Cryptographic Capability Computing

Michael LeMay
Intel Labs
USA

Joydeep Rakshit
Intel Labs
India

Sergej Deutsch
Intel Labs
USA

David M. Durham
Intel Labs
USA

Santosh Ghosh
Intel Labs
USA

Anant Nori
Intel Labs
India

Jayesh Gaur
Intel Labs
India

Andrew Weiler
Intel Labs
USA

Salmin Sultana
Intel Labs
USA

Karanvir Grewal
Intel Labs
USA

Sreenivas Subramoney
Intel Labs
India

ABSTRACT

Capability architectures for memory safety have traditionally required expanding pointers and radically changing microarchitectural structures throughout processors, while only providing superficial hardening. We hence propose Cryptographic Capability Computing (C^3) - the first memory safety mechanism that is stateless to avoid requiring extra metadata storage. C^3 retains 64-bit pointer sizes providing legacy binary compatibility while imposing minimal touchpoints. Pointers are encrypted to *unforgeably* (within cryptographic bounds) reference each object. Data is encrypted even in caches and entangled with pointers for both spatial and temporal object-granular protection. Pointers become like unique keys for each allocation. C^3 deploys a novel form of prediction for address translation that mitigates performance overheads even when addresses are partially encrypted. Use of a low-latency, low-area cipher from the NIST Lightweight Cryptography project avoids delaying loads by readying a data keystream by the time data is returned from the L1 cache. C^3 is compatible with legacy binaries. Simulated performance overhead on SPEC CPU2006 is negligible with no memory overhead, which is a big leap forward compared to the overheads imposed by past memory safety approaches. C^3 effectively replaces inefficient metadata with efficient cryptography.

CCS CONCEPTS

• Security and privacy → Security in hardware; **• Computer systems organization → Architectures**; **• Software and its engineering → Software safety**.

KEYWORDS

memory safety, capabilities, memory encryption

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
MICRO'21, October 18–22, 2021, Athens, Greece
© 2021 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-8557-2/21/10.
https://doi.org/10.1145/3466752.3480076

ACM Reference Format:
Michael LeMay, Joydeep Rakshit, Sergej Deutsch, David M. Durham, Santosh Ghosh, Anant Nori, Jayesh Gaur, Andrew Weiler, Salmin Sultana, Karanvir Grewal, and Sreenivas Subramoney. 2021. Cryptographic Capability Computing. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '21), October 18–22, 2021, Virtual Event, Greece.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3466752.3480076

1 INTRODUCTION

Memory safety vulnerabilities have afflicted computer systems for decades, accounting for ~70% of vulnerabilities [37], and the loosely-typed programming languages that give rise to them remain popular. Spatial memory safety vulnerabilities include buffer overflows to adjacent and non-adjacent allocations, or to fields within a single allocation. Use-After-Free (UAF) and uninitialized use are common examples of temporal memory safety vulnerabilities.

Many memory safety solutions have been proposed in the past, but they either suffer from high overheads and extensive touchpoints due to metadata storage and accesses or they only provide limited protections. For example, some approaches rely on fat pointers (*e.g.* 128 bits) containing bounds information and other metadata. These drastically increase the memory footprint of software and require the use of tagged memory to protect the integrity of metadata within the fat pointers [38, 54]. Other approaches assign a separate copy of metadata to every granule of each allocation for some fixed granule size, *e.g.* ARM® Memory Tagging Extension (MTE) with four tag bits for every 16-byte granule of data [44, 46]. Still other approaches assign separate metadata for every copy of a pointer, such as Intel® MPX that associates bounds with pointers in a multi-level table with high overheads [42, 57].

Authenticating or encrypting pointers to mitigate pointer corruption is an approach that avoids the need for separate metadata, but that does not protect data directly [9, 33]. For example, Heartbleed disclosed private keys without corrupting any pointers [17]. Authentication of base addresses for pointers can be used to generate keys for hash tables containing bounds to enforce spatial safety, and those can be invalidated when allocations are freed to enforce temporal safety as in Always-On memory Safety (AOS) [25]. However, this reintroduces the overheads of in-memory metadata.

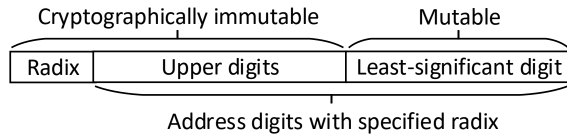

Figure 1: Every allocation is assigned a unique *Cryptographic Address (CA),* which is the basis for isolating each allocation cryptographically from all other allocations. The radix is required to be a power-of-two so that pointers can be represented with a 64-bit, binary legacy-compatible encoding.

Furthermore, pointer authentication consumes pointer bits that could otherwise be used for memory addressing, and it provides weak protection based on small authentication codes.

In this paper we propose *Cryptographic Capability Computing (C^3)* - the first stateless mechanism that enforces memory safety in a fully flexible memory layout without relying on any additional metadata besides what is encoded in a 64-bit pointer. **It replaces inefficient metadata memory accesses with efficient cryptography by assigning a unique and distinct cryptographically isolated space for each allocation.**

The isolated space for each allocation is identified by information encoded in a novel *Cryptographic Address (CA)* format. No additional metadata is needed. The CA format is illustrated in Figure 1. Each CA specifies a radix to identify what portion of the address is constant throughout the entire allocation. We will show that this pair of attributes is sufficient to uniquely identify every allocation throughout the linear address space, and that they can also distinguish allocations temporally. Cryptography mitigates forgery or corruption of the CA portion uniquely identifying the allocation. CAs have a legacy-compatible binary encoding usable with ordinary pointer arithmetic instructions to avoid requiring software recompilation. The linear memory layout is completely unchanged by C^3. C^3 encrypts data in memory and throughout the entire cache hierarchy. It entangles the data encryption for each allocation to the unforgeable (within cryptographic bounds) identity specified in its authorized CA. This makes pointers like unique keys for each allocation to enforce both spatial and temporal safety.

Many prior approaches emphasize generating exceptions for memory safety violations. This is undeniably useful for bug hunting. However, a novel aspect of C^3 is the observation that simply preserving data confidentiality and avoiding adversarial plaintext injection is adequate to enforce memory safety. C^3 does this without relying on the costly metadata needed for prior approaches that insist on generating exceptions.

Encrypting pointers rather than authenticating them as in past approaches avoids limiting address space sizes and strengthens the cryptographic protections for mitigating pointer corruption and forgery by permitting address bits to be encrypted. Pointers do not require special protections or tags beyond the cryptographic encoding itself.

Encrypting data in the cache holds the potential of more strongly mitigating physical attacks and threats from erroneous or compromised IP within SoCs.

C^3 contributes innovative techniques for avoiding disruptions from encrypted pointers and cache data to performance critical microarchitectural flows involving Translation Lookaside Buffer (TLB) lookups and the generation of keying material on the critical load path. We show that C^3 can be built on a modern high performance processor while causing negligible performance overheads.

This paper demonstrates that C^3 is a stateless mechanism that strongly mitigates prevalent categories of vulnerabilities.

The novel contributions of this paper include:

(1) First stateless spatial and temporal safety mechanism applicable to entire address space with no extra storage demands for metadata, no added restrictions on memory layouts, and no pointer size increases.
(2) Compact cryptographic pointer encoding that generates a unique encoding for every simultaneously active allocation and an expansive space for distinguishing allocations temporally. Up to 1 million times as strong as memory tagging.
(3) Microarchitectural optimizations for hiding the latency of pointer decryption, resulting in negligible 0.01% geomean overhead on a simulated subset of SPEC CPU2006.
(4) Legacy compatibility with no requirement for recompilation or kernel changes as demonstrated by running all 19 simulated C/C++ SPEC CPU2006 workloads on a Simics®-based functional simulator.
(5) Validated against relevant tests in the NIST Juliet suite, demonstrating the security efficacy of C^3.

The rest of this paper is organized as follows: §2 provides background, §3 enumerates requirements and the threat model, §4 describes the design of C^3 including microarchitectural optimizations, §5 evaluates C^3, §6 discusses related work, §7 describes limitations and future work, and §8 concludes this paper.

2 BACKGROUND

2.1 Memory Safety

Memory safety vulnerabilities have long been a major affliction for software written in loosely-typed languages such as C and C++. Data from Microsoft® and Google® LLC show around 70% of vulnerabilities fitting this category [7, 37], at least at the high severity level, with most of those vulnerabilities affecting heap allocations.

The most prevalent vulnerabilities violate either spatial or temporal safety. Spatial safety violations include buffer overflows and underflows, whereas temporal safety violations encompass UAF, in which a dangling pointer is used to reference a region of memory that was freed after the pointer to that region was generated, and uninitialized use. Type confusion is an additional type of vulnerability, in which data is processed as though it has a different type. For example, treating data as a pointer may permit pointer forgery. Some other type confusion vulnerabilities operate at the level of types defined in high-level programming languages.

The spatial safety category can be subdivided further into adjacent and non-adjacent overflows, with the former extending from an allocation into adjoining allocations and the latter skipping over adjoining allocations or randomly access memory to reach non-adjoining allocations. Non-adjacent overflow vulnerabilities have

become more common than adjacent overflow vulnerabilities starting in 2014 [37]. Spatial safety hardening approaches can vary in their effectiveness at mitigating each of these types of vulnerabilities.

Examples of both types of spatial vulnerabilities as well as UAF are provided in the following sample code listing. These are the types of vulnerabilities that we focus on in this paper. For the purposes of this example, assume that the heap allocator places each allocation at the lowest available address, including immediately reclaiming memory that is freed. Also assume that it stores allocator metadata separately from allocations, e.g. rather than interspersing it between allocations.

```
void vulnerable_function() {
  int *alloc1 = (int *)malloc(8*sizeof(int));
  int *alloc2 = (int *)malloc(8*sizeof(int));
  int *alloc3 = (int *)malloc(8*sizeof(int));
  alloc1[9] = 5; // adjacent overflow
  alloc1[16] = 5; // non-adjacent overflow
  free(alloc1);
  int *alloc4 = (int *)malloc(8*sizeof(int));
  alloc1[2] = 5; // UAF
}
```

Memory safety vulnerabilities may lead directly to information leakage or data corruption, or they may be used as just one or more links in an overall exploit chain that may lead to Control-Flow Integrity (CFI) violations and the execution of arbitrary code [37]. Thus, mitigating memory safety vulnerabilities offers a way to disrupt a wide variety of possible exploits.

