Cryptographic Least Privilege Enforcement for Scalable Memory Isolation

Martin Unterguggenberger, David Schrammel, Lukas Maar, Lukas Lamster, Vedad Hadžić, Stefan Mangard
Graz University of Technology: {firstname.lastname}@iaik.tugraz.at

***Abstract*—C/C++ computing systems constitute a significant share of our critical software infrastructure and face substantial security risks from memory exploitation. A single memory safety error can potentially lead to the compromise of the entire software system. To efficiently secure C/C++ computing systems without extensive software adaption, the processor must be able to restrict memory access to individual memory locations, thereby enforcing the principle of least privilege. The integration of lightweight and transparent isolation mechanisms that offer flexible and scalable memory protection is crucial to minimize the attack surface of software attacks.**

In this paper, we present cryptographic least privilege enforcement (CLPE), a novel mechanism for scalable memory isolation. Our lightweight ISA extension enforces cryptographic integrity checks for isolation granularities ranging from individual objects to arbitrarily sized protection domains. We achieve this through message authentication codes (MACs), linking pointers with specific access privileges that restrict access to memory resources. Our approach maintains compatibility with legacy software and only minimally increases the processor's microarchitectural complexity. We provide a formal model of our design, ensuring important properties of our ISA specification, and a hardware model, allowing functional and timing-accurate simulation. The simulated performance overhead of our hardware model shows an average overhead of 2.5–7.4 % for the SPEC CPU2017 benchmark suite.

***Index Terms*—memory safety, principle of least privilege.**

I. Introduction

Virtual memory-based process isolation provides strong memory isolation by separating processes at the operating system level. However, the lack of memory safety measures within a process introduces a substantial security risk: a single memory safety vulnerability [61] has the potential to completely compromise the target system. There is a strong need for processor enhancements that provide fine-grain, efficient, and transparent memory isolation to prevent the exploitation of software vulnerabilities.

The *principle of least privilege* [52] is a fundamental software paradigm typically enforced by the isolation of distinct entities. Least privilege memory isolation in hardware is particularly relevant to software built on memory-unsafe languages by restricting unauthorized access to resources in memory. Capability-based systems [14] implement the principle of least privilege at the architectural level. The CHERI ISA [69], [70], [71] provides scalable memory isolation through capability-based addressing. CHERI performs access checks in hardware using fat pointers where the base, bounds, and permissions are co-located with pointers to form capabilities. Capabilities restrict access to the specified memory ranges through dedicated capability registers and instructions. Additionally, capability-based systems allow for privilege management, e.g., the revocation of access privileges [19], [72]. Despite the security benefits of CHERI, its deployment can be challenging due to the significantly increased microarchitectural complexity and legacy incompatibilities.

In contrast, *lightweight ISA extensions* are largely transparent to the software (*i.e.*, introducing no new dependencies) and only require minimal hardware changes without affecting the microarchitectural design of the CPU pipeline. The ARM memory tagging extension (MTE) [57] and SPARC application data integrity (ADI) [1], for example, integrate tagged memory to enforce fine-grain access policies [25]. Tagged memory architectures associate metadata, called *memory tag*, with each memory granule on a defined size and granularity. Specifically, ARM MTE and SPARC ADI employ a 4-bit memory tag with a granularity of $16\,\mathrm{B}$ and $64\,\mathrm{B}$, respectively. Both mechanisms use pointer tagging and associate the tag with the corresponding memory location. Subsequent memory accesses perform *logical integrity* checks by comparing the pointer tag with the tag assigned to the memory location. This allows memory tagging to isolate individual objects, thus providing memory safety. However, due to the limited tag size of 4 bits, tag collisions are likely to occur, resulting in potential undetected memory safety errors.

Intel memory protection keys (MPK) [49] are a lightweight mechanism for page-based access control. MPK employs a 4-bit protection key stored in the page table entries (PTEs) along with a user space protection key register (PKRU) to implement page-level memory protection. PKRU-based protection schemes [10], [22], [55], [67] introduce transparent and efficient domain isolation using MPK's page-granular access policies. Similar to Intel MPK, ARM memory domains also use 4-bit domain IDs (akin to protection keys) for memory isolation. Both mechanisms can be considered coarse-grain memory tagging as they perform logical integrity checks at a $4\,\mathrm{kB}$ page granularity. However, due to the limited 4-bit key size, both Intel MPK and ARM memory domains offer only 16 distinct domains. This is restrictive as larger applications may use up to thousands of concurrent execution contexts. Both schemes also lack sub-page granular isolation, thus limiting their applicability to object-level isolation.

In summary, existing memory isolation techniques are limited in providing one or more of the following properties: strong detection capabilities, scalable isolation granularities, support for numerous concurrent execution contexts, and min-

imal microarchitectural complexity. There is a strong need for memory isolation mechanisms that unite these properties, thus enabling scalable isolation levels (*i.e.*, achieving memory safety and domain isolation) within a single user-space process while maintaining legacy compatibility.

In this paper, we present cryptographic least privilege enforcement (CLPE), a novel mechanism for memory isolation that scales efficiently through cryptography. Our lightweight ISA extension integrates *cryptographic integrity* checks using message authentication codes (MACs). Memory operations are transparently authenticated in hardware, utilizing recently introduced low-latency block cipher designs [4], [5], [7], [34]. A MAC is directly associated with each pointer, enabling fine-grain privilege management that supports frequent policy changes with cryptographic strength. As a result, our ISA extension detects memory safety errors with the security level of the cryptographic MAC, offering several advantages over established memory protection mechanisms:

(i) *Cryptographic least privilege policies*: The MAC associates each pointer with a specific address range and restricts memory access exclusively to that isolated memory region. Thus, our new hardware feature allows privilege-centered isolation for flexible memory protection similar to capability-enhanced processors, e.g., the CHERI ISA. In contrast to these processors, our hardware feature only minimally increases the microarchitectural complexity.

(ii) *Scalable isolation granularities*: Our cryptographic mechanism is efficiently scalable from single object isolation (*i.e.*, memory safety) to arbitrarily sized domain isolation, even supporting secure interleaved memory sharing. In contrast, mechanisms such as software-based fault isolation (SFI) [62], [68] cannot provide this flexibility due to their partitioning of contiguous memory regions. Other mechanisms that rely on page-level isolation (e.g., Intel MPK) are bound to their page granularity and cannot support finer levels of isolation.

(iii) *Flexible number of execution contexts*: Our cryptographically enforced memory isolation allows thousands of concurrent execution contexts (*i.e.*, protection domains) while supporting frequent policy changes. In contrast, isolation mechanisms such as Intel MPK and ARM memory domains only allow for 16 distinct domains, limiting their use.

(iv) *Lightweight ISA extension*: We only require minimal hardware changes, being adaptable to complex processor microarchitectures. Moreover, our binary and ABI-compatible solution can be easily integrated into legacy computing systems by linking our security-hardened allocator, e.g., to safeguard large and unmodified monolithic software.

In addition to the design of CLPE, we describe a formal model consisting of a Sail specification of the ISA extension and subsequently state and prove important properties of our design. Furthermore, we provide a prototype implementation consisting of a hardware model based on the gem5 simulator [9], [37], a hardened memory allocator, and a Linux kernel patch. The hardware model allows functional and timing-accurate simulation. We evaluate our prototype implementation in terms of system performance, highlighting an average overhead of 2.5–7.4 % for the subset of C benchmarks of SPEC CPU2017 [13]. Moreover, we also provide an extensive security analysis, including a systematic analysis and an empirical evaluation using the NIST Juliet [12] test suite.

Contributions. In summary, our main contributions are:

(1) **Cryptographic Least Privilege Enforcement.** We present a lightweight ISA extension for fine-grain memory isolation, enforcing the principle of least privilege through efficient and scalable cryptography.
(2) **Formal Model of the ISA Extension.** We describe and verify a formal ISA specification of our design.
(3) **Hardware Model for Evaluation.** We implement and evaluate a proof-of-concept prototype, demonstrating that our design is practical.
(4) **Systematic Security Analysis.** We systematically analyze the security of our ISA extension and empirically evaluate the efficacy of our design.

Outline. The remainder of the paper is organized as follows. Section II discusses the background of this work. Section III and Section IV describe the design and formal model of our ISA extension. Section V details our proof-of-concept implementation. Section VI and Section VII provide the security analysis and performance evaluation. Section VIII discusses related work, and Section IX concludes this work.

II. Background

This section provides background on memory safety errors and cryptographic memory protection.

A. Memory Safety Errors

Software developed using memory-unsafe programming languages, such as C and C++, grants developers substantial control over data stored in memory. However, programming bugs can lead to exploitable memory safety errors, a prevalent type of software vulnerability [40], [63]. Memory safety vulnerabilities allow an adversary to manipulate the program's state and consequently alter the behavior of the execution. Memory safety errors can be classified into spatial and temporal memory safety vulnerabilities [61].

