AFRL-RI-RS-TR-2022-136

SHUFFLED BLOCKLISTED MEMORY (MOMS)

COLUMBIA UNIVERSITY

SEPTEMBER 2022

FINAL TECHNICAL REPORT

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED

STINFO COPY

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE

■ **AIR FORCE MATERIEL COMMAND** ■ **UNITED STATES AIR FORCE** ■ **ROME, NY 13441**

NOTICE AND SIGNATURE PAGE

Using Government drawings, specifications, or other data included in this document for any purpose other than Government procurement does not in any way obligate the U.S. Government. The fact that the Government formulated or supplied the drawings, specifications, or other data does not license the holder or any other person or corporation; or convey any rights or permission to manufacture, use, or sell any patented invention that may relate to them.

This report is the result of contracted fundamental research deemed exempt from public affairs security and policy review in accordance with SAF/AQR memorandum dated 10 Dec 08 and AFRL/CA policy clarification memorandum dated 16 Jan 09. This report is available to the general public, including foreign nations. Copies may be obtained from the Defense Technical Information Center (DTIC) (http://www.dtic.mil).

AFRL-RI-RS-TR-2022-136 HAS BEEN REVIEWED AND IS APPROVED FOR PUBLICATION IN ACCORDANCE WITH ASSIGNED DISTRIBUTION STATEMENT.

FOR THE CHIEF ENGINEER:

/ S /
JONATHAN HEINER
Work Unit Manager

/ S /
GREGORY HADYNSKI
Computing & Communications Division
Information Directorate

This report is published in the interest of scientific and technical information exchange, and its publication does not constitute the Government's approval or disapproval of its ideas or findings.

REPORT DOCUMENTATION PAGE

1. REPORT DATE	2. REPORT TYPE	3. DATES COVERED	
		START DATE	END DATE
SEPTEMBER 2022	FINAL TECHNICAL REPORT	JULY 2020	APRIL 2022

4. TITLE AND SUBTITLE

Shuffled Blocklisted Memory (MOMS)

5a. CONTRACT NUMBER	5b. GRANT NUMBER	5c. PROGRAM ELEMENT NUMBER
FA8750-20-C-0210	N/A	62251D
5d. PROJECT NUMBER	**5e. TASK NUMBER**	**5f. WORK UNIT NUMBER**
		R308

6. AUTHOR(S)

Simha Sethumadhavan, Mohamed Tarek, Miguel Arroyo, Evgeny Manzhosov, Ryan Piersma

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Columbia University
116th & Broadway
New York NY 10027

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSOR/MONITOR'S ACRONYM(S)	11. SPONSOR/MONITOR'S REPORT NUMBER(S)
Air Force Research Laboratory/RITA 525 Brooks Road Rome NY 13441-4505	AFRL/RI	AFRL-RI-RS-TR-2022-136

12. DISTRIBUTION/AVAILABILITY STATEMENT

Approved for Public Release; Distribution Unlimited. This report is the result of contracted fundamental research deemed exempt from public affairs security and policy review in accordance with SAF/AQR memorandum dated 10 Dec 08 and AFRL/CA policy clarification memorandum dated 16 Jan 09.

13. SUPPLEMENTARY NOTES

14. ABSTRACT

Current software- and hardware-based memory safety solutions suffer from one or more of the following limitations: having complex metadata, lack in binary compatibility, offering incomplete protection, and being vulnerable to side-channels. Our proposed approach, MOMS, addresses these limitations as it inlines the necessary metadata for enforcing memory safety within the program data and uses fine-grained permutation to provide resiliency against certain classes of side-channel attacks. Further, our novel inlined metadata highly reduces the performance overheads of achieving memory safety while providing byte-granular protection and maintaining very low hardware overheads. Thus, our approach addresses the long standing problem of securing programs written in memory unsafe languages, such as C and C++.

15. SUBJECT TERMS

Hardware-based cyber, blacklisting, canary, data-centric cyber protections

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES
a. REPORT	b. ABSTRACT	C. THIS PAGE		
U	U	U	SAR	26

19a. NAME OF RESPONSIBLE PERSON	19b. PHONE NUMBER *(Include area code)*
JONATHAN HEINER	N/A

PREVIOUS EDITION IS OBSOLETE.

STANDARD FORM 298 (REV. 5/2020)
Prescribed by ANSI Std. Z39.18

TABLE OF CONTENTS

Section Page

LIST OF FIGURES

Figure Page

LIST OF TABLES

Table Page

1. SUMMARY

In this section we provide a brief overall view of the effort. Our project goal was to develop new memory safety mechanisms that provide high security guarantees for C/C++ programs at low hardware complexity and negligible performance overheads.

Current software- and hardware-based memory safety solutions suffer from one or more of the following limitations: having complex metadata, lack in binary compatibility, offering incomplete protection, and being vulnerable to side-channels. Our proposed approach, Shuffled Blocklisted Memory (MOMS), addresses these limitations as it inlines the necessary metadata for enforcing memory safety within the program data and uses fine-grained permutation to provide resiliency against certain classes of side-channel attacks. Further, our novel inlined metadata highly reduces the performance overheads of achieving memory safety while providing byte-granular protection and maintaining very low hardware overheads. Thus, our approach addresses the long standing problem of securing programs written in memory unsafe languages, such as C and C++.

