M-MAP: Multi-Factor Memory Authentication for Secure Embedded Processors

Syed Kamran Haider[†], Masab Ahmad[†], Farrukh Hijaz[†],
Astha Patni[†], Ethan Johnson[‡], Matthew Seita[*],
Omer Khan[†] and Marten van Dijk[†]

[†]University of Connecticut –
{syed.haider, masab.ahmad, farrukh.hijaz,
astha.patni, omer.khan, vandijk}@engr.uconn.edu
[‡]Grove City College, Grove City, PA – ethanjohnson@acm.org
[*]Rochester Institute of Technology, Rochester, NY – mss4296@rit.edu

August 26, 2015

Abstract

The challenges faced in securing embedded computing systems against multifaceted memory safety vulnerabilities have prompted great interest in the development of memory safety countermeasures. These countermeasures either provide protection only against their corresponding type of vulnerabilities, or incur substantial architectural modifications and overheads in order to provide complete safety, which makes them infeasible for embedded systems. In this paper, we propose M-MAP: a comprehensive system based on multi-factor memory authentication for complete memory safety, inspired by everyday user authentication factors. We examine certain crucial theoretical and practical implications of composing memory integrity verification and bounds checking protection schemes in a comprehensive system. Based on these implications, we implement M-MAP with hardware based memory integrity verification and software based bounds checking to achieve a balance between hardware modifications and performance. We demonstrate that M-MAP implemented on top of a lightweight out-of-order processor delivers complete memory safety with only 32% performance overhead on average, and incurs minimal hardware modifications and area overhead.

Contents

1 Introduction

The widespread proliferation of computer security vulnerabilities and attacks has generated intense interest in the development of processor architectures that provide inherent protection against weaknesses in both hardware and software [1]. Memory is fundamental to the concept of computation, and as such it is given a high degree of implicit trust by the central processing unit which relies on it. If an attacker can manipulate a computer's memory in even a limited fashion, he can often leverage this to wrest full control of program execution. For this reason, the most prevalent and dangerous classes of attacks have focused on exploitation of memory vulnerabilities [2].

Security attacks on a computer's memory can be broadly classified into software based and hardware based attacks. Figure 1 shows these two attack channels. Software based attacks are generally the attacks against *spatial* and/or *temporal* safety [3], and are launched by supplying a malicious input to the program, e.g. to exploit a *buffer overflow* vulnerability, through the legitimate I/O channels. In attacks on spatial safety, an attacker is able to abuse unchecked pointer dereferencing to read or write an inappropriate section of memory. Attacks on temporal safety involve performing a valid memory dereference at an inappropriate time: for instance, reading memory before it is initialized. Because of this indirect modification of memory by the adversary, we refer to such attacks as *indirect memory tampering* or simply indirect attacks. E.g., for buffer overflows, the attacker typically takes advantage of an unchecked string copy or a bounded memory copy, where the length of the structure to be copied is taken from user input, to write past the end of an array or structure and onto some other sensitive data. Other vulnerabilities in this class may allow direct arbitrary read/write of memory through a user-supplied pointer offset, or, more subtly, may inadvertently disclose sensitive regions of memory by making incorrect assumptions about the size or location of a structure being read (a famous example being the recent OpenSSL "Heartbleed" bug of 2014 [2]). Similarly reading memory that has since been reallocated can allow an attacker to forge data or function pointers. Attacks exploiting both spatial and temporal vulnerabilities have proven to be highly successful techniques, effective against many popular software products in the past and present. Hardware-based attacks on memory are the ones against its *data integrity* and *data freshness*, and typically involve various forms of "direct" attacks exploiting physical access to the device [4]. For performance reasons, many hardware devices, sockets, and ports both internal and external to a computer are given direct access to main memory. A malicious device can exploit this low-level access to read or write any portion of memory it has access to (often the entire 4GB 32-bit address space). Practical attacks of this nature have been demonstrated by easy-to-use tools such as Inception [5], which exploits DMA access through PCI/PCI Express interfaces such as FireWire, Thunderbolt, ExpressCard, and PC Card to completely bypass operating-system authentication mechanisms in systems running Windows, Mac OS X, or Linux [6]. We call such attacks as *direct memory tampering* or direct attacks.

Existing memory safety techniques can also be divided into two main groups. Most of the software based or indirect attacks can be prevented by *bounds checking* techniques which have been proposed in several different flavors. For hardware based or direct attacks, *memory integrity verification* schemes enable the trusted CPU to detect any illegitimate data modification [7]. We discuss both types of countermeasures in detail in the later sections.

Clearly, these schemes protect only against their corresponding type of attacks. Whereas a comprehensively secure processor architecture must provide protection against both hardware and software attacks as the system can be compromised if either one is possible. Embedded systems are often more vulnerable to both direct and indirect attacks because the adversary typically has

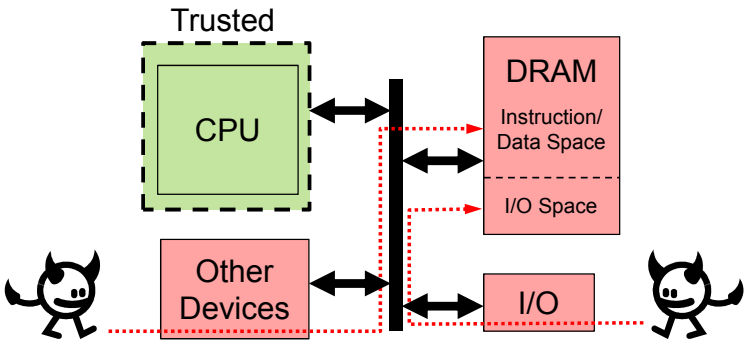


Figure 1: Memory Attacks: An adversary can alter the normal control flow of a program by either (a) Indirectly tampering with the main memory using legitimate I/O channels; or (b) Directly tampering with the main memory through a compromised device on the system bus.

the physical access to these devices. Therefore, such systems also need to prevent both types of attacks. As a direct consequence, in order to provide complete safety with minimal overheads, bounds checking and integrity verification techniques should coexist in the system. Potentially, these techniques can be implemented either in software, hardware or a combination of both.