Other types of software vulnerabilities outside of the memory safety category are also common, but they are often outgrowths from logic errors in high-level applications, languages, or frameworks, and thus not readily addressable by hardware.

The goal of memory-safety enforcement mechanisms is to prevent information leakage and adversarial control due to memory safety violations. A variety of mechanisms have been proposed, and we will next describe the category of mechanisms that includes C^3.

2.2 Memory and Pointer Encryption and Authentication

There has been a progression of encryption being applied more pervasively and deeply over time as hardware support has matured. Instructions for accelerating symmetric encryption have led to near-universal encryption of web traffic [19, 28]. Many storage devices have acceleration for full device encryption, if not also based on the acceleration instructions in the processor to which the device is attached. Hardware-accelerated total memory encryption has recently attained wide availability even on consumer-level devices to mitigate physical attacks on memory [40], but that does not extend into caches.

Pointers have been encrypted and authenticated in caches and registers in prior work. This mitigates attempts to use pointers corrupted via memory safety violations, which indirectly mitigates some memory safety violations. PointGuard encrypts entire pointers so that corrupting a pointer will result in a garbled linear address when that pointer is decrypted [9, 51]. PARTS authenticates pointers to generate an exception when a corrupted pointer is used [33].

The drawback of pointer authentication compared to pointer encryption is that a Pointer Authentication Code (PAC) consumes pointer bits whereas pointer encryption preserves pointer bit values that can be recovered later via decryption. Pointers have few unused bits that can be repurposed for a PAC, which limits the cryptographic strength of that PAC.

On the other hand, pointer authentication detects pointer corruption with a predictable probability, whereas the ability of pointer encryption mechanisms to detect pointer corruption depends on multiple factors that determine the likelihood of garbled, decrypted pointers referencing unmapped or inaccessible memory that leads to page faults: *1)* how many pointer bits are encrypted, *2)* the encryption algorithm, and *3)* the process-specific density of accessible data pages.

AOS uses a PAC to lookup bounds for an allocation to directly enforce spatial and temporal safety, but it does not encrypt data.

C^3 is the first mechanism that entangles data encryption with encoded CAs, and it does so in the cache. Bringing data encryption into the cache in this way is both a logical progression and a radical advance, since it integrates stateless, object-granular (we use the terms "object" and "allocation" interchangeably) control over encryption with memory access instructions for the first time. This converges software attack protections with physical attack mitigations and directly enforces both spatial and temporal memory safety.

3 REQUIREMENTS AND THREAT MODEL

The program is assumed to be initially benign but vulnerable to malicious inputs, *e.g.* from network packets or files.

We initially evaluate C^3 for protecting heap allocations due to the prevalence of heap vulnerabilities, and since we want to avoid requiring recompilation. Most exploits target the heap [3]. However, C^3 can also be used to harden stack and global allocations.

C^3 supports encrypted allocations up to 16GiB. Larger allocations can be handled without encryption, and pointers to them could not be abused to access encrypted allocations. Furthermore, C^3 can be extended to protect larger allocations by supporting additional pointer encoding options with a wider range of radixes.

C^3 meets or exceeds the exploit detection probabilities of other memory safety techniques, *e.g.* the 1/16 chance of an adversary bypassing memory safety checks in memory tagging.

Since C^3 is a capability architecture, if an adversary is directly able to harvest a valid (*i.e.*, non-stale) pointer, then accesses to the allocation referenced by that pointer are not considered memory safety violations.

Side channels and vulnerabilities or malicious behavior in privileged software or the heap allocator are considered out-of-scope for this paper, although invalid speculative overflows will similarly access ciphertext.

The focus of this paper is on hardening unprivileged, usermode software, but C^3 could be applied to privileged kernels and VMMs with system software support.

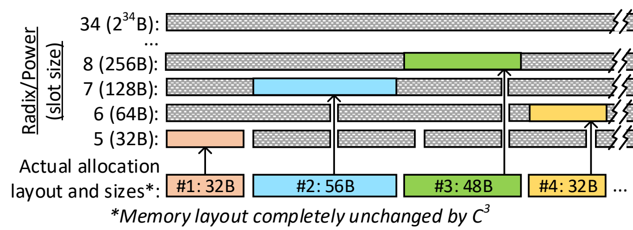

Figure 2: Best-fit assignment of allocations to power-of-two-aligned slots. Each radix implies a distinct set of naturally-aligned power-of-two slot boundaries to be used throughout the linear address space. Every slot is encrypted uniquely to isolate all allocations, even if they overlap in the linear space. C^3 does not disrupt the memory layout; allocations do not need to be padded to fill slots.

4 DESIGN

This section describes how C^3 mitigates both spatial and temporal vulnerabilities by binding stateless, object-granular data encryption in caches to CAs that are unique to each allocation.

4.1 Object-Granular Cryptographic Isolation

Each allocation is assigned a distinct Cryptographic Address (CA), which results in every allocation being uniquely encrypted. To understand how every allocation is distinguished spatially from all other simultaneously-valid allocations, it is helpful to think of each CA effectively defining a naturally-aligned power-of-two slot that contains the allocation in its entirety as depicted in Figure 2. Allocations do not need to be aligned to a power-of-two, *i.e.* there is no requirement for an allocation to fill its assigned slot.

To ensure that each allocation is assigned to a distinct slot, a "best-fit" criterion should be used to select the smallest slot that completely contains a given allocation, which will have the minimum space in the slot that is outside of the actual allocation. Consider that every allocation will then cross the midpoint of its assigned allocation slot. Only a single allocation can cross any particular slot's midpoint. If the allocation fits within some slot, but it does not cross that slot's midpoint, then it must fit more tightly in some smaller slot, and hence the best-fit algorithm would have selected that smaller slot. Therefore, for a given spatial memory layout, only a single allocation can be assigned to any slot.

The memory layout of allocations is not disrupted. The best-fit power-of-two slot of a particular allocation can overlap with other allocations, as we don't require padding to fill the slot. Even in this case, the data for the separate allocations remains cryptographically isolated due to the allocations each being assigned unique CAs to which the data encryption is bound. Using a pointer to an allocation in some slot to read or overwrite a different allocation within the bounds of the same slot results in the data for the second allocation being garbled. This preserves the confidentiality of that data and makes data corruption unpredictable to adversaries. Figure 3 shows how the flow of processing for C^3 mitigates covered memory safety violations.

Table 1: Comparison of mitigation levels provided by PointGuard [9, 51], PARTS [33], MTE [44], AOS [25], and C^3. Legend: ◇: Unmitigated, ◧: Mitigated with detection iff a corrupted pointer is used later, ■: Mitigated with detection, ●: Cryptographic mitigation plus possible detection.

Mechanism	OOB	UAF	Uninitialized use	Physical
PointGuard	◧	◧	◇	◇
PARTS	◧	◧	◇	◇
MTE	■	■	◇	◇
AOS	■	■	◇	◇
C^3	●	●	●	●

Examples of how OOB and UAF vulnerabilities are mitigated by the object-granular cryptographic isolation in C^3 are shown in Figure 4. Example (a) shows a pointer overflowing Allocation 1 into Allocation 2. Example (b) shows a pointer skipping from Allocation 1 all the way into Allocation 3. Allocation 3 has the same radix (64) as Allocation 1, but it has a different value for its least-significant radix-64 digit. Example (c) shows a stale pointer to Allocation 1 after it has been freed being used when the same underlying linear memory has been reused for Allocation 4. Note that Allocation 4 crosses a radix-64 boundary, so it requires a radix-128 slot. Since that is a different radix than was used for Allocation 1, they are cryptographically isolated even though the linear memory for Allocation 1 is reused in Allocation 4. In all of these cases, data will be garbled. Example (b) may generate an exception even before the data can be accessed, as will be explained below.

Furthermore, data encryption is distinctly well-suited to comprehensively mitigating uninitialized use throughout an entire allocation. C^3 breaks adversaries' control over uninitialized data values by encrypting stale data from previous allocations when read through a fresh CA with a different radix or version. This applies separately to every byte of the allocation, such that even a single remaining uninitialized byte is still mitigated by C^3. In contrast, the overheads of per-byte initialization metadata would be prohibitive.

Note that pointer authentication or encryption alone [9, 33, 51] is unable to mitigate some of the types of vulnerabilities addressed by C^3, which additionally entangles data encryption with unique, per-allocation CAs. For example, Heartbleed illustrates this benefit of C^3, since it only performs OOB reads that disclose data without corrupting pointers [17]. Fundamentally, the limitation of prior approaches based solely on pointer authentication or encryption is that unencrypted data in memory remains accessible.

In contrast, when a C^3 pointer is dereferenced that is corrupted, stale, or references an uninitialized allocation, the data will be garbled with high probability if the decrypted pointer even references accessible memory in the first place. We compare the types of violations that can be mitigated by C^3 and closely related memory safety mechanisms in Table 1. The "Physical" column represents threats to data confidentiality and pointer integrity from physical attacks and erroneous or compromised IP. C^3 can detect certain physical pointer corruptions and provides data confidentiality via encryption.

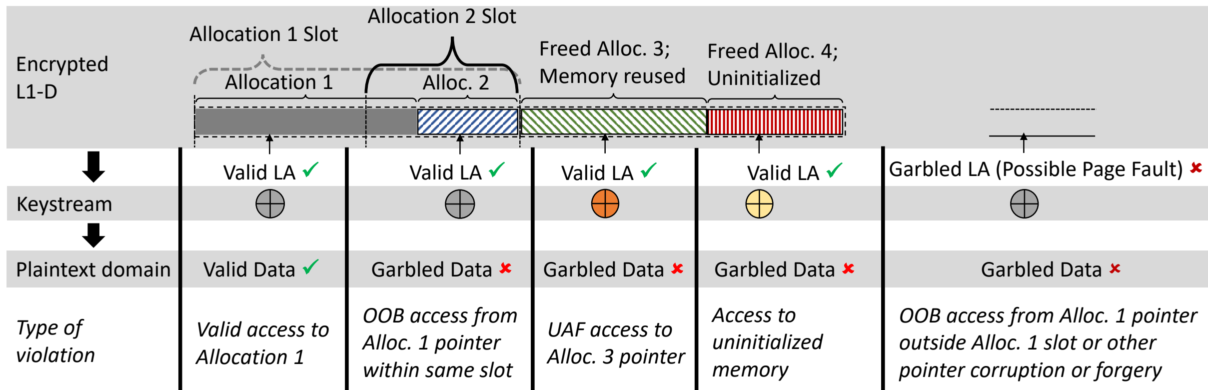

Figure 3: The flow of processing for C^3 includes CA decryption (§4.3) to obtain the Linear Address (LA) followed by data encryption or decryption based on a generated keystream (§4.4). In-scope memory safety violations result in garbled encryption or decryption with high probability. This enforces cryptographic isolation between different allocations both spatially and temporally, even within overlapping slots. Additionally, OOB accesses outside of the authorized slot for the pointer or other pointer corruption or forgery result in garbled linear addresses. This will secondarily generate a page fault with high probability assuming a sparse address space, hence leading to detection of the violation.

