPU benchmarks (≈ 5% g.m.), less latency, and more consistent performance in long-running SQLite, PostgreSQL, and gRPC workloads compared to prior work.

## 1 Introduction

Capability Hardware Enhanced RISC Instructions (CHERI), an instruction-set architecture (ISA) extensions which replaces conventional pointers with *capabilities*, is a prominent design for *hardware support for memory safety*. CHERI has so far been adapted to the MIPS, RISC-V, and Armv8-A ISAs and CHERI-enabled processors have been developed by Arm [33], Microsoft [11] and the RISC-V ecosystem. The baseline CHERI design inherently provides *spatial safety* . Prior attempts to augment CHERI with temporal safety [29, 65] fall short in terms of scalability, memory overhead, and incomplete security guarantees, such as the existence of an *use-after-reallocation/use-after-free* gap (§ 2.3).

**This paper and contributions.** We propose *colored capabilities*, an efficient mechanism for simultaneously *retracting* (not fully revoking)[1] entire classes of capabilities at once (§ 5.1). Our insight is that the performance and permission issues in CHERI's temporal-safety enforcement—particularly the use-after-reallocation/use-after-free mitigation gap—stem from its complete removal of *indirection*: central points of control for capability revocation. Colored capabilities introduce a *limited* form of indirection for *provenance tracking*, using a hardware-managed *provenance-validity table (PVT)* for efficiently retracting *dangling pointers*.

Using colored capabilities, we design an improved temporal-safety scheme for CHERI which eliminates the *use-after-reallocation/use-after-free* gap. A key benefit is a significantly reduced frequency required between capability-revocation sweeps compared to prior work (§ 5.2).

To demonstrate the practicality of our approach, we present PICASSO (Provenance Indirection-enabled CHERI Architecture for Scarcely Swept Objects), our realization of colored capabilities for the CHERI-RISC-V architecture. PICASSO is implemented on a field-programmable gate array (FPGA) softcore based on the speculative out-of-order CHERI-Toooba processor and in the QEMU full-system emulator. We integrate colored capabilities into the CheriBSD operating system (OS) and software stack. In summary, our contributions are:

- A hardware-software co-design for *colored capabilities* that extends CHERI with provenance tracking (§ 5).
- PICASSO: a realization of colored capabilities on the CHERI-RISC-V-QEMU full-system emulator and in an FPGA softcore based on the speculative out-of-order CHERI-Toooba processor IP (§ 6.1).

[‡]R.S. contributed to the research while at Ericsson Security Research.

[1]Retracting refers to *disabling access* for a *set of capabilities* at once, whereas revocation refers to permanently invalidating specific capabilities.

- Support for colored capabilities in the CHERI-enabled Clang/LLVM compiler and CheriBSD OS (§ 6.2).
- Demonstrating `PICASSO` successfully detects use-after-free and double-free defects in > 2800 NIST Juliet test cases without false positives (§ 7.1).
- Showing the efficiency of `PICASSO` via the SPEC standard central processing unit (CPU) benchmark showing small performance impact compared to the baseline CHERI-RISC-V processor (≈ 5% g.m.) and real-world SQLite, PostgreSQL, and gRPC benchmarks demonstrating less latency and more consistent performance compared to prior work (3%–8% overhead on average, § 7.2).

An extended version of this conference paper with supplementary material is available at [35]. The open-source PICASSO hardware and software stack is available at https://github.com/orgs/coloredcapabilities and at [36].

## 2 Background

### 2.1 Temporal Safety

Temporal safety relates to memory locations containing different data at different points in time during program execution. For example, after heap memory is freed by the application, accessing the memory *as though the original data is there* leads to undefined behavior. This can happen if the program holds a pointer to some memory, frees the memory but keeps the pointer (*dangling pointer*) and later tries to access the memory through the dangling pointer. The result is a *use-after-free (UAF)* bug. If the application reallocates the memory, a subsequent dereference of the dangling pointer leads to a *use-after-reallocation (UAR)* bug. UAF and UAR bugs are associated with non-deterministic behavior and stem from a confusion of *allocation provenance*: which distinct allocation a particular address to memory represents. In addition to crashes, these defects can lead to security vulnerabilities. For example, information leakage can occur if an object containing sensitive data overlaps with the dangling pointer; similarly, arbitrary code execution may be possible if the dangling pointer overlaps with an attacker-controlled address which is inadvertently used in an indirect function call.

### 2.2 The CHERI Capability Architecture

CHERI is an ISA extension that augments conventional ISAs with a capability-based hardware-software co-design for memory protection. Hardware-supported capabilities enforce *spatial safety* for code or data pointers. The CHERI ISA specification [63] defines capability representations in registers and memory, along with instructions to manipulate them safely.

As shown in Figure 1, a CHERI capability is twice the width of a native pointer type—128 bits on 64-bit platforms—and includes one additional *validity-tag* bit ①, stored separately to prevent tampering by non-capability-aware instruc-

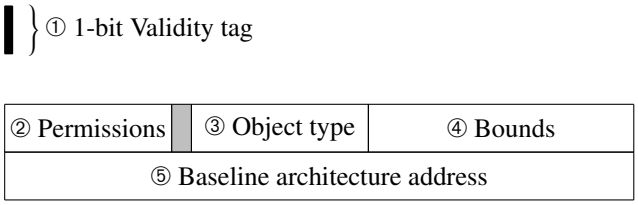



Figure 1: In-memory representation of CHERI capabilities adapted from Watson et al. [64]

tions. These tags are preserved by valid, capability-aware instructions but are invalidated by unauthorized manipulation or injection of arbitrary capabilities. In CHERI-enhanced architectures, general-purpose and address registers are extended to hold full capabilities. Each capability includes:

- **Permissions** ②: A bitmask defining allowed operations.
- **Object type (`otype`)** ③: A signed integer enabling temporary "sealing" and "unsealing" of capabilities for opaque pointer and fine-grained in-process isolation.
- **Bounds** ④: The valid memory range relative to the baseline architecture address ⑤, using a compressed encoding [68] to reduce the space taken by a capability at the cost of stricter alignment for larger object allocations.

CHERI enforces spatial safety by associating each allocation with a capability describing its valid range and permissions. New capabilities are derived from existing ones, while maintaining *monotonicity* that ensures that new capabilities cannot exceed the permissions or bounds of their parent. Sealed capabilities for compartmentalization and exception handling introduce limited non-monotonicity. Extensions to CHERI have also explored sandboxing [17], initialization safety [32, 34], safe speculation [30, 31], and side-channel resistance [25].

**Capability revocation.** In capability systems, granted authority must be *revocable*, rendering a capability *permanently unusable*. Historical designs often use *indirection* to address capability revocation [51]. Capability accesses in such systems occur through protected tables, providing a centralized point from where capabilities used throughout the system can be revoked. CHERI takes a different approach, avoiding indirection tables. Instead, it uses validity tags ① stored either in a dedicated memory region, or distributed throughout program memory and CPU register files. A capability's validity can thus be determined without implicit indirection on memory access, eliminating the performance bottlenecks associated with a centralized protection table. This design choice optimizes *memory accesses* over *efficient revocation*. Because CHERI capabilities do not explicitly track their provenance, revoking a capability (and all its derivatives) necessitates finding their in-memory and register-file representations through exhaustive search and invalidating the associated validity tags.

**Capability sealing.** A capability is "sealed" when its object type (`otype`) ③ is not `-1` (all ones). A sealed capability is *tem-*

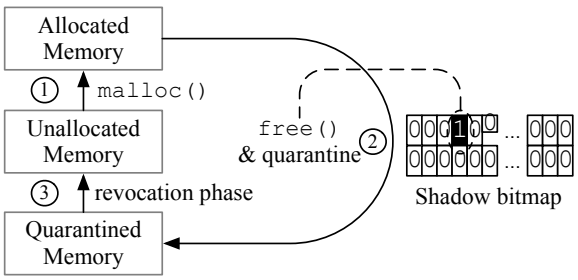



Figure 2: Memory allocation lifecycle in Cornucopia.

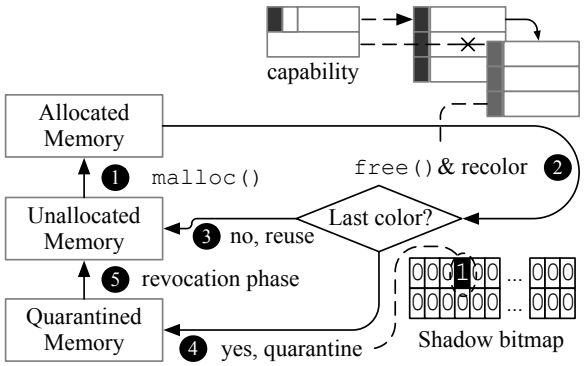



Figure 3: Allocation lifecycle in CHERI + Memory Tagging.

*porarily* rendered unusable until it is "unsealed". In CHERI ISAv9, sealing is primarily used to protect control-flow transfers between isolated, CHERI-enforced compartments.

**CHERI-enabled software stack.** CheriBSD is a CHERI-enhanced version of the FreeBSD OS, supporting CHERI-RISC-V both in emulation and hardware. It serves as a fully functional prototype demonstrating CHERI integration in a conventional OS. The CheriBSD kernel and userspace can be built in "*pure-capability*" CHERI C/C++, where all conventional memory pointers are replaced with capabilities by the CHERI-LLVM compiler.

## 2.3 CHERI Temporal Safety

**Cornucopia.** *Cornucopia* [29] enforces deterministic C/C++ temporal safety for heap allocations. It is integrated into CheriBSD via a *shim layer*, known as the `malloc` revocation shim (MRS), which sits on top of the underlying BSD libc memory allocator, and a kernel-resident, concurrent revocation service that periodically performs *sweeping revocation*—that is, periodically scans memory to revoke capabilities referencing freed memory. Freeing memory in Cornucopia's memory lifecycle (Figure 2) involves three separate steps:

① Freed memory is placed into a *quarantine* buffer.
② Quarantined regions are marked at word granularity in a *shadow bitmap* managed by the MRS.
③ Once the quarantine buffer exceeds a user-defined threshold, a revocation phase is triggered to find and invalidate stale capabilities referencing the quarantined memory.

To prevent UAR exploits, memory is not reused until revocation completes. However, this introduces several drawbacks: 1) **Dangling pointers remain valid during quarantine**: while quarantining can reduce exploitability it also *obscures* UAF defects. 2) **Increased memory overhead** due to delayed reuse of freed blocks. 3) **Performance trade-offs**: more frequent recycling reduces memory footprint and increases the effectiveness of the mitigation at the cost of higher performance overhead. Immediate revocation on `free()` (revoke-on-free) is, however, prohibitively expensive (Appendix C). 4) **Stop-the-world phase**: despite concurrent sweeping, Cornucopia halts all threads at the end to scan register files, recently modified pages, and kernel-held capabilities. This disrupts execution and aborts active system calls. Cornucopia trusts the MRS and allocator to maintain the quarantine buffer and bitmap. These structures are protected by an internal API, requiring the caller to hold a capability with the `CHERI_PERM_SW_VMEM` permission. Allocators can derive such capabilities from a higher-level "super" capability (e.g., via `mmap()` [14]), but the permission is stripped before returning capabilities to untrusted userspace. Cornucupia is also employed by the internal libc allocator and the thread-local storage memory management area, following a similar approach. However, it holds its own guaranteed memory and revocation mechanism.

**Cornucopia Reloaded.** Filardo et al. [29] identify Cornucopia's long *stop-the-world* pauses as its main limitation. Their *Cornucopia Reloaded* improves revocation concurrency by introducing *capability load barriers*, which associate each page table entry (PTE) with a *revocation epoch*. A new *epoch counter* in CPU control registers tracks completed atomic revocation sweeps. If a PTE's epoch differs from the counter, any capability load from that page traps, allowing the revoker to complete revocation before the CPU processes the load.

Because sweeps are atomic, pages with recent capability writes cannot retain stale capabilities, eliminating the need for global pauses. Applications thus avoid major stop-the-world events. However, within an active epoch, programs can still hold, copy, or dereference pointers to memory freed in that same epoch. Only allocations freed in earlier epochs are guaranteed to be revoked. Consequently, memory added to quarantine cannot be reused until at least a full epoch has passed. As with the original Cornucopia, this leads to deallocation delays and increased memory usage.

Sweeping revocation inherited from Cornucopia remains the most contested aspect of CHERI [50]. Alternatives involve accelerating CHERI temporal-safety enforcement by combining Cornucopia with approaches not relying on sweeps.