In this paper, we propose and implement M-MAP: an architecture based on Multi-factor Memory Authentication for secure embedded Processors. We first create an analogy between the daily life user authentication and memory authentication based on multiple *authentication factors*, and identify three important memory authentication factors necessary for complete memory safety. Then we explore the design tradeoffs of the composition of different memory safety techniques to achieve these authentication factors. We analytically argue which compositions/flavors of bounds checking and integrity verification techniques are secure as well as feasible in terms of performance and required modifications to the existing systems. Based on these arguments, we propose M-MAP which implements both integrity verification and bounds checking in an efficient manner to provide all memory authentication factors. Since hardware based bounds checking requires substantial hardware changes [8], M-MAP implements it in software. On the other hand, integrity verification is implemented in hardware since it offers better performance than its software counterpart, and only requires minimal changes at DRAM-Cache boundary. We evaluate our proposed architecture for an in-order processor and an out-of-order processor, both tailored for secure embedded systems applications. Our experimental results demonstrate only 32% performance overhead on average compared to an insecure system. We also show that the composition of countermeasures may introduce new overheads, which are otherwise not applicable.

2 Background of Memory Safety Schemes

2.1 Bounds Checking for Spatial and Temporal Safety

A great amount of research has been done on the subject of detecting and protecting against software-level memory safety violations. Among the various proposals are software only approaches [1], [3], [9], [10], [11], [12], [6], [13], [14], [15] and partially or fully hardware-assisted approaches [16], [8], [17], [4]. Hardware approaches such as CHERI [8] and HardBound [16] result

in much lower performance overheads. However, these specialized schemes have high area overheads, and require substantial modifications to the operating systems and Instruciton Set Architectures (ISAs). The rest of these approaches strike different balances of protection, performance, and compatibility with existing applications. A previous survey by Szekeres *et al.* [18] analyzes the current state of protection mechanisms available, and the corresponding exploit techniques that have been used to defeat them. Notably, it identified pointer-based checking as the only class of approaches to provide complete, non-probabilistic detection of both spatial and temporal memory safety vulnerabilities [12]. For our purposes, we will discuss a series of pointer-based checking approaches developed by Nagarakatte *et al.* [3], [9], [17], which represents the state of the art in complete memory safety solutions.

Pointer-based checking, also known as a capability system, treats each pointer as a "capability", or a key, carrying with it an associated set of access rights. Metadata representing these capabilities can be stored inline (as multi-value "fat pointers" [19]); this approach is common in high-level safe languages, such as Java and C#, and is also used in Cyclone [20], a safe dialect of C/C++. This is often acceptable for new programs, but does not address the problem posed by the huge body of unsafe C/C++ code already in existence, most of which cannot be practically ported to a safe language. Other notable works include Dynamic Information Flow Tracking [21] and [22], which track all suspicious data in program control flows, and Context based schemes [23] and [24]. However, these schemes suffer from high performance overheads, and high false positive rates that occur due to pointer aliasing [25]. We thus need a solution that provides full safety while preserving compatibility with existing C/C++ source code.

SoftBound [3] achieves this by creating a capability system based on a disjoint metadata space, tracking pointer capabilities in a distinct region of memory specifically reserved for that purpose. It is a complementary software-based solution implemented via compiler-level instrumentation, by adding custom passes to the LLVM compiler infrastructure [26]. It is based on a previous hardware-based scheme, HardBound [16], and provides complete spatial safety by maintaining a base address and bound corresponding to each pointer. Base and bound values are associated with pointers upon creation, and are propagated to any derived pointers. Each time a pointer is dereferenced, a check is performed to ensure that the effective address is within the allocated region; if it is not, a spatial safety violation is detected and the program is terminated. This approach allows arbitrary pointer arithmetic without compromising safety guarantees. SoftBound can additionally detect sub-object overflows by narrowing pointer bounds when creating pointers to structure members. Due to SoftBound being a state-of-the-art program control flow protection open-source framework, we employ it in our proposed architecture in this paper [27].

2.2 Memory Integrity Verification

Much of the research on memory integrity verification has remained hardware focused on Merkle Trees where address and data hashes are mapped to a binary tree, and then accessed and updated on off-chip DRAM read/writes. R. C. Merkle first introduced the idea of a hash tree (Merkle tree) for data integrity verification [28]. Since then, several research projects have been attempting to improve on the idea of Merkle trees. For instance, research has been done on creating energy-efficient memory integrity verification mechanisms for embedded systems by adding timestamps to help decrease the runtime of data checking [29]. An example of an attempt on optimizing Merkle tree is presented by Rogers et al. with a Bonsai Merkle tree [30]. Researchers have also been able to show that there are optimal parameters for software based hash trees, such as leaf block size

and tree depth, based on certain factors including the size of the used memory region [31]. And very recently, Szefer and Biedermann discussed a skewed Merkle tree approach to memory integrity checking, reasoning that some memory pages are accessed more often than others, so decreasing the paths from leaf nodes to the root for those pages could increase efficiency [32].

Suh *et al.* [33] describes two alternative schemes to guarantee memory integrity. The first is based on a traditional hash tree (Merkle tree), where each node contains the hash of the concatenation of its children. Only the root node must be stored in secure memory to maintain the integrity of the tree. The performance overhead of traversing the hash tree can be quite significant, but this can be dramatically reduced by caching hash chunks within trusted on-chip L2 cache [7]. The second scheme is a novel method involving incremental multiset hash functions [34], which are used to maintain a read/write log within trusted on-chip storage of all operations performed to untrusted off-chip memory; updates to this log are performed with minimal overhead, and the processor periodically performs an integrity check using the log. For applications that only need to guarantee memory integrity on a periodic basis (say, before performing certain critical operations), this can provide significantly lower overhead. However, an important implication here is that offline schemes like the multi-set hash functions implemented in software cannot be used in a stronger adversarial model, where besides the data memory also the instruction memory is untrusted.