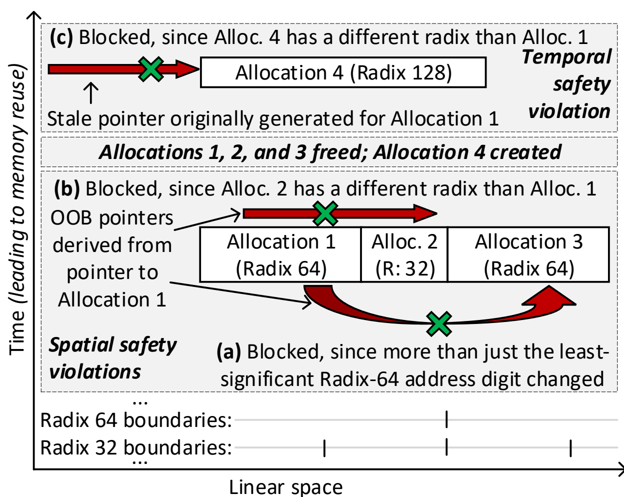

Figure 4: Examples of how object-granular cryptographic isolation in C^3 mitigates attempts to corrupt a pointer for OOB accesses (a and b) or to reuse it after the linear memory for its authorized allocation has been reallocated (c). The linear memory layout is completely unchanged by C^3. Only the radix-specific slot boundaries that are relevant to these examples are illustrated.

4.2 Pointer Encoding

C^3 preserves the 64-bit pointer width and ordinary pointer arithmetic support, although any pointer size may be supported. Each pointer is divided into distinct fields as illustrated in Figure 5 to represent a CA in a binary encoding.

The power field specifies the offset field width, with the value of all zeroes reserved to represent unencoded userspace pointers that point to unencrypted data. Even single-byte slots can be specified, which would impose too much overhead for memory tagging approaches that need to duplicate the tag for every granule (*e.g.* four tag bits per one-byte data granule would impose 50% memory overhead). Power binds the pointer to an object of a particular memory slot with the corresponding power-of-two size. The object is not required to completely fill the slot, so there is no requirement for padding.

The optional version field can be assigned randomly or in sequence for a given power-of-two slot as the slot is reused to prevent the encrypted data from being bound to the same CA across distinct allocations for temporal safety. Alternatively, the allocator could seek to avoid reusing linear memory with identical radixes while still permitting the memory to be reused and assigned different radixes so that there is no wasted memory. However, that may introduce more allocator complexity. The decrypted version field is ignored by the processor.

Both the encrypted address bits and the fixed address bits are immutable. If software attempts to change any of them or to forge a pointer, that will be detected with high probability when the corrupted pointer is used in a memory access. A change to an encrypted address bit is likely to change half of the decrypted bits of the decrypted address slice due to the diffusion property of the pointer encryption cipher as will be explained below. This incorrect decryption is likely to point to an unmapped or inaccessible page, generating a page fault, or to some uncontrollable page address. Certain paging modes treat some of the address bits in the encrypted address field as reserved, which causes them to serve as an implicit authentication code by being required to have some predictable value, *e.g.* all zeroes. Keep in mind that C^3 preserves object-granular

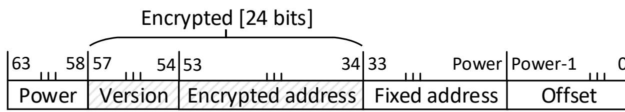

Figure 5: CA encrypted pointer format. "Power" is an exponent encoding the power-of-two radix/slot size that best fits the allocation the pointer is authorized to reference. Optional "Version" introduces cryptographic diversity, but an alternative is to avoid reusing a given power-of-two slot by changing the power used for the same underlying linear memory (*i.e.* no wasted memory). "Encrypted addr. (address)" contains the upper address bits that are both immutable and within the encrypted portion of the pointer. "Fixed address" contains plaintext bits indicating which slot with the specified power best fits the allocation. "Offset" bits are also plaintext, and software can modify them to point to different portions of the referenced allocation.

cryptographic isolation even when pointer corruption or forgery does not generate an exception.

Since the pointer bits are encrypted rather than being authenticated, the values of the underlying pointer bits can be recovered by decrypting the pointer. In contrast, pointer authentication as used in some previous approaches prevents the pointer bits containing the authentication code from being used for addressing [25, 33].

To detect changes to the fixed address bits, C^3 incorporates them as a tweak during pointer encryption. We describe the details of this below. Briefly, an identical tweak must be supplied during decryption as was supplied during encryption, or the decryption will be incorrect with a resulting random value with a high probability. We also incorporate the power bits in the tweak to detect unauthorized modifications of those fields.

Software is free to modify the offset bits. Since the supported radixes are all powers of two, the least-significant address digit of the specified radix can be represented as the corresponding number of offset bits.

If there are any allocations that do not fit into a 16GiB-aligned slot, they can be left unencrypted. The source of the 16GiB limit is that the offset field covering the allocation must be kept as plaintext. Thus, 34 plaintext pointer bits can represent at most an 16GiB-aligned slot. The fixed address bits field will not be present when the pointer refers to a 16GiB slot, since the entire 34 plaintext bits will be used as the offset.

Furthermore, a special power field value can be designated that represents 16GiB adjusted slots that are offset by 8GiB from 16GiB alignment boundaries, *i.e.* are shifted by half of their width. This can accommodate allocations spanning 16GiB alignment boundaries. Although the radix for such adjusted slots would match that for maximally-sized unadjusted slots, those would still effectively reside in separate, isolated cryptographic address spaces.

Unencoded pointers indicate to the processor that the access should be performed with data encryption and decryption disabled. Unencoded pointers can be detected in constant time, since the equivalent bit locations for the power field will be set to all zeroes or ones for userspace or supervisor pointers, respectively, to match

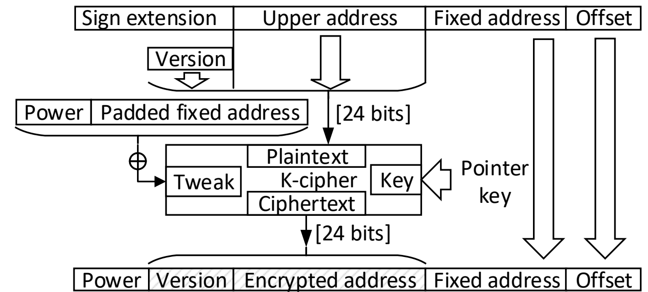

Figure 6: Flow for generating a CA given a plaintext input pointer and object context. Object context specifies the power-of-two slot fitting the object and its version to distinguish it from other allocations occupying the same slot at different times.

conventional address formats. Those power field values are not used for encoded pointers. Unencrypted accesses can be restricted to particular address ranges, *e.g.* using page table entry bits.

4.3 Pointer Encryption

C^3 encrypts pointers using a tweakable block cipher that provides diffusion and confusion and is resistant to known ciphertext attacks. It is named "K-cipher" [27]. No small block ciphers are currently standardized, although several are being considered for a standard. Other suitable small block ciphers with the same or different block sizes can be used in place of K-cipher. For example, 32-bit SIMON could be used if C^3 encrypted 32 pointer bits [2]. Existing state-of-the-art industry solutions like ARM® Pointer Authentication (PAuth) also use non-standard small block ciphers. Other prior arts incorporate this as well. For example, AOS uses a 16-bit Pointer Authentication Code (PAC), which is weaker than the 24-bit encrypted pointer slice in C^3.

Known ciphertext resistance makes K-cipher adequate for our threat model, since we assume that adversaries are constrained to be unable to generate or decrypt arbitrary encrypted pointers. They are only able to observe the encrypted pointers returned by the allocator in response to allocation requests. Those pointers do not reveal the underlying upper address bits, since those bits are encrypted.

The procedure for generating a CA is illustrated in Figure 6. It has the following inputs: *1)* Plaintext input pointer, *2)* Pointer encryption key, and *3)* Object context operand specifying the following information: *(a)* Power indicating the exponent for the power-of-two-aligned slot containing the allocation, and *(b)* Optional version field.

The pointer encoding algorithm may be implemented by a software routine or as a new instruction. Implementing it as an instruction enables the processor to protect the pointer encryption key by storing it in a register inaccessible from untrusted software (e.g. a write-only MSR) and enables hardware optimization of the pointer encoding algorithm.

If implemented as an instruction, the instruction may accept the plaintext input pointer and the object context operand as explicit instruction operands. It will also implicitly load the pointer

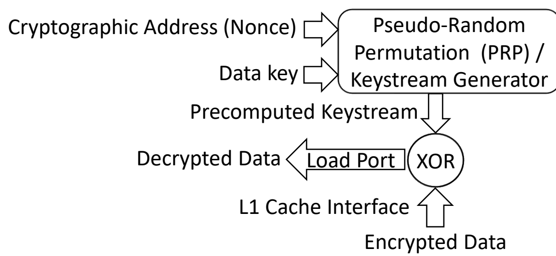

Figure 7: Data decryption during load execution of C^3 with parallel keystream generation to avoid delays. Data encryption during stores is similar, except that the flow of data through the XOR operation is reversed. Every allocation is uniquely encrypted due to each allocation being assigned a unique CA. The 64-bit CA itself is the only access-specific security input; no additional metadata is needed.

encryption key from the register that contains it. The OS would initialize the pointer encryption key register with a per-process key analogous to a per-process PAuth key. Other key management mechanisms may be employed to support various threat models.

The tweak input to the pointer encryption comprises all of the plaintext pointer components that need to be protected from being modified by the software. The fixed address bits field from the plaintext input pointer is padded to always have a width of 34 bits and to maintain the relative positions of each bit in the padded field (i.e. padding on the right). An XOR fold operation (or some other more sophisticated approach such as a lightweight permutation) for generating the tweak is needed due to the input fields being wider (40 bits) than the 24-bit block size of the pointer encryption cipher.

