Practical Byte-Granular Memory Blacklisting using Califorms*

Hiroshi Sasaki, Miguel A. Arroyo, M. Tarek Ibn Ziad, Koustubha Bhat, Kanad Sinha, Simha Sethumadhavan

SUMMARY

Our MICRO 2019 paper presents *Califorms*, a novel hardware primitive which provides both coarse- and fine-grained memory safety with low area and performance overheads. To the best of our knowledge, this is the first paper to propose a practical solution for detecting intra-object overflows (overflows within an object), one of the prominent open problems in area of memory safety and security.

Memory Safety. Historically, program memory safety violations have provided a significant opportunity for exploitation by attackers: for instance, Microsoft recently revealed that the root cause of around 70% of all exploits targeting their products are software memory safety violations [1]. We also often see news reporting that memory safety issues have led to real-world incidents. For instance, a recent Google Project Zero post said that hacked websites were used to indiscriminately attack individuals who visited them through vulnerabilities in iOS (subsequently Apple released a rebuttal statement) [2]. To address these threats, software checking tools (e.g., AddressSanitizer [3]) and commercial hardware support for memory safety (e.g., Oracle's ADI [4] and Intel's MPX [5]) 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 accesses 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 (2.2x performance and 1.1x memory overheads for EffectiveSan [6] and 1.7x performance and 2.1x memory overheads Intel MPX [5]) and/or need for making intrusive changes to the source code. Intra-object memory safety problems are a serious threat. They manifest in real-world scenarios such as type confusion vulnerabilities (e.g., CVE–2017–5115) and uninitialized data leaks through padding bytes (e.g., CVE–2014–1444), both recognized as high-impact security classes.

*H. Sasaki, M. A. Arroyo, M. Tarek Ibn Ziad, K. Bhat, K. Sinha, and S. Sethumadhavan, "Practical Byte-Granular Memory Blacklisting using Califorms," in Proceedings of the 52nd IEEE/ACM International Symposium on Microarchitecture (MICRO), pp.558–571, Oct. 2019.

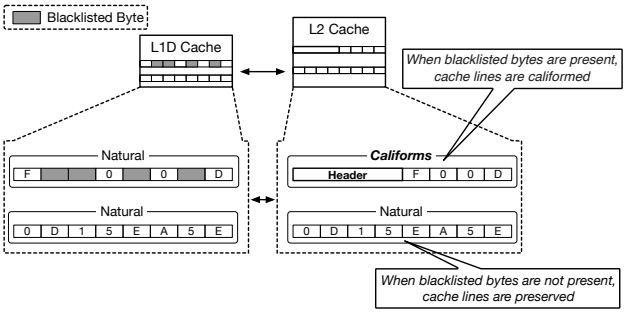

Figure 1: Califorms offers memory safety by detecting accesses to blacklisted bytes in memory. Blacklisted bytes are not stored beyond the L1 data cache and identified using a special header in the L2 cache (and beyond) resulting in very low overhead. The conversion between the formats happens when lines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

Califorms. Califorms is a novel hardware primitive which allows blacklisting of a memory location (i.e., if accessed due to programming errors or malicious attempts, it reports a privileged exception) at *byte granularity* with low area and performance overheads. The main obstacle to blacklisting a memory region at a fine granularity (e.g., to prevent intra-object overflows) is the associated overheads in maintaining the metadata. We solve this problem based on the following key observation: *a blacklisted region need not store its metadata (indicating it is blacklisted) separately, but can rather store them within itself (since it contains no useful data!).* With this principle, we utilize existing or added bytes between object elements to blacklist a region. This in-place compact data structure avoids additional operations for accessing the metadata making it very performant in comparison.

The challenge lies in how to reduce the additional hardware overhead required to identify normal data vs. metadata. A naive implementation requires additional one bit (to specify normal data or metadata) per byte, which results in 12.5% area overhead. We manage to reduce the overhead substantially, to one bit per cache line (typically 64 bytes, thus area overhead of 0.2%), by changing how data is stored within a cache line. For cache lines which contain metadata (within blacklisted bytes), the actual data is stored following the "header", which indicates the location of blacklisted bytes, as shown in Figure 1.

```
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 blacklisted_bytes[3];
  int i;
  char buf[64];
  void (*fp)();
}
```

(b) Opportunistic.

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

(c) Full.

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

(d) Intelligent.

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

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

Blacklisting with Califorms. One of the key ways in which we mitigate the overheads for fine-grained memory safety is by opportunistically harvesting padding bytes (which store no useful data) in programs and by blacklisting them. So how often do these occur in programs? Before we answer this 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 other languages and their runtime implementations. We found that 45.7% and 41.0% of structs within SPEC CPU2006 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 safety for the rest of the program. To offer protection for all defined compound data types, we can insert random sized blacklisted bytes, between every field of a `struct` or member of a `class` as in Listing 1(c) (full strategy). Random sized blacklisted bytes are chosen to provide a probabilistic defense as fixed sized blacklisted bytes can be jumped over by an attacker once she identifies the actual size (and the exact memory layout). Intuitively, the higher the unpredictability (or randomness) within the memory layout, the higher the security level we can offer.