**Composing CHERI and Memory Versioning.** Filardo [28, 29] proposes combining CHERI with *memory versioning* to close the UAR/UAF mitigation gap. A variant of this approach is included in CHERI ISAv9 [63, Appendix C.5]. As shown in Figure 3, both the pointer and its allocated memory region

are assigned a matching version ID (or "color") ❶. On free, the memory is recolored to invalidate stale pointers ❷, and the freed color becomes available for reuse ❸. This blocks dangling pointer use. However, the number of colors is limited, as version IDs are stored in dedicated *memory tags*. Due to performance constraints, architectures like Arm's Memory Tagging Extension (MTE) restrict tags to 4 bits, allowing only 16 unique IDs. Reusing colors risks collisions between valid and stale pointers, yielding only probabilistic UAF detection. To prevent this, once all version IDs are used ❹, freed memory is quarantined, and the system falls back to Cornucopia-style sweeping revocation ❺. A practical implementation for this design concept is not yet available.

## 3 System and Threat Model

**System Model.** We assume the same system model as CHERI [65] and that its spatial-safety properties hold. We rely on the underlying allocator ($\mathcal{A}lloc$) satisfying assumptions:

**A1**: $\mathcal{A}lloc$ is part of the trusted runtime.

**A2**: $\mathcal{A}lloc$ maintains correct bounds on capabilities it issues.

**A3**: $\mathcal{A}lloc$ can discern malformed input to `free()`.

**Threat Model.** We assume an adversary ($\mathcal{A}dv$) with the ability to manipulate program behavior through malicious input. Due to flaws in the victim program, such as faulty input validation, $\mathcal{A}dv$ can influence it to invoke arbitrary sequences of `malloc()` and `free()` calls (or C++ `new` and `delete` statements). Thanks to CHERI's architectural guarantees, $\mathcal{A}dv$ is unable to produce arbitrary, counterfeit capabilities. Nor is $\mathcal{A}dv$ capable of directly tampering with $\mathcal{A}lloc$'s data structures, control-data, or code beyond interaction through the `malloc()` and `free()` family of functions (**A1**). This is consistent with the threat model employed by prior work [29, 65].

CHERI capabilities are architecturally guaranteed to be *monotonically nonincreasing* (§ 2.2); the permissions and bounds set on a capability can be *decreased* by software (by deriving a new, more narrowly bounded capability), but cannot be *increased* beyond the parent capability permissions and bounds. Thus, **A2** implies that capabilities pointing to the heap fall within the bounds of (once) heap-allocated objects. **A3** in turn ensures that capabilities not pointing to beginning of legitimate heap-allocated objects are rejected by $\mathcal{A}lloc$. Invalid capabilities, as well as capabilities for non-heap-allocated data, are rejected by the MRS (§ 6.2).

The exploitation of uninitialized reads of heap-allocated memory objects are out of scope, but can be efficiently prevented using previously proposed CHERI extensions [34] or compiler-based uninitialized memory sanitization [19, 37, 43].

While temporal-safety concerns are also valid for stack-based allocations (c.f. *use-after-return* [44]), they are easier to protect [21], significantly rarer compared to heap-based use-after-free [69], and according to Lee et al. *"unlikely to be exploitable"* [38]. Compiler-based *escape analysis* can verify that references to stack variables do not outlive the enclosing stack frame and is widely-deployed in modern C and C++ toolchains [57, 59]. In fact, mitigations dating back a decade expect that run-time temporal-safety mitigation are only needed to dynamically allocated heap memory [21, 29, 38, 65, 69]. Consequently, following prior work [29, 65], we consider stack-based use-after-return out of scope. Additionally, similar to prominent prior implementations of CHERI on application-class processors, our threat model excludes transient-execution and other micro-architectural attacks and side-channels [62]. We discuss the gap with respect to safe speculation in § 9.2.

## 4 Goals and Challenges

Our goals are:

**G1**: Deterministically close the *UAF/UAR gap*.

**G2**: Allow the freed memory to be *immediately reused*.

**G3**: Reduce the *frequency of CHERI revocation sweeps*.

**Deterministically closing the UAF/UAR gap.** Designs that compose CHERI with memory versioning (§ 2.3) can only achieve *probabilistic* UAF mitigation if deployed without quarantining due to their limited tag space—as low as 4 bits. In contrast, we aim to achieve *deterministic* prevention of UAF. The inherent challenge is scaling the ability to efficiently track the *provenance*—which allocation a particular address to memory represents—to *millions* of allocations.

**Immediate reuse of freed memory.** A common source of memory overhead for each scheme discussed in § 2.3 is the quarantine buffer. Quarantining memory delays its availability for reuse until a revocation sweep occurs. This is necessary, as the CHERI design lacks a mechanism for *retracting* capabilities, that is, disabling capabilities whose provenance can be traced to particular allocation until their eventual revocation. Quarantining instead ensures *provenance validity* of capabilities by *artificially extending* the lifetime of the associated allocations beyond the `free()` call that de-allocates them until a time when all capabilities to that allocation have been revoked. Even though this is deferred until certain size criteria are reached, it may be costly in terms of performance and memory footprint for the application. For this reason, production-grade memory allocators, such as LLVM's Scudo, disable quarantining by default [58]. Therefore we target immediate reuse of freed memory without quarantine.

**Reducing the frequency of revocation sweeps.** Recall from § 2.2 that due to CHERI prioritizing the efficiency of memory accesses over the efficiency of revocation, the revocation sweep is a "*necessary evil*" that is inherent to the CHERI design. Our goal is not to fundamentally alter the core tenets

of CHERI. Rather, we want to significantly reduce the frequency of the necessary sweeps (without a quarantine buffer **G2**). The challenge of how to achieve reduced revocation frequency goes hand-in-hand with scaling the number of allocation provenances for capabilities that can be *simultaneously* tracked. By increasing the number of allocations that can simultaneously co-exist in a system while still being discernible from each other, the system can operate without risking overlapping capabilities with different allocation provenance being confused with one another and necessitating revocation of stale capabilities.

## 5 Colored Capability Design

Recall from § 2.2 that CHERI's design disregards revocation implemented via table-based lookups or indirection on pointer operations in favor of validity bits stored in tagged memory. However, as discussed in § 2.3, § 4, and shown in § 7.2, revoking all capabilities for an allocation at the moment that allocation is freed is prohibitively expensive in terms of run-time overhead in CHERI. To achieve **G1** to **G3** outlined in § 4 we aim to decouple the *validity of a capability's provenance*—the allocation the capability represents—from the *validity of the capability itself*. This requires introducing a form of *limited indirection* to the CHERI architecture. While many historical capability systems, including the initial design by Dennis and Van Horn [22], use table-based indirection to manage all capability metadata, CHERI explicitly has avoided such indirection as a design goal [63, 2.3.16]. The challenge we must overcome is introducing the necessary indirection in a manner which limits its impact only on the subset of capabilities whose provenances needs to be tracked, and reduce the run-time impact of table-based lookups by minimizing the information needed to validate provenance.

In this section we present the high-level idea behind the colored capability design (§ 5.1), how we apply colored capabilities for temporal-safety protection (§ 5.2), and describe the overall system architecture (§ 5.3).

### 5.1 High-Level Idea

Colored capabilities introduce a *limited* form of indirection to improve temporal safety. Whereas historical indirection designs centralize all capability metadata [22, 51], we demonstrate that 1-bit of indirect metadata for a set of colored capabilities is sufficient for accelerating temporal-safety enforcement. This allows *all* capabilities in a set referencing a given heap allocation to be *retracted* simultaneously. Retracted capabilities are disabled so that attempts to dereference them result in a CHERI fault (c.f. *capability sealing*, § 2.2). Retracted capabilities need to *eventually* be fully revoked, but do **not pose a risk for UAF** in the interim, thus **removing the need for quarantining freed memory**. Since CHERI's direct revocation aligns with the typical execution style of C-language

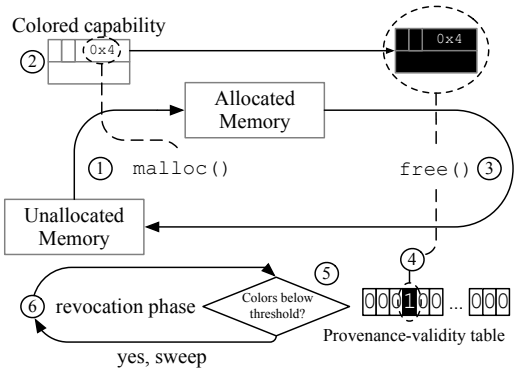



Figure 4: Colored capabilities allocation lifecycle.

programs–where pointers freely propagate through memory without enforced (capability-to-capability [51]) indirection–we instead target a mechanism for *indirect provenance tracking*. Recall from § 2 that CHERI's sealing encodes both the sealing state and the object type directly in the capability (the presence of a non-default object type doubles as both the indicator of a capability's sealing state and the conditions for its unsealing). In contrast, colored capabilities *decouple* the capability's *state* from its *type*: the *provenance identifier* remains embedded in the capability, but the provenance tracking state is expressed by a *provenance-validity bit (PVB)* stored separately in a *provenance-validity table (PVT)*, indexed by the provenance identifier. Colored capabilities only used to track heap allocations; non-heap capabilities are unaffected.

Colored capabilities derive their name from the use of "color" to distinguish allocation provenance in memory-versioning schemes (§ 2.3). We use *color* to refer to the `otype` of a capability while it represents a provenance identifier. We reserve a portion of CHERI's object type for conventional sealing. Unlike memory versioning, we only assign a color to colored capabilities, not the associated memory allocation. Instead, each entry in the PVT holds a single PVB indicating whether *all* capabilities with that provenance identifier are valid (the corresponding allocation is active). The provenance identifier in the capability determines if the capability is a colored capability or not. Hardware enforces provenance-validity checks for any load or store using a colored capability by looking up the corresponding PVB in the PVT. Any memory access using a colored capability, where the PVT indicates that the color (provenance identifier) is invalid, faults.

### 5.2 Temporal Safety via Colored Capabilities

Using colored capabilities and provenance tracking, we design the temporal memory safety scheme illustrated in Figure 4. On allocation ①, a unique provenance identifier, from a finite pool, is assigned to the capability ②, and its corresponding PVB in the PVT is marked *valid*. On de-allocation ③, this bit is marked *invalid* ④. As explained earlier, all load or stores via

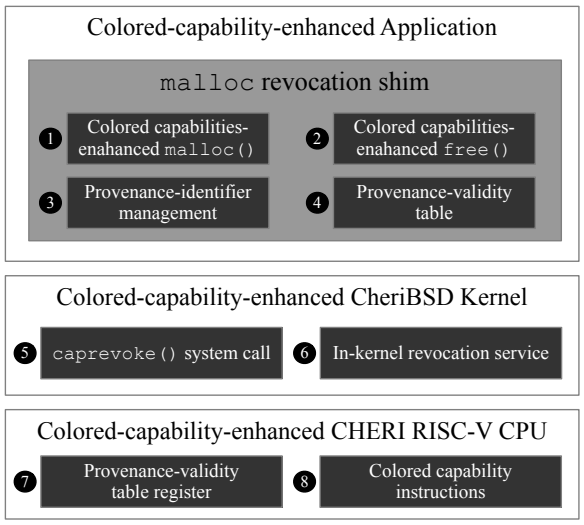



Figure 5: Colored capabilities overview

colored capabilities are subject to a hardware check against the PVT. Once a provenance identifier has been invalidated, the hardware ensures that all capabilities with that provenance identifier cannot be dereferenced. This eliminates the need for a quarantine buffer and shadow memory to track freed regions, allowing freed memory to be safely reused immediately.

The provenance identifier pool supports over two million entries per process with 128-bit capabilities (§ 6). In long-running programs, this pool may eventually be exhausted. To be able to retract and reuse previously assigned identifiers, colored capabilities still need a mechanism to identify and revoke such retracted capabilities. Once the identifier pool is depleted up to a pre-determined threshold ⑤, a revocation sweep is needed ⑥. Colored capability revocation is based on Cornucopia Reloaded's improved revocation sweep using capability load barriers and inherits the reduced stop-the-world periods, but does not require capability sweeps to be atomic. As provenance tracking already guarantees that capabilities pointing to freed memory cannot be referenced, even the in-kernel scanning for capabilities can occur asynchronously. A revocation pass for a given provenance identifier has to simply complete before that identifier is returned to the pool.