Spatial memory violations, such as out-of-bounds (OOB) errors, allow an attacker to either leak or modify sensitive data. A spatial memory violation occurs when a pointer is dereferenced outside the intended memory boundaries of the associated object. For instance, a linear buffer overflow accesses data adjacent to the memory location of the buffer, allowing this adjacent data to be leaked or corrupted. Moreover, an arbitrary memory access violation allows data manipulation in non-adjacent memory locations.

Temporal memory violations typically use dangling pointers (*i.e.*, pointers that refer to freed memory objects) to perform a use-after-free (UAF) access. Specifically, an attacker can misuse a dangling pointer to access the data of a freed memory object. The UAF error can be exploited to leak or corrupt potentially sensitive data that resides in the same memory location. UAF errors are further categorized depending on the memory state (whether the memory is free or has been

reallocated). UAF errors that affect memory in the reallocated state are critical. Precisely, if the memory has been reallocated, the attacker can access the reallocated and potentially sensitive object. Moreover, double-free errors are a special case of UAF, allowing an object to be freed twice.

B. Cryptographic Memory Protection

Cryptography is a versatile building block for system security. It addresses a wide range of security concerns through cryptographic primitives such as encryption and authentication. Various academic and commercial designs introduce different mechanisms based on cryptographic primitives to implement security measures for memory safety [53], [54], memory integrity [27], and software isolation [64].

Cryptographic pointer integrity [15], [39], [50] can be used to protect against control-flow hijacking attacks. ARM pointer authentication (PAuth) [50], for instance, uses a MAC encoded in the upper bits of the pointer to ensure pointer integrity while stored in memory. Security researchers have proposed ISA extensions [35], [46], [65] that leverage cryptographic building blocks to enforce memory safety. These techniques use cryptography to enforce confidentiality or integrity (or both) of data stored in memory. In addition, there have been academic proposals for secure enclave architectures [60] and DRAM integrity protection schemes [18], [27], [32], [33] based on cryptographic primitives.

III. Design

This section presents our novel ISA extension designed to enable fine-grain and scalable memory isolation through efficient low-latency cryptography. At its core, our design cryptographically binds memory access privileges to pointers, enabling *least privilege policies* that detect access violations with cryptographic strength. To achieve scalable isolation, we use a message authentication code (MAC) to perform fine-grain and implicit *cryptographic integrity* checks for every memory operation. Similar to pointer tagging, the MAC is encoded into the unused upper bits of the pointer. Thereby, we construct a *cryptographic lock-and-key* mechanism where access to a memory location is solely granted with the corresponding access privileges. The MAC represents the pointer's access privileges to specific memory regions, enforcing the principle of least privilege.

A. Threat Model

Our threat model is consistent with existing research on memory safety [21], [28], [35], [65], [73]. We assume that one or more memory safety vulnerabilities are present in the target user space program, granting an adversary arbitrary read-and-write capabilities. Additionally, we presume that the attacker has knowledge of the address space layout (*i.e.*, is able to bypass ASLR).

However, we assume Write-XOR-Execute is enabled, preventing the injection of arbitrary code. We also assume the operating system is trusted and free of exploitable programming errors. Note that microarchitectural attacks [30], [36] and fault attacks [29], [43] are beyond the scope of this work.

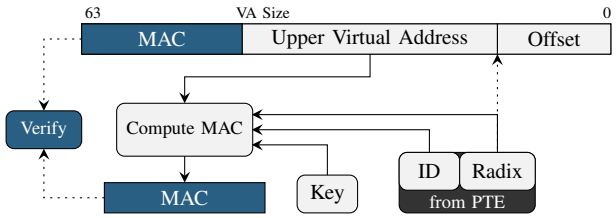

Fig. 1: The cryptographic lock-and-key mechanism for least privilege enforcement. Memory operations are implicitly verified using cryptographic integrity checks. The MAC is computed at every read and write operation and compared against the MAC encoded in the pointer.

B. Design Properties

We define the following key properties for our design to address the challenges of memory isolation for modern computing systems and legacy software:

- **Principle of least privilege:** Our design cryptographically enforces the least privilege principle for memory operations transparently in hardware.
- **Scalable isolation granularities:** Our approach supports memory isolation that scales from the level of individual objects (*i.e.*, memory safety) to the isolation of arbitrarily sized domains, depending on the security requirements of the respective software module.
- **Flexible number of execution contexts:** Our design allows the isolation of an arbitrary number of concurrent execution contexts (*i.e.*, protection domains) while also supporting frequent policy changes.
- **Lightweight ISA extension:** Our extension only minimally increases the CPU's microarchitectural complexity and preserves binary and ABI compatibility.

C. Secure Hardware Architecture

Our ISA extension incorporates cryptographic integrity checks, validating the pointer's access privileges on every memory operation. For this, we utilize MACs encoded in the pointer, representing its access privileges. This cryptographic lock-and-key approach allows us to enforce least privilege policies for scalable isolation granularities ranging from single-byte object-granular isolation to the isolation of arbitrarily sized protection domains.

Pointer Layout and MAC Generation. Figure 1 illustrates the cryptographic lock-and-key approach. Our scheme uses a 6-bit radix in combination with a 4-bit ID as page-granular metadata, both of which are stored in the PTE. The radix divides a pointer into two parts: an upper fixed address part and a lower offset part. The upper fixed part of the address is used for the MAC computation, together with the radix, the ID, and a secret per-process key. When the pointer is signed (e.g., during a memory allocation), the resulting MAC is encoded in the upper bits of the pointer. The derived MAC grants access privileges solely to memory with the power-of-two bounds set by the radix. The lower offset part of the pointer can be

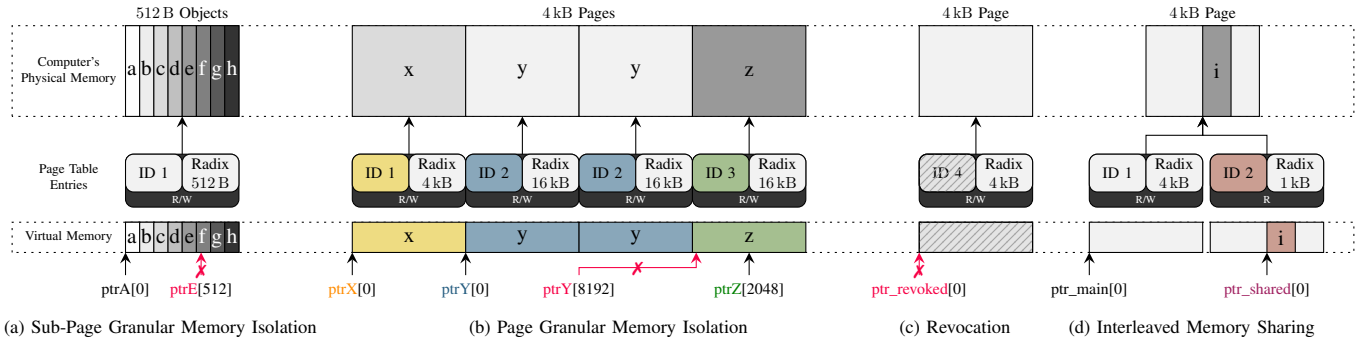

Fig. 2: Overview of the different cryptographic least privilege policies enforced by our ISA extension.

modified (e.g., through pointer arithmetic) to point to different locations within this memory region.

MAC Verification. Memory operations are implicitly verified using cryptographic integrity checks performed by the processor's microarchitecture. This validation restricts pointer dereferences to their designated memory region. During memory access, our hardware-enforced authentication recomputes the cryptographic MAC in the same way as the initial generation. The encoded and recomputed MAC are then compared for equality (cf. Figure 1). As a result, memory safety errors that allow dereferencing a pointer outside its access privileges will be detected by the MAC verification, thus preventing illegal memory access. Moreover, we use the ID to enforce different privilege policies at a page granularity. This allows us to directly revoke access privileges for pointers by updating the ID (or radix) of the PTE. The cryptographic integrity check validates that a pointer has the corresponding access privileges for a memory location, indicated by a successful MAC authentication. Note that the hardware architecture ensures validity for the entire data width of the memory operation. Any alteration of the upper fixed address, radix, or ID influences the MAC calculation (*i.e.*, memory accesses outside the intended memory region or revoked privileges), resulting in a failed MAC authentication and immediately triggering a hardware exception leading to program termination by the OS. Consequently, the pointer becomes unforgeable within the cryptographic bounds of the MAC and can only be dereferenced within its intended memory region.

Legacy Compatibility. Our pointer layout (*i.e.*, pointer tagging) is largely compatible with existing C/C++ computing systems. The pointer can be used for pointer arithmetics or array indexing within its allowed bounds. Furthermore, for compatibility with legacy software, we designate legacy memory locations using a distinct radix encoding (*i.e.*, all radix bits set to ones). This allows legacy memory locations to be accessed using legacy pointers without a MAC, while memory access to protected memory is always authenticated. Thus, legacy pointers cannot be used to access protected memory. Note that legacy and protected memory are not co-located within a single page. This allows us to be binary compatible without compromising security or usability.

D. Cryptographic Least Privilege Policies

In the following, we discuss four policies on how our design enforces scalable memory isolation and revocation of access privileges. Figure 2 provides an overview of the different policies enforced by our ISA extension.