2. INTRODUCTION

Memory safety violations in programs 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 [14]. Similarly, the Project Zero team at Google reports that memory corruption issues are the root cause of 68% of listed Common Vulnerability Enumerations (CVEs) for zero-day vulnerabilities between 2014 and 2019 [11].

To address the threat of memory safety, software checking tools (e.g., AddressSanitizer [15]) and fuzz testing are widely deployed. In software fuzz testing, binaries are instrumented with a tool like AddressSanitizer to detect memory safety vulnerabilities and run with inputs mutated from a set of exemplary inputs in the hopes of detecting bugs before deployment. Google has reported that it has been fuzzing about 25,000 machines continuously since 2016, which has resulted in the identification of many critical bugs in software such as Google Chrome and several open source projects [16]. Assuming 15 cents per central processing unit (CPU) hour for large memory machines—a requirement for reasonable performance on fuzz testing—the investment in software fuzzing for detecting memory errors could be close to a billion dollars at just one company.

Despite a Herculean effort by software vendors, memory safety vulnerabilities continue to slip through, ending up in deployed systems. Recognizing that pre-deployment fuzz tests can never be complete, companies have also proposed postdeployment crowdsourced fuzz testing [17], [59]. For instance, Mozilla recently created a framework for fuzzing software using a cluster of fuzzers made by users who are willing to trade and contribute their CPU resources (e.g., using office workstations after-hours for fuzz testing) [17]. Assuming that many companies participate and these tests run for enough time, on a global scale, the amount of energy invested in producing reliable software may be even higher than the amount of time running the software with crowdsourced testing. Thus, increasing the efficiency of memory error detection can have significant green benefits in addition to improving security and reliability.

Researchers and commercial vendors have also stepped up to the call to reduce inefficiencies in software testing and security. There is a long history of academic proposals that have continuously chipped away at these overheads for detecting memory safety vulnerabilities over the past 25 years. Commercial vendors have also proposed or manufactured hardware with support to mitigate these overheads (Intel's memory protection extensions (MPX) [8], ARM's memory tagging extension (MTE) [10], and Oracle's application data integrity (ADI) [9]).

In this report, we show that the overheads of providing memory safety can be decreased even further with novel hardware support. We present MOMS, a technique that integrates memory blocklisting with fine-grained permutation to provide resiliency against memory safety errors and certain classes of side-channel attacks (e.g., Meltdown). To blocklist illegal memory accesses, MOMS uses a novel Cache Line Formats, dubbed Califorms [13]. Califorms uses a two-fold approach for reducing memory safety overheads. First, instead of checking access bounds for each pointer access, Califorms blocklist all memory locations that should never be accessed. This reduces the additional work for memory safety such as comparing bounds.

Second, Califorms uses a novel metadata storage scheme for storing blocklisted information. The key observation is 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 automatically insert them using compiler transformation. In order to distinguish the dead bytes from normal bytes in memory, Califorms uses a compressed encoding that requires one bit per each 64B cache line. If an attacker accesses these dead (i.e., blocklisted) regions, we detect this rogue access without any additional metadata accesses as our metadata resides inline. To randomize the locations of the blocklisted (i.e., Califormed) bytes at runtime per each allocation, we use the allocation base address as a key for permuting the contents of the allocation data and blocklisted bytes.

The rest of the report is organized as follows. Section 3 presents the overall system overview. It details the instruction set extensions, hardware design, software changes, and operating system support needed for MOMS. We provide a detailed security analysis of the proposed solution in Section 4. Afterwards, we evaluate the hardware and software overheads of MOMS on 64- and 32-bit systems.

3. METHODS, ASSUMPTIONS, AND PROCEDURES

Our project can be broken into three main components that we discuss in the following subsections. First, the instruction set architecture (ISA) extensions, where we introduce new permuted load/store variants, alongside with instructions to compute the permutation and blocklist memory. Second, the hardware changes, where we introduce a permutation unit to the processor and add minimal modifications to the cache architecture, namely a new level 1 data cache (L1-D) permutation cache and cache converters between level 1 (L1) and level 2 (L2) caches. Third, the software design, where we use compiler support to take advantage of the new instructions and minimal operating system (OS) support to handle exceptions.

3.1 Instruction Set Architecture Extensions

MOMS adds the following instructions to the instruction set architecture (ISA).

3.1.1 pStore/pLoad <R1>, <R2>,<R3>

These instructions use three register operands. The values in registers R1 and R2 point to the store/load address and source/destination register as usual. The value in register R3 is reserved for the allocation permutation and is propagated by the compiler. Upon executing this instruction, the hardware uses the indexes in the permutation register to decide upon how to access the data cache.

3.1.2 computePerm <R1>,<R2>

This instruction takes a memory address (i.e., pointer) as input in R1 and returns the allocation permutation of this pointer in R2. This instruction is used to retrieve the correct permutation of pointers that are passed to different contexts (e.g., through function calls).

3.1.3 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.

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

	R2, R3		
	X, !Allow	!Set, Allow	Set, Allow
Regular Byte	Regular Byte	Exception	Security Byte
Security Byte	Security Byte	Regular Byte	Exception

3.2 Hardware Design

This section describes the main hardware components of MOMS, as shown in Figure 1.

