Practical Byte-Granular Memory Blacklisting using Califorms

Hiroshi Sasaki
Columbia University
sasaki@cs.columbia.edu

Miguel A. Arroyo
Columbia University
miguel@cs.columbia.edu

M. Tarek Ibn Ziad
Columbia University
mtarek@cs.columbia.edu

Koustubha Bhat[†]
Vrije Universiteit Amsterdam
k.bhat@vu.nl

Kanad Sinha
Columbia University
kanad@cs.columbia.edu

Simha Sethumadhavan
Columbia University
simha@cs.columbia.edu

ABSTRACT

Recent rapid strides in memory safety tools and hardware have improved software quality and security. While coarse-grained memory safety has improved, achieving memory safety at the granularity of individual objects remains a challenge due to high performance overheads usually between ~1.7x–2.2x. In this paper, we present a novel idea called *Califorms*, and associated program observations, to obtain a low overhead security solution for practical, byte-granular memory safety.

The idea we build on is called memory blacklisting, which prohibits a program from accessing certain memory regions based on program semantics. State of the art hardware-supported memory blacklisting, while much faster than software blacklisting, creates memory fragmentation (on the order of few bytes) for each use of the blacklisted location. We observe that metadata used for blacklisting can be stored in dead spaces in a program's data memory and that this metadata can be integrated into the microarchitecture by changing the cache line format. Using these observations, a Califorms based system proposed in this paper reduces the performance overheads of memory safety to ~1.02x–1.16x while providing byte-granular protection and maintaining very low hardware overheads. Moreover, the fundamental idea of storing metadata in empty spaces and changing cache line formats can be used for other security and performance applications.

CCS CONCEPTS

• **Security and privacy → Security in hardware**; • **Computer systems organization → Architectures**.

KEYWORDS

memory safety, memory blacklisting, caches

ACM Reference Format:
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 *The 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-52), October 12–16, 2019, Columbus, OH, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3352460.3358299

[†]Part of this work was carried out while the author was a visiting student at Columbia University.

Permission to make digital or hard copies of all or part 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 components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
MICRO-52, October 12–16, 2019, Columbus, OH, USA
© 2019 Association for Computing Machinery.
ACM ISBN 978-1-4503-6938-1/19/10…$15.00
https://doi.org/10.1145/3352460.3358299

1 INTRODUCTION

Historically, program memory safety violations have provided a significant opportunity for exploitation by attackers: for instance, Microsoft recently revealed that the root cause of more than half of all exploits are software memory safety violations [30]. To address this threat, software checking tools such as AddressSanitizer [25] and commercial hardware support for memory safety such as Oracle's ADI [21] and Intel's MPX [20] have enabled programmers to detect and fix memory safety violations before deploying software.

Current software and hardware-supported solutions excel at providing coarse-grained, inter-object memory safety which involves detecting memory access beyond arrays and heap allocated regions (`malloc`'d struct and class instances). However, they are not suitable for fine-grained memory safety (i.e., intra-object memory safety—detecting overflows within objects, such as fields within a struct, or members within a class) due to the high performance overheads and/or need for making intrusive changes to the source code [29]. Some real-world scenarios where intra-object memory safety problems manifest are type confusion vulnerabilities (e.g., CVE–2017–5115 [2]) and uninitialized data leaks through padding bytes (e.g., CVE–2014–1444 [1]), both recognized as high-impact security classes [13, 16].

For instance, a recent work that aims to provide intra-object overflow protection functionality incurs a 2.2x performance overhead [10]. These overheads are problematic because they not only reduce the number of pre-deployment tests that can be performed, but also impede post-deployment continuous monitoring, which researchers have pointed out is necessary for detecting benign and malicious memory safety violations [26]. Thus, a low overhead memory safety solution that can enable continuous monitoring and provide complete program safety has been elusive.

The source of overheads stem from how current designs store and use metadata necessary for enforcing memory safety. In Intel MPX [20], Hardbound [8], CHERI [31, 32], and PUMP [9], the metadata is stored for each pointer; each data or code memory access through a pointer performs checks using the metadata. Since C/C++ memory accesses tend to be highly pointer based, the performance and energy overheads of accessing metadata can be significant in such systems. Furthermore, the management of metadata, especially if it is stored in a disjoint manner from the pointer, can also create significant engineering complexity in terms of performance

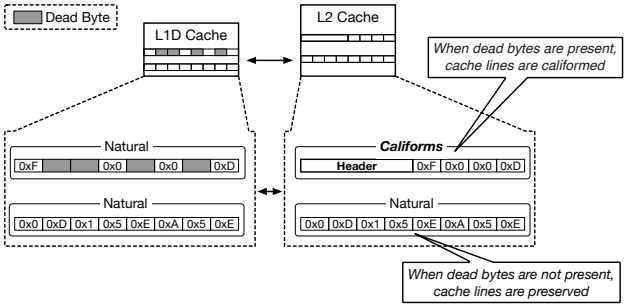


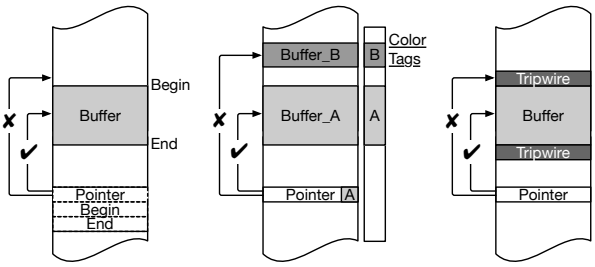
Figure 1: Califorms offers memory safety by detecting accesses to dead bytes in memory. Dead bytes are not stored beyond the L1 data cache and identified using a special header in the L2 cache (and beyond) resulting in very low overhead. The conversion between these formats happens when lines are filled or spilled between the L1 and L2 caches. The absence of dead bytes results in the cache lines stored in the same natural format across the memory system.

and usability. This was evidenced by the fact that compilers like LLVM and GCC dropped support for Intel MPX in their mainline after an initial push to integrate it into the toolchain [20].

Our approach for reducing overheads is two-fold. First, instead of checking access bounds for each pointer access, we blacklist all memory locations that should never be accessed. This reduces the additional work for memory safety such as comparing bounds. Second, we propose a novel metadata storage scheme for storing blacklisted information. We observe that by using dead memory spaces in the program, we can store metadata needed for memory safety for free for nearly half of the program objects. These dead spaces can occur for several reasons including language alignment requirements. When we cannot find naturally occurring dead spaces, we manually insert them. The overhead due to this dead space is smaller than traditional methods for storing metadata because of how we represent the metadata: our metadata is smaller (one byte) as opposed to multiple bytes with traditional whitelisting or blacklisting memory safety techniques.

A natural question is how the dead (more commonly referred to as *padding*) bytes can be distinguished from normal bytes in memory. A straightforward scheme results in one bit of additional storage per byte to identify if a byte is a dead byte; this scheme results in a space overhead of 12.5%. We reduce this overhead to one bit per 64B cache line (0.2% overhead) without any loss of precision by only reformatting how data is stored in cache lines. Our technique, *Califorms*, uses one bit of additional storage to identify if the cache line associated with the memory contains any dead bytes. For *califormed* cache lines, i.e., lines which contain dead bytes, the actual data is stored following the "header", which indicates the location of dead bytes, as shown in Figure 1.

With this support, it is easy to describe how a Califorms based system for memory safety works. Dead bytes, either naturally harvested or manually inserted, are used to indicate memory regions that should never be accessed by a program (i.e., blacklisting). If an attacker accesses these regions, we detect this rogue access without any additional metadata accesses as our metadata resides inline.



(a) Disjoint metadata whitelisting. **(b) Cojoined metadata whitelisting.** **(c) Inlined metadata blacklisting.**

Figure 2: Three main classes of hardware solutions for memory safety.

In theory, perfect blacklisting is a strictly weaker form of security than perfect whitelisting, but blacklisting is a more practical alternative because of its ease of deployment and low overheads. Informally, whitelisting techniques are applied partially to reduce overheads and maintain backward compatibility which reduces their security, while blacklisting techniques can be applied more broadly and thus offer more coverage due to their low overheads. Additionally, blacklisting techniques complement defenses in existing systems better since they do not require intrusive changes.

Our experimental results on the SPEC CPU2006 benchmark suite indicate that the overheads of Califorms are quite low: software overheads range from 2 to 14% slowdown (or alternatively, 1.02x to 1.16x performance overhead) depending on the amount and location of padding bytes used. Varying the number of padding bytes provides functionality for the user/customer to tune the security according to their performance requirements. Hardware induced overheads are also negligible, on average less than 1%. All of the software transformations are performed using the LLVM compiler framework using a front-end source-to-source transformation. These overheads are substantially lower compared to the state-of-the-art software or hardware supported schemes (viz., 2.2x performance and 1.1x memory overheads for EffectiveSan [10], and 1.7x performance and 2.1x memory overheads for Intel MPX [20]).

2 BACKGROUND

Hardware support for memory safety can be broadly categorized into the following three classes as presented in Figure 2.

Disjoint Metadata Whitelisting. This class of techniques, also called base and bounds, attaches bounds metadata with every pointer, bounding the region of memory they can legitimately dereference (see Figure 2(a)). Hardbound [8], proposed in 2008, provides spatial memory safety using this mechanism. Intel MPX [20], productized in 2016, is similar and introduces an explicit architectural interface (registers and instructions) for managing bounds information. Temporal memory safety was introduced to this scheme by storing an additional "version" information along with the pointer metadata and verifying that no stale versions are ever retrieved [18, 19]. BOGO [33] adds temporal memory safety to MPX by invalidating all pointers to freed regions in MPX's lookup table. Introduced about 35 years ago in commercial chips like Intel 432 and IBM System/38, CHERI [32] revived capability based

architectures with similar bounds-checking guarantees, in addition to having other metadata fields (e.g., permissions).[1] PUMP [9], on the other hand, is a general-purpose framework for metadata propagation, and can be used for propagating pointer bounds.