The pointer decode flow is the reverse of the pointer encode flow with the requirement that the same tweak value be supplied. Decoding also drops the optional version field, since it does not affect the generated linear address.

4.4 Data Encryption

We investigated cryptographic literature to identify minimal latency keystream generation techniques. An implementation analysis for Gimli reported a fully unrolled permutation with a 715.9 psec critical path, which results in a depth-4 pipeline that supports up to a 4.5Ghz clock [21]. We have implemented a similar keystream generator to minimize the cryptographic load and store latency and hence minimize overall performance overhead of the proposed C^3 technique. Other candidates from the NIST Lightweight Cryptography project [41] could similarly be used, e.g. Subterranean or Xoodyak [12, 13].

For a load, the keystream generation is started as soon as the Address Generation Unit (AGU) generates the CA for the instruction. In other words, a load execution pipeline and respective keystream generation pipeline proceed in parallel. Once both pipelines finish, the encrypted data fetched from the data-cache gets XORed with the keystream and returned to the respective load port as depicted in Figure 7. This avoids delays in the critical load path. Each process is assigned a unique data encryption key analogous to a per-process PAuth key.

One extra XOR latency in 7 nm FinFET is 4.5ps [49], which should fit in the final data cycle of L1 hit that typically just has data rotation. If there is a timing issue there, the extra XOR can be absorbed in the bypass network (before data is given to the consumers by the load). We are hence confident that this will not increase L1 latency. However, in case there are designs that cannot absorb this latency, an extra cycle would be needed on the L1 hit data-path for encrypted loads. We evaluate this possibility in §5.2.

4.5 Mitigation Strength

The 24-bit encrypted slice in the pointer provides a great deal more potential variation between pointers than is possible with the four-bit tag in MTE, for example, which strengthens UAF mitigation. This results in 1M times as many possible pointer encodings for the encrypted pointer slice compared to the tag value that serves the analogous role of mitigating UAF in MTE. Even within a given slot, the version field can provide cryptographic diversity. Furthermore, the allocator can make slot assignments unpredictable. The pointer encoding is unpredictable even if the slot location and version are known, since the pointer key may be assigned fresh for each process.

The strength of MTE to mitigate non-adjacent overflows drops back to the basic probability of guessing the tag for the targeted memory. In this instance as well, C^3 offers much greater strength due to its larger encrypted pointer slice.

This fundamental advantage of C^3 compared to MTE extends to other types of vulnerabilities such as pointer forgery and physical attacks.

4.6 Microarchitecture

In this section, we describe the microarchitecture to support the C^3 security capability in a performant manner within state-of-the-art processor designs. C^3 enforces temporally and spatially unique encryption of pointers and therefore the memory engine receives encrypted addresses for the load/store operations. Since the load/store pipeline and the cache hierarchy addressing circuitry is unmodified in C^3 and operates on plaintext addresses, it necessitates timely pointer decryption by the memory engine. In addition to pointer encryption, C^3 also enables data encryption. Therefore, during a load operation, the encrypted data needs to be decrypted before the write-back into the registers. Similarly, plaintext data from the core needs to be encrypted before being stored in the caches. We describe how the load and store pipelines are architected to induct C^3 in modern microprocessors.

4.6.1 Load pipeline.

Pipeline stages. Figure 8 illustrates the common pipeline stages for a load operation found ubiquitously in modern out-of-order processors [10, 23]. Firstly, the address generation unit (AGU) computes the virtual address (VA) of the load operation. Secondly, the translation-lookaside-buffer (TLB) is looked up for translating the virtual to physical address. Most processors use virtually-indexed and physical-tagged (VIPT) data caches, since it allows the L1 cache set read to be done in parallel to TLB lookup [31, 34, 58]. Thirdly,

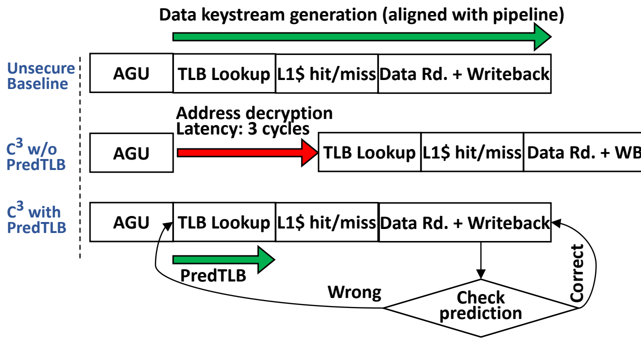

Figure 8: The load pipeline of the unsecure baseline, C^3 without PredTLB, and C^3 with PredTLB.

we have the L1 cache hit/miss decision, depending on the physical address provided by the TLB (on a TLB hit). Fourthly, on a L1 cache hit, the data is read out from the selected way in the set-associative L1 cache and the data is written back to the required consumers of the load.

TLB does not store cryptographic address translation. With C^3 enabled in a system, the AGU performs arithmetic on pointers in the same format used by software, *i.e.* as Cryptographic Addresses (CAs) for encrypted allocations and as unencoded pointers otherwise. However, the rest of the load pipeline works on plaintext addresses. The TLB does not store address translations for CA for the following reasons: (i) The TLB caches the page table entries, which are in turn populated by the OS; hence page tables are left in plaintext for legacy compatibility. (ii) A page may be shared between 2 different allocations. Since the CAs for these 2 allocations are different, we would need to allocate 2 separate TLB entries for the same translation. This would incur high TLB overhead which is difficult to accommodate since TLBs employ area- and power-hungry CAMs.

Predictive TLB lookup. The load pipeline in the presence of C^3 requires decryption of the CAs before the TLB lookup. The K-cipher, which is used for pointer encryption, is also used for pointer decryption. Decryption of CA introduces a 3 cycle K-cipher latency between the AGU and the TLB lookup directly on the critical path of the load pipeline. This additional latency incurs a significant 5% performance loss over our considered evaluation configuration (refer to Section 5.2). To mitigate this performance loss, we use a **Predictive TLB** (PredTLB) lookup which removes the constant decryption latency from the critical path to improve performance.

PredTLB is based on the insight that bits 33:0, *i.e.*, the lower 34 bits of the CA are actually in plaintext, which can be used for partial TLB lookup. With a page size of 4KiB, bits 11:0 denote the page offset of the address. PredTLB looks up the TLB structure with bits 33:12 (*i.e.*, 22 bits) of the CA, which are in plaintext, and performs a partial tag match. PredTLB predicts a TLB hit if any entry has a tag whose lower 22 bits match with bits 33:12 of the CA and provides the physical page number stored in that TLB entry as the corresponding translation. The L1 cache tag match uses this translation for hit/miss decisions and data lookup.

Since PredTLB does not wait for pointer decryption before TLB lookup, it completely bypasses the constant 3 cycle decryption latency from the critical path. However, PredTLB predicts a TLB hit/miss based on partial tag match, and it can result in a misprediction due to aliasing between 2 different virtual page numbers (VPNs) which have the same lower 22 bits. The result of the prediction from PredTLB is known after the decryption is completed, *i.e.*, before the data writeback. On a PredTLB misprediction, we cancel the load operation and re-dispatch it through the load pipeline. During this second try, the decrypted address is already available from the first pipeline dispatch of the load, and hence, the load proceeds through the pipeline and completes successfully. Therefore, a PredTLB misprediction incurs the latency of a useless pipepass. From our evaluations, we observe that PredTLB has a high successful prediction rate of 99.85% (refer Figure 10(b)), thereby enabling C^3 to bypass the constant decryption latency for 99.85% of the loads.

No performance overhead for data decryption. Since all the inputs for data keystream generation including the encrypted pointer are available right after the AGU, the generation is initiated after the AGU pipeline stage and happens in parallel with rest of the load pipeline. The keystream generation completes at the end of the write-back stage, when data is actually written back to the registers. The generated keystream is XORed with the encrypted data fetched from the L1 cache before the write-back. Essentially, the data decryption latency during a load operation is overlapped with the load pipeline, rendering no performance impact.

4.6.2 Store pipeline. All modern out-of-order processors split the store instruction into 2 independent sub-operations: store address (SA) and store data (SD); this split enables parallel/OoO execution of SA and SD, presenting performance improvement.

With C^3, the SA sub-operation gets a CA from the AGU. The SA needs to perform a TLB lookup to retrieve the physical address for the VA, which is encrypted. Therefore, we have a situation similar to the load pipeline, and we employ PredTLB for SA as well in order to prevent the addition of a constant decryption latency before the TLB lookup. The data to be stored also needs encryption. Hence, the keystream generation is started as soon as the CA is available and XORed with the data in the store-buffer (if available). In case the data is not present, the keystream is kept in the space of data, and XORed with the incoming data, when available. Similar to loads, the decryption is overlapped with the SA/SD pipelines and has no performance impact.

4.6.3 Memory engine microarchitecture. In this section, we describe the overall microarchitecture of the memory engine to support C^3. Addresses output by the AGU in CA format are decoded by the **Pointer Decryption** module. The CA is also used by the **Data Keystream** module to generate the keystream for data decryption (during loads) or encryption (during stores). The plaintext address is stored in the Load Buffer (LB) and Store Buffer (SB). The LB and SB entries are extended in C^3 to store the 24 bits of encrypted slice from the CA; these 24 bits can be concatenated with the lower 34 bits of the plaintext address to re-create the CA. This is especially necessary for timely load/store forwarding and memory disambiguation decisions. Either the load or store might not have the plaintext address available during the address comparisons.

LB Entry Extension: < 24 bits cryptographic slice >
SB Entry Extension: < 24 bits cryptographic slice, KeyAv >

Figure 9: Overview of the C^3 memory engine microarchitecture.

Hence, we can perform the address comparisons on the CAs for the corresponding load/store pair.

On a load, the L1 data cache returns encrypted data stored in the cache which is XORed with the already generated data keystream for retrieving the plaintext data. The forwarded data from a store to a younger load is encrypted, which is decrypted with the same data keystream. On a store, the store address operation provides the CA, which is used to generate the keystream for encrypting the data to be stored in cache. If the keystream is generated before the data is available (due to OoO execution of SA before SD), it is stored in place of the data in SB. Each SB entry is also extended to include a **Keystream Available** (KeyAv) bit which denotes whether the keystream is stored in the space for data. When the data is available, if KeyAv=1, it is XORed with the available keystream to generate the encrypted data, and stored in the L1 cache.