## 5.3 System Architecture

Figure 5 illustrates the high-level system architecture for a colored capability-enhanced CPU and OS. At the application-level, colored capabilities integrate with the regular `malloc`-family of functions and `free()` through enhanced versions of these memory management application programming interfaces (APIs) ❶, ❷ integrated via the `malloc` revocation shim (MRS, § 2.3). For brevity, we refer in Figure 4 and subsequent figures simply to colored capability-enhanced `malloc()` instead of the whole `malloc`-family of functions. When a new colored capability is issued by `malloc()`, or a memory object is freed by passing a colored capability to `free()`, the allocator must keep track of assigned and still unassigned provenance identifiers. The *provenance-identifier management* ❸ within the MRS is responsible for assigning each allocation a unique provenance identifier which is retained by any capabilities derived from the issued colored capability. Recall from § 3 that thanks to **A2** and monotonically non-increasing bounds (§ 2.2), any such derived capabilities are guaranteed to point within the bounds of the allocated object. When an allocation is freed, the `free()` implementation marks the corresponding provenance identifier as invalidated via the PVT ❹. This has the effect of simultaneously invalidating *all* colored capabilities pointing to the the relevant allocation since they share the corresponding provenance identifier.

Invalidated provenance identifiers are kept out of rotation by the provenance-identifier management until all corresponding capabilities have been revoked. This occurs when the number of unused IDs fall below a predetermined threshold, at which point any `malloc()` or `free()` call engages the kernel's revocation service ❻ by making a `caprevoke()` system call ❺. Other system software not using colored capabilities do not require modification (apart from adaptation to CHERI).

The colored capability-enhanced CPU core(s) is augmented via a *provenance-validity table register (PVTR)* ❼ which holds the virtual start address of the PVT available to the colored capability-enhanced instructions ❽. Load (`l[bhwd]`) and store (`s[bhwd]`) instructions, when given a colored capability as operand, lookup its corresponding PVB in the PVT.

# 6 Implementation

In this section, we present Provenance Indirection-enabled CHERI Architecture for Scarcely Swept Objects (`PICASSO`), our implementation of colored capabilities for the CHERI-RISC-V architecture. We implemented two versions of `PICASSO`: 1) a QEMU-system-CHERI128 full-system emulator, and 2) a softcore based on the CHERI-Toooba processor intellectual property (IP). For brevity, we focus on the Toooba implementation in the following sections, while the QEMU implementation is detailed in Appendix A.

## 6.1 `PICASSO` Extension for CHERI-RISC-V

To add colored capability architectural support to the CHERI-Toooba core, we modified 34 source files, totaling approximately 1,700 lines of changes spanning the build system, core CPU, ISA, memory pipeline, and CHERI capability logic. The most significant changes affect loads and stores.

**ISA extension.** Figure 6 illustrates our changes to the baseline CHERI-Toooba processor. A new control and status register (CSR) called the provenance-validity table register (PVTR) ❶ stores the base virtual address of the PVT ❷. Access to the PVTR is restricted to kernel mode, ensuring only privileged

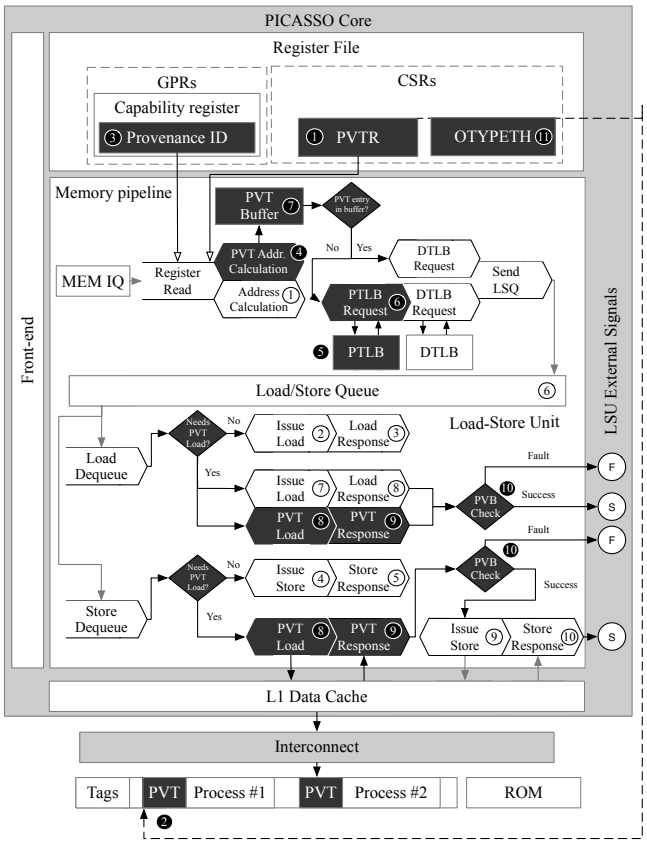



Figure 6: PICASSO hardware architecture.

code can control the PVT location. A new `ccsettype` instruction assigns provenance identifiers ❸ to capabilities. Recall from § 5.2 that Cornucopia requires allocator capabilities to hold the `CHERI_PERM_SW_VMEM` software-defined permission. Similarly, `ccsettype` requires this as a hardware-defined permission to ensure that only the trusted allocator can modify provenance identifiers. Loads and stores via capabilities with a provenance identifier (colored capabilities) implicitly read the corresponding 1-bit PVT entry, the provenance-validity bit (PVB), at `[PVTBase + ProvenanceID]` and fault if the PVB indicates the provenance is invalid, i.e., the corresponding capabilities have been retracted.

**Accessing the PVT.** The CHERI-Toooba memory pipeline consists of four main stages: *Register Read*, *Address Calculation* ①, *Address Translation*, *Issuing Load/Store* ②, ④, and *Load/Store Response* ③, ⑤. PICASSO extends the memory pipeline to distinguish between regular and colored capabilities. For colored capabilities, the provenance identifier is extracted from the source capability register ❸, while the PVT base address is read from the PVTR ❶. Address calculation ❹ for the PVT entry is performed concurrently with normal address calculation ①.

**PVT address translation.** To translate the physical address for a PVB, PICASSO adds a dedicated *provenance-translation lookaside buffer (PTLB)* ❺ to the memory pipeline. Accessing the PTLB introduces an additional pipeline stage ❻ with a one-cycle latency. PVT address translation feeds into the Load/Store Queue (LSQ) ⑥. Rather than adding separate LSQ entries for implicit PVT loads, each load or store entry is augmented to track its associated PVT load, preserving the existing queue structure. The PTLB is only used for address translation in implicit PVT loads; writes to the PVT use the standard store instructions and memory hierarchy (and require a corresponding capability with privileges to modify the PVT).

**PVT buffer.** To avoid PVT address translation and access for all colored capability loads and stores, PICASSO introduces a small PVT buffer ❼ that caches recently accessed 128-bit PVT words with valid PVBs. The buffer is indexed directly by the virtual PVB address, allowing it to bypass all of PTLB translation ❻, PVT load ❽, response ❾, and PVB check ❿. To maintain coherence, all buffer entries are invalidated on `fence` instructions; correctness relies on fences accompanying page-table updates and writes to the PVT. Each 128-bit word in the PVT buffer can contain 128 PVBs. The buffer is looked up combinationally to avoid adding latency to the common case and is therefore kept small: only 64 words, 4-way set-associative. Despite its size, it can buffer PVBs for 8192 objects, and achieves high hit rates in practice (Appendix C).

**Loads with colored capabilities.** For colored capability loads the data request and the PVT load to L1 data cache ⑦ are issued concurrently by the load-store unit (LSU). Although the cache services only one access per cycle, the two requests are independent and their latencies can overlap. A load can commit only after both responses ⑧ ❾ arrive, and PVB check ❿ succeeds. On mis-speculation, the LSU waits for outstanding responses before freeing the corresponding LSQ entry.

**Stores with colored capabilities.** For stores, the PVT load is issued early ❽, after address translation, but the store itself ⑨ is delayed until the PVT response ❾ and the PVB check ❿ passes. This ordering prevents use-after-free writes. If a store is squashed before issue, its LSQ entry must remain allocated until the PVT response. This requires additional waiting logic analogous to handling mis-speculated loads.

**Supporting millions of allocations.** In CHERI's 128-bit capability representation 18 bits are reserved for the `otype` which we reuse for provenance identifiers, allowing for $2^{18}$ distinct allocation provenances to be tracked. In our current design, applications benefit from a much larger provenance identifier space, which extends the period between needed invocations of the revocation mechanism. We extend the `otype` field by adding one reserved bit and two unused software-defined permission bits, resulting in a total of $2^{21}$ allocations.

**Compatibility with CHERI sealing.** Recall from § 2 that CHERI hardware prevents manipulation of any sealed capability. However, colored capabilities disallow de-referencing of capabilities with an invalid PVB. To support both the sealing mechanism and colored capabilities PICASSO provides a

way to detect which feature is in use whenever an `otype` or provenance identifier is set in a capability. PICASSO adds an additional CSR, otype threshold (OTYPETH) ⓫, that determines if the capability is interpreted as a normal, sealed or colored capability according to the following rule:

$$f(\texttt{otype}) = \begin{cases} \text{unsealed capability}, & \text{if } \texttt{otype} = -1 \\ \text{colored capability}, & \text{if } 0 < \texttt{otype} < \text{OTYPETH} \\ \text{sealed capability}, & \text{if } \texttt{otype} \geq \text{OTYPETH} \end{cases}.$$

This ensures that PICASSO is fully compatible with CHERI ISAv9 sealing and compartmentalization [63].

## 6.2 PICASSO Software Stack

We integrated colored capabilities into CheriBSD 25.03, resulting in ≈3 500 changed lines of code over 57 distinct files.

**Provenance Validity Table.** To efficiently track which provenance identifiers are valid, we map a PVT into each process's address space during the `execve()` system call. The PVT is placed directly under the stack, at a fixed virtual address determined when the kernel initializes based on the address space size. The virtual address of the PVT is stored in the PVTR, enabling the hardware to reference the PVT during load or store instructions. By using a fixed virtual address across all processes, we eliminate the need to update the PVTR during context switches. Cornucopia introduces the `cheri_revoke_get_shadow()` system call to provide a capability for its shadow bitmap to userspace; we augment this with a new flag that allows the colored-capability-enhanced MRS to request a capability to the PVT instead.

**Provenance-identifier management.** The MRS tracks which provenance identifiers are valid, which are in use, and which can be assigned to new capabilities. This requires it to keep track of $2^{21}$ (≈ 2 million) provenance identifiers in order to efficiently find and claim the first available one. When the number of unclaimed provenance identifiers reaches a user-defined threshold, less than 1% by default, the MRS releases the provenance identifiers associated with freed memory. The number of reclaimed IDs is program-dependent; however, a single sweep can release up to 99% of reserved provenance identifiers. Completely exhausting the provenance identifiers pool prevents further allocations until IDs can be reclaimed.

We use the BSD unit number (`unr`) allocator, `alloc_unr`, a mechanism used in the FreeBSD kernel for efficient allocation of integer identifiers by utilizing a doubly linked list of heap-allocated `unr` objects. It stores a compressed representation of provenance identifiers, keeping track of consecutive sequences of identifiers as runs instead of individual numbers. We optimize `alloc_unr` to batch provenance identifiers that are to be released and make corresponding changes to the `unr` data structure over a single iteration. For further details our modified implementation, refer to Appendix B. The BSD `unr` is a kernel-space facility. We adapt our design for userspace to avoid issuing system calls to manage `unr`s.

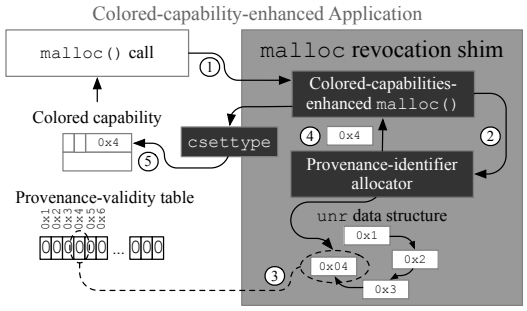



Figure 7: Allocation using colored capabilities.

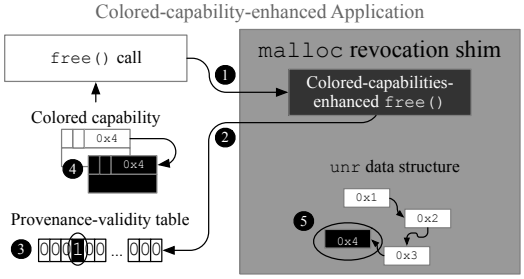



Figure 8: Free using colored capabilities.