One advantage of per pointer metadata stored separately from the pointer in a shadow memory region is that it allows compatibility with codes that use legacy pointer layouts. Typically, metadata storage overhead scales according to the number of pointers in principle but implementations generally reserve a fixed chunk of memory for easy lookup. Owing to this disjoint nature, metadata access requires additional memory operations, which some proposals seek to minimize with caching and other optimizations. Regardless, disjoint metadata introduces atomicity concerns potentially resulting in false positives and negatives or complicating coherence designs at the least (e.g., MPX is not thread-safe). Explicit specification of bounds per pointer also allows bounds-narrowing in principle, wherein pointer bounds can be tailored to protect individual elements in a composite memory object. However, commercial compilers do not support this feature for MPX due to the complexity of compiler analyses required. Furthermore, compatibility issues with untreated modules (e.g., unprotected libraries) also introduces real-world deployability concerns for these techniques. For instance MPX drops its bounds when protected pointers are modified by unprotected modules, while CHERI does not support it at all. MPX additionally makes bounds checking explicit, thus introducing a marginal computational overhead to bounds management as well.

Cojoined Metadata Whitelisting. Originally introduced in the IBM System/370 mainframes, this mechanism assigns a "color" to memory chunks when they are allocated, and the same color to the pointer used to access that region. The runtime check for access validity simply consists of comparing the colors of the pointer and accessed memory (see Figure 2(b)).

This technique is currently commercially deployed by Oracle as ADI [21],[2] which uses higher order bits in pointers to store the color. In ADI, color information associated with memory is stored in dedicated per line metadata bits while in cache and in extra ECC bits while in memory [26]. The use of ECC bits creates a restriction on the number of colors, however, if the colors can fit into ECC, metadata storage does not occupy any additional memory in the program's address space.[3] Additionally, since metadata bits are acquired along with concomitant data, extra memory operations are obviated. For the same reason, it is also compatible with unprotected modules since the checks are implicit as well. Temporal safety is achieved by assigning a different color when memory regions are reused. However, intra-object protection or bounds-narrowing is not supported as there is no means for "overlapping" colors. Furthermore, protection is also dependent on the number of metadata bits employed, since it determines the number of colors that can be assigned. So, while color reuse allows ADI to scale and limit metadata storage overhead, it can also be exploited by this vector. Another disadvantage of this technique, specifically due to inlining metadata in pointers, is that it only supports 64-bit architectures. Narrower pointers would not have enough spare bits to accommodate color information.

Inlined Metadata Blacklisting. Another line of work, also referred to as tripwires, aims to detect overflows by simply blacklisting a patch of memory on either side of a buffer, and flagging accesses to this patch (see Figure 2(c)). This is very similar to contemporary canary design [6], but there are a few critical differences. First, canaries only detect overwrites, not overreads. Second, hardware tripwires trigger instantaneously, whereas canaries need to be periodically checked for integrity, providing a period of attack to time of use window. Finally, unlike hardware tripwires, canary values can be leaked or tampered, and thus mimicked.

SafeMem [23] implements tripwires by repurposing ECC bits in memory to mark memory regions invalid, thus trading off reliability for security. On processors supporting speculative execution, however, it might be possible to speculatively fetch blacklisted lines into the cache without triggering a faulty memory exception. Unless these lines are flushed immediately after, SafeMem's blacklisting feature can be trivially bypassed. Alternatively, REST [28] achieves the same by storing a predetermined large random number, in the form of a 8–64B token, in the memory to be blacklisted. Violations are detected by comparing cache lines with the token when they are fetched. REST provides temporal memory safety by quarantining freed memory, and not reusing them for subsequent allocations. Compatibility with unprotected modules is easily achieved as well, since tokens are part of the program's address space and all access are implicitly checked. However, intra-object safety is not supported by REST owing to fragmentation overhead such heavy usage of tokens would entail.

Since Califorms operates on the principle of detecting memory accesses to dead bytes, which are in turn stored along with program data, it belongs to the inlined metadata class of defenses. However, it is different from other works in the class in one key aspect—granularity. While both REST and SafeMem naturally blacklist at the cache line granularity, Califorms can do so at the byte granularity. It is this property that enables us to provide intra-object safety with negligible performance and memory overheads, unlike previous work in the area. For inter-object spatial safety and temporal safety, we employ the same design principles as REST. Hence, our safety guarantees are a *strict superset* of those provided by previous schemes in this class (spatial memory safety by blacklisting and temporal memory safety by quarantining).

3 MOTIVATION

One of the key ways in which we mitigate the overheads for fine-grained memory safety is by opportunistically harvesting padding bytes in programs to store metadata. So how often do these occur in programs? Before we answer the question let us concretely understand padding bytes with an example. Consider the `struct A` defined in Listing 1(a). Let us say the compiler inserts a three-byte padding in between `char c` and `int i` as in Listing 1(b) because of the C language requirement that integers should be padded to their natural size (which we assume to be four bytes here). These types of paddings are not limited to C/C++ but also required by many

[1] A recent version of CHERI [31], however, manages to compress metadata to 128 bits and changes pointer layout to store it with the pointer value (i.e., implementing base and bounds as cojoined metadata whitelisting), accordingly introducing instructions to manipulate them specifically.

[2] ARM has a similar upcoming Memory Tagging feature [3], whose implementation details are unclear, as of this work.

[3] However, when a memory is swapped color bits are copied into memory by the OS.

```
struct A {
  char c;
  int i;
  char buf[64];
  void (*fp)();
}
```

(a) Original.

```
struct A_opportunistic {
  char c;
  /* compiler inserts padding
   * bytes for alignment */
  char padding_bytes[3];
  int i;
  char buf[64];
  void (*fp)();
}
```

(b) Opportunistic.

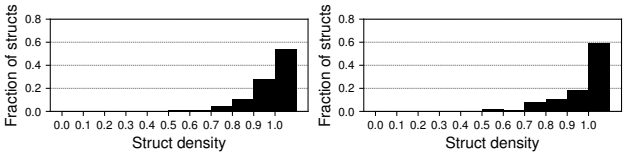
```
struct A_full {
  /* we protect every field with
   * random security bytes */
  char security_bytes[2];
  char c;
  char security_bytes[1];
  int i;
  char security_bytes[3];
  char buf[64];
  char security_bytes[2];
  void (*fp)();
  char security_bytes[1];
}
```

(c) Full.

```
struct A_intelligent {
  char c;
  int i;
  /* we protect boundaries
   * of arrays and pointers with
   * random security bytes */
  char security_bytes[3];
  char buf[64];
  char security_bytes[2];
  void (*fp)();
  char security_bytes[3];
}
```

(d) Intelligent.

Listing 1: (a) Original source code and examples of three security bytes harvesting strategies: (b) *opportunistic* uses the existing padding bytes as security bytes, (c) *full* protect every field within the struct with security bytes, and (d) *intelligent* surrounds arrays and pointers with security bytes.



(a) SPEC CPU2006 C and C++ benchmarks. **(b) V8 JavaScript engine.**

Figure 3: Struct density histogram of SPEC CPU2006 benchmarks and the V8 JavaScript engine. More than 40% of the structs have at least one padding byte.

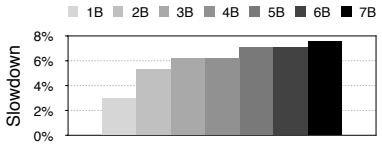


Figure 4: Average performance overhead with additional paddings (one byte to seven bytes) inserted for every field within structs (and classes) of SPEC CPU2006 C and C++ benchmarks.

other languages and their runtime implementations. To obtain a quantitative estimate on the amount of paddings, we developed a compiler pass to statically collect the padding size information. Figure 3 presents the histogram of struct densities for SPEC CPU2006 C and C++ benchmarks and the V8 JavaScript engine. Struct density is defined as the sum of the size of each field divided by the total size of the `struct` including the padding bytes (i.e., the smaller or sparse the struct density the more padding bytes the struct has). The results reveal that 45.7% and 41.0% of structs within SPEC and V8, respectively, have at least one byte of padding. This is encouraging since even without introducing additional padding bytes (no memory overhead), we can offer protection for certain compound data types restricting the remaining attack surface.

Naturally, one might inquire about the safety for the rest of the program. To offer protection for all defined compound data types, we can insert random sized padding bytes, also referred to as security bytes, between every field of a `struct` or member of a `class` as in Listing 1(c) (full strategy). Random sized security bytes are chosen to provide a probabilistic defense as fixed sized security bytes can be jumped over by an attacker once she identifies the actual size (and the exact memory layout). By carefully choosing the minimum and maximum of random sizes, we can keep the average size of security bytes small (few bytes). Intuitively, the higher the unpredictability (or randomness) within the memory layout, the higher the security level we can offer.

While the full strategy provides the widest coverage, not all of the security bytes provide the same security utility. For example, basic data types such as `char` and `int` cannot be easily overflowed past their bounds. The idea behind the intelligent insertion strategy is to prioritize insertion of security bytes into security-critical locations as shown in Listing 1(d). We choose data types which are most prone to abuse by an attacker via overflow type accesses: (1) arrays and (2) data and function pointers. In Listing 1(d), the array `buf[64]` and the function pointer `fp` are protected with random sized security bytes. While it is possible to utilize padding bytes present between other data types without incurring memory overheads, doing so would come at an additional performance overhead.