3 Authentication: User vs. Memory

Authentication is a process of verifying the claimed identity of a person or an entity through one or more authentication factors. User authentication, therefore, refers to the scenario where a person's identity is verified in order to grant him the authorization to a certain resource.

3.1 User Authentication Factors

All user authentication mechanisms mainly rely on one or more basic authentication factors which are as follows:

1. **Knowledge Factor:** Something that only the user *knows*, for example a PIN, a password or a pass phrase etc.

2. **Ownership Factor:** Something that only the user *has*, for example an ID card or a physical token etc.

3. **Inherence Factor:** Something that only the user *is* or *does*, for example signatures or the biometric identifiers such as fingerprints etc.

Authentication mechanisms based on pattern recognition [35] can also be considered as a new dimension towards the basic authentication factors. In combination with the basic authentication factors, user authentication may consider continuous authentication factors as well in order to continuously verify the identity of already authenticated users. As a user can be authenticated based on multiple (basic and/or continuous) authentication factors, the process is termed as multi-factor user authentication.

Table 1: Architectural Tradeoffs for Memory Safety

		Integrity Verification	
		Hardware	**Software**
Bounds Check	**HW**	+ Negligible performance overhead − Major hardware modifications	− Major hardware modifications − Significant Performance Overhead
	SW	+ Minimal hardware modifications − Moderate performance overhead	+ Compatibility across processors − Possible security flaws − Huge performance overhead

3.2 Memory Authentication Factors

Analogous to user authentication, we define multiple authentication factors for memory authentication in secure computer systems as follows:

1. **Integrity:** Memory integrity checking is the most crucial factor of memory authentication which answers the following question: "Did the device/CPU, which implements Memory Integrity Checking, write the data in memory?"

2. **Freshness:** To prevent an adversary from manipulating the memory in a way that the CPU reads a stale copy of data instead of the most recent one on a memory read, one must verify "When was the data written?"

3. **Spatial/Temporal Safety Verification:** To prevent program's control flow manipulation as a result of a spatial/temporal memory violation by the CPU itself (i.e. CPU writing the data in an illegitimate way) because of an exploit such as a buffer overflow, addresses the question "How was the data written?"

Based on these factors, a multi-factor memory authentication strategy can be defined which protects against all memory safety vulnerabilities and ensures a secure execution of the application.

4 Designing a Comprehensively Secure Processor Architecture

As explained earlier, memory safety countermeasures can be grouped into two logical classes, based on the types of attacks they aim to prevent, i.e. bounds checking and memory integrity verification. To provide complete memory safety, it is necessary to design a computational environment that integrates these two types of countermeasures in an efficient manner. In this section, we explore the design tradeoffs of different flavors of these protection schemes for a unified and comprehensively secure system. A summary of these tradeoffs is presented in Table 1.

4.1 Hardware vs. Software Integrity Verification

Memory integrity verification addresses direct attacks on memory, typically waged at the hardware level. Because such attacks are independent of program flow and can be made at any time, these protections must run in real time – that is, all loads and stores of insecure memory must be

instrumented to provide protection. In our adversarial model, we consider all layers of the memory hierarchy above the main memory to be physically secure. Therefore, memory integrity verification must be implemented *at least* above the boundary between main memory and lowest-level cache; higher-level implementations will provide equal security, but at correspondingly lower efficiencies.

A software implementation, for instance, might be useful for some purposes, but will incur extremely high performance overhead, since software has no visibility into the physical allocation of memory between DRAM and cache. To provide guaranteed protection, it must perform a costly check on every load or store, regardless of whether the addressed region is actually backed by insecure memory. Such software based approaches, however, would still require at least some basic hardware support (e.g. store the root hash on-chip) in order to provide fundamental security. Otherwise, an adversary can also manipulate e.g. the root hash in order to bypass the protection scheme.

Regardless of the basic hardware support, some software based schemes can still be circumvented by advanced attacks. For instance, Suh *et al.*'s scheme utilizing incremental multiset hash functions to facilitate periodic checks [33] implemented in software can alleviate the performance overhead as compared to a software based Merkle tree approach, but at the cost of potentially undermining security. If in hardware, such a scheme can "play back" recent memory operations during a periodic check, and guarantee termination of a compromised process at that time. In software, however, an attacker only needs a brief window of opportunity to overwrite instruction memory and effect full control of execution; the malicious code, not being instrumented with the necessary checks, will be unaffected by any software-level protections.

Hardware based schemes are also constrained by the fact that in the presence of strong adversaries, the integrity verification must be done online (i.e. in real time). As mentioned earlier, even hardware incremental multiset hash based scheme can only detect an attack at the next checkpoint. This could still harm in terms of privacy leakage type of attacks. Practically, then, to guarantee full security without drastically compromising performance, we need to implement Merkle tree based memory integrity verification in hardware, at the interface between main memory and lowest-level cache, where the roles of these protections are most naturally reflected. Since it only requires the addition of an integrity verification interface in DRAM controller, the required hardware changes are minimal.