Sub-Page Granular Memory Isolation. Within a page, several different memory objects can be isolated from each other, given that each of them resides in its own power-of-two region. Figure 2a shows an example in which a 4 kB page is used for eight 512 B objects. To isolate the objects against each other, we set the radix for this page to 512 B, ensuring that the signed pointers to these objects are restricted to accessing their object and are prohibited from accessing the other seven. As the upper fixed part of each pointer varies between distinct 512 B objects, an out-of-bounds (OOB) access results in a MAC mismatch and, thus, a verification failure. This radix-based isolation works with arbitrarily small (power-of-two) granularities down to a single byte.

Page Granular Memory Isolation. To isolate objects larger than a page, allocations need only be aligned to a page size (*i.e.*, 4 kB), as long as no other PTE exists within that power-of-two region that has the same radix and ID pair. The memory allocator can efficiently manage this policy. For example, all unused pages within the given power-of-two page can be left unmapped. Note that this does not use any additional physical memory. Alternatively, the virtual pages in that region can still be used as long as the radix or ID is different since a signed pointer can only access pages with correct PTE metadata.

Our scheme also allows access to memory regions of non-power-of-two size by *interleaving radices*. The ID is used to limit the access privileges of overlapping radix power-of-two sizes for large memory regions. For instance, a 12 kB (*i.e.*, 3 pages) memory object is given a radix value that allows it to access a 16 kB region (*i.e.*, 4 kB more than it should have access to). This 16 kB region has 4 PTEs, each mapping to 4 kB pages. However, the PTE of the last 4 kB page uses a different radix or ID. As a result, the first 12 kB can be isolated from the adjacent 4 kB.

Figure 2b depicts a similar case. There, the 8 kB memory region y gets a radix of 16 kB, as that particular region is not

8 kB-aligned. Since the two adjacent memory regions (*i.e.*, `x` and `z`) use different radices or IDs, all three objects are isolated from each other. Therefore, by properly selecting the ID values for large memory regions, we can securely interleave different radix sizes with no loss in security or usable memory. Notably, the same ID values can be reused freely for different non-overlapping radices.

Revocation. The radix defines the power-of-two size of the memory range that is cryptographically linked to the pointer, while the ID field acts as a version number. To revoke the access privileges of existing pointers, it is sufficient to update either the radix or the ID in the PTE, thus invalidating all previous pointers to objects on that page. If the same radix and ID pair is not reused for that page, a revoked pointer can never be used again without causing an exception. Figure 2c shows this where the memory of `ptr_revoked` used to have ID 4 but has since been updated.

Interleaved Memory Sharing. Figure 2d depicts how different radices may be used on a single physical page via memory aliasing. If a sub-range of memory needs to be shared (e.g., to an untrusted function), a new PTE can be generated with a different radix and ID. The PTE may also set different page permissions, such as *read-only*. The resulting pointer would then only have access to that sub-range limited by the radix.

E. Isolation of Objects and Domains

The cryptographic integrity checks provided by our ISA extension enable the isolation of individual objects (*i.e.*, memory safety) and the isolation of protection domains.

Spatial Memory Safety. We enforce spatial safety through specific memory management and alignment facilitated by binning memory allocators. Binning memory allocators group identical-sized objects in *buckets*. We take advantage of this structure, as all slots within a bucket have the same size. Thus, our page-granular metadata (*i.e.*, radix and ID) applies to the entire bucket. In our case, buckets use power-of-two sizes, which facilitates for the detection of spatial memory safety errors. Any spatial memory violation modifies the upper part of the virtual address, which results in a failed MAC authentication and is detected within the cryptographic security bounds.

Temporal Memory Safety. We enforce temporal memory safety by combining our mechanism with a memory quarantining strategy [2], [19]. Conceptually, freed memory is quarantined and only reused after ensuring temporal safety requirements. We isolate freed objects in quarantine for each page until all objects within that page are freed. Subsequently, we assign a new ID to that page, which automatically revokes access for potential dangling pointers. Since the updated ID results in a failed MAC authentication (within cryptographic bounds), it prevents potential dangling pointers from accessing objects on the respective page. This allows us to revoke and reuse virtual pages up to 16 times, as we have a 4-bit ID. After exhausting the provided ID range, we propose three approaches. First, one can opt to not use the virtual address of that page anymore, resulting in a slight decrease in usable virtual address space. Second, the particular virtual page can be reused for a different bucket size (*i.e.*, different radix). Third, memory scanning [8] can be used to identify potential dangling pointers before reuse.

Protection Domains. Similar to spatial memory safety, sub-page granular isolation is achieved through power-of-two memory alignment, which enforces radix-aligned memory protection. Larger (*i.e.*, at least page-sized) regions are page-size aligned and isolated by a unique ID within the radix region. Similar to temporal memory safety, when an execution context ends, we update the ID of all of the context's pages (up to 16 times). This prevents dangling pointers from old execution contexts, which may still exist elsewhere in the program, from being used.

Interleaved and Revocable Memory Sharing. As shown in Figure 2d, our design also supports secure (interleaved) shared memory between one or more protection domains. We use page table aliasing to establish shared memory between two distinct execution contexts, with privilege policies enforced by the radix and ID in the respective PTE. For instance, our design allows for fine-grain shared memory at the sub-page level through the radix in the aliased PTE. The corresponding pointer has access privileges only to the (sub-page) shared memory location, given a radix power-of-two alignment. Again, spatial violations are prevented by the radix boundaries enforced by the cryptographic integrity checks of the hardware feature. We also enforce the page-granular read-and-write permission present in the aliased PTE (via the `mprotect` system call), enabling read-only sharing. Revocation of aliased shared memory objects is achieved by updating the ID in the aliased PTE without having to unmap the shared memory entirely. Similar to temporal safety, an updated ID invalidates pointers to shared memory objects. By using more aliases, memory objects can be shared between multiple domains, each with different access permissions.

IV. Formal Model

In the following, we formalize the proposed ISA extension in the domain-specific language called Sail [3]. The reason we chose Sail is its powerful type system and versatile backends, as well as its use in the industry, e.g., for the official RISC-V specification [42]. Moreover, being able to export Sail code to logic formulas in the *satisfiability modulo theories* (SMT) format [6], we are able to formally verify desired properties of our ISA extension.

A. Sail Specification

Sail is a strongly typed language with automatic type inference. For the purpose of defining our ISA extension, we first define type aliases for bit-vectors of appropriate size for each of the key components, e.g., a 16-bit tag type `tag_t`, a 6-bit radix type `radix_t`, a 4-bit ID type `id_t`, etc. Moreover, we define utility functions for various parts of the existing ISA, bit-vector manipulation, and memory interface.

Signing Userspace Addresses. A key component of the proposed scheme is the signing of userspace addresses, with

Listing 1: Sail specification of useful signing utilities.

```
register key : key_t
val mac : (key_t, msg_t) -> tag_t

val sign : (uaddr_t, radix_t, id_t) -> tag_t
function sign(uaddr, radix, id) = {
  if radix == legacy_radix then
    return zeros(sizeof(tag_len));
  let mask : uaddr_t = ~(to_unary(radix));
  return mac(key, radix @ id @ (uaddr & mask));
}
```

Listing 2: Sail specification of the signing instruction.

```
val instr_sign : (rex_t, modrm_t, sib_t, disp_t) -> unit
function instr_sign(rex, modrm, sib, dis) = {
  let reg_id  : reg_t = decode_regid(rex, modrm);
  let src_val : u64_t = decode_src(rex, modrm, sib, dis);
  let _ @ uaddr : uaddr_t = gprs(reg_id);
  let _ @ radix : radix_t = src_val[ 7 .. 0];
  let _ @ id    : id_t    = src_val[15 .. 8];
  let tag = sign(uaddr, radix, id);
  gprs(reg_id) = tag @ 0b0 @ uaddr;
}
```

Listing 3: Sail specification of the additional checks performed before each memory access.

```
val new_check_memop : (vaddr_t, data_size_t) -> status_t
function new_check_memop(vaddr, len) = {
  let tag : tag_t @ raddr : raddr_t = vaddr;
  if raddr[47] == bitone then
    return check_memop(vaddr, len);

  let vpnum : vpnum_t @ _ = raddr;
  let radix : radix_t = radices(vpnum);
  let id : id_t = ids(vpnum);
  let mask : uaddr_t = ~(to_unary(radix));

  let _ @ uaddr_f : uaddr_t = raddr;
  let uaddr_l = uaddr_f + zero_extend(size_mask(len));
  if (uaddr_f & mask) != (uaddr_l & mask) then
    return Error_radix_overflow(uaddr_f, radix, len);

  let comp_tag = sign(uaddr_f, radix, id);
  if comp_tag != tag then
    return Error_tag_wrong(comp_tag, tag);

  return check_memop(zero_extend(raddr), len);
}
```

their associated radix and ID, using a cryptographic MAC and a per-process secret key. Listing 1 gives a formalization of this procedure, as described in Section III-C. Here, we define a global register `key` of the appropriate type `key_t`, and declare a function `mac` that maps a key and message to a cryptographic tag. Note here that Sail allows uninterpreted functions within specifications, as long as typing constraints are fulfilled, and we use this to leave the details of the `mac` function up to the actual implementation. Finally, Listing 1 defines the actual `sign` function. First, the function handles the special case for legacy radices, producing a zero tag. Otherwise, the resulting tag is computed using the `mac` function, whose message input is the concatenation of the radix, ID, and the userspace address with its lowermost bits masked out according to the radix.