In comparison to opportunistic harvesting, the other more secure strategies (full and intelligent) come at an additional performance overhead. We analyze the performance trend in order to decide how many security bytes can be reasonably inserted. For this purpose we developed an LLVM pass which pads every field of a `struct` and member of a `class` with fixed size paddings. We measure the performance of SPEC CPU2006 benchmarks by varying the padding size from one byte to seven bytes (since eight bytes is the finest granularity that state-of-the-art technique can offer [28]). The detailed evaluation environment and methodology are described later in Section 10.

Figure 4 demonstrates the average slowdown when inserting additional bytes for harvesting. As expected, we can see the performance overheads grow as we increase the padding size, mainly due to ineffective cache usage. On average the slowdown is 3.0% for one byte and 7.6% for seven bytes of padding. The figure presents the ideal (lower bound) performance overhead when fully inserting security bytes into compound data types; the hardware and software modifications we introduce add additional overheads on top of these numbers. We strive to provide a mechanism that allows the user to tune the security level at the cost of performance and thus explore several security byte insertion strategies to reduce the performance overhead in the paper.

4 THREAT MODEL

We assume a threat model comparable to that used in contemporary related works [28, 31, 32]. We assume the victim program to have one or more vulnerabilities that an attacker can exploit to gain arbitrary read and write capabilities in the memory; our goal is to prevent both spatial and temporal memory violations. Furthermore, we assume that the adversary has access to the source code of the program, therefore she is able to glean all source-level information. However, she does not have access to the host binary (e.g., server-side applications). Finally, we assume that all hardware is trusted—it does not contain and/or is not subject to bugs arising from exploits such as physical or glitching attacks. Due to its recent rise in relevance however, we maintain side-channel attacks in our design of Califorms within the purview of our threats. Specifically, we accommodate attack vectors seeking to leak the location and value of security bytes.

5 FULL SYSTEM OVERVIEW

The Califorms framework consists of multiple components we discuss in the following sections:

Architecture Support. A new instruction called `BLOC`, mnemonic for Blacklist LOCations, that blacklists memory locations at byte granularity and raises a privileged exception upon misuse of blacklisted locations (Section 6).

Microarchitecture Design. New cache line formats, or Califorms, that enable low cost access to the metadata—we propose different Califorms for L1 cache vs. L2 cache and beyond (Section 7).

Software Design. Compiler, memory allocator and operating system extensions which insert the security bytes at compile time and manages them via the `BLOC` instruction at runtime (Section 8).

At compile time, each compound data type (`struct` or `class`) is examined and security bytes are added according to a user defined insertion policy viz. opportunistic, full or intelligent, by a source-to-source translation pass. At execution time when compound data type instances are dynamically created in the heap, we use a new version of `malloc` that issues `BLOC` instructions to arrange the security bytes after the space is allocated. When the `BLOC` instruction is executed, the cache line format is transformed at the L1 cache controller (assuming a cache miss) and is inserted into the L1 data cache. Upon an L1 eviction, the L1 cache controller transforms the cache line to meet the Califorms of the L2 cache.

While we add additional metadata storage to the caches, we refrain from doing so for main memory and persistent storage to keep the changes local within the CPU core. When a califormed cache line is evicted from the last-level cache to main memory, we keep the cache line califormed and store the additional one metadata bit into spare ECC bits similar to Oracle's ADI [21, 26].[4] When a page is swapped out from main memory, the page fault handler stores the metadata for all the cache lines within the page into a reserved address space managed by the operating system; the metadata is reclaimed upon swap in. Therefore, our design keeps the cache line format califormed throughout the memory hierarchy. A califormed cache line is un-califormed only when the corresponding bytes cross the boundary where the califormed data cannot be understood by the other end, such as writing to I/O (e.g., pipe, filesystem or network socket). Finally, when an object is freed, the freed bytes are filled with security bytes and quarantined for offering temporal memory safety. At runtime, when a rogue load or store accesses a security byte the hardware returns a privileged, precise security exception to the next privilege level which can take any appropriate action including terminating the program.

[4] ADI stores four bits of metadata per cache line for allocation granularity enforcement while Califorms stores one bit for sub-allocation granularity enforcement.

Table 1: `BLOC` instruction K-map. X represents "Don't Care".

		R2, R3		
		X, $\overline{\text{Allow}}$	$\overline{\text{Set}}$, Allow	Set, Allow
Initial	Regular Byte	Regular Byte	Exception	Security Byte
	Security Byte	Security Byte	Regular Byte	Exception

6 ARCHITECTURE SUPPORT

6.1 BLOC Instruction

The format of the instruction is "`BLOC R1, R2, R3`". The value in register `R1` points to the starting (64B cache line aligned) address in the virtual address space, denoting the start of the 64B chunk which fits in a single cache line. Table 1 represents a K-map for the `BLOC` instruction. The value in register `R2` indicates the attributes of said region represented in a bit vector format (`1` to set and `0` to unset the security byte). The value in register `R3` is a mask to the corresponding 64B region, where `1` allows and `0` disallows changing the state of the corresponding byte. The mask is used to perform partial updates of metadata within a cache line. We throw a privileged exception when the `BLOC` instruction tries to set a security byte to an existing security byte, or unset a security byte from a normal byte.

The `BLOC` instruction is treated similarly to a store instruction in the processor pipeline since it modifies the architectural state of data bytes in a cache line. It first fetches the corresponding cache line into the L1 data cache upon an L1 miss (assuming a write allocate cache policy). Next, it manipulates the bits in the metadata storage to appropriately set or unset the security bytes.

6.2 Privileged Exceptions

When the hardware detects an access violation (i.e., access to a security byte), it throws a privileged exception once the instruction becomes non-speculative. There are some library functions which violate the aforementioned operations on security bytes such as `memcpy` so we need a way to suppress the exceptions. In order to whitelist such functions, we manipulate the exception mask registers and let the exception handler decide whether to suppress the exception or not. Although privileged exception handling is more expensive than handling user-level exceptions (because it requires a context switch to the kernel), we stick with the former to limit the attack surface. We rely on the fact that the exception itself is a rare event and would have negligible effect on performance.

7 MICROARCHITECTURE DESIGN

The microarchitectural support for our technique aims to keep the common case fast: L1 cache uses the straightforward scheme

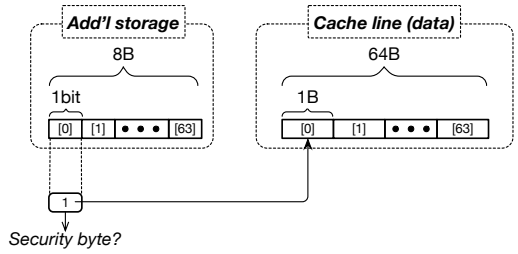


Figure 5: Califorms-bitvector: L1 Califorms implementation using a bit vector that indicates whether each byte is a security byte. HW overhead of 8B per 64B cache line.

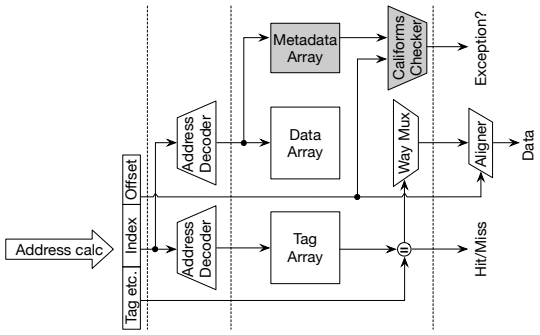


Figure 6: Pipeline diagram for the L1 cache hit operation. The shaded blocks correspond to Califorms components.

of having one bit of additional storage per byte. All califormed cache lines are converted to the straightforward scheme at the L1 data cache controller so that typical loads and stores which hit in the L1 cache do not have to perform address calculations to figure out the location of original data (which is required for Califorms of L2 cache and beyond). This design decision guarantees that the common case latencies will not be affected due to security functionality. Beyond the L1, the data is stored in the optimized califormed format, i.e., one bit of additional storage for the entire cache line. The transformation happens when the data is filled in or spilled from the L1 data cache (between the L1 and L2), and adds minimal latency to the L1 miss latency.

7.1 L1 Cache: Bit Vector Approach

To satisfy the L1 design goal we consider a naive (but low latency) approach which uses a bit vector to identify which bytes are security bytes in a cache line. Each bit of the bit vector corresponds to each byte of the cache line and represents its state (normal byte or security byte). Figure 5 presents a schematic view of this implementation *califorms-bitvector*. The bit vector requires a 64-bit (8B) bit vector per 64B cache line which adds 12.5% storage overhead for the L1 data cache (comparable to ECC overhead for reliability).

Figure 6 shows the L1 data cache hit path modifications for Califorms. If a load accesses a security byte (which is determined by reading the bit vector) an exception is recorded to be processed when the load is ready to be committed. Meanwhile, the load returns a pre-determined value for the security byte (in our design the value zero which is the value that the memory region is initialized to upon deallocation). Returning this fixed value is meant to be

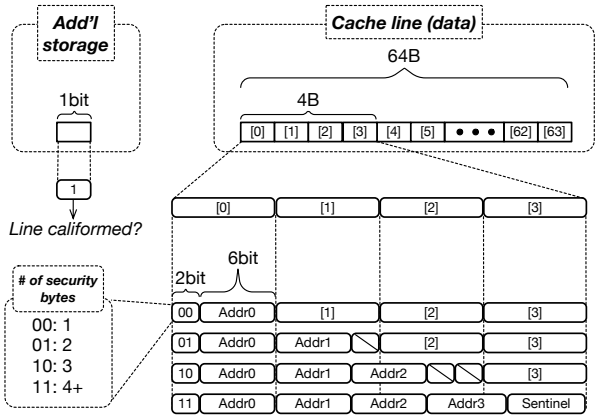


Figure 7: Califorms-sentinel that stores a bit vector in security byte locations. HW overhead of 1-bit per 64B cache line.

a countermeasure against speculative side-channel attacks that seek to identify security byte locations (discussed in greater detail in Section 9). On store accesses to security bytes, we report an exception when the store commits.