4.2 Hardware vs. Software Bounds Checking

Implementations of spatial and temporal safety protection (i.e. bounds checking) are constrained more tightly. Since these vulnerabilities manifest themselves in the logical (virtual) address space, therefore we cannot ignore the cache layers, which are part of that space, notwithstanding their physical safety. Hence bounds checking schemes *must* be implemented at the highest level of memory hierarchy, i.e. between L1 cache and CPU registers, since violations can occur even in the cache. This leaves the choice of implementing the protections in software, hardware, or some combination of the two (as have been demonstrated by SoftBound [3], Watchdog [10], and WatchdogLite [17], respectively); but all of these variations still operate at essentially the same level, albeit from different perspectives. Software based approaches happen to inherently implement the protections at the highest level of memory hierarchy and offer compatibility with existing systems and legacy code, but at a cost of high performance overhead. Conversely, hardware implementations introduce an integrity verification interface between CPU registers and L1 cache and offer better performance compared to the software based approaches, e.g. CHERI [8] reports only 15%-30% overhead –

though the hardware also has to perform frequent real time checks on registers-L1 cache boundary. Hardware approaches, however, are not feasible as they require substantial hardware changes leading to high area and cost overheads. Notice that spatiotemporal protections must be implemented on CPU-L1 Cache boundary, otherwise this would fundamentally compromise the safety guarantees they seek to provide.

4.3 Implications of Composition of Memory Safety Schemes

The two types of memory safety schemes, once composed together, complement each other to provide further protection against potential attacks. For instance, memory integrity verification allows spatio-temporal metadata to be safely stored in insecure memory; and likewise, spatial protection prevents corruption of metadata "shadow spaces". However, the logical separation of memory integrity verification from spatio-temporal protection provides a mutual orthogonality that allows various individual schemes to be substituted as long as they provide the same high-level safety guarantees. A unified software-only scheme, while not amenable to high performance, is possible, and may have value as a debugging tool; conversely, a hardware-only design can facilitate extremely high performance and binary compatibility with existing uninstrumented code, but at the cost of potentially significant circuit complexity. A practical compromise, such as a hardware implementation of memory integrity verification at the boundary between DRAM and lowest-level cache, combined with software-based or partially hardware-assisted bounds checking, can achieve a balance of performance overhead and hardware cost that is acceptable for a great deal of real-world applications.

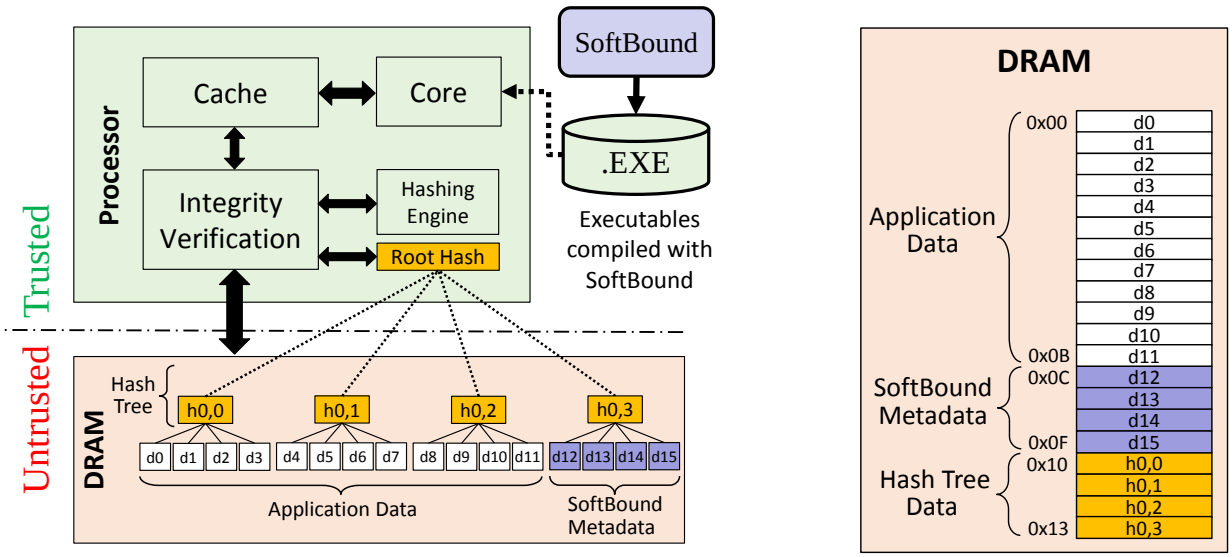
However, we expect that coexistence of the two schemes in the system may introduce new performance overheads. For instance, integrity verification engine may need to perform extra reads/writes to DRAM to verify the integrity of bounds checking metadata which would not be required in a system having only the integrity verification scheme. We observe this behavior in section 6.

4.4 Proposed Architecture

Based on the intuitions provided earlier in this section, we implement the following flavors of the two types of protection schemes to provide a holistic and practically secure system:

- Merkle Tree for integrity verification in Hardware
- SoftBound for bounds checking in Software

Figure 2a shows the architecture of our unified system. The processor is considered trusted and implements an integrity verification module which serves as an interface to the untrusted DRAM. All the communication between the processor and the DRAM is channeled through this trusted verification module. A balanced hash tree is maintained on top of the whole working set memory. The hash tree nodes are also stored in the untrusted DRAM, as shown in Figure 2b, except for the root hash which is stored on-chip. The lowest level cache is shared by both data and hash blocks, i.e. the hash blocks are also cached [7]. A cache partitioning scheme is used to reduce interference between regular cache data and hash blocks. $1/4^{th}$ of the cache space is allocated for caching hash blocks while the rest of the cache is used for regular data. For each off-chip access to a data block, the hash of data block is computed using an on-chip hashing engine. Then the nodes of its hash chain, which starts from its hash node leaf and goes up to the root hash, are searched



(a) M-MAP Architecture

(b) DRAM Layout

Figure 2: Architecture of the unified system with Merkle tree based memory integrity verification and SoftBound based bounds checking. "Application Data" represents the memory footprint of the uninstrumented program whereas "SoftBound Metadata" shows the additional space required by bounds checking data structures.

in the cache. Since the cache is trusted, the first hash node from the chain found in the cache can be used to verify data integrity. In the worst case, the complete hash chain up to the root hash is retrieved from the memory and is verified. Since in our implementation, the hash tree has 13 levels excluding the root hash level; therefore in the worst case, 13 additional cache lines need to be read for integrity verification per data read. To provide bounds checking protection, we recompile the application programs along with SoftBound which adds additional instructions to perform checking on each memory instruction. This results in an overall increase in total number of instructions of the program.