5 EVALUATION

5.1 Legacy Compatibility

We verified legacy compatibility with Simics®, which is a functional full-system simulator [36]. We used the instrumentation API of the Quick-Start Platform for x86 (QSP-x86) CPU model to functionally simulate handling of encoded addresses and encryption of data in memory.

The C^3 Simics® model intercepts load and store instructions and decrypts encrypted virtual addresses before the address translation. In addition, the model generates a keystream for the data by using the CA as input. Once the data is written to (read from) memory, it is XORed with the keystream to perform encryption (decryption). Loads and stores using an unencoded address bypass this instrumentation.

On top of the C^3 Simics® hardware model, we run an unmodified Clear Linux OS. To enable C^3 support in software, we modified the glibc v2.30 heap memory allocator to encrypt heap pointers before they are returned to the user application, while the actual memory management code is unchanged. In addition, `realloc()` and `free()` are modified to decrypt encrypted pointers provided as input. An application linked to such a library receives encrypted pointers upon dynamic memory allocation (e.g., `malloc()`) and can perform regular pointer arithmetic on them as if they were legacy pointers (e.g., array indexing).

Moreover, we modified system calls in glibc that use pointers as input arguments, *e.g.* pointers to buffers for exchanging data between the application and the kernel. An analogous challenge arose in the context of MTE, and various possible approaches for handling it have been proposed [26]. Since the application may provide a CA as input to a system call and the kernel only accepts addresses in the conventional (*i.e.* unencoded) format via system calls, we allocate an unencrypted buffer to serve as an interface between the application and the kernel. Before a system call, data is copied (with implicit decryption) into the temporary buffer and the pointer to the buffer is passed down to the system call. After returning from the system call, the data is copied back to the application's buffer, which encrypts the data if the application references that buffer using a CA.

Several memory and string routines, such as `memcpy`, `strlen()`, and `strcmp()`, needed to be adjusted to prevent benign OOB violations. Such functions are typically optimized to process strings in blocks using SIMD instructions, resulting in fetching data beyond the allocated region. We replaced these functions with an alternative implementation that avoids looking ahead beyond the allocation bounds in string buffers. Google® LLC observed a fundamentally similar challenge with MTE, which motivated them to modify various routines in the Android C library as well [8].

We used SPEC CPU2006 C and C++ workloads to verify the functionality of the model. The FORTAN-based workloads were excluded from this experiment, since they do not use the memory allocator in the C standard library. We compiled the unmodified workloads, linked them with the modified glibc shared library and executed the workloads inside the Simics® simulation.

In addition, we generated traces from the workloads running inside Simics® for an in-house, cycle-accurate simulator for performance evaluation.

5.2 Performance

In this section, we discuss the evaluation methodology and contrast microarchitectural simulation results for C^3 with state-of-the-art solutions like MTE and AOS. MTE has not yet been released in any processor implementations, so we are making assumptions about how metadata is stored, accessed, and cached in our model that may not correspond to future MTE releases.

We have done an optimistic evaluation of the prior work AOS, with geomean overheads lower than the 8.4% reported in the original AOS paper. Our core configuration is deeper than in the original AOS paper [25], reflecting current commercial processors. The bounds table resizing operation, which increases associativity, incurs no latency overhead. Similarly, searching each way, which may span multiple cachelines, is executed with zero latency overhead.

5.2.1 Evaluation Methodology. We simulate an x86 core using a cycle-accurate simulator that models a dynamically-scheduled OoO core clocked at 4GHz. The core parameters in our model are set to be similar to the Intel® Ice Lake processor [55]. We performed

Front End	4GHz, 5-wide fetch+decode, TAGE/ITAGE branch predictors [47], 32KiB, 8-way L1 instruction cache, 5-wide rename into OoO with macro and micro fusion.
Execution	384 ROB entries, 128 Load Queue entries, 72 Store Queue entries and 168 Issue Queue entries. 8 Execution units (ports) including 2 load ports, 2 store address ports, 2 store-data port, 4 ALU ports, 3 FP/AVX ports, 2 branch ports. 8 wide retire and full support for bypass. Aggressive memory disambiguation predictor. Out of order load scheduling to L1.
Caches	48KiB, 12-way L1 data caches with 5 cycles latency, 512KiB 16-way L2 cache (private) with round-trip latency of 17 cycles. 8 MB, 16-way shared LLC with 47 cycles round-trip latency. Aggressive multi-stream prefetching into L2 and LLC. PC based stride prefetcher at L1. The TLB configuration follows Ice Lake TLB configuration [11].
Memory	Two DDR4-2133 channels, two ranks per channel, eight banks per rank, 64 bits data-width per channel. 15-15-15-36 (tCAS-tRCD-tRP-tRAS) timing.

Table 2: Core parameters used in our simulator.

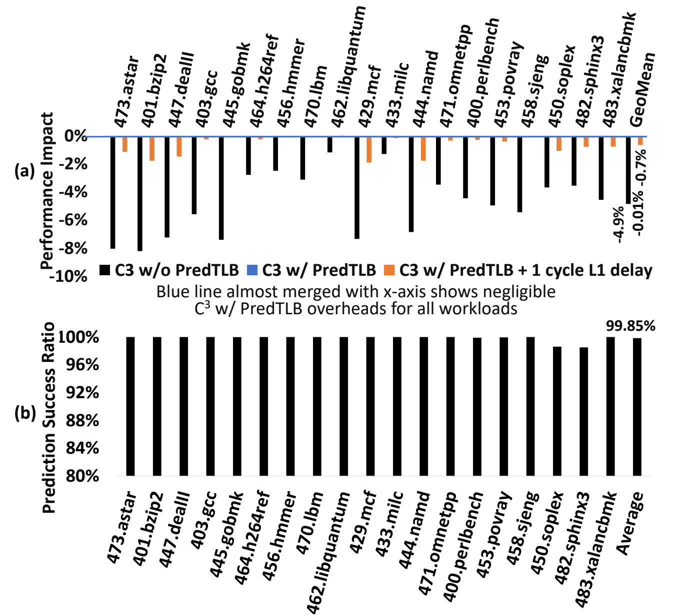

Figure 10: (a) Simulated performance overhead of C^3 without and with PredTLB. PredTLB reduces the performance overhead of C^3 significantly. Lower (*i.e.* more negative) percentages indicate more overhead. We evaluated an extra cycle of L1 latency to accommodate the XOR between the data and keystream for completeness, but it is unlikely to be needed. (b) The prediction success ratio of PredTLB. A higher success ratio means lower load/store cancellation and re-dispatching, resulting in lower performance impact.

these simulations in April 2021, reflecting the best known Ice Lake configuration. The primary core parameters are tabulated in Table 2. AOS mentions using an L1 bounds cache, *i.e.* basically a separate cache at L1 for metadata (L1-M). Since C^3 does not store/access any additional metadata, we first evaluate the proposals without an L1-M, and then discuss the impact of a separate L1-M on MTE and AOS. MTE imposes additional allocator overheads for updating duplicated, stateful tag metadata that this evaluation does not capture, which results in that aspect of our MTE evaluation being optimistic. We show our microarchitecture evaluations on all C/C++ workloads in the SPEC CPU2006 suite.

5.2.2 Performance impact of C^3. We first discuss the performance overhead of C^3, as shown in Figure 10(a) with and without PredTLB to illustrate its effectiveness. The average (GeoMean) performance overhead of C^3 over all workloads is 4.9% without the PredTLB optimization. This is primarily because C^3 inserts a constant decryption latency in the load pipeline, delaying the writeback of critical data to the core and stalling all dependent instructions. A successful prediction from PredTLB means the partial tag match by PredTLB will give the correct hit/miss decision and translation, *i.e.*, same as a full tag match in the baseline. Due to a high successful prediction ratio of 99.85% over all workloads, as shown in Figure 10(b), PredTLB is able to overlap this pointer decryption delay with the load pipeline for a majority of the loads. Across all workloads, we observed a negligible performance overhead of 0.01%. PredTLB enables providing the superior security guarantees of C^3 with negligible performance overhead. Figure 10(a) also illustrates the performance overheads of C^3+PredTLB with an extra delay cycle on the L1 hit data-path for the XOR. The overheads are 0.7% with no large outlier workloads. However, we are confident that will not be needed as we explained in §4.

5.2.3 Comparison against MTE/AOS without L1-M. We compare the performance overhead of C^3+PredTLB against competing memory safety proposals like MTE and AOS. Both MTE and AOS store and access required metadata at a separate physical address space than the data and require an additional metadata load for any memory access to the protected address regions in memory. Whereas the additional metadata address space needs to be cached in the data caches during execution, the additional metadata loads consume critical L1 cache bandwidth. Therefore, MTE and AOS performance is dependent on the metadata hit rate and available L1 bandwidth. C^3 offers a significant contrast to MTE and AOS since it neither stores metadata in a separate physical address space, nor does it need to spawn additional loads to fetch and verify this metadata.

Figure 11 illustrates the difference in performance overhead of C^3, MTE, and AOS. Whereas C^3 shows negligible performance overhead for all workloads, MTE and AOS have an average overhead of 7% and 3.6%, respectively, which are significantly high. Furthermore, MTE and AOS have large outliers. The performance overheads from existing security solutions are clearly extremely costly. On the contrary C^3 has negligible overheads, making it an even more attractive security solution.

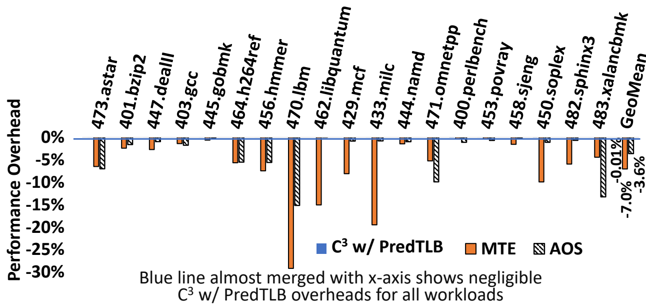

Figure 11: Simulated performance overhead comparisons of C^3 with PredTLB against MTE and AOS, without a dedicated metadata cache. Lower (*i.e.* more negative) percentages indicate more overhead.