**Userspace memory allocator.** We modify the CheriBSD MRS to support our allocation-provenance tracking mechanism similar to [29, 65]. Figure 7 shows how we assign provenance identifiers to newly allocated memory objects. Any `malloc()` call first invokes MRS ①. After the memory is allocated, the `unr` allocator ② claims an available provenance identifier ③ and assigns it to the newly created capability via `ccsettype` ⑤. This mechanism is also integrated into the internal libc and thread-local storage allocators.

Figure 8 shows our approach to invalidate provenance identifiers when memory is freed. When a `free()` call is made, it invokes MRS ❶, similar to `malloc()` calls. ***Unlike Cornucopia, we do not quarantine the memory.*** Instead, we read the provenance identifier from the capability ❷ and invalidate the corresponding PVB in the PVT ❸. Prior to invalidation, the MRS checks whether the PVB is already invalidated. This allows detection of double-free (DF) violations regardless of the underlying allocator. After that, the memory is freed ❹ and ready to be reused. However, the provenance identifier remains reserved by the `unr` to prevent capabilities to a new memory object from using an invalidated provenance identifier ❺. By default, the CheriBSD MRS rejects invalid capabilities (**A3** § 3); our extension validates any capability passed to free has a provenance identifier, ensuring that capabilities pointing to non-heap-allocated data are rejected.

**Revocation.** While PICASSO removes the need for a quarantine buffer, the finite number of provenance identifiers still requires periodic revocation in long-running programs to reclaim invalidated identifiers for reuse in the `unr`. Figure 9

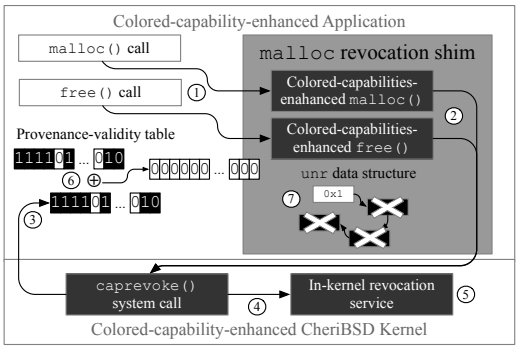



Figure 9: Colored capability revocation.

shows how we implement this revocation step to release invalidated provenance identifiers. During each memory allocation, the MRS monitors the population of reserved provenance identifiers against a predefined threshold ①. Exceeding this limit triggers a `caprevoke()` system call ② to initiate a kernel-level revocation sweep. The kernel then creates a snapshot of the current PVT ③ to establish a definitive set of provenance identifiers that are invalidated at that moment. Any provenance identifiers that become invalidated after this snapshot are excluded from the current revocation cycle and will remain in their invalidated state upon its completion. The kernel subsequently spawns a kernel thread to asynchronously revoke all invalidated capabilities ④. We employ the same revocation method ⑤ as described in Cornucopia Reloaded [29], with the exception that, rather than using a shadow bitmap, we inspect the copy of the PVT to determine whether the provenance identifier of the capability is marked as invalidated. Any subsequent malloc call will then poll the kernel to determine if the `caprevoke()` is finished. Once done, we clear the corresponding bits in the PVT that represent the provenance identifier values involved in the revocation ⑥. Then perform a batch release of all invalidated provenance identifiers within the `unr` ⑦ (Appendix B).

**PTLB page fault handling.** PVT pages can be evicted from physical memory to swap. De-referencing heap objects via colored capabilities can trigger page faults accessing the corresponding provenance identifier entry. User-space page faults are handled as usual, but colored capabilities passed to system calls can trigger provenance identifier check in kernel code. As the kernel does not expect memory accesses in privileged code to fault, we extend the CheriBSD page-fault handler to support faults for the PVT even in privileged mode.

# 7 Evaluation

We evaluate PICASSO across four dimensions: functionality, security, performance, and hardware area cost. For the performance evaluation we use the BESSPIN-GFE security evaluation platform, which provides out-of-the-box support for the CHERI-Toooba softcore. We replace the standard core with our PICASSO CHERI-Toooba, an extension of the CHERI-RISC-V Toooba FPGA softcore(RV64ACDFIMSUxCHERI) built on the open-source Bluespec RISC-V 64-bit Toooba core. We synthesize the system-on-chip (SoC) at 25MHz (default for BESSPIN-GFE) targeting the Xilinx Virtex UltraScale+ VCU118 FPGA. Functional and security evaluations are conducted using the hardware platform and a colored-capability–enhanced `QEMU-system-CHERI128` full-system emulator. We answer the research questions:

**RQ1**: Can PICASSO reliably detect UAF conditions?

**RQ2**: How does PICASSO impact the baseline processor in terms of hardware cost and performance?

**RQ3**: What is the performance and memory overhead of enforcing temporal-safety with PICASSO?

**RQ4**: To what extent does PICASSO reduce the frequency of CHERI revocation sweeps?

## 7.1 Security Evaluation

To answer **RQ1**, we evaluate PICASSO using version 1.3 of the U.S. NIST SARD Juliet Test Suite [13]. Juliet contains thousands of C/C++ test cases illustrating common programming flaws that can lead to memory vulnerabilities. The tests are organized by common weakness enumeration (CWE) categories, with "bad" cases containing known flaws and corresponding "good" cases representing the patched versions. We focus on CWE-416 (Use After Free) and CWE-415 (Double Free), encompassing 1385 vulnerable ("bad") and as many fixed ("good") C and C++ test cases. PICASSO detects all bad cases without false positives. Cornucopia fails to detect any bad CWE-416 cases in its default configuration, but achieves full success rate with revoke-on-free (RoF, § 2.3). RoF, however, comes with an impractical performance cost (Appendix C). Cornucopia detects all CWE-415 bad cases.

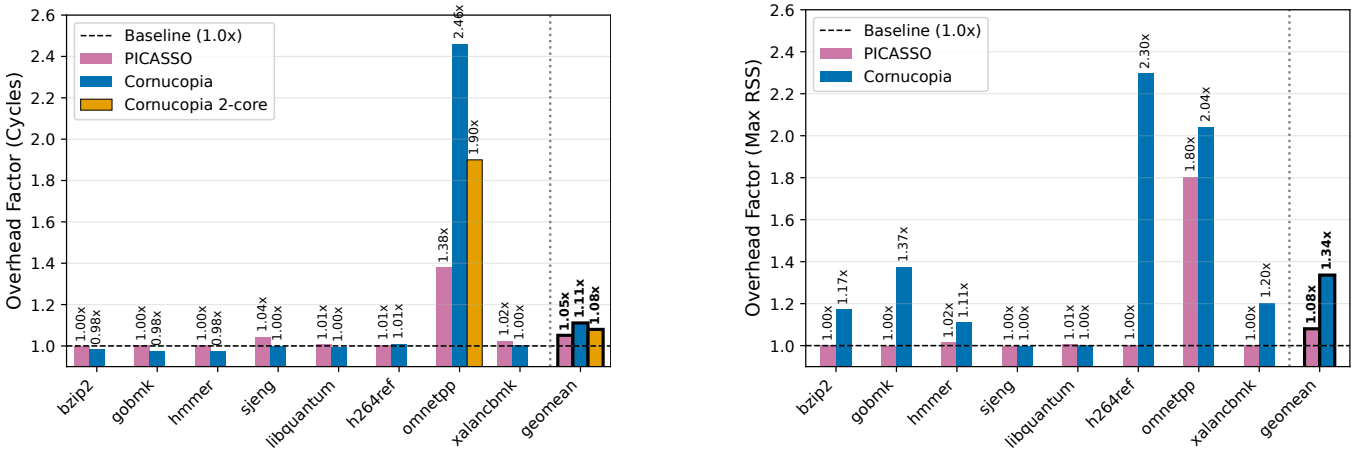
We additionally confirm PICASSO's efficacy in preventing real-world CVEs in BZip2 (CVE-2016-3189), libiberty (CVE-2016-4487), `nasm` (CVE-2017-10686, CVE-2019-8343), libzip (CVE-2019-17582), NGINX (CVE-2020-24346), LibreDWG (CVE-2022-35164), Lua (CVE-2019-6706), and other issues reported in `mjs` (issues 73 and 78) and `yasm` (issue 91) reproduced from the test suites in [5, 46].

## 7.2 Performance Evaluation

**Impact on baseline CHERI-Toooba processor.** To answer **RQ2**, we synthesize PICASSO CHERI-Toooba for the VCU118 FPGA. We also evaluated the design without the PVT buffer to isolate the logic overhead introduced by the optimization. Table 1 shows power and resource utilization reported by Xilinx Vivado 2019.1. Without the PVT buffer, the overheads are minimal (≈3% in logic and ≈4% in registers)

Table 1: Area and power costs on VCU118 @ 25MHz expressed in number of LUTs and registers, and Watts respectively.

| | logic | Δ(%) | memory | Δ(%) | registers | Δ(%) | power | Δ(%) |
|---|---|---|---|---|---|---|---|---|
| CHERI-Toooba | 688096 | - | 20113 | - | 419300 | - | 6.471 | - |
| PICASSO CHERI-Toooba without PVT buffer | 709869 | 21773(+3.16) | 20079 | -34(-0.17) | 436574 | 17274(+4.12) | 6.478 | 0.01 |
| PICASSO CHERI-Toooba with PVT buffer | 720347 | 32251(+4.69) | 21611 | 1498(+7.65) | 445129 | 25829(+6.16) | 6.515 | 0.04 |




(a) Normalized CPU cycles. We omit bars for the 2-core configuration when Cornucopia's revocation less frequently triggers.

(b) Normalized memory overhead measured as maximum resident set size (RSS).

Figure 10: SPEC CPU 2006 performance overhead for PICASSO (1.05× g.m.) and Cornucopia (1.11× g.m. in single-core, and 1.08× g.m. in 2-core configuration), and memory overhead of PICASSO (1.08× g.m.) and Cornucopia (1.34× g.m.).

compared to unmodified CHERI-Toooba. Incorporating the PVT buffer increases the overheads to ≈4.7% in logic ≈8% in memory and ≈6% in registers, a moderate impact justified by the performance tradeoff of the PVT buffer (Appendix C).

We investigated the performance overhead of our memory pipeline changes when not using the colored capabilities features caused by the additional pipeline latency introduced by PTLB lookup (❻ in Figure 6, § 6.1). This overhead is not fundamental to the approach, but is nevertheless an artifact of the current prototype. EEMBC CoreMark [24] incurred between 3.67% and 6.07% overhead depending on compilation mode on the VCU118, and MiBench incurred 1.32% overhead on average in simulation (see Tables 3 and 5 in Appendix C). Overheads in larger benchmarks were not as pronounced.

**Performance and memory overhead of enforcing temporal safety.** To answer **RQ3**, we measure the overhead of PICASSO relative to the CHERI-enabled memory allocator in CheriBSD 25.03, which is based on the Cornucopia Reloaded design. Unlike the allocator described by Filardo et al. [29], the CheriBSD implementation performs the `caprevoke()` system call asynchronously by spawning a new kernel thread that handles the revocation sweep. We utilize the Cornucopia configuration, where the quarantine limit is set to one-fourth of the total allocated memory. For brevity, we refer to the CheriBSD 25.03 allocator implementation as "Cornucopia".

We use a single-core CPU configuration, our results include the full cost of revocation: preempting the application by the revocation thread results in stop-the-world behavior. On multicore systems, this overhead could be reduced by overlapping the revocation sweep with application execution. The overhead we report is therefore an overestimate but enables a conservative assessment of the total system overhead introduced by the mechanism.