7.2 L2 Cache and Beyond: Sentinel Approach

For L2 and beyond, we take a different approach that allows us to recognize whether each byte is a security byte with fewer bits, as using the L1 metadata format throughout the system will increase the cache area overhead by 12.5%, which may not be acceptable. We propose *califorms-sentinel*, which has a 1-bit or 0.2% metadata overhead per 64B cache line. For main memory, we store the additional bit per cache line size in the DRAM ECC spare bits, thus completely removing any cycle time impact on DRAM access or modifications to the DIMM architecture.

The key insight that enables the savings is the following observation: *the number of bits required to address all the bytes in a cache line, which is six bits for a 64 byte cache line, is less than a single byte.* For example, let us assume that there is (at least) one security byte in a 64B cache line. Considering a byte granular protection there are at most 63 unique values (bytes) that non-security bytes can have. Therefore, we are guaranteed to find a six bit pattern that is not present in any of the normal bytes', for instance least significant, six bits. We use this pattern as a sentinel value to represent the security bytes in the cache line. Now if we store this six bit (sentinel value) as additional metadata, the storage overhead will be seven bits (six bits plus one bit to specify if the cache line is califormed) per cache line. In this paper we further propose a new cache line format which stores the sentinel value within a security byte to reduce the metadata overhead down to one bit per cache line.

As presented in Figure 7, califorms-sentinel stores the metadata into the first four bytes (at most) of the 64B cache line. Two bits of the first (0th) byte are used to specify the number of security bytes within the cache line: `00`, `01`, `10` and `11` represent one, two, three, and four or more security bytes, respectively. The sentinel is used only when we have more than four security bytes. If there is only one security byte in the cache line, we use the remaining six bits of the 0th byte to specify the location of the security byte, and the original value of the 0th byte is stored in the security byte. Similarly when there is two or three security bytes in the cache line,

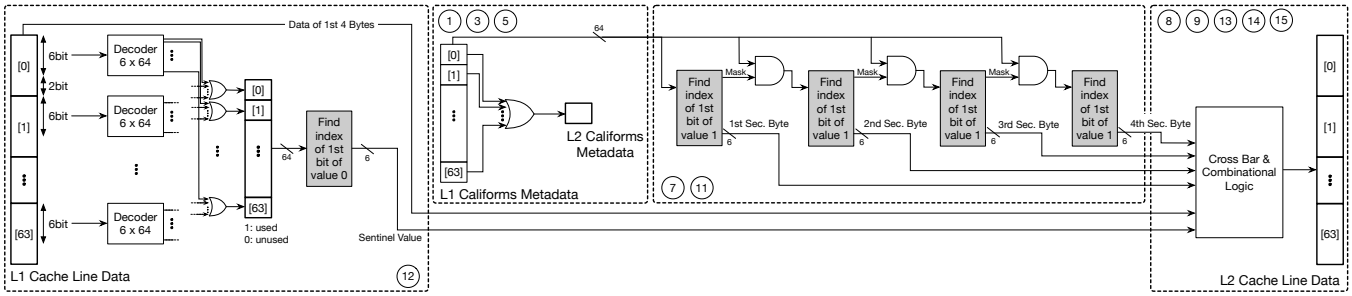


Figure 8: Logic diagram for Califorms conversion from the L1 cache (califorms-bitvector) to L2 cache (califorms-sentinel). The shaded blocks are constructed using 64 shift blocks followed by a single comparator. The circled numbers refer to the corresponding steps in Algorithm 1.

```
1: Read the Califorms metadata for the evicted line and OR them
2: if result is 0 then
3:    Evict the line as is and set Califorms bit to zero
4: else
5:    Set Califorms bit to one
6:    if num security bytes (N) < 4 then
7:       Get locations of first N security bytes
8:       Store data of first N bytes in locations obtained in 7
9:       Fill the first N bytes based on Figure 7
10:   else
11:      Get locations of first four security bytes
12:      Scan least 6-bit of every byte to determine sentinel
13:      Store data of first four bytes in locations obtained in 11
14:      Fill the first four bytes based on Figure 7
15:      Use the sentinel to mark the remaining security bytes
16:   end
17: end
```

Algorithm 1: Califorms conversion from the L1 cache (califorms-bitvector) to L2 cache (califorms-sentinel).

we use the bits of the second and third bytes to locate them. The key observation is that, we gain two bits per security byte since we only need six bits to specify a location in the cache line. Therefore when we have four security bytes, we can locate four addresses and have six bits remaining in the first four bytes. This remaining six bits can be used to store a sentinel value, which allows us to have any number of additional security bytes.

Although the sentinel value depends on the actual values within the 64B cache line, it works naturally with a write-allocate L1 cache (which is the most commonly used cache allocation policy in modern microprocessors). The cache line format is transformed upon L1 cache eviction and insertion (califorms-bitvector to/from califorms-sentinel), while the sentinel value only needs to be found upon L1 cache eviction (L1 miss). Also, it is important to note that califorms-sentinel supports critical-word first delivery since the security byte locations can be quickly retrieved by scanning only the first 4B of the first 16B flit.

7.3 L1 to/from L2 Califorms Conversion

Figure 8 and Algorithm 1 show the logic diagram and the high-level process of the spill (L1 to L2 conversion) module, respectively. The circled numbers in the figure refer to the corresponding steps in the algorithm. There are four components presented in the figure. From the left, the first block details the process of determining the sentinel value (line 12). We scan the least 6-bits of every byte, decode them, and OR the output to construct a used-values vector. The used-values vector is then processed by a "find-index" block to get the sentinel value. The find-index block takes a 64-bit input vector

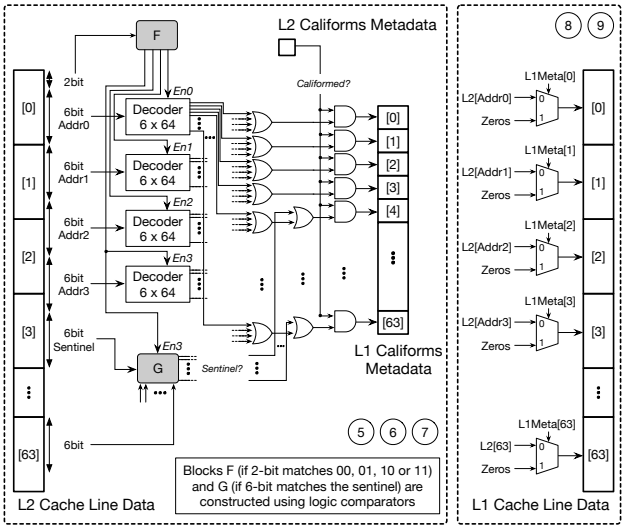


Figure 9: Logic diagram of Califorms conversion from the L2 cache (califorms-sentinel) to L1 cache (califorms-bitvector). The shaded block F and G consist of four and 60 comparators, respectively. The circled numbers refer to the corresponding steps in Algorithm 2.

```
1: Read the Califorms bit for the inserted line
2: if result is 0 then
3:    Set the Califorms metadata bit vector to [0]
4: else
5:    Check the least significant 2-bit of byte 0
6:    Set the metadata of byte[Addr[0-3]] to one based on 5
7:    Set the metadata of byte[Addr[byte==sentinel]] to one
8:    Set the data of byte[0-3] to byte[Addr[0-3]]
9:    Set the new locations of byte[Addr[0-3]] to zero
10: end
```

Algorithm 2: Califorms conversion from the L2 cache (califorms-sentinel) to L1 cache (califorms-bitvector).

and searches for the index of the first zero value. It is constructed using 64 shift blocks followed by a single comparator. In the second block, L1 cache (califorms-bitvector) metadata for the evicted line is ORed to construct the L2 cache (califorms-sentinel) metadata. The third block shows the logic for getting the locations of the first four security bytes (lines 7 and 11). It consists of four successive combinational find-index blocks (each detecting one security byte) in our evaluated design. This logic can be easily pipelined into four

stages if needed, to completely hide the latency of the spill process in the pipeline. Finally in the last block, we form the L2 cache line based on Figure 7.

Figure 9 shows the logic diagram for the fill (L2 to L1 conversion) module, as summarized in Algorithm 2. The shaded blocks F and G are constructed using logic comparators. The one bit metadata of L2 Califorms is used to control the value of the L1 cache (califorms-bitvector) metadata. The first two bits of the L2 cache line are used as inputs for the comparators (block F) to detect how many security bytes the cache line contain. Block F outputs four signals (En0 to En3) which enable the four decoders. Only if those two bits are 11, the sentinel value is read from the fourth byte and fed, with the least 6-bits of each byte, to 60 comparators simultaneously to set the rest of the L1 metadata bits. Such parallelization reduces the latency impact of the fill process.