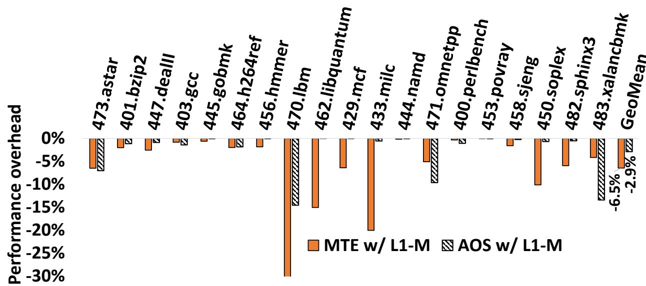

Figure 12: Simulated performance overhead comparisons of MTE and AOS with a dedicated metadata cache. Even with an L1-M, MTE and AOS have significant performance overhead. Lower (*i.e.* more negative) percentages indicate more overhead.

Since AOS has lower metadata footprint than MTE, it achieves higher metadata hit rates in the data cache, rendering its performance better than MTE. This is especially evident in 470.lbm, 462.libquantum, 433.milc.

5.2.4 Comparison against MTE/AOS with L1-M. We evaluate the performance overhead of MTE and AOS in the presence of a dedicated metadata cache, called L1-M cache. L1-M shifts the metadata caching out of the data caches into the L1-M, providing better metadata hit rates and additional bandwidth at the L1 level to serve metadata loads. However, these benefits are obtained by incurring the area/power overhead of an additional cache. Since C^3 does not store metadata, it does not need a costly metadata cache. Figure 12 illustrates the performance overheads of MTE and AOS with an 8KB L1-M are 6.5% and 2.9%, respectively. 456.hmmer in particular gains significantly with L1-M primarily because its baseline is bandwidth constrained, and therefore L1-M alleviates the bandwidth pressure by providing additional metadata bandwidth. Further, increasing L1-M even to 32KB improves metadata hit rates only by 1.5% with no tangible performance improvements. In conclusion, even with an L1-M, the performance overheads of MTE and AOS are significantly higher than C^3.

5.3 Efficacy

The NIST Juliet Test Suite for C/C++ contains 28,881 test case pairs of good and bad program behavior organized under 118 respective Common Weakness Enumerations (CWEs) [4]. These CWEs are subdivided in Common Vulnerabilities and Exposures (CVE) Variants. Each CVE variant is a unique behavior that is duplicated dozens of times with increasing levels of obfuscation towards static analysis tools. The test suite is designed to test static analysis tools; however, portions of Juliet still provide a reasonable evaluation of runtime defenses. We evaluated C^3 using tests for two relevant CWEs: CWE 122 "Heap Based Buffer Overflow" and CWE 416 "Use After Free". **The object-granular cryptographic isolation provided by C^3 mitigates all of the relevant sample vulnerabilities in Juliet for these two CWEs.** We will now describe our evaluation methodology and show detailed results.

5.3.1 Heap Buffer Overflow Evaluation Methodology. We separately report two possible types of overrun protection, either of which is an effective mitigation for each sample vulnerability: 1) exceptions generated due to detection of corrupted addresses, and 2) silent prevention of plaintext access via data encryption. The first type of protection is apparent when running the test, and we determined when the second type of protection is applicable via source code review of each CVE variant. Note that the second type of protection is adequate for effectively mitigating a vulnerability even if C^3 does not generate an exception. The cryptographic isolation between allocations still protects the data. A CVE variant is listed as detected (*i.e.* via an exception being generated) if and only if all of the workloads within it are detected. The results are shown in Figure 13a.

As Juliet was designed for static analysis, not every test case is applicable for testing a runtime defense. Some only generate memory unsafe behavior during an unpredictable subset of runs, and others never actually generate memory unsafe behavior on the heap at runtime. Intra-object overflows are out-of-scope for many memory safety mechanisms, including C^3, since the entire struct is allocated using one `malloc()`. These overflows only account for 1% of actual observed vulnerabilities reported in a recent analysis and are hard to exploit [24]. We filtered out such inapplicable test cases.

6.9% of the buffer overrun workloads already crash due to memory corruption even without C^3. Adding C^3 raised this crash/detection rate to 87.2%. Source code review concluded that 100% of in-scope variants are protected with the added consideration of data encryption.

5.3.2 Heap UAF Evaluation Methodology. The crucial property for mitigating UAF is actually to prevent access to a region of memory via a dangling pointer after the region has been reallocated [3]. However, all UAF tests in Juliet follow the pattern `malloc()` → write some data → `free()` → print the data → `exit()`, which only accesses a dangling pointer without ever reallocating the memory. We focused our analysis on what would happen if Juliet were to exhibit behavior more like realistic exploits by reallocating memory and then accessing a dangling pointer. Source code review determined that C^3 would mitigate this in all cases by allocating data with varying slots or versions. The results are shown in Figure 13b.

	Total Count	Crash/Detect Count				Protected with Data Encryption	
		without CC		with CC			
Workloads	2104	146	6.9%	1834	87.2%	2104	100.0%
CVE Variants	46	0	0%	40	87.0%	46	100.0%

(a) CWE 122: Heap buffer overrun detection and protection rates. The 6 variants not detected in the final row are all cases where the overflow amount is so small that it stays within the alignment boundary of eight bytes imposed by the glibc allocator. Thus, those overflows never cross a slot boundary. This also means that no other data can be allocated in the locations accessed by the overflow, so it is not an actual security vulnerability.

	Total Count	Protected with Varying Slot or Version	
Workloads	520	520	100.0%
CVE Variants	21	21	100.0%

(b) CWE 416: Heap UAF protection rates.

Figure 13: Efficacy of C^3 on NIST Juliet tests for CWEs 122 and 416.

6 RELATED WORK

This paper has mostly focused on comparing C^3 to past memory tagging approaches and AOS. Tagging associates a tag with each granule of memory, *e.g.* each 16-byte-aligned region [44, 46]. This tag duplication leads to high memory and performance overheads. AOS uses an authentication code of heap allocation base addresses as a hash key into a table of bounds, and that metadata access also leads to performance overheads [25]. The overheads for these approaches were analyzed in detail in earlier sections. C^3 completely avoids overheads due to metadata tables and introduces negligible cryptographic overheads instead.

We will now compare C^3 to other memory safety mechanisms besides MTE and AOS. Unless otherwise noted, this related work covers stack, heap, and globals, whereas per-allocation stack and global protection using C^3 is future work.

Pointer Authentication, Encryption, and Tagging. Some prior mechanisms authenticate pointers (*e.g.* PARTS [33]) or encrypt pointers (*e.g.* PointGuard [9, 51]) without relying on separate bounds metadata, but they only mitigate indirect effects of a limited subset of memory safety violations that happen to corrupt pointers used later as discussed in §4.1 with reference to Table 1. C^3 entangles data encryption with unique, per-allocation CAs to directly protect data confidentiality.

PARTS imposes 19.5% performance overhead and no memory overhead [33]. PointGuard pointer encryption imposes ~2% performance overhead in hardware and no memory overhead [51]. Pointer Tagging relies on tag bits rather than pointer authentication to detect pointer corruption [6].

PTAuth precedes each allocation with an ID incorporated as context when authenticating data pointers to enforce temporal safety with performance overhead of 26.5% (generating PACs in software) and memory overhead of 2% [18]. To support pointer offsets when checking accesses, it searches backwards for some distance until it finds a candidate ID that results in successful pointer authentication. This introduces non-constant overhead, whereas C^3 introduces no new metadata in memory.

Address Sanitizer. Address Sanitizer (ASan) associates a bit with each byte of memory indicating whether it is currently valid [45]. ASan inserts redzones marked as invalid to enforce spatial safety. To enforce temporal safety, ASan quarantines memory from freed allocations. However, ASan exhibits high memory (3.4X total) and performance (73% average) overheads with poor locality, and the lack of a tag value weakens its strength against non-adjacent overflows and pointer forgery. C^3 offers stronger protection than ASan by virtue of its large, encrypted pointer slice unforgeable within cryptographic limits to which data encryption is bound, and it avoids introducing costly metadata.

Metadata Tables Indexed by Storage Location of Pointer. Intel® MPX stores a 256-bit bounds table entry for every pointer, even when many pointers reference the same allocation. A previous analysis showed that this resulted in substantial performance overheads and an average 1.9-2.1X memory overhead [42]. Associating a separate bounds table entry with each pointer prevents direct temporal memory safety enforcement, since it is not feasible to directly identify all pointers that reference an allocation being freed. Subsequent work has shown that temporal safety can be enforced indirectly by unmapping some portions of the bounds table [57], but with overheads of 60% for performance and 36% for memory. BOGO does not enforce temporal safety on the stack. HardBound is another approach that uses a table of bounds to enforce spatial safety, although it also uses a compressed representation when possible, even inlining the bounds in some pointers [14]. C^3 avoids the memory and performance overheads due to storing metadata in memory and it directly enforces temporal safety.

SoftBound+CETS is a combined approach for enforcing spatial and temporal safety in software with a performance overhead of 116% [39]. It uses a bounds table analogously to MPX with an additional level of indirection through the bounds table entries to a slot containing a value that is shared across all copies of pointers to a particular allocation. Watchdog used the same fundamental approach for storing and checking metadata, but it extended hardware to reduce the performance overhead to 24% [38]. The indirection in these approaches further increases overhead, whereas C^3 enforces both spatial and temporal safety without relying on metadata stored in memory.

Metadata Tables with Index Embedded in Pointer. CUP places a bounds table index in upper pointer bits, which avoids duplicating bounds table entries and supports direct temporal safety enforcement with a performance overhead of 1.58X [5]. The use of a separate table has poor cache locality, especially in comparison to C^3 that does not rely on metadata in memory.

Capabilities with Metadata Embedded in Pointer. Capabilities allow software to prove its authorization by providing an unforgeable token to that effect [32]. The ability to cover an entire allocation using a single capability rather than requiring a copy of the tag for each granule of memory makes it more feasible to encode expressive security policies into capabilities, *e.g.* deterministically checked bounds that are byte-granular for many allocations, permissions, type specifiers, etc.

Numerous capability machine architectures have been implemented over several decades [32], and CHERI is one recent example with 128-bit pointers [54]. For pointer-intensive software, even doubling the pointer size can impose substantial memory and performance overheads, and it is incompatible with legacy binary code. Furthermore, indicating for each word of memory and register whether it contains a capability or ordinary data (*i.e.* using a tag bit) imposes additional overhead in memory and elsewhere in the design as well as extra software enabling burden.