5 Evaluation Methodology

The default architectural parameters used for evaluation are shown in Table 2. The baseline system models a single issue, in-order processor with physical address length of $48bits$. We also model a single issue, out-of-order core type for our evaluation. The DRAM size that an application can access is $4GB$, denoted by "effective DRAM size". It takes 100 cycles to complete a DRAM request. The hashing engine takes a $512bit$ block (i.e. one cache line) as input and computes a $128bit$ hash in 80 clock cycles. The hash tree is structured as a quad tree (i.e. each node has four child nodes) since the hashing engine can be fed four $128bit$ child hashes to compute their $128bit$ parent hash. For the $4GB$ working set with $64Bytes$ cache line size, the quad hash tree has 2^{26} leafs and $\log_4(2^{26}) = 13$ levels (excluding the root hash). Total number of nodes of a quad tree having L leafs is given by $\frac{4L-1}{3}$, therefore our hash tree has $\frac{2^{28}-1}{3}$ nodes, each of which occupies $16Bytes$ memory. Consequently the hash tree for memory integrity verification requires an additional $\approx 1.33GB$ of memory space in the insecure DRAM. The logic overhead of the hash engine is around 60000 1-bit gates [33].

Table 2: Architectural parameters for evaluation.

Parameter	Value
Number of Cores	1 @ 1 GHz
Compute Pipeline	
(i) In Order (In)	In-Order, Single-Issue
(ii) Out of Order (OoO)	OoO, Single-Issue. ROB: 32, Load/Store Queue: 10/8
Word Size	64 bits
Physical Address Length	48 bits
Memory Subsystem	
L1-I Cache per core	32 KB, 4-way Assoc., 1 cycle
L1-D Cache per core	32 KB, 4-way Assoc., 1 cycle
L2 Cache per core	1 MB, 16-way Assoc. Inclusive. Tag/Data: 2/6 cycles.
Cache Line Size	64 bytes
Effective DRAM Size	4 GB
DRAM Bandwidth	10 GBps per Controller
DRAM Latency	100 Clock Cycles

5.1 Performance Models

All experiments are performed using the core, cache hierarchy, and memory system models implemented within the Graphite simulator [36]. Memory integrity checking is faithfully modeled and integrated into Graphite. The Graphite simulator requires the memory system (including the cache hierarchy) to be functionally correct to complete simulation.

5.2 Simulated Configurations

We simulate the following configurations:

1. **Baseline** is a vanilla system without any memory integrity or bounds checking capability. Comparison against this system gives us an idea of how much overhead the individual and combined schemes will incur relative to a plain vanilla system.

2. **MI** implements memory integrity checking on top of the baseline system. It uses the Merkle tree approach to verify the integrity of data on each off-chip request.

3. **BC** uses SoftBound to detect buffer overflows on top of the baseline system. SoftBound adds extra instructions to perform bounds checking.

4. **MI_BC** implements both Merkle tree based memory integrity verification and SoftBound based bounds checking to provide a holistic multi-factor authentication framework.

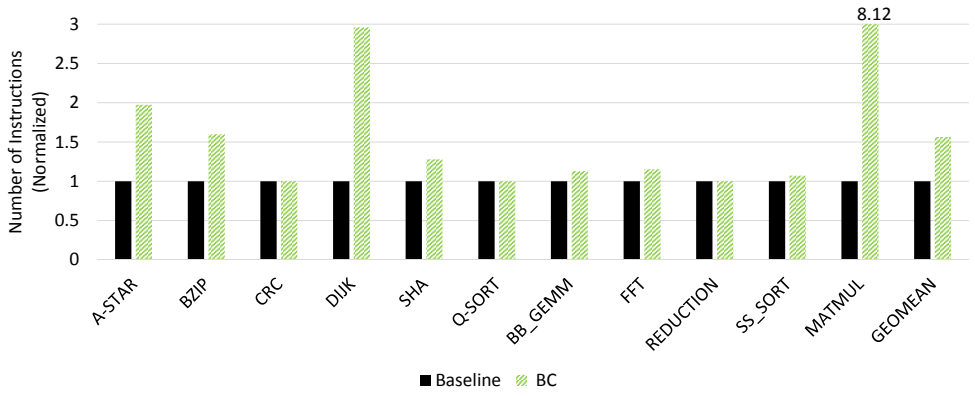


Figure 3: Number of instructions of baseline and bounds checking. The results are normalized to in-order insecure baseline.

5.3 Benchmarks and Evaluation Metrics

We simulate two SPEC [37] benchmarks (A-STAR, BZIP), four benchmarks from MiBench [38] (CRC, DIJKSTRA, SHA, and Q-SORT), four benchmarks from Aladdin [39, 40], (BB-GEMM, FFT, REDUCTION, and SS-SORT), and a matrix multiplication benchmark (MATMUL). We were unable to compile other benchmarks (such as SPLASH-2, PARSEC, rest of the benchmarks in SPEC and MiBench). For each simulation run, we measure the *DRAM accesses*, *instruction count*, and *Completion Time*. The DRAM accesses are broken down into 1) L2 Read Misses, 2) L2 Write Misses, 3) Dirty Evictions, and 4) Integrity Verification Accesses. The completion time is an aggregate of the following components:

1. **Compute latency:** The processing delay in compute pipeline including the private L1 hit latency.
2. **L1 to L2 cache latency:** The time spent accessing the L2 cache.
3. **L2 cache to off-chip memory latency:** The time spent accessing memory including the time spent communicating with the memory controller and the queuing delay incurred due to finite off-chip bandwidth.