We instrument `caprevoke()` to collect revocation counts, measure peak memory overhead using `time -l` in CheriBSD, which reports the maximum resident set size during execution, and collect hardware performance metrics using CheriBSD's `minimal_counts_stats` tool. We evaluate PICASSO's performance impact across the following benchmarks:

**SPEC CPU2006 INT.** We run the subset of the SPEC CPU2006 INT benchmarks [2] that execute on the baseline CHERI system without code modifications [52]. Similar to prior work [52], we use the *train* input set, as even these inputs per configuration require over 24 hours to complete on FPGA. Figure 10 shows the overall cycle and memory normalized overhead, for the CheriBSD allocator on the unmodified CHERI-Toooba core (Cornucopia) and for PICASSO. In 7 out of 8 INT tests, the number of allocations is too few to trigger revocation in PICASSO, however, it triggers in most cases for Cornucopia. In four tests, Cornucopia achieves better performance than the baseline since it postpones memory deallocation and instead places freed objects into the quaran-

tine buffer, which involves less bookkeeping than reclaiming the memory. For omnetpp, PICASSO and Cornucopia trigger the revocation mechanism 76 and 2792 times, respectively, resulting in a worst-case overhead of 38% and 268%. To understand the impact of parallelizing revocation, we obtain measurements for the SPEC CPU 2006 INT benchmarks on an unmodified CHERI-Toooba in a 2-core configuration from the Cambridge Computer Laboratory. Parallelizing revocation reduces the Cornucopia overhead to 90% relative to our baseline for omnetpp but has no effect in cases where revocation less frequently triggers. PICASSO in a single-core configuration still outperforms Cornucopia in the 2-core configuration. PICASSO shows negligible memory overhead in most cases. However, unlike Cornucopia, whose memory consumption peaks as the quarantine buffer grows to its maximum size, PICASSO's memory usage increases during revocation due to handling two PVT tables and the memory consumed by the unr allocator (§ 6.2). We observe a worst-case memory overhead in PICASSO of 80% during revocation in omnetpp, lower than Cornucopia's in all benchmarks. To quantify the impact on DRAM, we collect hardware counters that characterize DRAM traffic. PICASSO increases DRAM traffic by 21% (g.m.), whereas Cornucopia increases it by 65% (g.m.) (see Figure 15 in Appendix C).

**SQLite.** We further evaluate PICASSO using SQLite 3.22.0 [3] with its built-in speedtest1 benchmark, which consists of multiple phases of database operations such as create, insert, reorder, and delete. A complete list of benchmark phase descriptions can be found in Table 8 in Appendix C of [35]. Figure 11 illustrates the performance overhead across these distinct operational phases. Overall, PICASSO incurs a 8% performance overhead on average, while Cornucopia incurs a 11% average performance overhead and Cornucopia exhibits a 99% memory overhead, whereas PICASSO incurs no observable memory overhead. Unlike Cornucopia, which suffers performance spikes up to 62% during revocation, PICASSO maintains a consistent profile with a peak overhead of only 12%. In the SQLite speedtest1 benchmark, Cornucopia triggers 272 revocations while PICASSO triggers zero.

**PostgreSQL pgbench.** To evaluate PICASSO impact on latency we run the PostgreSQL server [1, 20] on the CHERI-Toooba core with pgbench executed on a separate host connected over the local network. We configure a scale factor of 10 and execute $10^3$ transactions with a single client. To reduce I/O overhead and ensure single-threaded execution suitable for the single-core CHERI-Toooba processor, we disable fsync, synchronous commit, full page writes, statistics collection, and parallel workers. As illustrated in Figure 12, which plots individual transaction latencies, Cornucopia exhibits periodic latency spikes occurring approximately every 20 transactions (60 allocations per transaction). Each latency spike represents a tail-latency penalty, with p99 latency increasing by over 300%, due to the non-concurrent revocation sweep in a single-core environment (see Figure 12 in Appendix C). Overall, PICASSO degrades transactions per second (TPS) by 6.2%, compared to 23.1% for Cornucopia. We measure that PICASSO requires revocation only once every $66 \times 10^3$ transactions (vs. once every 20 in Cornucopia). For a production-speed core similar to Arm Morello (2.5GHz), we estimate this corresponds to one revocation every 2.5 min vs. one every 71.4 ms for Cornucopia [29].

**gRPC QPS.** We run a gRPC 1.54.2 QPS [60], client-server workload. Similar to pgbench setup, we run the server on the FPGA, and run the clients on a separate host. To isolate and measure only gRPC server overhead, we use a custom gRPC scenario: transport security is disabled, each of the client and server is a single process, both client and server are synchronous and apply latency-focused tuning to a minimal stack, the client opens 1 gRPC channel with 8 outstanding messages. The client sends messages to the server and awaits responses, while measuring throughput and latency percentiles. Each run lasts 120 seconds with an initial 15-second warmup, and each configuration is executed twice with 0-byte request sizes. Figure 13 shows the different latency percentiles. For p50 latency, PICASSO incurs a 3% overhead, while Cornucopia incurs a 7% overhead. At p99, PICASSO incurs only a 5% overhead, whereas Cornucopia exhibits a 3.7× increase due to revocation. Overall, PICASSO introduces a 3% Queries Per Second (QPS) degradation, while Cornucopia incurs a 15.6% degradation. To further evaluate when PICASSO triggers revocation, we run the same benchmark for 8 hours, more than 8 million allocations, resulting in a total of four revocations. These four revocations over 8 hours do not affect any latency percentile, even up to p999. In comparison, Cornucopia triggers seven revocations within just 120 seconds.

**Performance and memory overhead analysis**. PICASSO overhead is bounded by the revocation frequency, determined by the ratio of allocations to frees. For example, omnetpp revocations reclaim ≈80% of provenance identifiers on average. As the number of allocations increase, revocation counts scale proportionally but given a comparable reclamation rate of provenance identifiers the per-revocation cost remains similar. PICASSO can fall back to Cornucopia non-provenance-tracked allocations if provenance identifiers are exhausted. As revocation rate scales with available provenance identifiers, PICASSO 's advantage widens relative to Cornucopia under allocation/free pressure as Cornucopia's revocation frequency depends on the amount of quarantined memory. PVT memory overhead is 256 KiB per process (§ 6.1, § 6.2) independent of the amount of heap memory consumed since PVBs are associated with capabilities, not allocations. Fragmentation of provenance identifiers also contribute to PICASSO's memory overhead due to memory consumed by the unr allocator.

**Reduction in revocations.** To answer **RQ4**, we collect statistics on the number of allocations and revocations as discussed for each individual benchmark above. As shown in Table 2, PICASSO does not trigger revocation in the majority of benchmarks. Our further experiment with long-running PostgreSQL

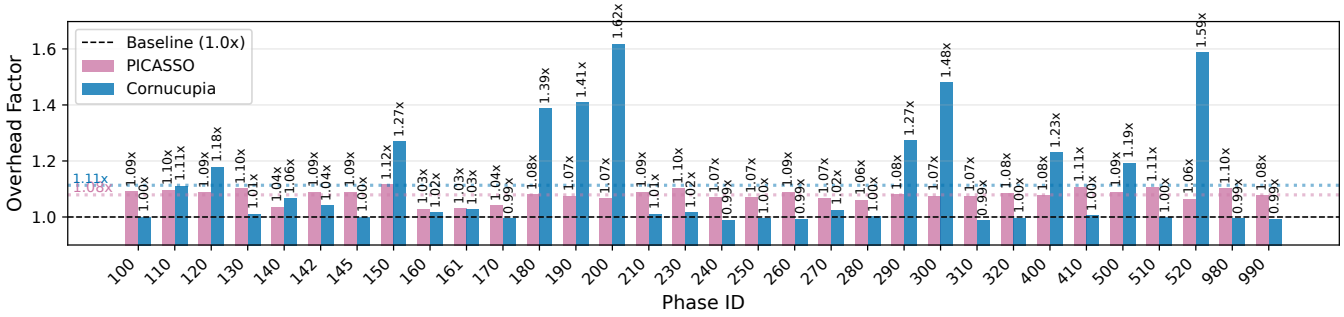



Figure 11: Normalized Performance overhead of PICASSO (1.08× avg., 1.12× max.) and Cornucopia (1.12× avg., 1.62× max.) across SQLite speedtest1 operational phases. Phase description and raw data are available in Table 8 in Appendix C of [35].

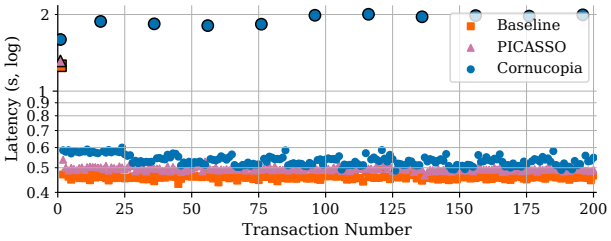



Figure 12: Latency for 200 sample pgbench transactions. Markers denote individual response times. Initial latency spike at $t = 1$ is due to connection establishment.

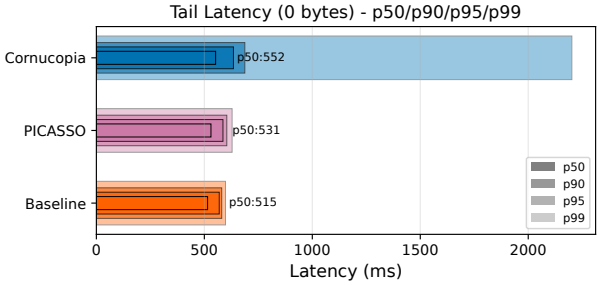



Figure 13: gRPC QPS latency percentile. Raw data are available in Table 6 in Appendix C.

and gRPC shows that PICASSO reduces the revocation time significantly without increasing the latency percentile even with p999. This demonstrates that PICASSO eliminates frequent revocation sweeps for long-running software, significantly reducing overall system load and overhead.

## 8 Related Work

Research on UAF broadly follows two strategies: *detection*, which identifies violations during program execution, and *prevention*, which blocks exploitation of freed memory.

**Detection.** "*Lock-and-key*" schemes [15, 18, 23, 40, 45] pair each pointer with a "key" and each memory object with a "lock". When an object is freed, its lock is invalidated. At dereference, the key and lock are compared to detect temporal violations. These methods resemble PICASSO's provenance-based detection but require complex static data-flow analysis and run-time instrumentation of pointer dereferences, leading to high run-time overhead (e.g., CETS [45] incurs 48%). PTAuth [27] applies a similar scheme using Arm pointer authentication [49] by creating an authentication between a pointer and its object using a random ID, but suffers from costly base-address lookups for interior pointers (pointers to the middle of objects). DangNull [38] automatically nullifies all pointers to freed objects by tracing pointer references through run-time instrumenting, at a steep 80% run-time overhead. In contrast, PICASSO is a hardware feature that detects UAF without compiler transformations or pointer analysis. Another class of detection relies on *page-permission–based* schemes [21, 61]. In these schemes each heap object is allocated on a separate virtual page. When an object is freed, access to the corresponding page is revoked. Any dangling

Table 2: Allocations and revocations for § 7.2 benchmarks.

| | | # Revocations | |
|---|---|---|---|
| **Benchmark** | **# Allocations** | **Cornucopia** | **PICASSO** |
| bzip2 | 87 | 3 | 0 |
| gobmk | 121361 | 17 | 0 |
| hmmer | 170126 | 16 | 0 |
| sjeng | 5 | 0 | 0 |
| libquantum | 108 | 1 | 0 |
| h264ref | 38270 | 12 | 0 |
| omnetpp | 129620400 | 2792 | 76 |
| xalancbmk | 1057345 | 2 | 0 |
| SQLite | 224917 | 272 | 0 |
| **Benchmark** | **# Transactions** | **Cornucopia** | **PICASSO** |
| PostgreSQL | ≈66000 | ≈ 3300 | 1 |
| gRPC | ≈100000 | ≈ 400 | 1 |

dereference immediately triggers a hardware fault, avoiding the need to instrument dereference sites.

*Tag-based memory versioning* assigns matching tags to pointer and objects and reassigns new tags to objects when they are freed. This is probabilistic due to tag collisions, especially with small tag spaces (e.g., ARM MTE's 4-bit tags). Combining CHERI with memory versioning (§ 2.3) mitigates the "reallocation gap" by reassigning tags before quarantining freed memory. However, with 4-bit versioning, they can track only 16 distinct allocations before memory tags collide. In contrast, PICASSO supports over two million unique allocation-provenance identifiers without relying on memory tagging thus enabling better scaling, deterministic protection.

**Prevention.** Prevention approaches avoid reusing memory until all references to freed regions are eliminated [6, 26, 42, 48, 61], or reduce the likelihood of successful exploitation by delaying reuse of freed blocks [9, 12, 41, 47, 55, 56]. Another strategy employs virtual address–based allocators [5,66], exploiting the abundant 64-bit address space to ensure each allocation receives a unique virtual address and is never reused at the same location. These techniques do not detect developer mistakes, but can block UAF-based attacks outright. These schemes also assign unique identifiers to every allocation to prevent temporal aliasing. This makes their memory overhead scale with application memory usage. In contrast, PICASSO's main source of memory overhead stems from the PVT; a fixed cost per process since provenance identifiers are associated with capabilities and PVBs describes whether the allocation which provenance the corresponding capabilities relate is still is valid; they do not enumerate unique allocations.