7.4 Load/Store Queue Modifications

Since the BLOC instruction updates the architectural state, it is functionally a store instruction and handled as such in the pipeline. However, there is a key difference: unlike a store instruction, the BLOC instruction should not forward the value to a younger load instruction whose address matches within the load/store queue (LSQ) but instead return the value zero. This functionality is required to provide tamper-resistance against side-channel attacks. Additionally, upon an address match, both load and store instructions subsequent to an in flight BLOC instruction are marked for Califorms exception; exception is thrown when the instruction is committed to avoid any false positives due to misspeculation.

In order to detect an address match in the LSQ with a BLOC instruction, first a cache line address should be matched with all the younger instructions. Subsequently upon a match, the value stored in the LSQ for the BLOC instruction which contains the mask value (to set/unset security bytes) is used to confirm the final match. To facilitate a match with a BLOC instruction, each LSQ entry should be associated with a bit to indicate whether the entry contains a BLOC instruction. Detecting a complete match may take multiple cycles, however, a legitimate load/store instruction should never be forwarded a value from a BLOC instruction, and thus the store-to-load forwarding from a BLOC instruction is not on the critical path of the program (i.e., its latency should not affect performance), and we do not evaluate its effect in our evaluation. Alternately, if LSQ modifications are to be avoided, the BLOC instructions can be surrounded by memory serializing instructions (i.e., ensure that BLOC instructions are the only in flight memory instructions).

8 SOFTWARE DESIGN

We describe the memory allocator, compiler and the operating system changes to support Califorms in the following.

8.1 Dynamic Memory Management

We can consider two approaches to applying security bytes: (1) **Dirty-before-use.** Unallocated memory has no security bytes. We set security bytes upon allocation and unset them upon deallocation; or (2) **Clean-before-use.** Unallocated memory remains filled with security bytes all the time. We clear the security bytes (in legitimate data locations) upon allocation and set them upon deallocation.

Ensuring temporal memory safety in the heap remains a non-trivial problem [30]. We therefore choose to follow a *clean-before-use* approach in the heap, so that deallocated memory regions remain protected by security bytes.[5] In order to provide temporal memory safety (to mitigate use-after-free exploits), we do not reallocate recently freed regions until the heap is sufficiently consumed (quarantining). Additionally, both ends of the heap allocated regions are protected by security bytes in order to provide inter-object memory safety. Compared to the heap, the security benefits are limited for the stack since temporal attacks on the stack (e.g., use-after-return attacks) are much rarer. Hence, we apply the *dirty-before-use* scheme on the stack.

8.2 Compiler Support

Our compiler-based instrumentation infers where to place security bytes within target objects, based on their type layout information. The compiler pass supports three insertion policies: the first *opportunistic* policy supports security bytes insertion into existing padding bytes within the objects, and the other two support modifying object layouts to introduce randomly sized security byte spans that follow the *full* or *intelligent* strategies described in Section 3. The first policy aims at retaining interoperability with external code modules (e.g., shared libraries) by avoiding type layout modification. Where this is not a concern, the latter two policies help offer stronger security coverage—exhibiting a tradeoff between security and performance.

8.3 Operating System Support

We need the following support in the operating system:

Privileged Exceptions. As the Califorms exception is privileged, the operating system needs to properly handle it as with other privileged exceptions (e.g., page faults). We also assume the faulting address is passed in an existing register so that it can be used for reporting/investigation purposes. Additionally, for the sake of usability and backwards compatibility, we have to accommodate copying operations similar in nature to `memcpy`. For example, a simple `struct` to `struct` assignment could trigger this behavior, thus leading to a potential breakdown of software with Califorms support. Hence, in order to maintain usability, we allow whitelisting functionality to suppress the exceptions. This can either be done with a privileged store (requiring a syscall) or an unprivileged store. Both options represent different design points in the performance-security tradeoff spectrum.

Page Swaps. As we have discussed in Section 5, data with security bytes is stored in main memory in a califormed format. When a page with califormed data is swapped out from main memory, the page fault handler needs to store the metadata for the entire page into a reserved address space managed by the operating system; the metadata is reclaimed upon swap in. The kernel has enough address space in practice (kernel's virtual address space is 128TB for a 64-bit Linux with 48-bit virtual address space) to store the

[5] It is natural to use a variant of BLOC instruction which bypasses (does not store into) the L1 data cache, just like the non-temporal (or streaming) load/store instructions (e.g., `MOVNTI`, `MOVNTQ`, etc) when deallocating a memory region; deallocated region is not meant to be used by the program and thus polluting the L1 data cache with those memory is harmful and should be avoided. However, we do not evaluate the use of such instructions in this paper.

metadata for all the processes on the system since the size of the metadata is minimal (8B for a 4KB page or 0.2%).

9 SECURITY DISCUSSION

9.1 Hardware Attacks and Mitigations

Metadata Tampering Attacks. A key feature of Califorms is the absence of metadata that is accessible by the program via regular load-stores. This makes our technique immune to attacks that explicitly aim to leak or tamper metadata to bypass the defense. This, in turn, implies a smaller attack surface as far as software maintenance/isolation of metadata is concerned.

Bit-Granular Attacks. Califorms's capability of fine-grained memory protection is the key enabler for intra-object overflow detection. However, our byte granular mechanism is not enough for protecting bit-fields without turning them into `char` bytes functionally. This should not be a major detraction since security bytes can still be added around composites of bit-fields.

Side-Channel Attacks. Our design takes multiple steps to be resilient to side-channel attacks. Firstly, we purposefully avoid having our hardware modifications introduce timing variances to avoid timing based side-channel attacks. Additionally, to avoid speculative execution side channels ala Spectre [15], our design returns zero on a load to camouflage security byte with normal data, thus preventing speculative disclosure of metadata. We augment this further by requiring that deallocated objects (heap or stack) be zeroed out in software [17]. This is to reduce the chances of the following attack scenario: consider a case if the attacker somehow knows that the padding locations should contain a non-zero value (for instance, because she knows the object allocated at the same location prior to the current object had non-zero values). However, while speculatively disclosing memory contents of the object, she discovers that the padding location contains a zero instead. As such, she can infer that the padding there contains a security byte. If deallocations were accompanied with zeroing, however, this assumption can be made with a lower likelihood. Hence, making Califorms return a fixed value (zero), complemented by software actively zeroing out unused locations, reduces the attacker's probability of speculatively predicting security byte locations, as well as leaking its exact value.

9.2 Software Attacks and Mitigations

Coverage-Based Attacks. For emitting `BLOC` instructions to work on the padding bytes (in an object), we need to know the precise type information of the allocated object. This is not always possible in C-style programs where `void*` allocations may be used. In these cases, the compiler may not be able to infer the correct type, in which case intra-object support may be skipped for such allocations. Similarly, our metadata insertion policies (viz., intelligent and full) require changes to the type layouts. This means that interactions with external modules that have not been compiled with Califorms support may need (de)serialization to remain compatible. For an attacker, such points in execution may appear lucrative because of inserted security bytes getting stripped away in those short periods. We note however that the opportunistic policy can still remain in place to offer some protection. On the other hand, for those interactions that remain oblivious to type layout modifications (e.g., passing a pointer to an object that shall remain opaque within the external module), our hardware-based implicit checks have the benefit of persistent tampering protection, even across binary module boundaries.

Whitelisting Attacks. Our concession of allowing whitelisting of certain functions was necessary to make Califorms more usable in common environments without requiring significant source modifications. However, this also creates a vulnerability window wherein an adversary can piggy back on these functions in the source to bypass our protection. To confine this vector, we keep the number of whitelisted functions as minimal as possible.

Derandomization Attacks. Since Califorms can be bypassed if an attacker can guess the security bytes location, it is crucial that it be placed unpredictably. For the attacker to carry out a guessing attack, the virtual address of the target object has to be leaked, in order to overwrite a certain number of bytes within that object. To know the address of the object of interest, she typically has to scan the process's memory: the probability of scanning without touching any of the security bytes is $(1-P/N)^O$ where O is number of allocated objects, N is the size of each object, and P is number of security bytes within that object. With 10% padding ($P/N = 0.1$), when O reaches 250, the attack success goes to 10^{-20}. If the attacker can somehow reduce O to 1, which represents the ideal case for the attacker, the probability of guessing the element of interest is $1/7^n$ (since we insert 1–7 wide security bytes), compounding as the number of padding spans to be guessed ($= n$) increases.

The randomness is, however, introduced statically akin to `randstruct` plugin introduced in recent Linux kernels which randomizes structure layout of those which are specified (it does not offer detection of rogue accesses unlike Califorms do) [5, 12]. The static nature of the technique may make it prone to brute force attacks like BROP [4] which repeatedly crashes the program until the correct configuration is guessed. This could be prevented by having multiple binaries of the same program with different padding sizes or simply by better logging, when possible. Another mitigating factor is that BROP attacks require specific type of program semantics, namely, automatic restart-after-crash with the same memory layout. Applications with these semantics can be modified to spawn with a different padding layout in our case and yet satisfy application level requirements.

10 PERFORMANCE EVALUATION

10.1 Hardware Overheads

Cache Access Latency Impact of Califorms. Califorms adds additional state and operations to the L1 data cache and the interface between the L1 and L2 caches. The goal of this section is to evaluate the access latency impact of the additional state and operations described in Section 7. Qualitatively, the metadata area overhead of L1 Califorms is 12.5%, and the access latency should not be impacted as the metadata lookup can happen in parallel with the L1 data and tag accesses; the L1 to/from L2 Califorms conversion should also be simple enough so that its latency can be completely hidden. However, the metadata area overhead can increase the L1 access latency and the conversions might add little latency. Without loss of generality, we measure the access latency impact of adding

Table 2: Area, delay and power overheads of Califorms (GE represents gate equivalent). L1 Califorms (califorms-bitvector) adds negligible delay and power overheads to the L1 cache access.