6 Results

In this section we discuss the results of the simulated configurations in terms of the completion time, number of DRAM accesses, and instruction count. We highlight the different tradeoffs involved and evaluate the feasibility of the different schemes.

6.1 Instruction Count

Figure 3 shows the instruction count of the bounds checking scheme normalized to the baseline. The number of instructions in MI are the same as baseline, as MI is a hardware based scheme and does not add any additional instructions. However, BC add substantial number of instructions on top of the baseline. The benchmarks that are memory-bound incur higher overhead compared to the benchmarks that are compute-bound. For example, the instruction count for memory-bound

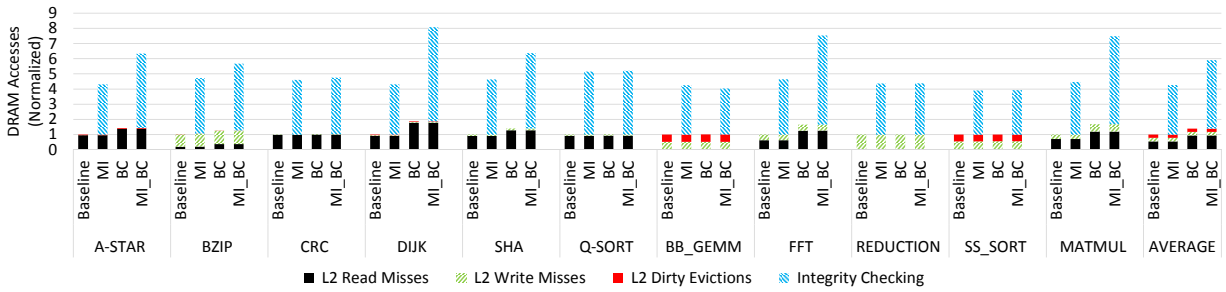


Figure 4: DRAM accesses breakdown of the evaluated schemes. The results are normalized to in-order insecure baseline.

DIJKSTRA and MATMUL increases to 2.96× and 8.12× respectively. On the other the instruction count for compute-bound CRC and Q-SORT is practically the same as the baseline. This is due to the fact that the memory-bound benchmarks have higher number of load/store instructions. These load/store instructions result in large number of bounds checking instructions being inserted. The geometric mean (GEOMEAN) of instruction count for bounds checking across all workloads shows 56% increase over baseline.

6.2 DRAM Accesses

Figure 4 shows the number of DRAM accesses of the simulated configurations for in-order processor. As there can be multiple DRAM accesses on each L2 cache miss to load the hash nodes of the required hash chain, the number of DRAM accesses in MI increases substantially. We can also see a significant increase in the DRAM access count in BC. This is because the number of instructions are very high compared to the baseline, as seen in section 6.1.

The increase in DRAM accesses in the individual schemes add up to an even higher number in MI_BC. Clearly, a naïve thinking would be that the overheads of the two individual protection schemes add up in MI_BC. However, we observe that the coexistence of MI and BC in a comprehensive system leads to additional overheads than simply the sum of their individual overheads. This is because of the fact that bounds checking causes additional off-chip memory accesses: for a BC only configuration, these accesses would only be normal DRAM accesses. Whereas in MI_BC configuration, these DRAM requests lead to further accesses to retrieve the hash nodes form the memory to perform integrity verification, and hence cause overall slowdown.

In the baseline system, the compute-bound workloads show a low number of DRAM accesses because of their computation intensive behavior. In MI, the number of accesses become 4×, however, this access count does not have any significant impact on system performance as evident from Figure 5. The reason being the low frequency of DRAM accesses. On the other hand, memory-bound workloads exhibit a large number of DRAM accesses. This number balloons up under MI and BC, impacting the overall system performance.

The number of DRAM accesses increase substantially in MI_BC over MI and BC due to the increased pressure on LLC. There is a chain reaction effect being seen in the DRAM access count in MI_BC. The accesses made to load the hash blocks from the DRAM dominates the overall DRAM accesses. Under MI and MI_BC, ≈ 77% of the total DRAM accesses are made to verify memory integrity.

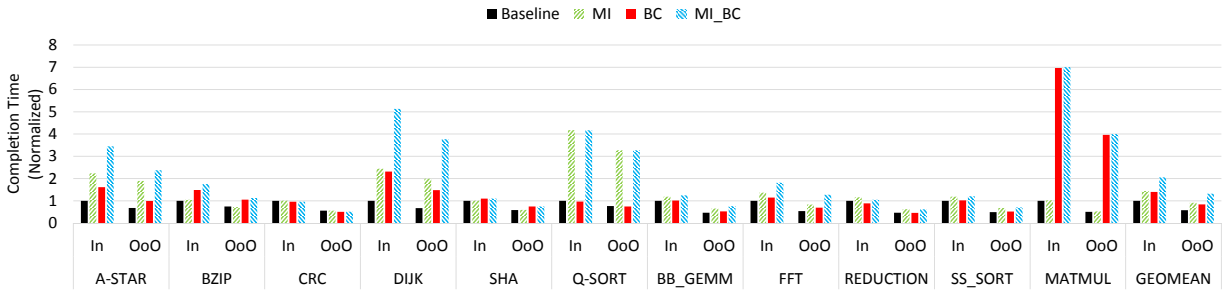


Figure 5: Completion time of the evaluated schemes under in-order and out-of-order processors. The results are normalized to in-order insecure baseline.

6.3 Completion Time

Figure 5 shows the completion time of the simulated configurations for both in-order and out-of-order processors. The results are normalized to that of in-order baseline. The completion time is affected directly by the increase in number of instructions and DRAM accesses. This is why we see the completion time increasing going from the baseline to MI and BC to MI_BC.