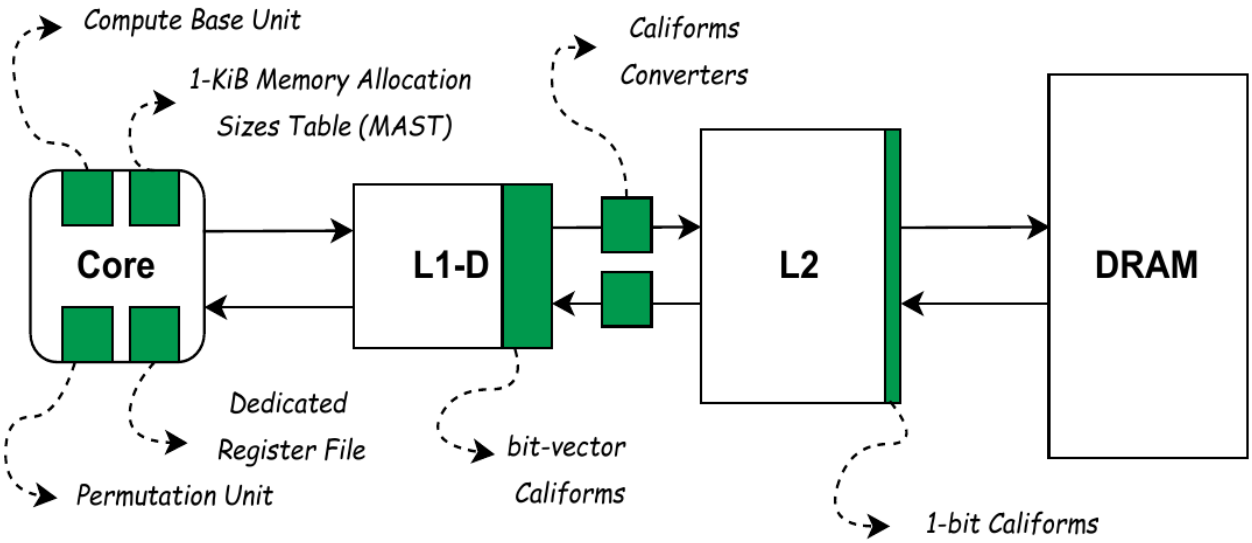

Figure 1: Hardware Block Diagram with MOMS components

3.2.1 Memory Allocation Size Table

The Memory Allocation Size Table (MAST) is a hardware structure, which is initialized at program startup with a process's allocation size configuration. The table is designed to work with binning allocators. The MAST enables MOMS to support generic (i.e., non-powers-of-two) allocation sizes for each memory allocator bin.

3.2.2 Base Computing Module

Pointers can be passed from one context to another. As MOMS relies on simple intra-procedural compiler analysis, it needs to recompute the base address every time a pointer is loaded from memory (e.g., double pointers) or used as a function argument. This feature is currently implemented as part of the computePerm instruction that invokes the Base Computing Module followed by the permutation unit.

In this work, we use a simple binning allocator that divides the heap into N equally sized bins. Based on our experiments, using 64 distinct bins is sufficient to balance performance and memory utilization. Thus, we use a 64-entry MAST with an entry size of 16B resulting in a total size of 1KB. Each entry holds an 8B size field and an 8B inverse size field. The size field of the nth entry is used to hold the allocation size used for the nth allocator bin. The inverse size field is an optimization that is discussed later. As a program's heap is contiguous, we use a single hardware register to store the starting address of the program heap and use it to derive the starting address of all bins. Some binning allocators (e.g., TCmalloc and Jemalloc) may change the allocation size used by one bin at runtime if all objects in the bin are freed. In this case, the allocator can simply update the MAST entry with the new size.

This module takes a 64-bit pointer operand, `ptr`, and computes its base address using `floor(ptr/size(ptr)) * size(ptr)`. While `size(ptr)` requires one MAST lookup, the division operation is costly. MOMS uses a common optimization that replaces the expensive division `(ptr/size(ptr))` with a cheaper multiplication `(ptr * (1/size(ptr)))` by using fixed-point arithmetic. This approach is feasible since the set of allocation sizes is constant, and thus the set of allocation size reciprocals can be pre-calculated and stored in the MAST alongside with allocation size.

3.2.3 Permutation Unit

The central component of our hardware modifications is the permutation unit. We use a Benes network capable of creating (n!) permutations out of n inputs, as shown in Figure 2. Given an ordered indexes as input, we generate permutation indexes that are used to define how data bytes are laid in memory. The output of the permutation unit is stored in a dedicated register file. The control signals for this network are driven from the object base address, size, and the per-process key. The Benes network has a secondary benefit of being scalable, meaning, the network can grow or shrink dependent on the die area/processor requirements.