L1 Califorms	Area (GE)	Delay (*ns*)	Power (*mW*)
L1 Overheads	[+18.69%] 412,263.87	[+1.85%] 1.65	[+2.12%] 16.17
Fill Module	8,957.16	1.43	0.18
Spill Module	34,561.80	5.50	0.52

califorms-bitvector on a 32KB direct mapped L1 cache in the context of a typical energy optimized tag and data, formatting L1 pipeline with multicycle fill/spill handling. For the implementation we use the 65nm TSMC core library, and generate the SRAM arrays with the ARM Artisan memory compiler.

Table 2 summarizes the results for the L1 Califorms (califorms-bitvector). As expected, the overheads associated with the califorms-bitvector are minor in terms of delay (1.85%) and power consumption (2.12%). We found the SRAM area to be the dominant component in the total cache area (around 98%) where the overhead is 18.69% (higher than 12.5%).

The results of fill/spill modules are reported separately in the bottom half of Table 2. The latency impact of the fill operation is within the access period of the L1 design. Thus, the transformation can be folded completely within the pipeline stages that are responsible for bringing cache lines from L2 to L1. The timing delay of the less performance sensitive spill operation is larger than that of the fill operation (5.5*ns* vs. 1.4*ns*) as we use pure combinational logic to construct the califorms-sentinel format in one cycle, as shown in Figure 8. This cycle period can be reduced by dividing the operations of Algorithm 1 into two or more pipeline stages. For instance, getting the locations of the first four security bytes (lines 7 and 11) consists of four successive combinational blocks (each detecting one security byte) in our evaluated design. This logic can be easily pipelined into four stages. Therefore we believe that the latency of both the fill and spill operations can be minimal (or completely hidden) in the pipeline.

Performance with Additional Cache Access Latency. Our VLSI implementation results imply that there will be no additional L2/L3 latency imposed by implementing Califorms. However, this might not be the case depending on several implementation details (e.g., target clock frequency) so we pessimistically assume that the L2/L3 access latency incurs additional one cycle latency overhead. In order to evaluate the performance of the additional latency posed by Califorms, we perform detailed microarchitectural simulations.

We run SPEC CPU2006 benchmarks with ZSim [24] processor simulator for evaluation. All the benchmarks are compiled with Clang version 6.0.0 with "`-O3 -fno-strict-aliasing`" flags. We use the `ref` input sets and representative simulation regions are selected with PinPoints [22]. We do not warmup the simulator upon executing each SimPoint region, but instead use a relatively large interval length of 500M instructions to avoid any warmup issues. MaxK used in SimPoint region selection is set to 30.[6] Table 3 shows the parameters of the processor, an Intel Westmere-like out-of-order core which has been validated against a real system whose performance and microarchitectural events to be commonly within 10% [24]. We evaluate the performance when both L2 and L3 caches incur additional latency of one cycle.

[6]For some benchmark-input pairs we saw discrepancies in the number of instructions measured by PinPoints vs. ZSim and thus the appropriate SimPoint regions might not be simulated. Those inputs are: `foreman_ref_encoder_main` for `h264ref` and `pds-50` for `soplex`. Also, due to time constraints, we could not complete executing SimPoint for `h264ref` with `sss_encoder_main` input and excluded it from the evaluation.

Table 3: Hardware configuration of the simulated system.

Core	x86-64 Intel Westmere-like OoO core at 2.27GHz
L1 inst. cache	32KB, 4-way, 3-cycle latency
L1 data cache	32KB, 8-way, 4-cycle latency
L2 cache	256KB, 8-way, 7-cycle latency
L3 cache	2MB, 16-way, 27-cycle latency
DRAM	8GB, DDR3-1333

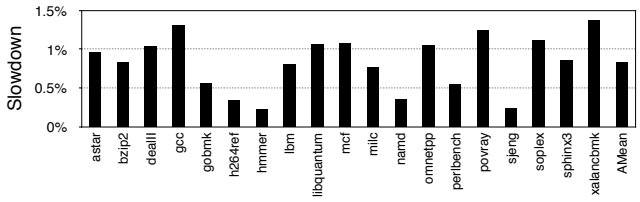


Figure 10: Slowdown with additional one-cycle access latency for both L2 and L3 caches.

As shown in Figure 10 slowdowns range from 0.24% (`hmmer`) to 1.37% (`xalancbmk`). The average performance slowdown is 0.83% which is well in the range of error when executed on real systems.

10.2 Software Performance Overheads

Our evaluations so far revealed that the hardware modifications required to implement Califorms add little or no performance overhead. Here, we evaluate the overheads incurred by the software based changes required to enable inter-/intra-object and temporal memory safety with Califorms: the effect of underutilized memory structures (e.g., caches) due to additional security bytes, the additional work necessary to issue `BLOC` instructions (and the overhead of executing the instructions themselves), and the quarantining to support temporal memory safety.

Evaluation Setup. We run the experiments on an Intel Skylake-based Xeon Gold 6126 processor running at 2.6GHz with RHEL Linux 7.5 (kernel 3.10). We omit `dealII` and `omnetpp` due to library compatibility issues in our evaluation environment, and `gcc` since it fails when executed with the memory allocator with inter-object spatial and temporal memory safety support. The remaining 16 SPEC CPU2006 C/C++ benchmarks are compiled with our modified Clang version 6.0.0 with "`-O3 -fno-strict-aliasing`" flags. We use the `ref` inputs and run to completion. We run each benchmark-input pair five times and use the shortest execution time as its performance. For benchmarks with multiple `ref` inputs, the sum of the execution time of all the inputs are used as their execution times. We use the arithmetic mean to represent the average slowdown.[7]

We estimate the performance impact of executing a `BLOC` instruction by emulating it with a dummy store instruction that writes some value to the corresponding cache line's padding byte. Since

[7]The use of arithmetic mean of the speedup (execution time of the original system divided by that of the system with additional latency) means that we are interested in a condition where the workloads are not fixed and all types of workloads are equally probable on the target system [11, 14].

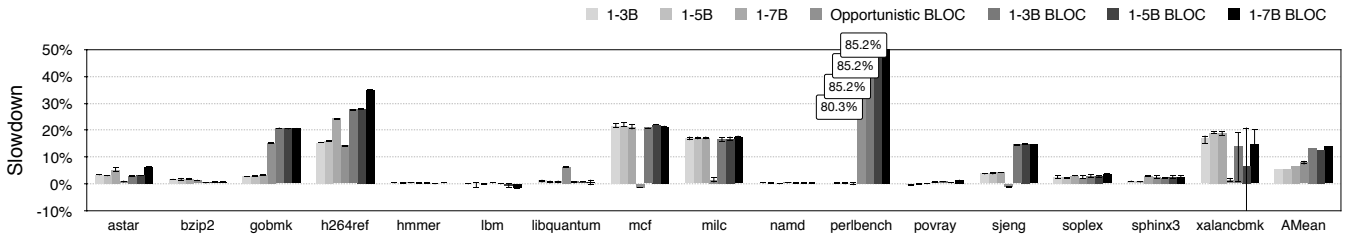


Figure 11: Slowdown of the opportunistic policy, and full insertion policy with random sized security bytes (with and without BLOC instructions). The average slowdowns of opportunistic and full insertion policies are 6.2% and 14.2%, respectively.

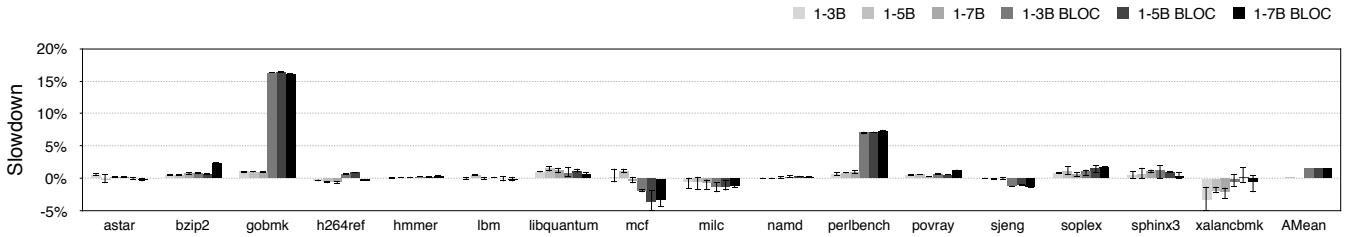


Figure 12: Slowdown of the intelligent insert policy with random sized security bytes (with and without BLOC instructions). The average slowdown is 2.0%.

a single BLOC instruction is able to caliform the entire cache line, issuing one dummy store instruction per to-be-califormed cache line suffices. In order to issue the dummy stores, we implement a LLVM pass to instrument the code to hook into memory allocations and deallocations. We then retrieve the type information to locate the padding bytes, calculate the number of dummy stores and the address they access, and finally emit them. Therefore, all the software overheads we need to pay to enable Califorms are accounted for in our evaluation.

For the random sized security bytes, we evaluate three variants: we fix the minimum size to one byte while varying the maximum size to three, five and seven bytes (i.e., on average the amount of security bytes inserted are two, three and four bytes, respectively). In addition, in order to account for the randomness introduced by the compiler, we generate three different versions of binaries for the same setup (e.g., three versions of `astar` with random sized paddings of minimum one byte and maximum three bytes). The error bars in the figure represent the minimum and the maximum execution times among 15 executions (three binaries × five runs) and the average of the execution times is represented as the bar.