New Signing Instruction. While the proposed approach is ISA agnostic, we give a concrete example of how the signing could be implemented in an extension to the x86-64 ISA. Listing 2 shows the specification of a new x86-64 instruction for the computation of signatures. The instruction has two 64-bit operands; the first always being a register, and the second coming from either another register or memory. On x86-64, operands are decoded from a ModR/M byte, an optional REX instruction prefix, and in the case of memory operands, an optional SIB byte and optional address displacement of up to 32-bit [23]. These decoding details are abstracted away in Listing 2 through the `decode_regid` and `decode_src` utilities. The userspace address is extracted from the first operand, the radix and ID are extracted from the two lowermost bytes of the second operand, ignoring the unused uppermost bits. Finally, the tag is computed with the `sign` function, appended to the userspace address, and written back to the first operand. No flags are affected.

Memory Access Checks. Finally, we also formally specify the checks performed for each aligned memory access, as shown in Listing 3. The check consists of several phases. First, the provided virtual address is split into the `tag` and canonical address part `raddr`. In case of kernel addresses, the standard access check is performed. Next, the virtual page number `vpnum` is extracted from `raddr` and used for the TLB lookup, respectively page table walk, to retrieve the radix and ID. Furthermore, we check whether the whole memory access of length `len` is within the same radix region, and give an error otherwise. Finally, the tag is recomputed using the `sign` function and compared to the provided tag. If there is a mismatch, an error is produced, and otherwise, the standard access check is performed with the clean userspace address. Note here that non-canonical memory accesses now also produce the tag mismatch error.

B. Verifying Specification Properties

An interesting capability of the Sail language is the writing of *properties*, which are specially defined Sail functions that produce Boolean outputs, and can be turned into a logic formula. Afterwards, one can use off-the-shelf SMT solvers, such as Microsoft's Z3 [16], to prove the given functions always return true. We have implemented and checked several properties our specification should fulfill. We give a brief (informal) overview in the following:

- **Legacy compatibility:** Unprotected legacy pointers can access unprotected memory, same as before extension.
- **Tag consistency:** It is not possible to successfully access a canonical userspace address with the wrong tag.
- **Region isolation:** All wide memory accesses that overflow into an adjacent memory region produce an error.

During the development of the properties, Z3 routinely found counterexamples, e.g., whenever we were missing an assumption. After we refined the property definitions accordingly, Z3 was able to instantaneously prove that the properties hold. Moreover, we believe the given specification should be useful for verifying functional correctness of real-world hardware implementations later on [3], [11].

V. Implementation

In this section, we detail our prototype implementation, including a hardware model, allocator, and kernel patch.

A. Hardware Model

Our prototype is built on the open-source gem5 simulator [9], [37] (gem5 version 22.1.0.0), a widely used full-system simulator for the implementation and evaluation of hardware extensions [28], [32], [33], [38], [65]. The gem5 prototype features a functional and timing-accurate hardware model of our ISA extension, integrating the cryptographic integrity checks into the CPU's microarchitecture. Moreover, it is fully parameterizable regarding the MAC computation latencies. While our design is ISA agnostic, we implement our gem5 prototype specifically for the x86-64 architecture. Implementing our ISA extension requires a small set of modifications to the CPU microarchitecture. The system integration includes the following changes to our gem5 simulator:

First, we implement hardware address masking to enable a pointer tagging mechanism. The gem5 simulator utilizes *memory requests*, which are issued by a software module responsible for managing load and store operations. These requests provide a high-level abstraction for all memory operations of the underlying ISA. We modify the load and store unit to mask the address before generating the memory request. This allows us to propagate the upper bits of the pointer (*i.e.*, the pointer tag) as metadata associated with the memory request. The specific number of upper bits utilized for the hardware address masking depends on the virtual address mode. For instance, the 48-bit addressing mode allows us to use the 16 uppermost bits to encode the MAC. The hardware address masking propagates the MAC as metadata for every issued memory request.

Next, we perform the integrity checks (*i.e.*, hardware authentication) during the translation lookaside buffer (TLB) lookup for every memory operation accessing protected pages. The gem5 simulator performs the address translation for every memory request issued by the hardware model. The page table walker resolves the virtual-to-physical mapping and caches the corresponding page table entry (PTE) in the TLB. In addition, the memory management unit (MMU) performs access permission checks (e.g., read, write, execute) using the permission bits stored in the PTE. Our mechanism incorporates the radix and ID in the PTE, which can be set by the OS kernel using the syscall interface (`mprotect`-like system call). Similar to the other page permission checks, we integrate our cryptographic integrity checks into the MMU. As shown in Listing 3, we use a low-latency block cipher to recompute the MAC using parts of the address, radix, and ID (with a unique per-process secret key), and compare it against the MAC metadata propagated through the memory request. Additionally, the MMU ensures validity for the entire data width of the memory request. A MAC mismatch indicates a memory safety error and triggers an exception for the corresponding memory request. Access to protected pages is only granted for pointers with the corresponding access privileges. We reserve a special radix value to identify legacy memory pages, allowing legacy pointers to access the memory locations. The synchronous cryptographic integrity checks of our gem5 hardware model influence the L1 access latency since the MAC validation needs to be finished before processing the received data. Suitable cipher candidates for our ISA extension are BipBip [7], QARMA [4], QARMAv2 [5], and SPEEDY [34]. Depending on the block cipher in use and the processor's clock frequency, the cipher latency varies between 1–3 cycles. Our gem5 hardware model is fully parameterizable regarding cipher latencies, which is reflected in our performance evaluation.

Lastly, we extend the x86-64 instruction set to integrate the custom `Sign` instruction specified in Listing 2. Alternatively, one could also implement a software-based signing operation with the same functionality.

B. Memory Allocator

Our prototype implementation is based on the musl C standard library (musl-libc version 1.2.3). Leveraging our novel hardware feature, we modify the binning memory allocator to enforce our cryptographic least privilege isolation on all heap-allocated objects. As such, it is binary-compatible and can be linked to existing C applications. We instrument all `malloc`-related function calls to return and operate on signed pointers. When requesting a new page from the operating system, the allocator also sets the ID and radix through the `mprotect` system call. As each bucket contains only power-of-two slots, all pages within that bucket share that same radix. Our allocator conforms to the slot sizes by rounding up the allocated size to the next power-of-two. When handing out pointers to the userspace application, e.g., before returning from `malloc`, the allocator signs the returned pointer using the `Sign` instruction of our extended x86-64 ISA. Since the ID and radix are used for the signature, the allocator stores both in per-bucket metadata to avoid additional system calls during pointer signing. Moreover, to ensure temporal memory safety, the memory allocator quarantines freed objects within a page until all objects on that page are freed. Once all objects are freed, the allocator assigns a new ID for that page, thus invalidating any dangling pointers referencing that page.

C. Linux Kernel Patch

To enable the setting of ID and radix values in the PTEs, we extend the `mprotect` system call with additional arguments (similar to `pkey_mprotect`). Similar to other (PTE) permission bits, we store our page metadata in the `vm_area_struct` structure. We encode the 6-bit radix and 4-bit ID into the PTE, repurposing previously unused bits by the ISA (*i.e.*, x86-64 PTE [24]). We repurpose the four Intel MPK bits for our implementation. In addition, 9 bits (bits 58–52 and 10–9) are currently unused and ignored, which leaves a total of 3 bits for future use. Note that in practice, several different options are available. For instance, it is also possible to encode the ID and radix in the upper physical address bits. However, this would reduce the addressable physical memory.

VI. Security Analysis

This section provides a systematic analysis (*i.e.*, on memory errors) of our design and an empirical security evaluation.

A. Systematic Analysis

Our ISA extension enables scalable memory isolation with cryptographic strength. The prototype implementation uses the 48-bit virtual addressing mode, which allows for a 16-bit MAC. Note that using the 39-bit virtual addressing mode would allow us to encode a 25-bit MAC. For this analysis, we generalize the MAC size to M bits. Assuming a cryptographically secure MAC, a collision (*i.e.*, second preimage resistance) occurs with a probability of 2^{-M}. An attacker could also try to predict the correct MAC value to access data in an arbitrary memory location. This forgery attempt leads with a high probability $(1 - 2^{-M})$ to a hardware exception.

An attacker could attempt to forge a legacy pointer to bypass the cryptographic integrity checks. However, our design prohibits any memory access using legacy pointers to protected memory. Thus, it is not possible to bypass our protection mechanism by using legacy pointers.