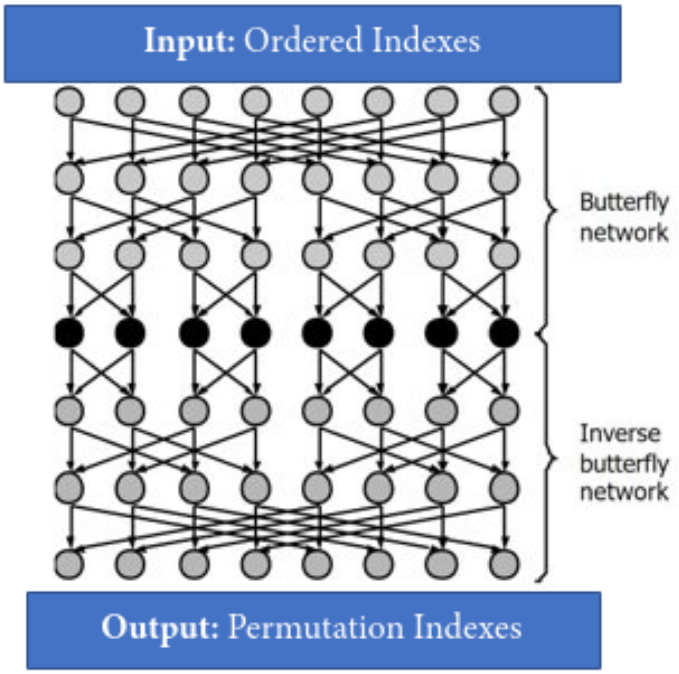

Figure 2: Permutation Network

3.2.4 Dedicated Register File

As our pLoad and pStore instructions use a third register operand, they may introduce register pressure. Thus, MOMS adds a set of architectural registers that the compiler can exclusively use for holding and propagating the permutation vector. The new registers are saved in a separate register file that is accessed in parallel to the regular register file

3.2.5 Califorms Hardware Extensions

MOMS use Califorms for blocklisting illegal memory accesses. The microarchitectural support for Califorms aims to keep the common case fast: L1 cache uses the straightforward scheme 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.

3.3 Software Design

In this section, we describe the memory allocator, compiler and operating system changes to support our project.

3.3.1 Dynamic Memory Management

One of MOMS' key contributions is making the allocation size an architectural feature (i.e., sharing the allocation size information between software and hardware). To enable this feature, MOMS requires binning memory allocators, in which a memory page is used to allocate objects of the same size. MOMS does not add any constraints on how the allocator manages its internal metadata (e.g., free lists). MOMS only intercepts calls to common memory management operations. For example, MOMS intercepts all calls to malloc/new and tags the returned pointer with a random 16-bit value for ensuring temporal memory safety. Additionally, MOMS issue BLOC instructions to mark the bytes of allocated memory as blocklisted/permit-listed based on the paddings layout.

Upon deallocation, MOMS intercepts the calls to free/delete to remove the tag bits and mark the free'd bytes as blocklisted before calling the allocator's own free/delete API.

3.3.2 Compiler Support

3.3.2.1 Source-to-Source Transformation.

In order to achieve intra-allocation memory safety, we use a source-to-source transformation (Califorms). Califorms is implemented using Clang's rewriter interface. First, we perform an abstract syntax tree (AST) traversal over each translation unit to collect a whole program view of composite data types (e.g., structs) and their usages. This step allows us to infer where to place security bytes within target objects, based on their type layout information. Then, we perform a second traversal to perform the actual rewriting.

Our Califorms transformation supports three insertion policies: The first opportunistic policy supports security bytes insertion into existing padding bytes within the objects. The second full policy inserts security bytes between all fields within the object. The third intelligent policy modifies the object layout to introduce randomly sized security bytes around security critical fields only, such as arrays and pointers. 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.

3.3.2.2 Heap Instrumentation.

To guarantee spatial protection, we implement an instrumentation pass at the low level virtual machine (LLVM) intermediate representation (IR) level that replaces program loads and stores with our new instructions, pLoad and pStore. To prepare the allocation base address register operand, we use simple function-level analysis to propagate the pointers returned by malloc or new intra-procedurally. To handle out-of-context pointers (e.g., those that are loaded from memory or passed as function arguments), our compiler pass inserts computePerm instructions in the corresponding locations to resolve the allocation base address and generate the corresponding permutation.

3.3.2.3 Stack & Global Instrumentation.

In order to achieve full memory safety on all memory segments, we extend MOMS to protect objects that are allocated on the stack and global memory. At compile time, MOMS instruments all stack and global allocations (e.g., alloca) to use the same bins, which are used to satisfy heap allocations. This way MOMS uses a unified method to enforce memory safety on all program memory segments. To avoid overheads related to allocating stack objects on the heap, we adopt the same pointer mirroring and memory aliasing techniques used in prior work [7].

3.3.3 Operating System Support

3.3.3.1 MAST Initialization.

During program initialization, the memory allocator needs to pass the allocation size information to the hardware. This is a one time task that can be done with a special system call or by writing to a hardware-mapped memory region. The size of the table is fixed, as described in Section 3.2.1.

3.3.3.2 Context Switching.

Upon a context switch, MOMS requires the operating system (OS) to store the MAST (and the dedicated register file contents) of the interrupted process and update the MAST and register file of the new process. Both the MAST and the register file contents are of fixed size and can be stored as part of the process control block. This step is likely to add minimal overhead (a few load and store instructions takes ≤ 0.1μS) to the OS context switch (typically 3–5μS).

3.3.3.3 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 permit-listing functionality to suppress the exceptions.

3.3.3.4 Page Swaps.

As MOMS relies on Califorms to detect access violation, 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 metadata for all the processes on the system since the size of the metadata is minimal (8B for a 4KB page or 0.2%).

3.4 Security Analysis

In this section, we discuss how MOMS is resilient to common exploit types.