System Design. The Califorms framework consists of three major components:

- **Architecture Support.** A new ISA instruction that blacklists memory locations at byte granularity and raises a privileged exception upon misuse of blacklisted locations.
- **Microarchitecture Design.** New cache line formats, or Califorms, that enable low cost access to the metadata—we propose different Califorms for L1 vs. L2 and beyond.
- **Software Design.** Compiler, memory allocator and operating system extensions which insert the blacklisted bytes at compile time and manages them via the new ISA instruction at runtime.

Performance. Our evaluation results reveal that our prototyped hardware modifications add little or no impact on cache access latencies (negligible performance overheads), while the software modifications have 2.0 to 14.0% slowdowns, depending on the blacklisted bytes insertion policy.

REFERENCES

[1] Matt Miller. Trends, challenge, and shifts in software vulnerability mitigation. BlueHat IL, 2019.

[2] A very deep dive into ios exploit chains found in the wild. https://googleprojectzero.blogspot.com/2019/08/a-very-deep-dive-into-ios-exploit.html. [Online; accessed 25-Oct-2019].

[3] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. AddressSanitizer: a fast address sanity checker. In *USENIX ATC '12*, 2012.

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

[5] Oleksii Oleksenko, Dmitrii Kuvaiskii, Pramod Bhatotia, Pascal Felber, and Christof Fetzer. Intel MPX explained: a cross-layer analysis of the Intel MPX system stack. *ACM POMACS*, 2(2):28:1–28:30, June 2018.

[6] Gregory J Duck and Roland H C Yap. EffectiveSan: type and memory error detection using dynamically typed C/C++. In *PLDI '18*, 2018.

LONG-TERM IMPACT

The problem of memory safety errors is a long-standing issue which has annoyed programmers to no end since the first Morris Worm was released in 1988. Since then, it has driven researchers to find *practical* solutions for the last 30 years. Proposed solutions have since ranged from languages that offer memory safety, to static analysis/compiler techniques to identify and fix these errors before deployment, to runtime hardware techniques for continuous detection/mitigation. Even with these continuous efforts, we have yet to solve this problem or even mitigate it to a significant extent. Why? The main reason for this is that very few of the solutions are practical, which should meet the following conditions: (a) be performance- and energy-efficient, (b) be applicable to a large fraction of devices, and (c) be cost-efficient to develop and deploy. We discuss how each of these concerns are addressed by Califorms and hence offer a significant chance of having long-term impact.

- **Performance- and Energy-Efficient.** According to end users demands, the performance and energy overheads of memory safety solutions have to be as low as possible, and definitely not greater than those due to programming in a memory safe language (unfortunately most software-based solutions will not meet this requirement). Further, while the increased awareness of security is a recent (and welcome) development, as performance gains wither in the post-accelerator age, it is likely that system designers will perform increasingly heroic, and perhaps, riskier optimizations that can have unknown security consequences. Califorms fits these requirements and has one of the lowest overheads reported among prior work.
- **Applicable to a Large Fraction of Devices.** Security is a full-system property, i.e., security of the system is defined by the most insecure entity in the environment. For instance, consider a home full of connected devices where a firewall sanitizes external accesses. Even if the firewall works as advertised, a (memory) vulnerability in the internet connectivity of one of the devices (e.g., thermal regulators, stoves, etc.) can completely compromise the security of the entire system. With limited silicon, power, and software budgets, introduction of non-trivial software and/or hardware schemes is a non-starter for IoT and cyber-physical systems. Yet, securing them is highly critical as these devices increasingly form the backbone of our infrastructure and personal lives. Califorms is not just limited to 64-bit architectures, in fact, its design is architecture width agnostic. Califorms can be easily applied across the spectrum of high-end enterprise to deeply embedded systems. In contrast, current hardware based whitelisting solutions, commercial or research, are limited to 64-bit architectures because of their fundamental design assumptions and complexities. Califorms's minimal footprint makes it a good primitive for security in the post-Moore's Law era.
- **Cost-Efficient to Develop and Deploy.** From the programmers point of view, there are two main impediments to the use of programming language approaches towards memory safety. First, programmers are often unwilling to change familiar workflows, and second, there is a significant amount of legacy code that is just too expensive/impossible to convert. Even annotations for security has proven to be difficult to bring to practice. While it would be valuable to program in newer languages, unsafe languages are here for the foreseeable future. Califorms is transparent and thus satisfies the requirements of (majority of) programmers who do not have the resources/incentive to invest in securing legacy code. For hardware vendors, its simplicity implies Califorms can be integrated in their existing design, even in the presence of tight development and power/area budgets, and importantly, easily validated.

Impact on Researchers and Industry. The key design choice which enabled Califorms to meet these goals is to implement *memory blacklisting* in hardware. Whitelisting approach currently dominates in the research community when facing memory safety problems. However, in this paper we advocate a blacklisting approach (as opposed to a whitelisting approach) and encourage researchers/industry to conduct further research in this direction. 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. Additionally, blacklisting techniques complement defenses in existing systems better since they do not require intrusive changes.

Finally, Califorms is a broader and more general concept than what is presented in the paper, which has potential applications other than memory safety (not restricting itself to security applications). For instance, since it can essentially be used to mark boundaries, we imagine Califorms can be used to enable and throttle aggressive and/or inaccurate microarchitectural speculation and prediction mechanisms. Exploring these benefits are topics of our current research.

CITATION FOR TEST OF TIME AWARD IN 10 YEARS

This paper makes the first workable byte-granular memory safety solution for a diverse environment ranging from high-performance general purpose to resource-constrained IoT devices.