In addition, our ISA extension achieves memory safety (*i.e.*, object-granular isolation) through the use of a binning memory allocator, which locates same-size objects on the same buckets (and pages). Moreover, every memory object is aligned to a power-of-two size represented by the radix. Thus, linear buffer overflows are prevented since overflowing from one radix into an adjacent one is detected by the memory access checks. Similarly, arbitrary memory safety violations are detected by the integrity checks, *i.e.*, MAC authentication.

A special case of spatial memory vulnerability is the sub-object memory violation, where data is corrupted within an object outside the intended internal boundaries (e.g., C structures). We exclude sub-object memory violations since our protection mechanism operates on an object granularity. Furthermore, recent studies show that sub-object violations only account for around 1 % of observed vulnerabilities [40].

We prevent use-after-free (UAF) errors through memory quarantining [2], [19]. UAF errors are caused by dangling pointers referencing already freed memory locations. Here, we need to distinguish two cases: UAF in the freed and reallocated state. UAF in the freed state defines that no object has been reallocated on the same chunk of memory, and the reallocated state defines that a new object is assigned to the previously freed chunk of memory. Moreover, dangling pointers to memory in the reallocated state can be exploited to modify the data of the new memory object. To counteract UAF errors, our memory allocator puts freed objects into quarantine and waits until all objects within the page are freed. Subsequently, we assign a new ID to the page, and the page can be reused. Thus, memory accesses using a dangling pointer (UAF in the reallocated state) are mitigated by the integrity checks since the IDs mismatch. We can reuse every page until all unique IDs are used and afterwards return the page to the OS or scan the memory for potential dangling pointers [8]. Note that our design cannot detect UAF in the freed state due to this imprecision; however, protecting the reallocated state is typically the crucial property of UAF.

Similarly, double-free errors are detected. In the freed state, a double-free is detected since the memory chunk is in quarantine. In the reallocated state, a double-free is detected by the integrity check of an access before freeing the object.

Moreover, uninitialized use of memory is also classified as a temporal safety vulnerability. While this threat is not directly addressed by our design, it can be relatively easily mitigated by enforcing memory initialization (pool zeroing) before usage.

TABLE I: The results of the empirical security evaluation for CWE-122 and CWE-416 of the NIST Juliet test suite.

CWE	Description	Number of Test Cases	Passed
CWE-122	Heap buffer overrun	1368 Test Cases	100 %
CWE-416	Use-after-free (UAF)	138 Test Cases	100 %

B. Empirical Security Evaluation

We use the Juliet [12] test suite (Juliet version 1.3) for our empirical security evaluation to demonstrate the efficacy of our design. Juliet provides test cases for common types of memory safety errors, categorized as common weakness enumerations (CWEs) [41], divided into good and bad types for each test case. In particular, we focus on the subset of C benchmarks for CWE-122 *heap buffer overrun* and CWE-416 *use-after-free* of the Juliet test suite. Note that Juliet was initially designed for testing static analysis tools, not for runtime testing. As a result, some test cases may require specific input stimuli, cause the test program to crash, or not necessarily trigger an actual memory safety violation. In order to reasonably evaluate Juliet's benchmarks, we use the Address Sanitizer (ASan) [56] to identify the subset of benchmarks that trigger detectable memory safety errors. Additionally, we exclude sub-object memory violations as they are not covered in our prototype implementation.

Furthermore, our design requires minor adjustments for some of the Juliet benchmarks. For instance, some test cases of CWE-122 need to be adapted to trigger an actual heap overrun since our memory allocator aligns all memory objects to bucket sizes. With these adaptations, an overflow that would originally corrupt padded memory locations (which are not security-critical) will now corrupt the memory of an adjacent or non-adjacent object. Similarly, CWE-416 needs to be adapted as some of these benchmarks perform the UAF access in the freed state. We instrument the benchmark to perform the UAF error in the reallocated state (the crucial UAF detection property). Table I highlights the results of the empirical security evaluation using CWE-122 and CWE-416 of the NIST Juliet test suite. We define a test case as successful if the cryptographic integrity checks of our ISA extension trigger a hardware exception. In summary, our scheme successfully detects all memory safety errors for the subset of CWE-122 (heap buffer overrun) and CWE-416 (use-after-free) test cases identified by ASan.

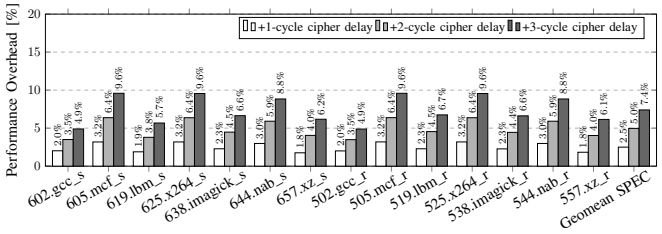

Fig. 3: The simulated relative runtime overhead of heap memory isolation using different cipher latencies, ranging from 1–3 cycles, for the SPEC CPU2017 benchmark suite.

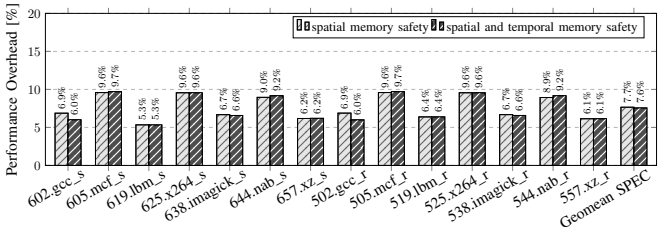

Fig. 4: The simulated relative runtime overhead of heap memory safety through single object isolation with a 3-cycle latency for the SPEC CPU2017 benchmark suite.

VII. Evaluation

This section evaluates the system performance of our prototype and discusses the area and alignment overhead.

A. Performance Evaluation

The evaluation of our hardware model uses a subset of C programs of the SPEC CPU2017 [13] benchmark suite, compiled using our modified musl-libc with `-O3` optimization level. All benchmarks are executed in full-system mode running Linux (kernel 5.15.67) with our patch applied.

Configuration. For our performance evaluation, we use the following gem5 configuration. We use the TimingSimpleCPU model with a clock frequency of 3 GHz. We instantiate a private 8-way set associative cache with a 16 kB L1 instruction cache, and a 64 kB L1 data cache with 64 B cache lines. In addition, we instantiate a 16-way set associative shared L2 cache with a size of 8 MB, acting as a last-level cache (LLC). We configure the cache access latencies using parameters derived from recent Intel processors, *i.e.*, the L1 cache with 5 cycles and the L2 cache with 17-cycle latency. These parameters are consistent with existing work on system simulation [28], [35], [65]. The main memory of our system consists of an 8 GB 2400 MHz DDR4 DRAM.

Simulation Results. We use the gem5 simulator to measure the execution time of the workloads. The baseline is measured using an unmodified gem5 model executing the workloads compiled with the uninstrumented musl-libc. Note that the perlbench benchmark is excluded due to toolchain issues with the musl-libc. Figure 3 shows the simulated relative runtime overhead of heap memory isolation (*i.e.*, domain isolation) using cipher latencies ranging from 1 to 3 cycles. Depending on the chosen block cipher, the cryptographic integrity checks increase the L1 access latency by single or multiple cycles to verify the MAC. Table II illustrates our chosen cipher candidates and their latency overheads. We report an average overhead of 2.5–7.4 % for heap isolation, depending on the chosen low-latency block cipher, for SPEC CPU2017.

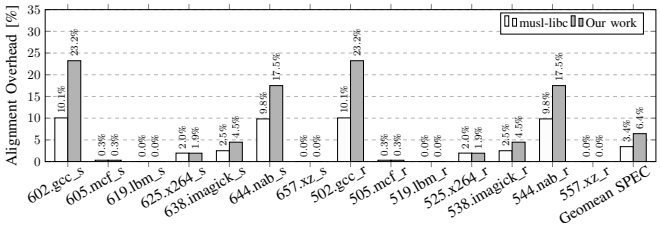

Fig. 5: The alignment overhead of our design for object isolation for the SPEC CPU2017 benchmark suite.

TABLE II: Latency and area overhead of cipher candidates.

Cipher	Latency‡ ps	Cycles 3 GHz	Cycles 4 GHz	Cycles 5 GHz	Area‡ µm²	Area‡ GE
`BIPBIP-DEC` [7]	327	1	2	2	6554	33843
`QARMAv1-64` ($r = 5$) [4]	354	2	2	2	2771	14091
`QARMAv1-64` ($r = 7$) [4]	513	2	3	3	3863	19649
`QARMAv2-64` ($r = 4$) [5]	305	1	2	2	2381	12110
`QARMAv2-64` ($r = 7$) [5]	497	2	2	3	3754	19095
`QARMAv2-64` ($r = 9$) [5]	618	2	3	4	4738	24100

‡ Numbers are taken from QARMAv2 [5] for the Nangate 15 nm process.

In addition, Figure 4 shows the simulated relative runtime overhead of heap memory safety through single object isolation (*i.e.*, spatial and temporal safety) with a 3-cycle latency for SPEC CPU2017. Compared to domain isolation, enforcing memory safety with our design incurs a small additional overhead due to alignment and quarantining. Our evaluation shows an average overhead of 7.7 % for spatial safety and 7.6 % for spatial and temporal safety of heap objects.