3.4.1 Buffer under-/over-flows.

MOMS defends against the exploitation of buffer overflows (and underflows) by hiding the mapping between program objects and their actual layout in memory. Even if the attacker has access to the source code and/or binary image of the victim program, they cannot infer the layout of the victim object. The same object can have multiple layouts based on its location in memory. With high probability, with MOMS in place, the attacker cannot corrupt or leak information that can be used to mount many exploit variants. Our protection applies to both inter- and intra-object safety as Califorms blocklisted bytes can detect intra-object violations.

3.4.2 Use-after-frees.

MOMS provides temporal memory safety via the alias address space. The same allocated virtual/physical memory region can have up to 2^16 different aliases (each alias having its own permutation). This alias address space is sufficient to thwart use-after-free attacks, in which the type of the freed object aligns with the confused type of the new object, having a big impact on the reliability of these exploits [5].

Heap Feng Shui attacks exploit a memory allocator's determinism to arrange memory so that it is favorable for an attacker to manipulate a victim allocation. Similarly, MOMS relies on the alias address space to generate multiple permutations for the same memory region, making it impractical to infer any information about an object layout based on another object, even if both have the same type.

3.4.3 Uninitialized Reads.

While MOMS does not explicitly zero out memory that may have held security sensitive data, the fact that the memory is left permuted after a free is sufficient in many cases. The next program to use the same memory region will be assigned a different permutation key by default, making it impractical to recover the data by mistake. Additionally, an attacker trying to access this sensitive data would need the appropriate permutation in order to unscramble the memory. The same applies when peeking at memory with a variadic function misuse attack [6].

3.4.4 Control-Flow Hijacking and Data-Oriented Attacks.

Given a memory error, attackers can gain arbitrary memory read/write primitives. Attackers can then leverage such primitives to launch different attacks, such as control-flow hijacking, information leakage, or data-only attacks. MOMS effectively mitigates all of those attacks as it makes it harder for the attacker to utilize the memory read/write primitives. For instance, it is impractical to hijack the control-flow of the program (e.g., by overwriting a function pointer in a C struct) if the whole struct is permuted with 64 factorial different permutations. The same argument holds even for the more critical data-only attacks that corrupt the program without changing its control-flow. MOMS provides sufficient probabilistic guarantees for protecting the security- critical data structures of a program.

3.4.5 Memory errors in uninstrumented code.

MOMS unpermutes the data that is passed to uninstrumented code (e.g., library functions) while the rest of the program data remains permuted. So, if the uninstrumented code has a memory vulnerability it may only reliably corrupt the portion of data that is passed to it. Unlike other techniques that provide no security guarantees for uninstrumented code, MOMS reduces the attack surface by keeping the rest of the program data permuted.

3.4.6 Speculative Execution Attacks.

With MOMS, utilizing speculative exploits is more challenging for an attacker. Not only is the data permuted, but the speculative (instrumented) load additionally uses a different permutation to access the permuted data. Consider an attacker that tries to speculatively load the secret value a[i] using an out-of-bound index, i, as shown in Listing 1. In this case, they will end up with an unpredictable value in secret due to our security primitives which permute the address of a[i].

Listing 1: Example speculative execution attack.

```
if (i < sizeof(a)) { // mispredicted branch
    secret = a[i];
    val = b[64 * secret]; // secret is leaked
}
```

3.4.7 Hardware Memory Violations.

Let us consider an attacker who wants to leak a function pointer from a struct that has ten other fields. Using a side-channel, the attacker leaks the entire struct. However, due to our permutation they would not be able to recognize the needed function pointer (or even reconstruct the struct layout). By the same principle, MOMS provides indirect protection against other types of attacks such as RowHammer and ColdBoot attacks.

4. RESULTS AND DISCUSSION

We evaluate MOMS across multiple dimensions. First, we measure the hardware overheads of our changes. Second, we analyze the performance overheads of MOMS using the Standard Performance Evaluation Corporation (SPEC) CPU2017 benchmarks.

4.1 Hardware Evaluation

MOMS requires minimal hardware changes to the core and caches. Qualitatively, MOMS requires a 1KB MAST and extra logic to compute the allocation base address (namely, one subtract, one shift, two 64-bit multipliers), a permutation network, and additional state and operations to the L1 data cache and the interface between the L1 and L2 caches. We implement all the designs using Verilog and synthesize them with the Synopsys Design Compiler and the 45nm NangateOpenCell library. We generate the static random access memory (SRAM) arrays (for MAST, Califorms bits, and the tag/data arrays) with OpenRAM [2].

4.1.1 Permutation Unit Implementation

As mentioned in Section 3.2.3, we implement the core of the Permutation Unit as a Benes network. We implement three network configurations to showcase the flexibility of the approach: 16-by-16, 32-by-32, and 64-by-64. Table 2 summarizes the results of the permutation networks implementation. Overall, all three networks have minimal latencies of 440 ps to 580 ps, modest cell counts, and area/power overheads. As expected, as the number of inputs increases, the area/power consumption increases as well at approximately the same rate. For example, a 32-by-32 network uses almost double the area (an increase of 132%), double the cell count (an increase of 137%), and twice the power – 19.5 mW, compared to a 16-by-16 network.