It is difficult to revoke tagged capabilities to freed allocations, since the capabilities may be scattered widely. For example, this may involve quarantining freed heap allocations until they can be swept up, *e.g.* as in CHERIvoke and Cornucopia [20, 56]. Those impose added overheads on performance (additional 4.7% and 5.8%, respectively) and heap memory usage (*e.g.* additional 33%), which C^3 avoids.

Approaches that do not expand the pointers trade off expressiveness and flexibility, *e.g.* by requiring even moderately-sized allocations to be aligned to coarse-grained boundaries and limiting address space size [29].

CHEx86 hybridizes capabilities and bounds tables, and it maintains the bounds tables for heap allocations beneath the ISA layer to obviate the need for recompilation [48]. This introduces substantial processor complexity, as well as the overheads from separate bounds tables that C^3 avoids by not relying on metadata in memory. CHEx86 imposes 14% performance overhead and 38% memory overhead.

Tripwires. Tripwire mitigations place special markers within memory regions that should never be accessed.

REST inserts a relatively large tripwire at the L2 level and deeper in aligned memory regions and tags corresponding regions at the L1 level [50]. There is a tension between memory overheads from large tripwires and an increased probability of a legitimate data value colliding with a small tripwire.

Califorms uses an inline header specifying such "dead" memory regions with byte-granularity, but it requires even more tag bits at the L1 level and even a per-cacheline tag bit at L2 and deeper levels as well as recompilation [43]. The primary limitation of tripwire approaches is that exploits may bypass tripwires. Recall that non-adjacent overflows have become more common than adjacent overflows [37]. C^3 cryptographically isolates allocations to mitigate both adjacent and non-adjacent overflows.

MemTracker associates metadata with each word of memory, as does MTE, but MemTracker is not a memory tagging mechanism [53]. MemTracker stores metadata in a linearly-mapped table. A variety of state machines can be programmed, e.g. to enforce heap spatial safety by marking chunk headers delimiting heap allocations as tripwires. This imposes an average performance overhead of 1.4% for SPEC CPU2000 with one bit of memory overhead per eight-byte word of data, 1.6%. C^3 provides stronger protections than MemTracker by mitigating non-adjacent buffer overflows and UAF.

Temporal Safety Mechanisms Inspired by Garbage Collection. MarkUs sweeps memory to identify dangling heap pointers, and it prevents freeing memory referenced by dangling pointers [1]. It imposes 10% performance overhead and 16% memory overhead. pSweeper sweeps memory concurrently, with performance overheads of 12.5% and memory overheads of 112.5% when sweeping is performed continuously on spare cores [35]. False negatives can occur in either approach when dangling pointers are hidden, whereas C^3 would still mitigate UAF in such scenarios.

Other Memory Safety Mechanisms. The CheckedPtr3 approach for Chromium places metadata just prior to each allocation and encodes the distance from metadata into the pointer [22]. However, the distance field can overflow rendering memory safety checks impossible. The field also needs to be separately updated each time the pointer is updated, whereas the upper encoded portions of C^3 pointers remain constant.

Duck *et al.* showed how address bits can be overloaded to encode sizes for enforcing spatial safety, but that imposes a rigid layout in the linear address space [15, 16].

C^3 directly mitigates UAF without needing to track object relationships unlike DangNull and DangSan [30, 52].

7 LIMITATIONS AND FUTURE WORK

We are now investigating how to implement C^3 in actual processors. Other potential areas of future work are described below, as well as limitations.

C^3 can be paired with compiler enhancements to harden stack and global variables. The enhanced compiler can insert instructions to encrypt pointers to those variables.

The current lack of encryption for shared data and IO buffers limits mitigating vulnerabilities involving that memory. However, C^3 surpasses prior approaches by encrypting process private memory to help address threats from untrusted processes and accelerators.

Future work can support key sharing for those regions across contexts and with accelerators. For example, C^3 can extend to accelerators to provide a unified cryptographic addressing layer to enforce memory safety atop Shared Virtual Memory (SVM).

8 CONCLUSION

C^3 is the first stateless technique to enforce memory safety without requiring any additional metadata or memory layout changes. It binds data encryption to a cryptographically protected, radix-bound pointer encoding that is unique to each allocation. The use of low-latency ciphers replaces computationally expensive and storage-hungry metadata checks with efficient cryptographic computations. The encrypted pointer format mitigates pointer corruption and forgery without consuming large numbers of pointer bits, in contrast to pointer authentication approaches. C^3 strongly mitigates prevalent categories of vulnerabilities, and it can be extended to mitigate even more with a unified cryptographic approach. C^3, by encrypting pointers and data, goes further by fundamentally extending protections against even physical adversaries and vulnerabilities in hardware by exposing only ciphertext across caches and memory. Detailed performance evaluation shows that C^3 can be implemented in modern high performance processors with negligible performance overheads.

ACKNOWLEDGMENTS

We appreciate the feedback from the anonymous shepherd and other reviewers, Anjo Vahldiek-Oberwagner, and Lauren Biernacki.

REFERENCES

[1] Sam Ainsworth and Timothy M. Jones. 2020. MarkUs: Drop-in use-after-free prevention for low-level languages. In *Proceedings of the 2020 IEEE Symposium on Security and Privacy*. 578–591. https://doi.org/10.1109/SP40000.2020.00058

[2] Ray Beaulieu, Douglas Shors, Jason Smith, Stefan Treatman-Clark, Bryan Weeks, and Louis Wingers. 2013. The SIMON and SPECK Families of Lightweight Block Ciphers. Cryptology ePrint Archive, Report 2013/404. https://eprint.iacr.org/2013/404.

[3] Joe Bialek, Ken Johnson, Matt Miller, and Tony Chen. 2020. *Security Analysis of Memory Tagging*. Technical Report. Microsoft. https://github.com/microsoft/MSRC-Security-Research/blob/master/papers/2020/Security%20analysis%20of%20memory%20tagging.pdf

[4] Tim Boland and Paul E. Black. 2012. Juliet 1.1 C/C++ and Java Test Suite. *Computer* 45, 10 (2012), 88–90.

[5] Nathan Burow, Derrick McKee, Scott A. Carr, and Mathias Payer. 2018. CUP: Comprehensive User-Space Protection for C/C++. In *Proceedings of the 2018 Asia Conference on Computer and Communications Security (ASIACCS '18)*. Incheon, Republic of Korea, 381–392. https://doi.org/10.1145/3196494.3196540

[6] Tony Chen and David Chisnall. 2019. *Pointer Tagging for Memory Safety*. Technical Report. Microsoft. 23 pages. https://www.microsoft.com/en-us/research/uploads/prod/2019/07/Pointer-Tagging-for-Memory-Safety.pdf

[7] Chromium Project. 2020. Memory safety - The Chromium Projects. https://www.chromium.org/Home/chromium-security/memory-safety

[8] Peter Collingbourne. 2019. Add arm64 string.h function implementations for use with hardware supporting MTE. https://android.googlesource.com/platform/bionic/+/900d07d6a1f3e1eca8cdbb3b1db1ceeec0acc9e2

[9] Crispin Cowan, Steve Beattie, John Johansen, and Perry Wagle. 2003. PointGuardTM: Protecting Pointers From Buffer Overflow Vulnerabilities. In *Proceedings of the 12th USENIX Security Symposium*. Washington, DC, USA. https://www.usenix.org/legacy/event/sec03/tech/full_papers/cowan/cowan_html/

[10] Ian Cutress. 2019. Examining Intel's Ice Lake Processors: Taking a Bite of the Sunny Cove Microarchitecture. *AnandTech* (July 2019). https://www.anandtech.com/show/14514/examining-intels-ice-lake-microarchitecture-and-sunny-cove/3

[11] Ian Cutress. 2019. The Ice Lake Benchmark Preview: Inside Intel's 10nm. *AnandTech* (Aug. 2019). https://www.anandtech.com/show/14664/testing-intel-ice-lake-10nm/2

[12] Joan Daemen, Seth Hoffert, Michaël Peeters, Gilles Van Assche, and Ronny Van Keer. 2019. Xoodyak, a lightweight cryptographic scheme. In *NIST Lightweight Cryptography Project*. https://csrc.nist.gov/CSRC/media/Projects/lightweight-cryptography/documents/round-2/spec-doc-rnd2/Xoodyak-spec-round2.pdf

[13] Joan Daemen, Pedro Maat Costa Massolino, and Yann Rotella. 2019. The Subterranean 2.0 cipher suite. In *NIST Lightweight Cryptography Project*. https://csrc.nist.gov/CSRC/media/Projects/lightweight-cryptography/documents/round-2/spec-doc-rnd2/subterranean-spec-round2.pdf

[14] Joe Devietti, Colin Blundell, Milo M. K. Martin, and Steve Zdancewic. 2008. Hardbound: Architectural Support for Spatial Safety of the C Programming Language. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XIII)*. New York, NY, USA, 103–114. https://doi.org/10.1145/1346281.1346295

[15] Gregory J. Duck and Roland H. C. Yap. 2016. Heap Bounds Protection with Low Fat Pointers. In *Proceedings of the 25th International Conference on Compiler Construction (CC '16)*. ACM, New York, NY, USA, 132–142. https://doi.org/10.1145/2892208.2892212

[16] Gregory J. Duck, Roland H. C. Yap, and Lorenzo Cavallaro. 2017. Stack Bounds Protection with Low Fat Pointers. In *Proceedings of the NDSS Symposium 2017*. https://doi.org/10.14722/ndss.2017.23287

[17] Zakir Durumeric, Frank Li, James Kasten, Johanna Amann, Jethro Beekman, Mathias Payer, Nicolas Weaver, David Adrian, Vern Paxson, Michael Bailey, and J. Alex Halderman. 2014. The Matter of Heartbleed. In *Proceedings of the 2014 Internet Measurement Conference (IMC '14)*. Association for Computing Machinery, New York, NY, USA, 475–488. https://doi.org/10.1145/2663716.2663755

[18] Reza Mirzazade Farkhani, Mansour Ahmadi, and Long Lu. 2021. PTAuth: Temporal Memory Safety via Robust Points-to Authentication. In *Proceedings of the 30th USENIX Security Symposium*. https://www.usenix.org/conference/usenixsecurity21/presentation/mirzazade

[19] Adrienne Porter Felt, Richard Barnes, April King, Chris Palmer, Chris Bentzel, and Parisa Tabriz. 2017. Measuring HTTPS adoption on the web. In *Proceedings of the 26th USENIX Conference on Security Symposium (SEC'17)*. USA, 1323–1338.