The compute-bound benchmarks fare well on completion time, as they do in instruction count and DRAM access count. The reason being that they load a small amount of data from off-chip and then work on it in the trusted on-chip area. Furthermore, they do not contain a high percentage of load/store instructions. This is evident from the results of CRC, SHA, BB-GEMM, and SS-SORT. Each of these workloads shows a slowdown of $< 25\%$ in MI_BC under the in-order processor.

The memory-bound workloads have to communicate with the untrusted off-chip more frequently because of their large memory footprint and a high percentage of load/store instructions. Both of these facts add up and result in a substantial slowdown. DIJKSTRA and MATMUL are two such workloads and they show slowdowns of $> 5\times$ under MI_BC scheme for an in-order processor. A-STAR is another such benchmark but with lower slowdown.

The baseline out-of-order processor is able to hide most of the latency and shows a 42% reduction in completion time over baseline in-order processor. This advantage of OoO processor carries over to the evaluated schemes and MI_BC only shows a slowdown of 32% over in-order insecure baseline. In comparison, the slowdown of in-oder processor for MI_BC is 105% over the in-order insecure baseline. Furthermore, the OoO processor improves the worst performing workload, MATMUL, by $3\times$ in MI_BC over the in-order counterpart.

6.4 Proposed Solution for Secure Embedded Systems

In-order processors are prevalent in embedded systems because of its cost-effectiveness. The simpler design leads to lower energy consumption, however, the performance is sub-optimal. One can deploy a small out-of-order processor in embedded system setting to improve the performance. However, it comes with increase in energy consumption and area footprint.

The out-of-order processor evaluated in this paper models a small single-issue processor with rather small ROB and load/store queues. This does not add a huge area overhead and keeps the microarchitecture simple. This processor enables MI_BC with a modest overhead of 32% in completion time over in-order baseline. Therefore, we propose MI_BC running on top of a simple out-of-order processor as the recommended solution for a secure embedded system.

7 Further Considerations

Networked embedded systems for which physical access is not possible are only vulnerable to attacks through the network by remote adversaries. In such an environment, the embedded systems only need to protect the network I/O channels against spatial/temporal safety violations through malicious I/Os. This can be achieved through the bounds checking techniques. Since the adversary has no direct access to the system, therefore integrity verification is not required in such environment which leads to lower overheads.

In this paper, we discuss the implications and constraints of different memory protection schemes mainly for single-core systems. However, it is equally interesting to explore the possibilities and limitations of memory safety techniques for multi-core systems as well. In multi-core systems, one does not necessarily need to implement integrity verification in hardware in order to get better performance. Instead, software based schemes can exploit the available hardware resources to run integrity verification in parallel with the actual application on separate cores. This could also lead to minimal performance overhead without any substantial hardware support.

8 Conclusion

Spatio-temporal vulnerabilities and memory integrity attacks pose serious challenges in designing secure embedded systems. Conventional memory safety schemes provide protection against either spatio-temporal vulnerabilities or memory integrity attacks. Solutions that provide complete memory safety guarantees come with substantial architectural modifications and overheads, making them infeasible for embedded systems. In this paper we examine key theoretical and practical implications of implementing the conventional protections in a comprehensive secure processor design. Based on these implications, we propose a holistic memory authentication framework, called M-MAP, for complete memory safety. M-MAP implements hardware based memory integrity verification along with software based bounds checking in order to keep a balance between hardware modifications and performance. We propose to implement M-MAP on top of a lightweight out-of-order processor which delivers complete memory safety with a modest overhead of 32% on average. This enables a low cost solution geared towards secure embedded systems.

References

[1] J. P. Anderson, "Computer security technology planning study," Oct 1972.

[2] Codenomicon, "Heartbleed bug," http://heartbleed.com/, 2014.

[3] S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic, "Softbound: Highly compatible and complete spatial memory safety for c," in *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation*.

[4] S. Chen, M. Kozuch, T. Strigkos, B. Falsafi, P. B. Gibbons, T. C. Mowry, V. Ramachandran, O. Ruwase, M. Ryan, and E. Vlachos, "Flexible hardware acceleration for instruction-grain program monitoring," *ACM SIGARCH Computer Architecture News*, vol. 36, no. 3.

[5] C. Maartmann-Moe, "Inception, a firewire physical memory manipulation and hacking tool," http://www.breaknenter.org/projects/inception/.

[6] M. Castro, M. Costa, and T. Harris, "Securing software by enforcing data-flow integrity," in *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*.

[7] B. Gassend, G. Suh, D. Clarke, M. van Dijk, and S. Devadas, "Caches and hash trees for efficient memory integrity verification," in *High-Performance Computer Architecture, 2003. Proceedings. The Ninth International Symposium on*.

[8] J. Woodruff, R. N. Watson, D. Chisnall, S. W. Moore, J. Anderson, B. Davis, B. Laurie, P. G. Neumann, R. Norton, and M. Roe, "The cheri capability model: Revisiting risc in an age of risk," in *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, 2014.

[9] S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic, "Cets: Compiler enforced temporal safety for c," in *Proceedings of the 2010 International Symposium on Memory Management*.

[10] S. Nagarakatte, M. Martin, and S. A. Zdancewic, "Watchdog: Hardware for safe and secure manual memory management and full memory safety," in *Proceedings of the 39th International Symposium on Computer Architecture*, 2012.

[11] P. Akritidis, M. Costa, M. Castro, and S. Hand, "Baggy bounds checking: An efficient and backwards-compatible defense against out-of-bounds errors." in *USENIX Security Symposium*.

[12] E. D. Berger and B. G. Zorn, "Diehard: probabilistic memory safety for unsafe languages," in *ACM SIGPLAN Notices*, vol. 41, no. 6.

[13] D. Dhurjati and V. Adve, "Backwards-compatible array bounds checking for c with very low overhead," in *Proceedings of the 28th international conference on Software engineering*.