B. Area and Alignment Overhead

Our design introduces an area overhead due to the block cipher required for the MAC computation. We approximate this overhead with a single cipher instance per CPU core. Table II provides an overview of potential cipher candidates. Note that additional instances might be required depending on the cache architecture and block cipher implementation.

Our work isolates individual heap objects by aligning and padding allocations, which creates a memory overhead. Figure 5 shows the alignment overhead of our design compared to the musl-libc for the SPEC CPU2017 benchmark suite. Specifically, sub-page granular objects are aligned to radix-sized slots. Note that this alignment overhead is mostly negligible compared to the overhead of larger allocations. Importantly, for allocation larger than a page, we only need to align the memory to the next page granularity since we can isolate pages using different IDs.

VIII. Related Work

Hardware-Assisted Bounds-Checking. Several research proposals have introduced bounds-checking mechanisms, initially established using software-based approaches (e.g., CCured [47], Cyclone [26], or SoftBound [45]), and have now been efficiently integrated into hardware. Hardware-assisted mechanisms incorporate additional support for bounds checks into the processor architecture (where the bounds metadata is either stored inline, adjacent, or disjoint) [17], [31], [44], [48], [51], [74], [75]. In contrast, our design achieves scalable memory protection through cryptographic integrity checks.

Capability Architectures. Capability-based systems like the M-Machine [14], CHEx86 [59], and CHERI [69], [70], [71] utilize capability-based addressing to enforce the principle of least privilege on the architectural level. CHERI leverages a single-bit memory tag to protect capabilities in memory. Moreover, CHERI integrates capability register into the processor architecture to enforce spatial memory safety by checking the object's bounds on memory interactions. In addition, Cornucopia [19], [20] provides temporal memory safety for CHERI heaps. However, CHERI's 128-bit capabilities can cause incompatibilities with legacy software. Furthermore, CHERI extensively reshapes the processor architecture, which can be hard to integrate into complex CPU designs. Contrarily, we enforce least privilege to secure legacy computing systems by only minimally increasing microarchitectural complexity.

Tagged Memory Architectures. Memory tagging [25], [58], [66], either implemented in software or hardware, employs a *lock-and-key* mechanism to enforce logical integrity checks. ISA extensions, like the ARM memory tagging extension (MTE) [57] and SPARC application data integrity (ADI) [1], [58], integrate tagged memory into the system architecture, propagating the memory tags through the memory hierarchy. However, ARM MTE, which uses four tag bits per 16 B, and SPARC ADI, using four tag bits per 64 B, yield a high collision probability of 6.25 %, potentially leading to undetected memory safety violations. Furthermore, memory tagging has the disadvantage that its logical integrity scales poorly in terms of security. An increased tag size incurs an unmanageable high memory overhead, making larger tag sizes impractical to implement. Moreover, conventional tagged architectures rely on additional DRAM requests (issued by the memory controller) to receive the tag metadata from the main memory, increasing DRAM pressure and leading to a non-negligible performance overhead. Unlike logical integrity, our cryptographic integrity elegantly scales in security and isolation granularity while only relying on PTE metadata.

Cryptographic Memory Safety. Cryptographic primitives such as authentication and encryption are already established for the enforcement of cryptographic pointer integrity [15], [39] and are recently adapted for memory safety [53], [54]. Cryptographically sealed pointers [65] and CrypTag [46], for example, enforce spatial and temporal memory safety using a cryptographic lock-and-key approach. While cryptographically sealed pointers detect memory safety violations using a cryptographic MAC in combination with efficiently scaled memory tagging, CrypTag [46] applies cryptographically tagged memory by leveraging a memory encryption engine to implicitly encode the memory tags into the MAC of the authenticated encryption. Both countermeasures rely on additional DRAM fetches to propagate (sub-) cache line-granular metadata similar to tagged memory (incurring comparable performance overheads) to apply integrity protection. Cryptographic capability computing (C^3) [35] partially encrypts the pointer and uses the encrypted part, called cryptographic address (CA), as nonce for a keystream generator. Memory accesses enforce CA decryption followed by data encryption or decryption utilizing the keystream. This way, C^3 can either generate exceptions due to corrupted addresses or result in access to garbled data. The detection of memory safety errors typically requires additional metadata for integrity checks. We achieve fine-grain memory isolation by solely relying on PTE metadata (instead of tagged memory) that does not increase the DRAM pressure.

Memory Protection Keys. Intel memory protection keys (MPK) [49] implement access policies based on logical integrity that are controlled by the user space protection key register (PKRU). The protection keys are stored inside the page table entries (PTEs) and checked in the MMU during the page translation procedure, *i.e.*, by comparing whether the protection key matches the current policies of the PKRU register. For instance, ERIM [67], Hodor [22], and Donky [55] use MPK to enforce in-process domain isolation. Note that MPK supports page-granular access policies for 16 protection domains, which can be limiting for modern software systems that may require thousands of separate execution contexts.

IX. Conclusion

In this paper, we presented cryptographic least privilege enforcement (CLPE), a scalable isolation mechanism enforcing the principle of least privilege with cryptographic strength. The ISA extension detects memory safety errors by incorporating cryptographic integrity checks based on message authentication codes (MACs). Specifically, our design effectively scales through cryptography, allowing the isolation of individual objects (*i.e.*, memory safety) and the isolation of protection domains. We provide a formal model of our ISA extension, along with a hardware model that enables functional and timing-accurate simulation. Moreover, our mechanism only requires lightweight architectural changes while preserving legacy code compatibility. The simulated performance overhead of our hardware model demonstrates the efficiency of our design, reporting an average overhead of 2.5–7.4 % for the subset of C programs of the SPEC CPU2017 benchmark suite.

Acknowledgment

We thank the anonymous reviewers for their valuable feedback that improved this work. This project has received funding from the Austrian Research Promotion Agency (FFG) via the AWARE project (FFG grant number 891092) and the SEIZE project (FFG grant number 888087). Additional funding was provided by a generous gift from Intel.

References

[1] Kathirgamar Aingaran, Sumti Jairath, Georgios K. Konstadinidis, Serena Leung, Paul Loewenstein, Curtis McAllister, Stephen Phillips, Zoran Radovic, Ram Sivaramakrishnan, David Smentek, and Thomas Wicki. M7: Oracle's Next-Generation Sparc Processor. *IEEE Micro*, 35:36–45, 2015.

[2] Sam Ainsworth and Timothy M. Jones. MarkUs: Drop-in use-after-free prevention for low-level languages. In *S&P*, 2020.

[3] Alasdair Armstrong, Thomas Bauereiss, Brian Campbell, Alastair Reid, Kathryn E. Gray, Robert M. Norton, Prashanth Mundkur, Mark Wassell, Jon French, Christopher Pulte, Shaked Flur, Ian Stark, Neel Krishnaswami, and Peter Sewell. ISA Semantics for ARMv8-A, RISC-V, and CHERI-MIPS. *ACM on Programming Languages*, 3:71:1–71:31, 2019.

[4] Roberto Avanzi. The QARMA Block Cipher Family. Almost MDS Matrices Over Rings With Zero Divisors, Nearly Symmetric Even-Mansour Constructions With Non-Involutory Central Rounds, and Search Heuristics for Low-Latency S-Boxes. *IACR Transactions on Symmetric Cryptology*, 2017:4–44, 2017.

[5] Roberto Avanzi, Subhadeep Banik, Orr Dunkelman, Maria Eichlseder, Shibam Ghosh, Marcel Nageler, and Francesco Regazzoni. The QARMAv2 Family of Tweakable Block Ciphers. *IACR Transactions on Symmetric Cryptology*, 2023:25–73, 2023.

[6] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. The SMT-LIB Standard: Version 2.6. Technical report, Department of Computer Science, The University of Iowa, 2017. Available at `www.SMT-LIB.org`.

[7] Yanis Belkheyar, Joan Daemen, Christoph Dobraunig, Santosh Ghosh, and Shahram Rasoolzadeh. BipBip: A Low-Latency Tweakable Block Cipher with Small Dimensions. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2023:326–368, 2023.

[8] Anton Bikineev, Michael Lippautz, and Hannes Payer. Retrofitting Temporal Memory Safety on C++. https://security.googleblog.com/2022/05/retrofitting-temporal-memory-safety-on-c.html, 2022. Accessed: 2023-07-26.

[9] Nathan L. Binkert, Bradford M. Beckmann, Gabriel Black, Steven K. Reinhardt, Ali G. Saidi, Arkaprava Basu, Joel Hestness, Derek Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib Bin Altaf, Nilay Vaish, Mark D. Hill, and David A. Wood. The gem5 Simulator. *SIGARCH Computer Architecture News*, 39:1–7, 2011.

[10] William Blair, William K. Robertson, and Manuel Egele. ThreadLock: Native Principal Isolation Through Memory Protection Keys. In *ASIACCS*, 2023.