Performance of the Opportunistic and Full Insertion Policies with BLOC Instructions. Figure 11 presents the slowdown incurred by three set of strategies: full insertion policy (with random sized security bytes) *without* BLOC instructions, the opportunistic policy *with* BLOC instructions, and the full insertion policy *with* BLOC instructions. Since the first strategy does not execute BLOC instructions it does not offer any security coverage, but is shown as a reference to showcase the performance breakdown of the third strategy (cache underutilization vs. executing BLOC instructions).

First, we focus on the three variants of the first strategy, which are shown in the three left most bars. We can see that different sizes of (random sized) security bytes does not make a large difference in terms of performance. The average slowdown of the three variants are 5.5%, 5.6% and 6.5%, respectively. This can be backed up by our results shown in Figure 4, where the average slowdowns of additional padding of two, three and four bytes ranges from 5.4% to 6.2%. Therefore in order to achieve higher security coverage without losing performance, using a random sized bytes of, minimum of one byte and maximum of seven bytes, is promising. When we focus on individual benchmarks, we can see that a few benchmarks including `h264ref`, `mcf`, `milc` and `omnetpp` incur noticeable slowdowns (ranging from 15.4% to 24.3%).

Next, we examine the opportunistic policy *with* BLOC instructions, which is shown in the middle (fourth) bar. Since this strategy does not add any additional security bytes, the overheads are purely due to the work required to setup and execute BLOC instructions. The average slowdown of this policy is 7.9%. There are benchmarks which encounter a slowdown of more than 10%, namely `gobmk`, `h264ref` and `perlbench`. The overheads are due to frequent allocations and deallocations made during program execution, where we have to calculate and execute BLOC instructions upon every event (since every compound data type requires security bytes management). For instance `perlbench` is notorious for being malloc-intensive, and reported as such elsewhere [25].

Lastly the third policy, the full insertion policy *with* BLOC instructions, offers the highest security coverage in Califorms based system with the highest average slowdown of 14.0% (with the random sized security bytes of maximum seven bytes). Nearly half (seven out of 16) the benchmarks encounter a slowdown of more than 10%, which might not be suitable for performance-critical environments, and thus the user might want to consider the use of the following intelligent insertion policy.

Performance of the Intelligent Insertion Policy with BLOC Instructions. Figure 12 shows the slowdowns of the intelligent insertion policy with random sized security bytes (*with* and *without* BLOC instructions, in the same spirit as Figure 11). First we focus

Table 4: Security comparison against prior hardware techniques. *Achieved with bounds narrowing. †Although the hardware supports bounds narrowing, CHERI foregoes it since doing so compromises capability logic [7].‡Execution compatible, but protection dropped when external modules modify pointer. §Limited to 13 tags. ¶Allocator should randomize allocation predictability.

Proposal	Protection Granularity	Intra-Object	Binary Composability	Temporal Safety
Hardbound [8]	Byte	✓*	✗	✗
Watchdog [18]	Byte	✓*	✗	✓
WatchdogLite [19]	Byte	✓*	✗	✓
Intel MPX [20]	Byte	✓*	✗‡	✗
BOGO [33]	Byte	✓*	✗‡	✓
PUMP [9]	Word	✗	✓	✓
CHERI [32]	Byte	✗†	✗	✗
CHERI concentrate [31]	Byte	✗†	✗	✗
SPARC ADI [21]	Cache line	✗	✓	✓§
SafeMem [23]	Cache line	✗	✓	✗
REST [28]	8–64B	✗	✓	✓¶
Califorms	Byte	✓	✓	✓¶

on the strategy without executing `BLOC` instructions (the three bars on the left). The performance trend is similar such that the three variants with different random sizes have little performance difference, where the average slowdown is 0.2% with the random sized security bytes of maximum seven bytes. We can see that none of the programs incurs a slowdown of greater than 5%. Finally with `BLOC` instructions (three bars on the right), `gobmk` and `perlbench` have slowdowns of greater than 5% (16.1% for `gobmk` and 7.2% for `perlbench`). The average slowdown is 1.5%, where considering its security coverage and performance overheads the intelligent policy might be the most practical option for many environments.

11 COMPARISON WITH PRIOR WORK

Tables 4, 5, and 6 summarize the security, performance, and implementation characteristics of the hardware based memory safety techniques discussed in Section 2, respectively. Califorms has the advantage of requiring simpler hardware modifications and being faster than disjoint metadata based whitelisting systems. The hardware savings mainly stem from the fact that our metadata resides with program data; it does not require explicit propagation while additionally obviating all lookup logic. This significantly reduces our design's implementation costs. Califorms also has lower performance and energy overheads since it neither requires multiple memory accesses, nor does it incur any significant checking costs. However, Califorms can be bypassed if accesses to security bytes can be avoided. This safety-vs.-complexity tradeoff is critical to deployability and we argue that our design point is more practical. This is because designers have to contend with integrating these features to already complicated processor designs, without introducing additional bugs while also keeping the functionality of legacy software intact. This is a hard balance to strike [20].

On the other hand, ideal cojoined metadata mechanisms would have comparable slowdowns and similar compiler requirements. However, practical implementations like ADI exhibits some crucial differences from the ideal.

- It is limited to 64-bit architectures, which excludes a large portion of embedded and IoT processors that operate on 32-bit or narrower platforms.
- It has finite number of colors since available tag bits are limited—ADI supports 13 colors with 4 tag bits. This is important because reusing colors proportionally reduces the safety guarantees of these systems in the event of a collision.
- It operates at the coarse granularity of cache line width, and hence, is not practically applicable for intra-object safety.

On the contrary, Califorms is agnostic of architecture width and is better suited for deployment over a more diverse device environment. In terms of safety, collision is not an issue for our design. Hence, unlike cojoined metadata systems, our security does not scale inversely with the number of allocations in the program. Finally, our fine-grained protection makes us suitable for intra-object memory safety which is a non-trivial threat in modern security [16].

12 CONCLUSION

Califorms is a hardware primitive which allows blacklisting a memory location at byte granularity with low area and performance overhead. A key observation behind Califorms is that a blacklisted region need not store its metadata separately but can rather store them within itself; we utilize byte-granular existing or added space between object elements to blacklist a region. This in-place compact data structure avoids additional operations for fetching the metadata making it very performant in comparison. Further, by changing how data is stored within a cache line we reduce the hardware area overheads substantially. Subsequently, if the processor accesses a blacklisted byte or a security byte, due to programming errors or malicious attempts, it reports a privileged exception.

To provide memory safety, we use Califorms to insert security bytes between and within data structures (e.g., between fields of a `struct`) upon memory allocation and clear them on deallocation. Notably, by doing so, Califorms can even detect intra-object overflows in a practical manner, thus addressing one of the prominent open problems in area of memory safety and security.

ACKNOWLEDGMENTS

This work was partially supported by ONR N00014–16–1–2263, ONR N00014–17–1–2788, ONR N00014–15–1–2173, DARPA HR0011–18–C–0017, and a gift from Bloomberg. The authors thank Prof. Mingoo Seok for access to SRAM timing measurement tools. Any opinions, findings, conclusions and recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the US government or commercial entities. Simha Sethumadhavan has a significant financial interest in Chip Scan Inc.

Table 5: Performance comparison against previous hardware techniques.

Proposal	Metadata Overhead	Memory Overhead	Performance Overhead	Main Operations
Hardbound [8]	0–2 words per ptr, 4b per word	$\propto$ # of ptrs and prog memory footprint	$\propto$ # of ptr derefs	1–2 mem ref for bounds (may be cached), check μops.
Watchdog [18]	4 words per ptr	$\propto$ # of ptrs and allocations	$\propto$ # of ptr derefs	1–3 mem ref for bounds (may be cached), check μops.
WatchdogLite [19]	4 words per ptr	$\propto$ # of ptrs and allocations	$\propto$ # of ptr ops	1–3 mem ref for bounds (may be cached), check & propagate insns.
Intel MPX [20]	2 words per ptr	$\propto$ # of ptrs	$\propto$ # of ptr derefs	2+ mem ref for bounds (may be cached), check & propagate insns.
BOGO [33]	2 words per ptr	$\propto$ # of ptrs	$\propto$ # of ptr derefs	MPX ops + ptr miss exception handling, page permission mods.
PUMP [9]	64b per cache line	$\propto$ Prog memory footprint	$\propto$ # of ptr ops	1 mem ref for tags, may be cached, fetch and chk rules; propagate tags.
CHERI [32]	256b per ptr	$\propto$ # of ptrs and physical mem	$\propto$ # of ptr ops	1+ mem ref for capability (may be cached), capability management insns.
CHERI concentrate [31]	Ptr size is 2x	$\propto$ # of ptrs	$\propto$ # of ptr ops	Wide ptr load (may be cached), capability management insns.
SPARC ADI [21]	4b per cache line	$\propto$ Prog memory footprint	$\propto$ # of tag (un)set ops	(Un)set tag.
SafeMem [23]	2x blacklisted memory	$\propto$ Blacklisted memory	$\propto$ # of ECC (un)set ops	Syscall to scramble ECC, copy data content.
REST [28]	8–64B token	$\propto$ Blacklisted memory	$\propto$ # of arm/disarm insns	Execute arm/disarm insns.
Califorms	Byte granular security byte	$\propto$ Blacklisted memory	$\propto$ # of `BLOC` insns.	Execute `BLOC` insns.

Table 6: Implementation complexity comparison against previous hardware techniques.