[14] G. C. Necula, J. Condit, M. Harren, S. McPeak, and W. Weimer, "Ccured: type-safe retrofitting of legacy software," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 27, no. 3.

[15] N. Nethercote and J. Seward, "Valgrind: a framework for heavyweight dynamic binary instrumentation," in *ACM Sigplan Notices*, vol. 42, no. 6.

[16] J. Devietti, C. Blundell, M. M. Martin, and S. Zdancewic, "Hardbound: architectural support for spatial safety of the c programming language," *ACM SIGOPS Operating Systems Review*, vol. 42, no. 2.

[17] S. Nagarakatte, M. M. Martin, and S. Zdancewic, "Watchdoglite: Hardware-accelerated compiler-based pointer checking," in *Proceedings of the Annual IEEE/ACM International Symposium on Code Generation and Optimization*.

[18] L. Szekeres, M. Payer, T. Wei, and D. Song, "Sok: Eternal war in memory," in *Security and Privacy (SP), 2013 IEEE Symposium on*.

[19] A. Kwon, U. Dhawan, J. M. Smith, T. F. Knight, Jr., and A. DeHon, "Low-fat pointers: Compact encoding and efficient gate-level implementation of fat pointers for spatial safety and capability-based security," in *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*.

[20] T. Jim, J. G. Morrisett, D. Grossman, M. W. Hicks, J. Cheney, and Y. Wang, "Cyclone: A safe dialect of c." in *USENIX Annual Technical Conference, General Track*.

[21] G. E. Suh, J. W. Lee, D. Zhang, and S. Devadas, "Secure program execution via dynamic information flow tracking," in *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2004.

[22] F. Qin, C. Wang, Z. Li, H.-s. Kim, Y. Zhou, and Y. Wu, "Lift: A low-overhead practical information flow tracking system for detecting security attacks," in *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, 2006.

[23] A. Azevedo de Amorim, N. Collins, A. DeHon, D. Demange, C. Hriţcu, D. Pichardie, B. C. Pierce, R. Pollack, and A. Tolmach, "A verified information-flow architecture," in *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2014.

[24] M. S. Simpson and R. K. Barua, "Memsafe: Ensuring the spatial and temporal memory safety of c at runtime," *Softw. Pract. Exper.*, vol. 43, no. 1, Jan. 2013.

[25] B. Hackett and A. Aiken, "How is aliasing used in systems software?" in *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2006.

[26] C. Lattner and V. Adve, "Llvm: A compilation framework for lifelong program analysis & transformation," in *Code Generation and Optimization, 2004. CGO 2004. International Symposium on.*

[27] M. Ahmad, S. K. Haider, F. Hijaz, M. van Dijk, and O. Khan, "Exploring the performance implications of memory safety primitives in many-core processors executing multi-threaded workloads," in *Proc. of the Fourth Workshop on Hardware and Arch. Support for Security and Privacy*, ser. HASP '15. NY, USA: ACM, 2015.

[28] R. C. Merkle, "Protocols for public key cryptography," in *IEEE Symposium on Security and Privacy.*

[29] S. Nimgaonkar, M. Gomathisankaran, and S. P. Mohanty, "Tsv: A novel energy efficient memory integrity verification scheme for embedded systems," *Journal of Systems Architecture*, vol. 59, no. 7, 2013.

[30] B. Rogers, S. Chhabra, M. Prvulovic, and Y. Solihinn, "Towards fast hardware memory integrity checking with skewed merkle trees," in *International Symposium on Microarchitecture - MICRO*, 2007.

[31] D. Dopson, "Softecc: A system for software memory integrity checking," Master's thesis, Massachusetts Institute of Technology.

[32] J. Szefer and S. Biedermann, "Towards fast hardware memory integrity checking with skewed merkle trees," in *Proceedings of the Third Workshop on Hardware and Architectural Support for Security and Privacy.*

[33] G. E. Suh, D. Clarke, B. Gassend, M. v. Dijk, and S. Devadas, "Efficient memory integrity verification and encryption for secure processors," in *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO 36. Washington, DC, USA: IEEE Computer Society, 2003.

[34] D. Clarke, S. Devadas, M. van Dijk, B. Gassend, and G. Suh, "Incremental multiset hash functions and their application to memory integrity checking," in *Advances in Cryptology - ASIACRYPT 2003*, ser. Lecture Notes in Computer Science, C.-S. Laih, Ed., 2003, vol. 2894.

[35] V. Grindle, S. K. Haider, J. Magee, and M. van Dijk, "Virtual fingerprint - image-based authentication increases privacy for users of mouse-replacement interfaces," in *Universal Access in Human-Computer Interaction. Access to the Human Environment and Culture*, ser. Lecture Notes in Computer Science. Springer International Publishing, 2015.

[36] J. Miller, H. Kasture, G. Kurian, C. Gruenwald, N. Beckmann, C. Celio, J. Eastep, and A. Agarwal, "Graphite: A distributed parallel simulator for multicores," in *High Performance Computer Architecture (HPCA), 2010 IEEE 16th International Symposium on*, Jan 2010.

[37] J. L. Henning, "Spec cpu2006 benchmark descriptions," *ACM SIGARCH Computer Architecture News*, vol. 34, no. 4, 2006.

[38] M. Guthaus, J. Ringenberg, D. Ernst, T. Austin, T. Mudge, and R. Brown, "Mibench: A free, commercially representative embedded benchmark suite," in *Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop on*, Dec 2001.

[39] Y. S. Shao, B. Reagen, G.-Y. Wei, and D. Brooks, "Aladdin: A pre-rtl, power-performance accelerator simulator enabling large design space exploration of customized architectures," in *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, 2014.

[40] B. Reagen, R. Adolf, Y. Shao, G.-Y. Wei, and D. Brooks, "Machsuite: Benchmarks for accelerator design and customized architectures," in *Workload Characterization (IISWC), 2014 IEEE International Symposium on*, Oct 2014.