**Temporal-safety solutions for CHERI.** Apart from quarantine-based approaches (§ 2.3) , temporal-safety solutions for CHERI also include the CHERIoT real-time operating system (RTOS) [10, 11]. CHERIoT is geared towards embedded systems without virtual memory that use a "flat" physical memory space shared between programs (tasks) on the device. These systems avoid heap allocation and preallocate all needed memory, requiring total memory equal to the sum of all component's peak memory usage. C++ `new` and `delete` operators are likely to still use the `malloc`-family of functions for dynamic allocations. CHERIoT enables safe reuse of memory across RTOS tasks. All allocations/frees require an allocator capability tied to a specific quota, and pointers can only be freed using the same capability that allocated them, preventing unauthorized releases. A *revocation bitmap* (1-bit per 8 bytes heap, similar to Cornucopia's shadow bitmap § 2.3) tracks freed objects. When capabilities load from memory to registers, a *load filter* checks the revocation bit associated capability's base and clears the validity tag if the capability points to freed memory, preventing dangling pointers. Freed objects are quarantined while periodic revocation sweeps to invalidate any lingering stale capabilities. Because the load filter prevents new invalid pointers from forming, this sweep can be done safely in the background. Unlike § 2.3 approaches, CHERIoT deterministically prevents de-referencing dangling pointers. However, this requires the hardware-managed load filter check every capability load against a bitmap of *freed memory*, making the CHERIoT approach only suitable for resource-constrained devices with limited memory (and manageable revocation bitmap sizes). Further, unlike PICASSO, CHERIoT still relies on quarantining memory.

**In Summary.** PICASSO addresses the UAF/UAR gap, can extend the revocation period without requiring any memory to be quarantined, and can detect the UAF bugs deterministically, without any compiler instrumentation and pointer analysis.

# 9 Discussion and Future Work

## 9.1 Historical and Current Relevance

**Rethinking indirection.** Hennessy and Patterson [4] reflected on how early security concepts from the 1970's, including protection domains and rings, as seen in systems like Multics [53], and even capabilities [39], ended up underused, partly due to a lack of *architectural techniques* necessary to make them sufficiently fast and performant. Today, with the resurgence of capabilities through CHERI, it is timely to revisit these historical ideas using modern design tools.

As noted in § 2.2, CHERI avoids table-based indirection for revocation, opting instead for tagged memory to track capability validity. CHERI ISAv9 [63, Appendix C.7.2] discusses indirect capabilities as an experimental feature, favoring limited tagging for performance reasons. With *colored capabilities* and PICASSO, we argue that rejecting indirection outright also lead to suboptimal designs. While indirection on *every* capability access is costly, our approach shows that *selective* indirection, applied only where beneficial, combined with microarchitectural optimizations, can minimize overhead.

**Applicability to other processors.** PICASSO targets application processors whereas embedded systems are better served by CHERIoT's load filter [10, 11]. CHERIoT's revocation bitmap, however, scales poorly to virtual-memory systems since its size depends on the total available memory whereas PICASSO's PVT scales with numbers of provenance identifiers reserved from `otype`. PICASSO 's ISA-level design (PVTR, `ccsettype`, PVB check) can be applied to any core. Our implementation of colored capabilities on CHERI-Toooba with the microarchitectural PVT buffer and PTLB optimizations demonstrate that our design can be applicable to superscalar, out-of-order CHERI processors.

**Relevancy for CHERI standardization.** CHERI is in a pre-commercialization phase and is being standardized by the CHERI Task Group within RISC-V International, under the proposed "Y" extension (called "Zcheri" during development). 64-bit RISC-V (RV64Y) is receiving additional extensions [7, 8] enabling temporal-safety enforcement similar to Cornucopia Reloaded. Unlike CHERI ISAv9, which uses

five bits for capability load/store barrier controls [63, 4.3.12], RV64Y simplifies the encoding due to limited available free PTE bits. This, however, requires changes to the revocation state machine and may degrade Zcheri revocation performance compared to CHERI ISAv9.

PICASSO is orthogonal to optimizing the revocation sweep; it reduces how often revocation is needed. Furthermore, PICASSO does not require changes to the PTE. Consequently, PICASSO-style improvements are potentially interesting to future CHERI standardization as a complement for improving temporal-safety enforcement performance. We believe colored capabilities are compatible with the currently available work-in-progress drafts of the Zcheri specification, although `otype` has not yet been ratified as a Zcheri feature. As fully compatible CPUs design are not yet readily available we leave integrating PICASSO into Zcheri as future work.

## 9.2 Limitations and Future Work

**Safe speculative execution.** It is possible for PVT checks that miss the PVT buffer to be bypassed in transient execution in our current Toooba implementation, as the PVT and data loads are issued in parallel. Stores are delayed until after the check and cannot be bypassed (see in § 6.1). Future work could explore the security/performance tradeoff of adapting defensive specification, as in Capability Speculation Contracts [30, 31], or integration with state-of-the-art taint tracking [25, 70] to selectively delay loads until after PVT checks.

**PVT coherence on multi-core.** PVT coherence in core-private caches is maintained by the cache coherence protocol. Furthermore, free() applies fence instructions to discard PVBs cached in the PVT buffer (§ 6.1). In multi-core scenarios, our current implementation would require all cores to perform a fence before a PVT update is guaranteed to be seen by all cores. Future work would add a hardware mechanism to broadcast invalidated colors to other cores to selectively "shootdown" modified entries.

**Software-Design Optimization.** We observe that high fragmentation within the provenance identifiers space causes memory bloat in the `unr` allocator. To address this, future work could explore alternative allocators, such as those used in GCC [16] or the Linux ID allocator [67]. Another approach involves the direct use of a bitmap accelerated by RISC-V hardware extensions such as Bitmanip [54] or Vector [71] for efficient provenance identifier searching. Additionally, the current revocation process requires a full copy of the PVT. Implementing a Copy-on-Write (CoW) strategy would significantly reduce this memory overhead and avoid unnecessary duplication during the revocation cycle.

**Dynamically-adjustable PVT.** The favorable performance trade-off of colored capabilities depends on how many provenance identifiers can be encoded in the capability format. In § 6 and § 7, we describe PICASSO with a 21-bit provenance identifier on CHERI-Toooba. A larger identifier space allows more allocations between revocation sweeps, further improving overall system performance. However, it also has costs: 1) It consumes finite bits from the capability representation, competing with other CHERI extensions, 2) It increases memory usage for the PVT and `unr` storage, 3) It raises revocation overhead due to more `unr` entries. For example, CHERI-RISC-V with pointer masking [71] could support up to $2^{37}$ provenance identifiers, but would require a 16 GiB PVT—impractical for most use cases. To address this, we envision allocating a small PVT at process startup, which can grow dynamically when more identifiers are needed. This is especially beneficial for short-lived programs which use only a fraction of available identifiers. Smaller initial PVTs reduce memory footprint without impacting performance.

# 10 Conclusion

We introduce *colored capabilities*, a novel hardware-software co-design that enhances CHERI's temporal memory safety through efficient, *bulk retraction* of capabilities. By adding limited indirection to CHERI's capability model, colored capabilities addresses key shortcomings of existing revocation-based approaches–notably, mitigating UAF vulnerabilities without requiring quarantining memory management. Our implementation, PICASSO, demonstrates the practicality of this approach by integrating colored capabilities into CHERI-RISC-V, the CheriBSD OS, and the LLVM toolchain. Evaluation on thousands of NIST Juliet test cases, real-world CVEs, SPEC CPU benchmarks, and latency-sensitive real-world benchmarks confirm that colored capabilities improve temporal safety with reasonable performance overhead ($3\% - 8\%$ overhead on average).

# 11 Ethical Considerations

We identify three stakeholder groups affected by this research: hardware/software vendors implementing CHERI systems, CHERI system users, and malicious actors.

**Vendors** could face harm if vulnerabilities are exploited before patches are deployed. We mitigated this risk by relying on synthetic software benchmarks with known flaws, namely the NIST SARD Juliet Test suite, and previously known CVEs for our security evaluation. The Juliet Test Suite contains only artificial code, curated for the specific purpose of test security scanners. Further, the shortcomings of prior approaches to CHERI temporal safety are previously known and well-documented by prior research. Consequently, we expect no potential harm to vendors from research that aims to address these shortcomings.

**Users** could experience security breaches if temporal-safety vulnerabilities (e.g., use-after-free) are exploited. Our colored capabilities approach helps vendors detect and prevent such vulnerabilities, enhancing user security. All testing

was performed on dedicated test systems without involving production user devices. Since CHERI is predominantly deployed in pre-production hardware platforms, such as prototype systems, development boards, and FPGA testbeds, we expect no potential harm to end users.

**Malicious actors** might theoretically misuse our methods for vulnerability discovery. However, executing PICASSO on the FPGA platform supported by our artifact is orders of magnitude slower than alternative software-based approaches for vulnerability detection, such as sanitizer-instrumented binaries. Consequently, the PICASSO artifact does not provide additional capabilities beyond existing tools. We encountered no unexpected ethical issues during the research.

**Decision:** We reached the decision to proceed with and publish the research since we did not identify potential harms to the stakeholders we identified.

## 12 Open Science

Our artifact can be accessed at: https://github.com/coloredcapabilities/colored-artifact and at https://doi.org/10.5281/zenodo.20353117 [36]. PICASSO can be evaluated in three evaluation environments:

- **Bluespec Simulator:** Reproduces the results in Table 3 in Appendix C. Detailed instructions are available in Bluespec_simulation.md.
- **QEMU:** Reproduces the security evaluations in § 7.1. Documentation is provided in QEMU.md.
- **FPGA:** Reproduces security evaluation in § 7.1 and all other performance results. § 7.2 requires a Xilinx VCU118 FPGA (UltraScale+ Virtex) and a Vivado license (version 2019.1). See FPGA.md for setup details.

## Acknowledgments

We thank our anonymous Shepherd and our collegues at Ericsson: Santeri Paavolainen, Sini Ruohomaa, and Zoltán Turányi for their feedback on this manuscript. We are equally grateful to the University of Cambridge Computer Laboratory Security Group members, particularly Alfredo Mazzinghi, Peter Rugg, Jessica Clarke, Robert Watson, and Simon Moore.

This work is supported in part by the Wallenberg Visiting Professor Program, the Natural Sciences and Engineering Research Council of Canada (grant number RGPIN-2026-04826), and the Government of Ontario (RE011-038). Views expressed in the paper are those of the authors and do not necessarily reflect the position of the funders.

## References

[1] PostgreSQL. URL: https:/postgresql.org/.

[2] SPEC CPU 2006. URL: https://spec.org/cpu2006/.

[3] SQLite Home Page. URL: https://sqlite.org/.

[4] John Hennessy and David Patterson 2017 ACM A.M. Turing Award Lecture, June 2018. URL: https://www.youtube.com/watch?v=3LVeEjsn8Ts.

[5] Junho Ahn, et al. BUDAlloc: Defeating use-after-free bugs by decoupling virtual address management from kernel. In *USENIX Security '24*, pages 181–197, August 2024. USENIX. URL: https://www.usenix.org/conference/usenixsecurity24/presentation/ahn.

[6] Sam Ainsworth and Timothy M. Jones. MarkUs: Drop-in use-after-free prevention for low-level languages. In *S&P '20*, pages 578–591. IEEE, May 2020. doi:10.1109/SP40000.2020.00058.

[7] Thomas Aird, et al. "Svy" Extension, Version 1.0 for RV64Y (Stable state), August 2025. URL: https://riscv.github.io/riscv-cheri/#section_priv_cheri_vmem.

[8] Thomas Aird, et al. "Svucrg" Extension, Version 1.0 for RV64Y (Stable state), August 2025. URL: https://riscv.github.io/riscv-cheri/#section_cheri_priv_crg_ext.

[9] Periklis Akritidis. Cling: A memory allocator to mitigate dangling pointers. In *USENIX Security '10*, page 12, August 2010. USENIX. URL: https://www.usenix.org/conference/usenixsecurity10/cling-memory-allocator-mitigate-dangling-pointers.

[10] Saar Amar, et al. CHERIoT: Rethinking security for low-cost embedded systems. Technical Report MSR-TR-2023-6, Microsoft, February 2023. URL: https://www.microsoft.com/en-us/research/wp-content/uploads/2023/02/cheriot-63e11a4f1e629.pdf.