Proposal	Core	Caches/TLB	Memory	Software
Hardbound [8]	μop injection & logic for ptr meta, extend reg file and data path to propagate ptr meta	Tag cache and its TLB	N/A	Compiler & allocator annotates ptr meta
Watchdog [18]	μop injection & logic for ptr meta, extend reg file and data path to propagate ptr meta	Ptr lock cache	N/A	Compiler & allocator annotates ptr meta
WatchdogLite [19]	N/A	N/A	N/A	Compiler & allocator annotates ptrs, compiler inserts meta propagation and check insns
Intel MPX [20]	Unknown (closed platform [27], design likely similar to Hardbound)			Compiler & allocator annotates ptrs, compiler inserts meta propagation and check insns
BOGO [33]	Unknown (closed platform [27], design likely similar to Hardbound)			MPX mods + kernel mods for bounds page right management
PUMP [9]	Extend all data units by tag width, modify pipeline stages for tag checks, new miss handler	Rule cache	N/A	Compiler & allocator (un)sets memory, tag ptrs
CHERI [32]	Capability reg file, coprocessor integrated with pipeline	Capability caches	N/A	Compiler & allocator annotates ptrs, compiler inserts meta propagation and check insns
CHERI concentrate [31]	Modify pipeline to integrate ptr checks	N/A	N/A	Compiler & allocator annotates ptrs, compiler inserts meta propagation and check insns
SPARC ADI [21]	Unknown (closed platform)			Compiler & allocator (un)sets memory, tag ptrs
SafeMem [23]	N/A	N/A	Repurposes ECC bits	Original data copied to distinguish from hardware faults
REST [28]	N/A	1–8b per L1D line, 1 comparator	N/A	Compiler & allocator (un)sets tags, allocator randomizes allocation order/placement
Califorms	N/A	8b per L1D line, 1b per L2/L3 line	Use unused/spare ECC bit	Compiler & allocator mods to (un)set tags, compiler inserts intra-object spacing

REFERENCES

[1] 2014. CVE–2014–1444. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-1444. [Online; accessed 30-Aug-2019].

[2] 2017. CVE–2017–5115. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-5115. [Online; accessed 30-Aug-2019].

[3] ARM. 2018. ARM A64 instruction set architecture for ARMv8-A architecture profile. https://static.docs.arm.com/ddi0596/a/DDI_0596_ARM_a64_instruction_set_architecture.pdf.

[4] Andrea Bittau, Adam Belay, Ali Mashtizadeh, David Mazi e res, and Dan Boneh. 2014. Hacking blind. In *IEEE S&P '14: Proceedings of the 35th IEEE Symposium on Security and Privacy*.

[5] Kees Cook. 2017. Introduce struct layout randomization plugin. https://lkml.org/lkml/2017/5/26/558.

[6] Crispin Cowan, Calton Pu, Dave Maier, Heather Hintony, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, and Qian Zhang. 1998. StackGuard: automatic adaptive detection and prevention of buffer-overflow attacks. In *USENIX Security '98: Proceedings of the 7th USENIX Security Symposium*.

[7] Brooks Davis, Khilan Gudka, Alexandre Joannou, Ben Laurie, A Theodore Markettos, J Edward Maste, Alfredo Mazzinghi, Edward Tomasz Napierala, Robert M Norton, Michael Roe, Peter Sewell, Robert N M Watson, Stacey Son, Jonathan Woodruff, Alexander Richardson, Peter G Neumann, Simon W Moore, John Baldwin, David Chisnall, James Clarke, and Nathaniel Wesley Filardo. 2019. CheriABI: enforcing valid pointer provenance and minimizing pointer privilege in the POSIX C run-time environment. In *ASPLOS '19: Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems*.

[8] Joe Devietti, Colin Blundell, Milo M K Martin, and Steve Zdancewic. 2008. HardBound: architectural support for spatial safety of the C programming language. In *ASPLOS XIII: Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*.

[9] Udit Dhawan, Catalin Hritcu, Raphael Rubin, Nikos Vasilakis, Silviu Chiricescu, Jonathan M Smith, Thomas F Knight, Jr, Benjamin C Pierce, and Andre DeHon. 2015. Architectural support for software-defined metadata processing. In *ASPLOS '15: Proceedings of the 20th International Conference on Architectural Support for Programming Languages and Operating Systems*.

[10] Gregory J Duck and Roland H C Yap. 2018. EffectiveSan: type and memory error detection using dynamically typed C/C++. In *PLDI '18: Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*.

[11] Lieven Eeckhout. 2010. *Computer architecture performance evaluation methods* (1st ed.).

[12] Nur Hussein. 2017. Randomizing structure layout. https://lwn.net/Articles/722293/.

[13] Yuseok Jeon, Priyam Biswas, Scott Carr, Byoungyoung Lee, and Mathias Payer. 2017. HexType: efficient detection of type confusion errors for C++. In *CCS '17: Proceedings of the 24th ACM SIGSAC Conference on Computer and Communications Security*.

[14] Lizy Kurian John. 2004. More on finding a single number to indicate overall performance of a benchmark suite. *ACM SIGARCH Computer Architecture News* 32, 1 (March 2004), 3–8.

[15] Paul Kocher, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. 2019. Spectre attacks: exploiting speculative execution. In *IEEE S&P '19: Proceedings of the 40th IEEE Symposium on Security and Privacy*.

[16] Kangjie Lu, Chengyu Song, Taesoo Kim, and Wenke Lee. 2016. UniSan: proactive kernel memory initialization to eliminate data leakages. In *CCS '16: Proceedings of the 23rd ACM SIGSAC Conference on Computer and Communications Security*.

[17] Alyssa Milburn, Herbert Bos, and Cristiano Giuffrida. 2017. SafeInit: comprehensive and practical mitigation of uninitialized read vulnerabilities. In *NDSS '17: Proceedings of the 2017 Network and Distributed System Security Symposium*.

[18] Santosh Nagarakatte, Milo M K Martin, and Steve Zdancewic. 2012. Watchdog: hardware for safe and secure manual memory management and full memory safety. In *ISCA '12: Proceedings of the 39th International Symposium on Computer Architecture*.

[19] Santosh Nagarakatte, Milo M K Martin, and Steve Zdancewic. 2014. WatchdogLite: hardware-accelerated compiler-based pointer checking. In *CGO '14: Proceedings of the 12th IEEE/ACM International Symposium on Code Generation and Optimization*.

[20] 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 (June 2018), 28:1–28:30.

[21] Oracle. 2015. Hardware-assisted checking using Silicon Secured Memory (SSM). https://docs.oracle.com/cd/E37069_01/html/E37085/gphwb.html.

[22] Harish Patil, Robert Cohn, Mark Charney, Rajiv Kapoor, Andrew Sun, and Anand Karunanidhi. 2004. Pinpointing representative portions of large Intel® Itanium® programs with dynamic instrumentation. *MICRO-37: Proceedings of the 37th IEEE/ACM International Symposium on Microarchitecture*.

[23] Feng Qin, Shan Lu, and Yuanyuan Zhou. 2005. SafeMem: exploiting ECC-memory for detecting memory leaks and memory corruption during production runs. In *HPCA '05: Proceedings of the IEEE 11th International Symposium on High Performance Computer Architecture*.

[24] Daniel Sanchez and Christos Kozyrakis. 2013. ZSim: fast and accurate microarchitectural simulation of thousand-core systems. In *ISCA '13: Proceedings of the 40th International Symposium on Computer Architecture*.

[25] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. 2012. AddressSanitizer: a fast address sanity checker. In *USENIX ATC '12: Proceedings of the 2012 USENIX Annual Technical Conference*.

[26] Kostya Serebryany, Evgenii Stepanov, Aleksey Shlyapnikov, Vlad Tsyrklevich, and Dmitry Vyukov. 2018. Memory tagging and how it improves C/C++ memory safety. *arXiv.org* (Feb. 2018). arXiv:cs.CR/1802.09517v1

[27] Junjing Shi, Qin Long, Liming Gao, Michael A. Rothman, and Vincent J. Zimmer. 2018. Methods and apparatus to protect memory from buffer overflow and/or underflow. International patent WO/2018/176339.

[28] Kanad Sinha and Simha Sethumadhavan. 2018. Practical memory safety with REST. In *ISCA '18: Proceedings of the 45th International Symposium on Computer Architecture*.

[29] Dokyung Song, Julian Lettner, Prabhu Rajasekaran, Yeoul Na, Stijn Volckaert, Per Larsen, and Michael Franz. 2019. SoK: sanitizing for security. In *IEEE S&P '19: Proceedings of the 40th IEEE Symposium on Security and Privacy*.

[30] David Weston and Matt Miller. 2016. Windows 10 mitigation improvements. Black Hat USA.

[31] Jonathan Woodruff, Alexandre Joannou, Hongyan Xia, Anthony Fox, Robert Norton, David Chisnall, Brooks Davis, Khilan Gudka, Nathaniel W Filardo, , A Theodore Markettos, Michael Roe, Peter G Neumann, Robert Nicholas Maxwell Watson, and Simon Moore. 2019. CHERI concentrate: practical compressed capabilities. *IEEE Trans. Comput.* 68, 10 (Oct. 2019), 1455–1469.

[32] Jonathan Woodruff, Robert N M Watson, David Chisnall, Simon W Moore, Jonathan Anderson, Brooks Davis, Ben Laurie, Peter G Neumann, Robert Norton, and Michael Roe. 2014. The CHERI capability model: revisiting RISC in an age of risk. In *ISCA '14: Proceedings of the 41st International Symposium on Computer Architecture*.

[33] Tong Zhang, Dongyoon Lee, and Changhee Jung. 2019. BOGO: buy spatial memory safety, get temporal memory safety (almost) free. In *ASPLOS '19: Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems*.