Table 2: Summary of the VLSI implementation of the Benes Networks

Network	Clock Period [ns]	Area [m^2]	Cell Count	Power [mW]
16-by-16	0.440	2693	1370	9.18
32-by-32	0.500 (+13.6%)	6260 (+132%)	3254 (+137%)	19.5 (+112%)
64-by-64	0.580 (+31.8%)	14172 (+426%)	7561 (+451%)	39.6 (+331%)

4.1.2 Cache Implementation

The metadata area overhead of our L1 changes (aka 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 hardware overheads by adding MOMS to a typical energy-optimized 32KB direct-mapped L1 cache. The baseline cache is made of eight 8B SRAM banks, each storing 512 words. To enable the permutation of the data, we break down the 8B cache SRAM banks into eight single-byte banks and add scatter-gather circuitry to enable the read/write of data with the byte granularity. Thus, upon each read/write only the needed banks are used, reducing the power consumption of the cache.

Table 3 summarizes the very large scale integration (VLSI) implementation results of the caches. The percentages in the parenthesis indicate the increase compared to the appropriate baseline, except for the 8b banks baseline configuration, which is compared to the 64b banks baseline. For example, 8b banks + Perm has a 0.9% longer clock cycle compared to the 8b banks baseline. Overall, the impact of MOMS on the clock cycle of the cache is minimal – in the worst case an increase of 2%. With the new design, the overheads of the SRAM decoders and encoders are not shared by 8B of data but instead are confined to each single byte bank. Thus, the area and power overheads are mostly caused by the breakdown of 8B SRAM banks in the cache into eight single-byte banks: an increase of 103% (64b banks and 8b banks baselines in Table 3).

Table 3: Summary of the VLSI implementation of the caches

Design	Clock Period [ns]	Area [m^2]	Memory Area [m^2]	Cell Count	Power [mW]
64b banks, baseline	1.520	1342775	53910	2913	61
64b banks, Cforms	1.540 (+1.3%)	1677699 (+24.9%)	388185	3274 (+12.4%)	74 (+21.3%)
8b banks, baseline	1.525	2734049 (+103%)	2728109	3046	110
8b banks, Perm	1.540 (+0.9%)	2750885 (+0.6%)	2728109	16054 (+427%)	121 (+10%)
8b banks, Perm+Cfroms	1.555 (+2%)	3085690 (+12.8%)	3062384	16248(+433%)	134 (+22%)

4.2 Software Evaluation

Our VLSI measurements show that MOMS hardware modifications add no performance overhead. In this section, we evaluate the software-based overheads. MOMS instructions pLoad and pStore are similar to regular loads and stores. Thus, they do not increase code size. While our instructions use one more register operand compared to regular memory instructions, the extra register pressure is compensated for by adding a MOMS-specific register file (i.e., similar to Intel MPX). The additional functionality performed by our instructions can be totally hidden within the processor pipeline. However, MOMS requires a binning memory allocator and invokes additional instructions (BLOC to mark certain memory bytes as blocklisted and computePerm to compute the allocation base address and permutation of arbitrary pointers when they are loaded from memory).

Without loss of generality, we implement MOMS on top of a simple binning allocator (Binning-Malloc [3]) that divides the virtual memory into 64 regions, each of size 32GB. Each region is used to satisfy heap-allocation requests of a unique size. Stack and global memory allocations are satisfied using special carved out sections of the same 32GB regions. To estimate the computePerm instruction overheads, we implement an IR pass using the LLVM/Clang compiler [4] to instrument the code and insert two multiply, mul, instructions followed by an XoR and a store instead of computePerm instructions in the corresponding locations. The additional XoR instruction is used to emulate the additional one cycle latency for the permutation network. We use a store to make sure the instruction is not omitted by compiler optimizations.

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 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 create

wrappers for memory allocations and deallocations (i.e., malloc/new and free/delete). 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.

4.2.1 Evaluation Setup.

We run our experiments on a bare-metal Intel Skylake-based Xeon Gold 6126 processor running at 2.6GHz with Red Hat Enterprise Linux (RHEL) 7.5 (kernel 3.10). We implement MOMS using Clang 4.0.0 and normalize its performance against a vanilla baseline. To get better insight on MOMS overheads, we also run a malloc-only version that only uses the binning memory allocator with no bounds checking (BinningMalloc) and a permutation-only version that shuffles memory objects without adding the blocklisted bytes (PermutationOnly). Our MOMS variant includes the overheads of using the binning allocator, the permutation, and the Califorms bytes around sub-object arrays (to handle intra-object overflows). Furthermore, we compare MOMS against AddressSanitizer (ASan) and Intel MPX, as representatives of pre- and post-deployment memory safety solutions, respectively.

We use the SPEC CPU2017 benchmark suite with ref inputs and run to completion. To minimize variability, each benchmark is executed 5 times and the average of the execution times is reported.

4.2.2 Performance Results.

Figure 3 summarizes the performance overheads of SPEC CPU2017 for different configurations normalized to the Vanilla (insecure) baseline using 64-bit binaries. The geometric mean of each tool is as follows: ASan (2.07x), MPX (2.06x), SoftwareEBB (2.0x), Binning-Malloc (1.05x), Permutation (1.11x), and MOMS (1.12x). The main reason for MOMS overheads comes from the underlying memory allocator, which introduces 5% overheads, and the permutation, which introduces an additional 6% runtime cost. Our Califorms intelligent policy only adds 1% overheads on top of the permutation and binning memory allocator.