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

[21] Santosh Ghosh, Michael Kounavis, and Sergej Deutsch. 2020. Gimli Encryption in 715.9 psec. Cryptology ePrint Archive, Report 2020/336. https://eprint.iacr.org/2020/336

[22] haraken. 2020. CheckedPtr2 and CheckedPtr3. https://docs.google.com/document/d/14TsvTgswPUOQuQoI9TmkFQnuSaFD8ZLHRvzapNwl5vs

[23] John L Hennessy and David A Patterson. 2011. *Computer architecture: a quantitative approach*. Elsevier.

[24] Nicolas Joly, Saif ElSherei, and Saar Amar. 2020. *SECURITY ANALYSIS OF CHERI ISA*. Technical Report. Microsoft. https://github.com/microsoft/MSRC-Security-Research/blob/master/papers/2020/Security%20analysis%20of%20CHERI%20ISA.pdf

[25] Y. Kim, J. Lee, and H. Kim. 2020. Hardware-based Always-On Heap Memory Safety. In *53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '20)*. Los Alamitos, CA, USA, 1153–1166. https://doi.org/10.1109/MICRO50266.2020.00095

[26] Andrey Konovalov. 2019. LKML: Andrey Konovalov: [PATCH v19 00/15] arm64: untag user pointers passed to the kernel. https://lkml.org/lkml/2019/7/23/728

[27] M. Kounavis, S. Deutsch, S. Ghosh, and D. Durham. 2020. K-Cipher: A Low Latency, Bit Length Parameterizable Cipher. In *2020 IEEE Symposium on Computers and Communications (ISCC '20)*. https://doi.org/10.1109/ISCC50000.2020.9219582 ISSN: 2642-7389.

[28] Michael E. Kounavis, Xiaozhu Kang, Ken Grewal, Mathew Eszenyi, Shay Gueron, and David Durham. 2010. Encrypting the internet. *ACM SIGCOMM Computer Communication Review* 40, 4 (Aug. 2010), 135–146. https://doi.org/10.1145/1851275.1851200

[29] Albert Kwon, Udit Dhawan, Jonathan M. Smith, Thomas F. Knight, Jr., and Andre DeHon. 2013. Low-fat Pointers: Compact Encoding and Efficient Gate-level Implementation of Fat Pointers for Spatial Safety and Capability-based Security. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security (CCS '13)*. New York, NY, USA, 721–732. https://doi.org/10.1145/2508859.2516713

[30] Byoungyoung Lee, Chengyu Song, Yeongjin Jang, Tielei Wang, Taesoo Kim, Long Lu, and Wenke Lee. 2015. Preventing Use-after-free with Dangling Pointers Nullification. In *Proceedings of the NDSS Symposium 2015*. ISOC. https://www.ndss-symposium.org/ndss2015/ndss-2015-programme/preventing-use-after-free-dangling-pointers-nullification/

[31] Jongmin Lee and Soontae Kim. 2012. Adopting TLB index-based tagging to data caches for tag energy reduction. In *Proceedings of the 2012 ACM/IEEE International Symposium on Low Power Electronics and Design*. 231–236.

[32] Henry M. Levy. 1984. *Capability-based Computer Systems*. Digital Press. https://homes.cs.washington.edu/~levy/capabook/

[33] Hans Liljestrand, Thomas Nyman, Kui Wang, Carlos Chinea Perez, Jan-Erik Ekberg, and N. Asokan. 2019. PAC it up: Towards Pointer Integrity using ARM Pointer Authentication. In *Proceedings of the 28th USENIX Security Symposium*. 177–194. https://www.usenix.org/conference/usenixsecurity19/presentation/liljestrand

[34] Moritz Lipp, Vedad Hažić, Michael Schwarz, Arthur Perais, Clémentine Maurice, and Daniel Gruss. 2020. Take A Way: Exploring the Security Implications of AMD's Cache Way Predictors. In *Proceedings of the 15th ACM Asia Conference on Computer and Communications Security*. 813–825.

[35] Daiping Liu, Mingwei Zhang, and Haining Wang. 2018. A Robust and Efficient Defense against Use-after-Free Exploits via Concurrent Pointer Sweeping. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS '18)*. Toronto, Canada, 1635–1648. https://doi.org/10.1145/3243734.3243826

[36] Peter S. Magnusson, Magnus Christensson, Jesper Eskilson, Daniel Forsgren, Gustav Hållberg, Johan Högberg, Fredrik Larsson, Andreas Moestedt, and Bengt Werner. 2002. Simics: A full system simulation platform. *Computer* 35, 2 (Feb. 2002), 50–58. https://doi.org/10.1109/2.982916

[37] Matt Miller. 2019. Trends, Challenges, and Strategic Shifts in the Software Vulnerability Mitigation Landscape. https://www.youtube.com/watch?v=PjbGojjnBZQ.

[38] Santosh Nagarakatte, Milo M. K. Martin, and Steve Zdancewic. 2012. Watchdog: hardware for safe and secure manual memory management and full memory safety. *SIGARCH Comput. Archit. News* 40, 3 (June 2012), 189–200. https://doi.org/10.1145/2366231.2337181

[39] Santosh Nagarakatte, Jianzhou Zhao, Milo M.K. Martin, and Steve Zdancewic. 2010. CETS: Compiler Enforced Temporal Safety for C. In *Proceedings of the 2010 International Symposium on Memory Management (ISMM '10)*. New York, NY, USA, 31–40. https://doi.org/10.1145/1806651.1806657

[40] Intel® Newsroom. 2020. Intel Launches World's Best Processor for Thin-and-Light Laptops: 11th Gen Intel Core. https://newsroom.intel.com/news-releases/11th-gen-tiger-lake-evo/

[41] NIST Information Technology Laboratory Computer Security Resource Center. [n. d.]. Lightweight Cryptography. https://csrc.nist.gov/projects/lightweight-cryptography

[42] Oleksii Oleksenko, Dmitrii Kuvaiskii, Pramod Bhatotia, Pascal Felber, and Christof Fetzer. 2018. Intel MPX Explained: A Cross-Layer Analysis of the Intel MPX

System Stack. *Proceedings of the ACM on Measurement and Analysis of Computing Systems* 2, 2, Article 28 (June 2018). https://doi.org/10.1145/3224423

[43] Hiroshi Sasaki, Miguel A. Arroyo, M. Tarek Ibn Ziad, Koustubha Bhat, Kanad Sinha, and Simha Sethumadhavan. 2019. Practical Byte-Granular Memory Blacklisting using Califorms. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. Columbus OH USA, 558–571. https://doi.org/10.1145/3352460.3358299

[44] Kostya Serebryany. 2019. ARM Memory Tagging Extension and How It Improves C/C++ Memory Safety. *USENIX ;login:* 44, 2 (2019), 5. https://www.usenix.org/publications/login/summer2019/serebryany

[45] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. 2012. AddressSanitizer: A Fast Address Sanity Checker. In *Proceedings of the 2012 USENIX Annual Technical Conference (USENIX ATC '12)*. Boston, MA, 309–318. https://www.usenix.org/conference/atc12/technical-sessions/presentation/serebryany

[46] Kostya Serebryany, Evgenii Stepanov, Aleksey Shlyapnikov, Vlad Tsyrklevich, and Dmitry Vyukov. 2018. Memory Tagging and how it improves C/C++ memory safety. *arXiv* (Feb. 2018). http://arxiv.org/abs/1802.09517

[47] André Seznec. 2011. A new case for the TAGE branch predictor. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*. 117–127.

[48] Rasool Sharifi and Ashish Venkat. 2020. CHEx86: Context-Sensitive Enforcement of Memory Safety via Microcode-Enabled Capabilities. In *ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. Valencia, Spain, 762–775. https://doi.org/10.1109/ISCA45697.2020.00068

[49] Etienne Sicard. 2017. *Introducing 7-nm FinFET technology in Microwind*. https://hal.archives-ouvertes.fr/hal-01558775

[50] K. Sinha and S. Sethumadhavan. 2018. Practical Memory Safety with REST. In *ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. 600–611. https://doi.org/10.1109/ISCA.2018.00056

[51] Nathan Tuck, Brad Calder, and George Varghese. 2004. Hardware and Binary Modification Support for Code Pointer Protection From Buffer Overflow. In *37th International Symposium on Microarchitecture (MICRO-37)*. 209–220. https://doi.org/10.1109/MICRO.2004.20 ISSN: 1072-4451.

[52] Erik van der Kouwe, Vinod Nigade, and Cristiano Giuffrida. 2017. DangSan: Scalable Use-after-free Detection. In *Proceedings of the Twelfth European Conference on Computer Systems (EuroSys '17)*. Belgrade, Serbia, 405–419. https://doi.org/10.1145/3064176.3064211

[53] G. Venkataramani, B. Roemer, Y. Solihin, and M. Prvulovic. 2007. MemTracker: Efficient and Programmable Support for Memory Access Monitoring and Debugging. In *IEEE 13th International Symposium on High Performance Computer Architecture (HPCA)*. http://dx.doi.org/10.1109/HPCA.2007.346205

[54] Robert NM Watson, Jonathan Woodruff, Peter G. Neumann, Simon W. Moore, Jonathan Anderson, David Chisnall, Nirav Dave, Brooks Davis, Khilan Gudka, Ben Laurie, and others. 2015. Cheri: A hybrid capability-system architecture for scalable software compartmentalization. In *IEEE Symposium on Security and Privacy*. https://www.ieee-security.org/TC/SP2015/papers/6949a020.pdf

[55] WikiChip [n. d.]. *Ice Lake (client) - Microarchitectures - Intel*. WikiChip. https://en.wikichip.org/wiki/intel/microarchitectures/sunny_cove

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

[57] Tong Zhang, Dongyoon Lee, and Changhee Jung. 2019. BOGO: Buy Spatial Memory Safety, Get Temporal Memory Safety (Almost) Free. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '19)*. Providence, RI, USA, 631–644. https://doi.org/10.1145/3297858.3304017

[58] Tianhao Zheng, Haishan Zhu, and Mattan Erez. 2018. SIPT: Speculatively indexed, physically tagged caches. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 118–130.