[11] Saar Amar, et al. CHERIoT: Complete Memory Safety for Embedded Devices. In *MICRO '23*, pages 641–653, October 2023. ACM. doi:10.1145/3613424.3614266.

[12] Emery D. Berger and Benjamin G. Zorn. DieHard: Probabilistic memory safety for unsafe languages. *ACM SIGPLAN Notices*, 41(6):158–168, June 2006. doi:10.1145/1133255.1134000.

[13] Tim Boland and Paul E. Black. Juliet 1.1 C/C++ and Java Test Suite. *Computer*, 45(10):88–90, October 2012. doi:10.1109/MC.2012.345.

[14] Jacob Bramley, et al. Picking a CHERI Allocator: Security and Performance Considerations. In *ISMM*

*2023*, pages 111–123, June 2023. ACM. doi:10.1145/3591195.3595278.

[15] Juan Caballero, et al. Undangle: Early detection of dangling pointers in use-after-free and double-free vulnerabilities. In *ISSTA '12*, pages 133–143, July 2012. ACM. doi:10.1145/2338965.2336769.

[16] Paolo Carlini, et al. The bitmap_allocator, August 2025. URL: https://gcc.gnu.org/onlinedocs/gcc-15.2.0/libstdc++/manual/manual/bitmap_allocator.html.

[17] David Chisnall, et al. CHERI JNI: Sinking the Java Security Model into the C. In *ASPLOS '17*, pages 569–583, April 2017. ACM. doi:10.1145/3037697.3037725.

[18] Haehyun Cho, et al. ViK: Practical mitigation of temporal memory safety violations through object ID inspection. In *ASPLOS '22*, pages 271–284, February 2022. ACM. doi:10.1145/3503222.3507780.

[19] Jim Chow, et al. Shredding Your Garbage: Reducing Data Lifetime Through Secure Deallocation. In *USENIX Security '15*, July 2005. USENIX. URL: https://www.usenix.org/conference/14th-usenix-security-symposium/shredding-your-garbage-reducing-data-lifetime-through.

[20] CTSRD. CTSRD-CHERI/postgres, June 2019. URL: https://github.com/CTSRD-CHERI/postgres.

[21] Thurston H. Y. Dang, Petros Maniatis, and David Wagner. Oscar: A practical page-permissions-based scheme for thwarting dangling pointers. In *USENIX Security '17*, pages 815–832, August 2017. USENIX. URL: https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/dang.

[22] Jack B. Dennis and Earl C. Van Horn. Programming semantics for multiprogrammed computations. *Communications of the ACM*, 9(3):143–155, March 1966. doi:10.1145/365230.365252.

[23] Gregory J. Duck and Roland H. C. Yap. EffectiveSan: Type and memory error detection using dynamically typed C/C++. In *PLDI '18*, pages 181–195, June 2018. ACM. doi:10.1145/3192366.3192388.

[24] EEMBC. Eembc/coremark. Embedded Microprocessor Benchmark Consortium, May 2018. URL: https://github.com/eembc/coremark.

[25] Hossam ElAtali, et al. BLACKOUT: Data-Oblivious Computation with Blinded Capabilities. In *CCS '25*, pages 2039–2053, November 2025. ACM. doi:10.1145/3719027.3765169.

[26] Márton Erdős, Sam Ainsworth, and Timothy M. Jones. MineSweeper: A "clean sweep" for drop-in use-after-free prevention. In *ASPLOS '22*, pages 212–225, February 2022. ACM. doi:10.1145/3503222.3507712.

[27] Reza Mirzazade Farkhani, Mansour Ahmadi, and Long Lu. PTAuth: Temporal Memory Safety via Robust Points-to Authentication. In *USENIX Security '21*, pages 1037–1054. USENIX, 2021. URL: https://www.usenix.org/conference/usenixsecurity21/presentation/mirzazade.

[28] Nathaniel Filardo. For discussion: MSR's CHERI+MTE composition, August 2024. URL: https://github.com/riscv/riscv-cheri/issues/340.

[29] Nathaniel Wesley Filardo, et al. Cornucopia Reloaded: Load Barriers for CHERI Heap Temporal Safety. In *ASPLOS '24*, volume 2 of *ASPLOS '24*, pages 251–268, April 2024. ACM. doi:10.1145/3620665.3640416.

[30] Franz A. Fuchs, et al. Safe Speculation for CHERI. In *ICCD '24*, pages 364–372, November 2024. IEEE. doi:10.1109/ICCD63220.2024.00063.

[31] Franz A Fuchs, et al. Architectural Contracts for Safe Speculation. In *ICCD '23*, pages 578–586, November 2023. IEEE. doi:10.1109/ICCD58817.2023.00093.

[32] Aïna Linn Georges, et al. Efficient and provable local capability revocation using uninitialized capabilities. In *POPL '21*, volume 5 of *POPL '21*, pages 6:1–6:30, January 2021. doi:10.1145/3434287.

[33] Richard Grisenthwaite. Arm Morello Evaluation Platform -Validating CHERI-based Security in a High-performance System. In *HCS '22*, pages 1–22. IEEE, August 2022. doi:10.1109/HCS55958.2022.9895591.

[34] Merve Gülmez, et al. Mon CHÉRI: Mitigating Uninitialized Memory Access with Conditional Capabilities. In *S&P '15*, pages 829–847, April 2025. IEEE. doi:10.1109/SP61157.2025.00133.

[35] Merve Gülmez, et al. PICASSO: Scaling CHERI Use-After-Free Protection to Millions of Allocations using Colored Capabilities, February 2026. arXiv:2602.09131, doi:10.48550/arXiv.2602.09131.

[36] Merve Gülmez et al. PICASSO artifact, May 2026. doi:10.5281/zenodo.20353117.

[37] Nicolas Joly, Saif ElSherei, and Saar Amar. Security Analysis of CHERI ISA. Technical report, Microsoft Security Response Center, October 2020. URL: https://msrc.microsoft.com/blog/2020/10/security-analysis-of-cheri-isa/.

[38] Byoungyoung Lee, et al. Preventing Use-after-free with Dangling Pointers Nullification. In *NDSS '15*, 2015. Internet Society. doi:10.14722/ndss.2015.23238.

[39] Henry M. Levy. *Capability-Based Computer Systems*. Butterworth-Heinemann, 1984. URL: https://homes.cs.washington.edu/~levy/capabook/.

[40] Yuan Li, et al. PACMem: Enforcing Spatial and Temporal Memory Safety via ARM Pointer Authentication. In *CCS '22*, pages 1901–1915, November 2022. ACM. doi:10.1145/3548606.3560598.

[41] Beichen Liu, Pierre Olivier, and Binoy Ravindran. SlimGuard: A Secure and Memory-Efficient Heap Allocator. In *Middleware '19*, pages 1–13, December 2019. ACM. doi:10.1145/3361525.3361532.

[42] Daiping Liu, Mingwei Zhang, and Haining Wang. A Robust and Efficient Defense against Use-after-Free Exploits via Concurrent Pointer Sweeping. In *CCS '18*, pages 1635–1648, October 2018. ACM. doi:10.1145/3243734.3243826.

[43] Alyssa Milburn, Herbert Bos, and Cristiano Giuffrida. SafeInit: Comprehensive and Practical Mitigation of Uninitialized Read Vulnerabilities. 2017. Internet Society. doi:10.14722/ndss.2017.23183.

[44] MITRE. CWE-562: Return of Stack Variable Address (4.17), February 2024. URL: https://cwe.mitre.org/data/definitions/562.html.

[45] Santosh Nagarakatte, et al. CETS: Compiler enforced temporal safety for C. *SIGPLAN Not.*, 45(8):31–40, June 2010. doi:10.1145/1837855.1806657.

[46] Manh-Dung Nguyen, et al. Binary-level Directed Fuzzing for Use-After-Free Vulnerabilities. pages 47–62, 2020. URL: https://www.usenix.org/conference/raid2020/presentation/nguyen.

[47] Gene Novark and Emery D. Berger. DieHarder: Securing the heap. In *CCS '10*, pages 573–584, October 2010. ACM. doi:10.1145/1866307.1866371.

[48] Chanyoung Park and Hyungon Moon. Efficient Use-After-Free Prevention with Opportunistic Page-Level Sweeping. 2024. Internet Society. doi:10.14722/ndss.2024.24804.

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

[50] Alex Rebert and Christoph Kern. Secure by Design: Google's Perspective on Memory Safety. Technical Report, Google, March 2024. URL: https://research.google/pubs/secure-by-design-googles-perspective-on-memory-safety/.

[51] David D. Redell. Naming and Protection in Extendible Operating Systems. Technical Report MAC TR-140, Massachussets Institute of Technology, November 1974. URL: https://apps.dtic.mil/sti/tr/pdf/ADA001721.pdf.

[52] Peter David Rugg. Efficient spatial and temporal safety for microcontrollers and application-class processors. Technical Report UCAM-CL-TR-984, University of Cambridge, July 2023. URL: https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-984.pdf.

[53] Jerome H. Saltzer. Protection and the Control of Information Sharing in Multics. *Commun. ACM*, 17(7):388–402, July 1974. doi:10.1145/361011.361067.

[54] Ved Shanbhogue. "B" Extension for Bit Manipulation, Version 1.0.0 :: RISC-V Ratified Specifications Library, April 2024. URL: https://docs.riscv.org/reference/isa/unpriv/b-st-ext.html.

[55] Sam Silvestro, et al. FreeGuard: A Faster Secure Heap Allocator. In *CCS '17*, pages 2389–2403, October 2017. ACM. doi:10.1145/3133956.3133957.

[56] Sam Silvestro, et al. Guarder: A tunable secure allocator. In *USENIX Security '18*, pages 117–133, August 2018. USENIX. URL: https://www.usenix.org/conference/usenixsecurity18/presentation/silvestro.

[57] GCC Team. Options to Request or Suppress Warnings: -Wno-return-local-addr, August 2025. URL: https://gcc.gnu.org/onlinedocs/gcc-15.2.0/gcc/Warning-Options.html#index-Wno-return-local-addr.

[58] LLVM Team. Diagnostic flags in Clang, March 2025. URL: https://releases.llvm.org/20.1.0/tools/clang/docs/DiagnosticsReference.html#wreturn-stack-address.

[59] LLVM Team. Scudo Hardened Allocator, March 2025. URL: https://releases.llvm.org/20.1.0/docs/ScudoHardenedAllocator.html.

[60] Craig Tiller, et al. gRPC, June 2025. URL: https://github.com/grpc/grpc/tree/v1.54.2.

[61] Erik van der Kouwe, Vinod Nigade, and Cristiano Giuffrida. DangSan: Scalable Use-after-free Detection. In *EuroSys '17*, pages 405–419, April 2017. ACM. doi:10.1145/3064176.3064211.

[62] Robert N M Watson, et al. Arm Morello Programme: Architectural security goals and known limitations. Technical Report UCAM-CL-TR-982, University of Cambridge, July 2023. URL: https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-982.pdf.

[63] Robert N. M. Watson, et al. Capability Hardware Enhanced RISC Instructions: CHERI Instruction-Set Architecture (Version 9). Technical Report UCAM-CL-TR-987, University of Cambridge, September 2023. doi:10.48456/TR-987.

[64] Robert N M Watson, et al. Capability Hardware Enhanced RISC Instructions: CHERI Instruction-Set Architecture (Version 7). Technical Report UCAM-CL-TR-927, University of Cambridge, June 2019. doi:10.48456/TR-927.

[65] Nathaniel Wesley Filardo, et al. Cornucopia: Temporal Safety for CHERI Heaps. In *S&P '20*, pages 608–625. IEEE, May 2020. doi:10.1109/SP40000.2020.00098.

[66] Brian Wickman, et al. Preventing Use-After-Free Attacks with Fast Forward Allocation. In *USENIX Security Symposium '21*, pages 2453–2470. USENIX, August 2021. URL: https://www.usenix.org/conference/usenixsecurity21/presentation/wickman.

[67] Matthew Wilcox. ID Allocation, November 2025. URL: https://www.kernel.org/doc/html/v6.18/core-api/idr.html#id-allocation.

[68] Jonathan Woodruff, et al. CHERI Concentrate: Practical Compressed Capabilities. *IEEE Transactions on Computers*, 68(10):1455–1469, October 2019. doi:10.1109/TC.2019.2914037.

[69] Yves Younan. FreeSentry: Protecting Against Use-After-Free Vulnerabilities Due to Dangling Pointers. In *NDSS '15*, 2015. Internet Society. doi:10.14722/ndss.2015.23190.