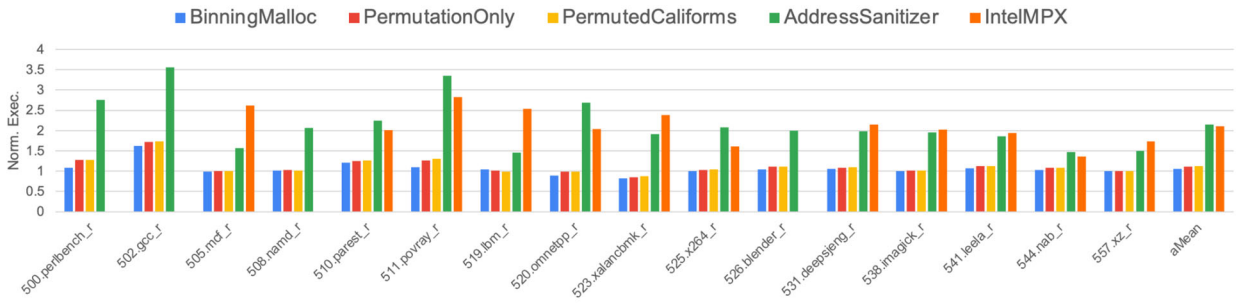

Figure 3: Performance overheads of the SPEC CPU2017 benchmarks for different tools normalized to their corresponding baseline using 64-bit binaries

We repeated the analysis with 32-bit executables on the same real x86 machine. Figure 4 shows the performance overheads for the different tools. We noticed negligible overheads for the

modified binning allocator whereas the permutation-only and permuted Califorms (aka MOMS) causes 4% and 5% slowdowns, respectively.

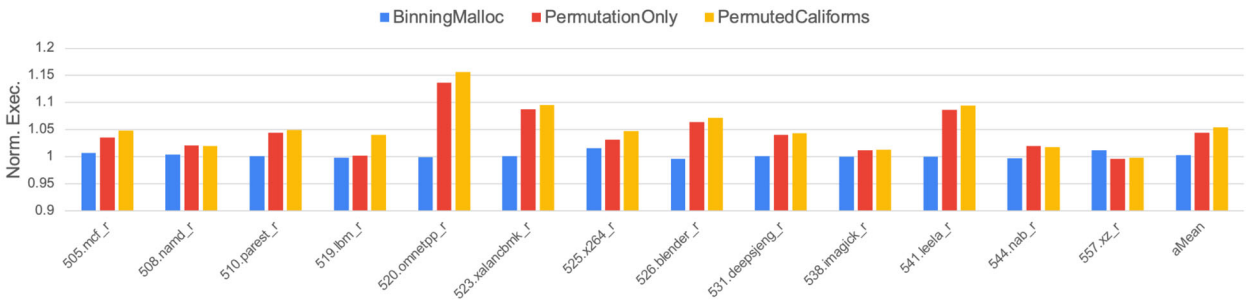

Figure 4: Performance overheads of the SPEC CPU2017 benchmarks for different tools normalized to their corresponding baseline using 32-bit binaries

5. CONCLUSIONS AND FUTURE WORK

The project shows that it is possible to achieve high levels of security at low complexity with low performance overhead. We present a novel hardware support, called MOMS, which combines byte-granular memory blocklisting with data permutation in order to provide resiliency against memory safety vulnerabilities and certain side-channel attacks such as Meltdown.

A key observation behind MOMS is that a blocklisted 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 blocklist a region. This in-place compact data structure avoids additional operations for fetching the metadata making it very performant compared to prior work. Additionally, we substantially reduce the hardware area overheads by changing how data is stored within a cache line. Subsequently, if the processor accesses a blocklisted byte or a security byte, due to programming errors or malicious attempts, it reports a privileged exception.

In order to randomize the locations of the blocklisted bytes at runtime, we use a hardware-based permutation network that generates a unique memory layout per each memory allocation using the allocation base address as a key. This way the allocation data and blocklisted bytes are randomly shuffled in memory, increasing the chances of detecting memory safety attackers and making it harder for side-channel attacks to recover the original allocation layout.

Our VLSI implementation results show that MOMS can be integrated into the microarchitecture with low area and power overheads. Furthermore, our software evaluation shows that MOMS introduces low runtime overheads for both 32- and 64-bit systems.

REFERENCES

[1] Air Force Research Laboratory, "American National Standard Institue / National Information Standards Organziation (ANSI/NISO) Standard Z39.18-2005," AFRL, 2010.

[2] M. R. Guthaus, J. E. Stine, S. Ataei, B. Chen, B. Wu, and M. Sarwar, “OpenRAM: An open-source memory compiler,” in ICCAD ’16: Proceedings of the 35th International Conference on Computer-Aided Design, Austin, TX, USA, 2016.

[3] G. J. Duck and R. H. C. Yap, “An extended low fat allocator API and applications,” arXiv preprint arXiv:1804.04812, 2018.

[4] C. Lattner and V. Adve, “LLVM: a compilation framework for lifelong program analysis transformation,” in CGO ’04: Proceedings of the International Symposium on Code Generation and Optimization, San Jose, CA, USA, 2004, pp. 75–86.