[11] Roderick Bloem, Barbara Gigerl, Marc Gourjon, Vedad Hadzic, Stefan Mangard, and Robert Primas. Power Contracts: Provably Complete Power Leakage Models for Processors. In *CCS*, 2022.

[12] Tim Boland and Paul E. Black. Juliet 1.1 C/C++ and Java Test Suite. *Computer*, 45:88–90, 2012.

[13] James Bucek, Klaus-Dieter Lange, and Jóakim von Kistowski. SPEC CPU2017: Next-Generation Compute Benchmark. In *ICPE*, 2018.

[14] Nicholas P. Carter, Stephen W. Keckler, and William J. Dally. Hardware Support for Fast Capability-based Addressing. In *ASPLOS*, 1994.

[15] Crispin Cowan, Steve Beattie, John Johansen, and Perry Wagle. PointGuard™: Protecting Pointers from Buffer Overflow Vulnerabilities. In *USENIX Security*, 2003.

[16] Leonardo Mendonça de Moura and Nikolaj S. Bjørner. Z3: An Efficient SMT Solver. In *TACAS*, 2008.

[17] Joe Devietti, Colin Blundell, Milo M. K. Martin, and Steve Zdancewic. HardBound: Architectural Support for Spatial Safety of the C Programming Language. In *ASPLOS*, 2008.

[18] Ali Fakhrzadehgan, Yale N. Patt, Prashant J. Nair, and Moinuddin K. Qureshi. SafeGuard: Reducing the Security Risk from Row-Hammer via Low-Cost Integrity Protection. In *HPCA*, 2022.

[19] Nathaniel Wesley Filardo, Brett F. Gutstein, Jonathan Woodruff, Sam Ainsworth, Lucian Paul-Trifu, Brooks Davis, Hongyan Xia, Edward Tomasz Napierala, Alexander Richardson, John Baldwin, David Chisnall, Jessica Clarke, Khilan Gudka, Alexandre Joannou, A. Theodore Markettos, Alfredo Mazzinghi, Robert M. Norton, Michael Roe, Peter Sewell, Stacey D. Son, Timothy M. Jones, Simon W. Moore, Peter G. Neumann, and Robert N. M. Watson. Cornucopia: Temporal Safety for CHERI Heaps. In *S&P*, 2020.

[20] Nathaniel Wesley Filardo, Brett F. Gutstein, Jonathan Woodruff, Jessica Clarke, Peter Rugg, Brooks Davis, Mark Johnston, Robert M. Norton, David Chisnall, Simon W. Moore, Peter G. Neumann, and Robert N. M. Watson. Cornucopia Reloaded: Load Barriers for CHERI Heap Temporal Safety. In *ASPLOS*, 2024.

[21] Tommaso Frassetto, Patrick Jauernig, Christopher Liebchen, and Ahmad-Reza Sadeghi. IMIX: In-Process Memory Isolation EXtension. In *USENIX Security*, 2018.

[22] Mohammad Hedayati, Spyridoula Gravani, Ethan Johnson, John Criswell, Michael L. Scott, Kai Shen, and Mike Marty. Hodor: Intra-Process Isolation for High-Throughput Data Plane Libraries. In *USENIX ATC*, 2019.

[23] Intel. Intel 64 and IA-32 Architectures Software Developer's Manual Volume 2A: Instruction Set Reference, A-L. https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-vol-2a-manual.pdf, 2023. Accessed: 2023-02-26.

[24] Intel. Intel 64 and IA-32 Architectures Software Developer's Manual Volume 3A: System Programming Guide, Part 1. https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-vol-3a-part-1-manual.pdf, 2023. Accessed: 2023-02-26.

[25] Samuel Jero, Nathan Burow, Bryan C. Ward, Richard Skowyra, Roger Khazan, Howard E. Shrobe, and Hamed Okhravi. TAG: Tagged Architecture Guide. *ACM Computing Surveys*, 55:124:1–124:34, 2023.

[26] Trevor Jim, J. Gregory Morrisett, Dan Grossman, Michael W. Hicks, James Cheney, and Yanling Wang. Cyclone: A Safe Dialect of C. In *USENIX ATC*, 2002.

[27] Jonas Juffinger, Lukas Lamster, Andreas Kogler, Maria Eichlseder, Moritz Lipp, and Daniel Gruss. CSI:Rowhammer - Cryptographic Security and Integrity against Rowhammer. In *S&P*, 2023.

[28] Yonghae Kim, Jaekyu Lee, and Hyesoon Kim. Hardware-based Always-On Heap Memory Safety. In *MICRO*, 2020.

[29] Yoongu Kim, Ross Daly, Jeremie S. Kim, Chris Fallin, Ji-Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu. Flipping Bits in Memory Without Accessing Them: An Experimental Study of DRAM Disturbance Errors. In *ISCA*, 2014.

[30] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre Attacks: Exploiting Speculative Execution. In *S&P*, 2019.

[31] Albert Kwon, Udit Dhawan, Jonathan M. Smith, Thomas F. Knight Jr., and André DeHon. Low-Fat Pointers: Compact Encoding and Efficient Gate-Level Implementation of Fat Pointers for Spatial Safety and Capability-based Security. In *CCS*, 2013.

[32] Lukas Lamster, Martin Unterguggenberger, David Schrammel, and Stefan Mangard. HashTag: Hash-based Integrity Protection for Tagged Architectures. In *USENIX Security*, 2023.

[33] Lukas Lamster, Martin Unterguggenberger, David Schrammel, and Stefan Mangard. Voodoo: Memory Tagging, Authenticated Encryption, and Error Correction through MAGIC. In *USENIX Security*, 2024.

[34] Gregor Leander, Thorben Moos, Amir Moradi, and Shahram Rasoolzadeh. The SPEEDY Family of Block Ciphers Engineering an Ultra Low-Latency Cipher from Gate Level for Secure Processor Architectures. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2021:510–545, 2021.

[35] Michael LeMay, Joydeep Rakshit, Sergej Deutsch, David M. Durham, Santosh Ghosh, Anant Nori, Jayesh Gaur, Andrew Weiler, Salmin Sultana, Karanvir Grewal, and Sreenivas Subramoney. Cryptographic Capability Computing. In *MICRO*, 2021.

[36] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown: Reading Kernel Memory from User Space. In *USENIX Security*, 2018.

[37] Jason Lowe-Power, Abdul Mutaal Ahmad, Ayaz Akram, Mohammad Alian, Rico Amslinger, Matteo Andreozzi, Adrià Armejach, Nils Asmussen, Srikant Bharadwaj, Gabe Black, Gedare Bloom, Bobby R. Bruce, Daniel Rodrigues Carvalho, Jerónimo Castrillón, Lizhong Chen, Nicolas Derumigny, Stephan Diestelhorst, Wendy Elsasser, Marjan Fariborz, Amin Farmahini Farahani, Pouya Fotouhi, Ryan Gambord, Jayneel Gandhi, Dibakar Gope, Thomas Grass, Bagus Hanindhito, Andreas Hansson, Swapnil Haria, Austin Harris, Timothy Hayes, Adrian Herrera, Matthew Horsnell, Syed Ali Raza Jafri, Radhika Jagtap, Hanhwi Jang, Reiley Jeyapaul, Timothy M. Jones, Matthias Jung, Subash Kannoth,

Hamidreza Khaleghzadeh, Yuetsu Kodama, Tushar Krishna, Tommaso Marinelli, Christian Menard, Andrea Mondelli, Tiago Mück, Omar Naji, Krishnendra Nathella, Hoa Nguyen, Nikos Nikoleris, Lena E. Olson, Marc S. Orr, Binh Pham, Pablo Prieto, Trivikram Reddy, Alec Roelke, Mahyar Samani, Andreas Sandberg, Javier Setoain, Boris Shingarov, Matthew D. Sinclair, Tuan Ta, Rahul Thakur, Giacomo Travaglini, Michael Upton, Nilay Vaish, Ilias Vougioukas, Zhengrong Wang, Norbert Wehn, Christian Weis, David A. Wood, Hongil Yoon, and Éder F. Zulian. The gem5 Simulator: Version 20.0+. *CoRR*, abs/2007.03152, 2020.

[38] Evgeny Manzhosov, Adam Hastings, Meghna Pancholi, Ryan Piersma, Mohamed Tarek Ibn Ziad, and Simha Sethumadhavan. Revisiting Residue Codes for Modern Memories. In *MICRO*, 2022.

[39] Ali José Mashtizadeh, Andrea Bittau, Dan Boneh, and David Mazières. CCFI: Cryptographically Enforced Control Flow Integrity. In *CCS*, 2015.

[40] Matt Miller. Trends, challenges, and strategic shifts in the software vulnerability mitigation landscape. https://github.com/Microsoft/MSRC-Security-Research/blob/master/presentations/2019_02_BlueHatIL/2019_01%20-%20BlueHatIL%20-%20Trends%2C%20challenge%2C%20and%20shifts%20in%20software%20vulnerability%20mitigation.pdf, 2019. Accessed: 2023-02-26.

[41] MITRE. Common Weakness Enumeration. https://cwe.mitre.org/, 2006-2023. Accessed: 2023-02-26.