[70] Jiyong Yu, et al. Speculative Taint Tracking (STT): A Comprehensive Protection for Speculatively Accessed Data. In *MICRO-52*, pages 954–968, October 2019. ACM. doi:10.1145/3352460.3358274.

[71] Adam Zabrocki, et al. RISC-V Pointer Masking, Version 1.0 (Ratified), October 2024. URL: https://raw.githubusercontent.com/riscv/riscv-j-extension/master/zjpm-spec.pdf.

# A QEMU Implementation

We implemented a functional model of PICASSO in QEMU-system-CHERI128 to enable kernel development and functional validation prior to hardware development. The QEMU implementation mirrors the hardware ISA extensions, including the PVTR and the `ccsettype` instruction, as well as the compressed capability format modifications, the expanded 21-bit object type field identical to the hardware implementation. The primary architectural difference lies in the PVT access model. Whereas Toooba has its own translation lookaside buffer (TLB) and buffer, QEMU performs PVT lookups equentially during capability validation. When a load or store uses a provenance identifiers, the emulator computes the PVB address and reads the corresponding PVT word from emulated memory before the instruction completes. To optimize the common case, QEMU first probes its software TLB for a cached translation. If found, the PVT word is read directly from host memory, avoiding full address translation. On a TLB miss, the emulator falls back to complete guest address translation with proper exception handling.

# B Optimized `unr` allocator

As described in section 6.2, we utilize the `unr` from FreeBSD to manage both claimed and available provenance identifiers. This approach enables us to store the state of all provenance identifiers in a compressed form, implemented as a linked list of `unr` objects. These objects can represent either a `run` or a `bitmap`. A `run` is a series of claimed or available numbers represented by a type (claimed or available) and a length. For example, 50 consecutive allocated numbers can be represented in an efficient way using: {state = "claimed", len = 50}. To save memory, multiple small `runs` can be combined into one `bitmap` representation of a fixed size (in our case 512 bits), for example: {state = "claimed", len = 3} + {state = "available", len = 2} + {state = "claimed", len = 3} = "11100111". These bitmaps are formed where they replace at least three `runs`, because this is the point where the bitmap takes up less memory then the runs it replaced.

The approach of leveraging runs of claimed and available provenance identifiers offers two advantages. Firstly, it optimizes memory usage by representing consecutive claimed or available IDs as a single run rather than tracking each ID individually. This compression is effective because, when claiming the first available provenance identifiers sequentially, long runs of claimed IDs naturally form. Secondly, finding the first available number is efficient because the sequence of claimed numbers will always start with a consecutive run from the beginning. This ensures that the first unclaimed number immediately follows this claimed run.

However, releasing numbers incurs significant overhead, which becomes problematic when releasing more than 1 million provenance identifiers simultaneously. This overhead

arises from the three steps involved in releasing an ID. First, the system iterates over the doubly linked list to find the `unr` objects that contain the to be released ID. Second, it restructures the data to indicate that the number is now available by resizing the current and/or neighboring `unr` objects. Finally, it optimizes memory usage by searching for any new bitmaps that can be formed.

To optimize the batch-releasing process, we eliminate the overhead of the first and last steps by releasing IDs in sorted order and scanning the allocator exactly once. As we traverse the allocator, we free each number on the fly, avoiding a separate search for every release. Simultaneously, we look for opportunities to form a "first good bitmap" rather than waiting to compute the absolute optimal bitmaps. Whenever the current layout yields any memory savings by switching from runs to a bitmap, we convert immediately, even if it's not the maximal compression. This approach reduces memory usage with minimal additional overhead.

## C Complementary Evaluation

**`PICASSO` PVT Buffer Optimization.** We evaluate the performance impact of the PVT buffer using the SPEC CPU2006 `bzip2` benchmark. While bzip2 includes three distinct inputs for the *train* workload, we utilize a single representative input for this evaluation. The workload performs 29 allocations, executing compression and decompression operations on each. Figure 16 illustrates the normalized hardware performance counter overheads, comparing `PICASSO` without the PVT buffer against the optimized design, both normalized to the baseline. Without the PVT buffer, `PICASSO` incurs a 1.7× cycle overhead, primarily due to a 2.39× increase in data cache loads. This is due to the frequent memory accesses required to verify allocation-provenance validity (PVBs). In contrast, with the PVT buffer enabled, the overhead drops to 1.002×. The buffer effectively eliminates the performance penalty of PVT lookups by caching recently accessed PVBs.

**`PICASSO` Comparison with Cornucopia-revoke-on-free (RoF) on SQLite.**: To address the UAF, Cornucopia can be configured in a revoke-on-free mode, which triggers a global revocation sweep upon every `free` operation. From a security standpoint, `PICASSO` and Cornucopia-RoF provide equivalent protection against UAF/UAR. However, a key mechanical distinction remains: `PICASSO` does not perform a full memory scan upon deallocation; instead, it retracts the capability, deferring the computationally expensive global sweep until the provenance identifiers space is exhausted. We evaluate Cornucopia-RoF using the SQLite `speedtest1` benchmark under the same configuration described in § 7.2. Figure 14 shows the cycles, instruction and memory overhead for Cornucopia, Cornucopia-RoF and `PICASSO`. Our results demonstrate that while Cornucopia-RoF reduces memory overhead from 1.99× to 1.1× (by eliminating the quarantine buffer) compare to Cornucopia, this comes at a prohibitive performance cost of 6.10× overhead relative to the baseline. In contrast, `PICASSO` achieves equivalent security guarantees without the penalty of frequent memory scans.

Table 3: PICASSO overhead on MiBench

| Benchmark | Baseline | PICASSO | Overhead (%) |
|---|---|---|---|
| adpcm_decode | 3325012 | 3380972 | 1.68% |
| adpcm_encode | 3048620 | 3083658 | 1.15% |
| aes | 74548 | 75133 | 0.78% |
| bitcount | 2632864 | 2632886 | 0.00% |
| blowfish | 1393763 | 1465512 | 5.15% |
| crc | 14435 | 14509 | 0.51% |
| dijkstra | 1427512 | 1432030 | 0.32% |
| limits | 4434 | 4436 | 0.05% |
| patricia | 1610777 | 1632902 | 1.37% |
| picojpeg | 2038299 | 2074049 | 1.75% |
| qsort | 514081 | 513281 | -0.16% |
| randmath | 38221 | 38601 | 0.99% |
| rc4 | 67938 | 68072 | 0.20% |
| rsa | 49450 | 52004 | 5.16% |
| sha | 1464582 | 1474223 | 0.66% |
| Average | | | 1.31% |

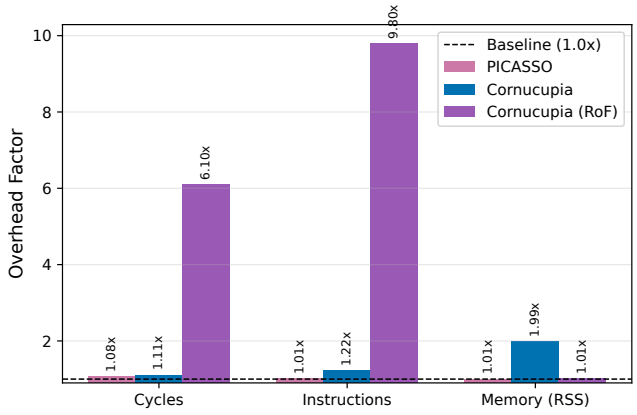



Figure 14: Normalized cycles, instructions, and memory overhead for SQLite speedtest1 for Cornucopia, Cornucopia-RoF and `PICASSO`.

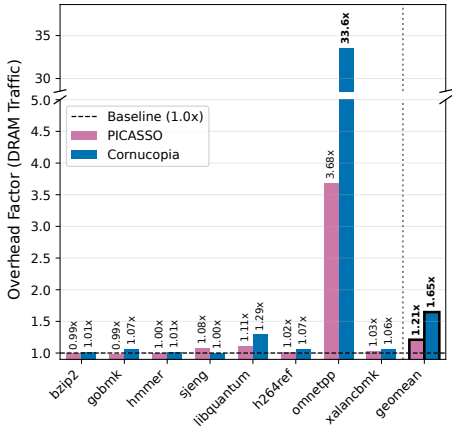



Figure 15: Approximation of DRAM traffic overhead.

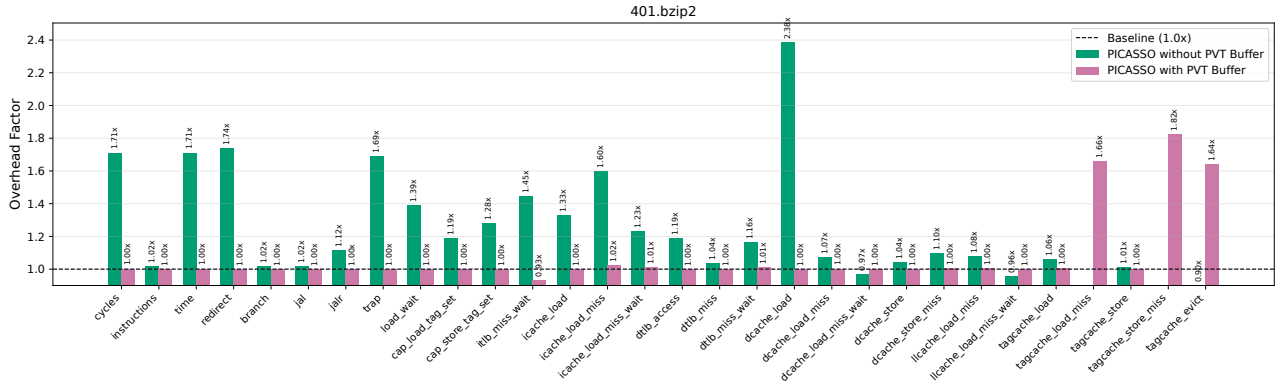



Figure 16: SPEC CPU 2006 bzip performance overhead for `PICASSO` without PVT buffer and `PICASSO` with PVT buffer.

Table 4: PostgreSQL pgbench: TPS and transaction latency percentiles (ms) with overhead vs. baseline

| Scenario | TPS | P50 | P90 | P99 | P99.9 |
|---|---|---|---|---|---|
| Baseline | 2.10 | 463.3 | 519.1 | 538.8 | 1129.9 |
| PICASSO | 1.97 (-6.2%) | 494.6 (+6.7%) | 554.7 (+6.9%) | 573.5 (+6.4%) | 1165.1 (+3.1%) |
| Cornucopia | 1.61 (-23.1%) | 538.9 (+16.3%) | 606.8 (+16.9%) | 2182.3 (+305.0%) | 2291.3 (+102.8%) |

Table 5: Performance cost on VCU118 @ 25MHz expressed as CoreMark test results for $5x10^3$ iterations. The CoreMark score for a processor is reported as CoreMark-iterations-per-second-per-core-MHz. The Δ is relative to CHERI-Toooba Core results.

| | Total ticks | Δ | | Total time (sec) | Δ | | Iterations/sec | Δ | | Score |
|---|---|---|---|---|---|---|---|---|---|---|
| **CHERI-Toooba** | | | | | | | | | | |
| baseline (nocap) | 938984056 | – | | 37 | – | | 135 | – | | 5.4 |
| purecap | 954633763 | +15649707 | +1.67% | 38 | +1 | +2.70% | 131 | -4 | −2.96% | 5.24 |
| **PICASSO CHERI-Toooba** | | | | | | | | | | |
| baseline (nocap) | 973452912 | +34468856 | +3.67% | 38 | +1 | +2.70% | 131 | -4 | −2.96% | 5.24 |
| purecap | 995997744 | +57013688 | +6.07% | 39 | +2 | +5.41% | 128 | -7 | −5.19% | 5.12 |

Table 6: gRPC benchmark: QPS and latency percentiles with overhead vs. baseline

| Scenario | QPS | P50 (ms) | P90 (ms) | P95 (ms) | P99 (ms) |
|---|---|---|---|---|---|
| Baseline | 15.24 | 515.1 | 569.0 | 580.4 | 598.0 |
| PICASSO | 14.80 (-2.9%) | 530.7 (+3.0%) | 586.2 (+3.0%) | 604.0 (+4.1%) | 628.5 (+5.1%) |
| Cornucopia | 12.87 (-15.6%) | 552.3 (+7.2%) | 634.8 (+11.6%) | 687.4 (+18.4%) | 2202.0 (+268.2%) |