[5] GoogleProjectZero, “One perfect bug: Exploiting type confusion in flash,” 2015. [Online]. Available:https://googleprojectzero.blogspot.com/2015/07/one-perfect-bug-exploiting-type20.html

[6] C. W. Enumeration, “CWE-134: Use of externally-controlled format string.” [Online]. Available: https://cwe.mitre.org/data/definitions/134. html

[7] G. J. Duck, R. H. C. Yap, and L. Cavallaro, “Stack bounds protection with low fat pointers,” in NDSS ’17: Proceedings of the 24th Annual Network and Distributed System Security Symposium, San Diego, CA, USA, February 2017.

[8] O. Oleksenko, D. Kuvaiskii, P. Bhatotia, P. Felber, and C. Fetzer, “Intel mpx explained: A cross-layer analysis of the intel mpx system stack,” Proceedings of the ACM on Measurement and Analysis of Computing Systems, vol. 2, no. 2, p. 28, 2018.

[9] Oracle, “Hardware-assisted checking using silicon secured memory (SSM),” 2015. [Online]. Available: https://docs.oracle.com/cd/E3706901/html/E37085/gphwb.html

[10] ARM, “Memory tagging extension: Enhancing memory safety through architecture,” https://community.arm.com/developer/ip-products/processors/b/processors-ip-blog/posts/enhancing-memory-safety , 2019, [Online].

[11] GoogleProjectZero, “0day in the wild,” 2019. [Online]. Available: https://googleprojectzero.blogspot.com/p/0day.html

[12] L. Szekeres, M. Payer, T. Wei, and D. Song, “SoK: Eternal war in memory,” in SP ’13: Proceedings of the 2013 IEEE Symposium on Security and Privacy, San Francisco, CA, USA, 2013, pp. 48–62.

[13] H. Sasaki, M. A. Arroyo, M. Tarek Ibn Ziad, K. Bhat, K. Sinha, and S. Sethumadha- van, “Practical byte-granular memory blacklisting using Califorms,” in MICRO’52: Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture, Columbus, OH, USA, 2019, 558—571.

[14] M. Miller, “Trends, challenges, and strategic shifts in the software vulnerability mitigation landscape,” 2019. [Online]. Available: https://github.com/microsoft/MSRC-Security-Research/blob/master/presentations/2019_02_BlueHatIL/2019_01%20-%20BlueHatIL%20-%20Trends%2C%20challenge%2C%20and%20shifts%20in%20software%20vulnerability%20mitigation.pdf

[15] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, “AddressSanitizer: a fast address sanity checker,” in ATC ’12: Proceedings of the 2012 USENIX Annual Technical Conference, 2012.

[16] M. Bohme and B. Falk, “Fuzzing: On the exponential cost of vul- ¨ nerability discovery,” in Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, 2020, pp. 713–724.

[17] MozillaSecurity, “VIRGO: Crowdsourced fuzzing cluster,” 2019. [Online]. Available: https://github.com/MozillaSecurity/virgo

APPENDIX A - Publications

[1] Hiroshi Sasaki, Miguel A. Arroyo, Mohamed Tarek Ibn Ziad, KoustubhaBhat, Kanad Sinha, and Simha Sethumadhavan. “Practical Byte-Granular Memory Blacklisting using Califorms”, In MICRO-52, Columbus, Ohio, USA, October 2019.

[2] Mohamed Tarek Ibn Ziad, Miguel A. Arroyo, Evgeny Manzhosov, R. Piersma, and Simha Sethumadhavan. “No-FAT: Architectural Support for Low Overhead Memory Safety Checks”, In ISCA-48, June 2021.

[3] Mohamed Tarek Ibn Ziad, Miguel A. Arroyo, and Simha Sethumadhavan. “SPAM: Stateless Permutation of Application Memory”, Technical Report, https://arxiv.org/pdf/2007.13808.pdf

[4] Mohamed Tarek Ibn Ziad, Miguel A. Arroyo, Evgeny Manzhosov, and Simha Sethumadhavan. “ZeRØ: Zero-Overhead Resilient Operation Under Pointer Integrity Attacks”, In ISCA-48, June 2021.

[5] Mohamed Tarek Ibn Ziad, Miguel A. Arroyo, Evgeny Manzhosov, Vasileios P. Kemerlis, and Simha Sethumadhavan. “EPI: Efficient Pointer Integrity For Securing Embedded Systems”, In SEED ’21, Worldwide Event, September 2021.

LIST OF SYMBOLS, ABBREVIATIONS AND ACRONYMS

ADI	Application Data Integrity
ASan	Address Sanitizer
AST	Abstract Syntax Tree
BLOC	Blocklisting LOCation
Califorms	Cache Line Formats
CPU	Central Processing Unit
CVE	Common Vulnerability Enumeration
IR	Intermediate Representation
ISA	Instruction Set Architecture
L1	Level 1 Cache
L1-D	Level 1 Data Cache
L2	Level 2 Cache
LLVM	Low Level Virtual Machine
MPX	Intel Memory Protection Extensions
MAST	Memory Allocation Size Table
MOMS	Mother Of all Memory Security techniques
MTE	Memory Tagging Extension
OS	Operating System
pLoad	permuted Load
pStore	permuted Store
RHEL	Red Hat Enterprise Linux
SPEC	Standard Performance Evaluation Corporation
SRAM	Static Random Access Memory
VLSI	Very Large Scale Integration