[42] Prashanth Mundkur, Rishiyur S. Nikhil, Jon French, Brian Campbell, Robert Norton-Wright, Alasdair Armstrong, Thomas Bauereiss, Shaked Flur, Christopher Pulte, Peter Sewell, Alexander Richardson, Hesham Almatary, Jessica Clarke, Nathaniel Wesley Filardo, Peter Rugg, and Scott Johnson. RISC-V Sail Model, 2019. Accessed: 2023-12-05.

[43] Kit Murdock, David F. Oswald, Flavio D. Garcia, Jo Van Bulck, Daniel Gruss, and Frank Piessens. Plundervolt: Software-based Fault Injection Attacks against Intel SGX. In *S&P*, 2020.

[44] Santosh Nagarakatte, Milo M. K. Martin, and Steve Zdancewic. Watchdog: Hardware for Safe and Secure Manual Memory Management and Full Memory Safety. In *ISCA*, 2012.

[45] Santosh Nagarakatte, Jianzhou Zhao, Milo M. K. Martin, and Steve Zdancewic. SoftBound: Highly Compatible and Complete Spatial Memory Safety for C. In *PLDI*, 2009.

[46] Pascal Nasahl, Robert Schilling, Mario Werner, Jan Hoogerbrugge, Marcel Medwed, and Stefan Mangard. CrypTag: Thwarting Physical and Logical Memory Vulnerabilities using Cryptographically Colored Memory. In *ASIACCS*, 2021.

[47] George C. Necula, Scott McPeak, and Westley Weimer. CCured: Type-Safe Retrofitting of Legacy Code. In *POPL*, 2002.

[48] Oleksii Oleksenko, Dmitrii Kuvaiskii, Pramod Bhatotia, Pascal Felber, and Christof Fetzer. Intel MPX Explained: A Cross-layer Analysis of the Intel MPX System Stack. In *SIGMETRICS*, 2018.

[49] Soyeon Park, Sangho Lee, and Taesoo Kim. Memory Protection Keys: Facts, Key Extension Perspectives, and Discussions. *IEEE Security & Privacy*, 21:8–15, 2023.

[50] Qualcomm. Pointer Authentication on ARMv8.3. https://www.qualcomm.com/content/dam/qcomm-martech/dm-assets/documents/pointer-auth-v7.pdf, 2017. Accessed: 2023-02-26.

[51] Gururaj Saileshwar, Rick Boivie, Tong Chen, Benjamin Segal, and Alper Buyuktosunoglu. HeapCheck: Low-cost Hardware Support for Memory Safety. *ACM Transactions on Architecture and Code Optimization*, 19:10:1–10:24, 2022.

[52] Jerome H. Saltzer and Michael D. Schroeder. The Protection of Information in Computer Systems. *Proceedings of the IEEE*, 63:1278–1308, 1975.

[53] David Schrammel, Salmin Sultana, Karanvir Grewal, Michael LeMay, David M. Durham, Martin Unterguggenberger, Pascal Nasahl, and Stefan Mangard. MEMES: Memory Encryption-Based Memory Safety on Commodity Hardware. In *SECRYPT*, 2023.

[54] David Schrammel, Martin Unterguggenberger, Lukas Lamster, Salmin Sultana, Karanvir Grewal, Michael LeMay, David M. Durham, and Stefan Mangard. Memory Tagging using Cryptographic Integrity on Commodity x86 CPUs. In *EuroS&P*, 2024.

[55] David Schrammel, Samuel Weiser, Stefan Steinegger, Martin Schwarzl, Michael Schwarz, Stefan Mangard, and Daniel Gruss. Donky: Domain Keys - Efficient In-Process Isolation for RISC-V and x86. In *USENIX Security*, 2020.

[56] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. AddressSanitizer: A Fast Address Sanity Checker. In *USENIX ATC*, 2012.

[57] Kostya Serebryany. ARM Memory Tagging Extension and How It Improves C/C++ Memory Safety. *login Usenix Magazine*, 2019.

[58] Kostya Serebryany, Evgenii Stepanov, Aleksey Shlyapnikov, Vlad Tsyrklevich, and Dmitriy Vyukov. Memory Tagging and how it improves C/C++ memory safety. Technical report, Google Security Engineering, 2018.

[59] Rasool Sharifi and Ashish Venkat. CHEx86: Context-Sensitive Enforcement of Memory Safety via Microcode-Enabled Capabilities. In *ISCA*, 2020.

[60] Stefan Steinegger, David Schrammel, Samuel Weiser, Pascal Nasahl, and Stefan Mangard. SERVAS! Secure Enclaves via RISC-V Authenticryption Shield. In *ESORICS*, 2021.

[61] Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Song. SoK: Eternal War in Memory. In *S&P*, 2013.

[62] Gang Tan. Principles and Implementation Techniques of Software-Based Fault Isolation. *Foundations and Trends in Privacy and Secruity*, 1(3):137–198, 2017.

[63] Adrian Taylor, Andrew Whalley, Dana Jansens, and Nasko Oskov. An update on Memory Safety in Chrome. https://security.googleblog.com/2021/09/an-update-on-memory-safety-in-chrome.html, 2021. Accessed: 2023-02-26.

[64] Martin Unterguggenberger, Lukas Lamster, David Schrammel, Martin Schwarzl, and Stefan Mangard. TME-Box: Scalable In-Process Isolation through Intel TME-MK Memory Encryption. In *NDSS*, 2025.

[65] Martin Unterguggenberger, David Schrammel, Lukas Lamster, Pascal Nasahl, and Stefan Mangard. Cryptographically Enforced Memory Safety. In *CCS*, 2023.

[66] Martin Unterguggenberger, David Schrammel, Pascal Nasahl, Robert Schilling, Lukas Lamster, and Stefan Mangard. Multi-Tag: A Hardware-Software Co-Design for Memory Safety based on Multi-Granular Memory Tagging. In *ASIACCS*, 2023.

[67] Anjo Vahldiek-Oberwagner, Eslam Elnikety, Nuno O. Duarte, Michael Sammler, Peter Druschel, and Deepak Garg. ERIM: Secure, Efficient In-process Isolation with Protection Keys (MPK). In *USENIX Security*, 2019.

[68] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. Efficient Software-Based Fault Isolation. In *SOSP*, 1993.

[69] Robert N. M. Watson, Peter G. Neumann, Jonathan Woodruff, Michael Roe, Hesham Almatary, Jonathan Anderson, John Baldwin, Graeme Barnes, David Chisnall, Jessica Clarke, Brooks Davis, Lee Eisen, Nathaniel Wesley Filardo, Franz A. Fuchs, Richard Grisenthwaite, Alexandre Joannou, Ben Laurie, A. Theodore Markettos, Simon W. Moore, Steven J. Murdoch, Kyndylan Nienhuis, Robert Norton, Alexander Richardson, Peter Rugg, Peter Sewell, Stacey Son, and Hongyan Xia. Capability Hardware Enhanced RISC Instructions: CHERI Instruction-Set Architecture (Version 9). https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-987.pdf, 2023. Accessed: 2023-10-26.

[70] Robert N. M. Watson, Jonathan Woodruff, Peter G. Neumann, Simon W. Moore, Jonathan Anderson, David Chisnall, Nirav H. Dave, Brooks Davis, Khilan Gudka, Ben Laurie, Steven J. Murdoch, Robert M. Norton, Michael Roe, Stacey D. Son, and Munraj Vadera. CHERI: A Hybrid Capability-System Architecture for Scalable Software Compartmentalization. In *S&P*, 2015.

[71] Jonathan Woodruff, Robert N. M. Watson, David Chisnall, Simon W. Moore, Jonathan Anderson, Brooks Davis, Ben Laurie, Peter G. Neumann, Robert M. Norton, and Michael Roe. The CHERI capability model: Revisiting RISC in an age of risk. In *ISCA*, 2014.

[72] Hongyan Xia, Jonathan Woodruff, Sam Ainsworth, Nathaniel Wesley Filardo, Michael Roe, Alexander Richardson, Peter Rugg, Peter G. Neumann, Simon W. Moore, Robert N. M. Watson, and Timothy M. Jones. CHERIvoke: Characterising Pointer Revocation using CHERI Capabilities for Temporal Memory Safety. In *MICRO*, 2019.

[73] Mengyao Xie, Chenggang Wu, Yinqian Zhang, Jiali Xu, Yuanming Lai, Yan Kang, Wei Wang, and Zhe Wang. CETIS: Retrofitting Intel CET for Generic and Efficient Intra-process Memory Isolation. In *CCS*, 2022.

[74] Shengjie Xu, Wei Huang, and David Lie. In-Fat Pointer: Hardware-Assisted Tagged-Pointer Spatial Memory Safety Defense with Subobject Granularity Protection. In *ASPLOS*, 2021.

[75] Mohamed Tarek Ibn Ziad, Miguel A. Arroyo, Evgeny Manzhosov, Ryan Piersma, and Simha Sethumadhavan. No-FAT: Architectural Support for Low Overhead Memory Safety Checks. In *ISCA*, 2021.