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†
†Electrical and Computer Engineering, University of Connecticut, Storrs, CT 06269, USA
Email: {syed.haider, masab.ahmad, farrukh.hijaz, astha.patni, omer.khan, vandijk}.engr.uconn.edu
†Grove City College, Grove City, PA, USA. Email: ethanjohnson@acm.org
‡Rochester Institute of Technology, Rochester, NY, USA. Email: mss4296@rit.edu

***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. We examine certain crucial implications of composing memory integrity verification and bounds checking 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, while incurring minimal hardware modifications, and area overhead.**

I. Introduction

Security attacks on a computer's memory can be broadly classified into software based and hardware based attacks. Software based attacks are generally the attacks against *spatial* and/or *temporal* safety [1], and are launched by supplying a malicious input to the program, e.g. to exploit a *buffer overflow* vulnerability. We refer to such attacks as *indirect memory tampering* or simply indirect attacks. Hardware-based attacks on memory are the ones against *data integrity* and *data freshness*, and typically involve various forms of "direct" attacks exploiting physical access to the device. For performance reasons, devices such as DMA 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. We call such attacks as *direct memory tampering* or direct attacks.[1]

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. For hardware based or direct attacks, *memory integrity verification* schemes enable the trusted CPU to detect any illegitimate data modification [2]. 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. 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 identify multiple important memory *authentication factors* necessary for complete memory safety, namely *Integrity*, *Freshness*, and *Spatial/Temporal Safety Verification*. 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 memory safety techniques i.e., bounds checking and integrity verification, are secure as well as feasible in terms of performance and required hardware modifications. 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. 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.

II. Background of Memory Safety Schemes

Bounds Checking — A great amount of research has been done on detecting and protecting against software-level memory safety violations. Among the various proposals are software only approaches [1] and partially or fully hardware-assisted approaches [3]. Hardware approaches result in lower performance overheads. However, these schemes require substantial modifications to operating systems and ISAs. Pointer-based checking, also known as a capability system, treats each pointer as a "capability", or a key, carrying a set of access rights [1]. Other notable works include Information Flow Tracking [4], and Context schemes [5]. However, these schemes suffer from high performance overheads, and high false positive rates that occur due to pointer aliasing. We thus need a solution that provides full safety while preserving compatibility with existing C/C++ source code. SoftBound [1] provides a capability system based on a disjoint metadata space. It is a complementary software-based solution that maintains a base address and bound corresponding to each pointer. On pointer dereferences, checks are performed to ensure that effective addresses are within a secure allocated region. Due to SoftBound being a state-of-the-art program protection framework [6], we employ it in our proposed architecture.

[1]This research is partially supported by NSF grant CNS-1413996 for "MACS: A Modular Approach to Cloud Security" and partially supported by Center for Hardware Assurance, Security, and Engineering (CHASE), UConn.

TABLE I: 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

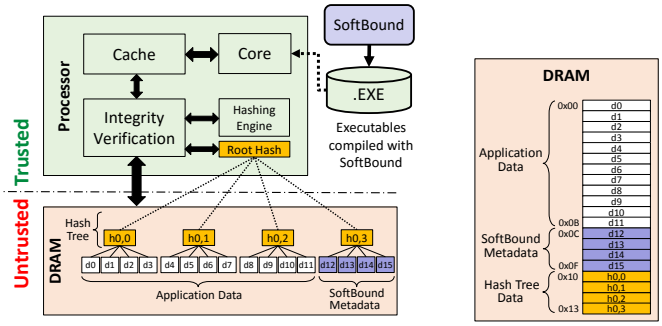
Memory Integrity Verification — Most research on memory integrity verification has remained hardware focused on Merkle Trees where address and data hashes are mapped to a binary tree, and then accessed/updated on off-chip DRAM read/writes. Several optimizations, such as Bonsai Trees [7], have been proposed to improve on the idea of Merkle trees. Suh *et al.* [8] describes two alternative schemes to guarantee memory integrity. The first one is based on a traditional hash tree (Merkle tree), where each node contains the hash of the concatenation of its children, and the root node is stored on-chip in secure memory. The second scheme involves incremental multiset hash functions which are used to maintain (within trusted on-chip storage) a read/write log of all operations performed to untrusted off-chip memory. This can provide significantly lower overhead for applications that only need memory integrity verification on a periodic basis. However, an implication here is that offline schemes like multi-set hash functions cannot be used in a stronger adversarial model, where both data memory and instruction memory are untrusted.

III. M-MAP Architecture

We identify the following memory authentication factors vital for complete memory safety: *Integrity* i.e. "Who wrote the data in memory?", *Freshness* i.e. "When was the data written?", and *Spatial/Temporal Safety Verification* i.e. "How was the data written?". In this section, we explore the design tradeoffs of different flavors of memory safety schemes[2], and then present our proposed architecture for a comprehensively secure system providing all memory authentication factors. A summary of the design tradeoffs is presented in Table I.

Memory integrity verification addresses direct attacks on memory, typically waged at the hardware level. Since such attacks are independent of program flow and can be made at any time, these protections must run in real time. 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. A software implementation might be useful for some purposes, but will incur extremely high performance overhead. 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. Hardware based schemes are constrained by the fact that in the presence of strong adversaries, the integrity verification must be done online (i.e. in real time). Even hardware incremental multiset hash based scheme can only detect an attack at the next checkpoint, exposing security concerns. Practically, then, to guarantee full security without drastically compromising performance, we need to implement a Merkle tree based memory integrity verification in hardware.

[2]Detailed arguments can be found in the extended version of this paper [9].



(a) M-MAP Architecture (b) DRAM Layout

Fig. 1: 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.

Bounds checking (for spatial and temporal safety) can be implemented in software, hardware, or some combination of the two (as have been demonstrated by SoftBound [1], Watchdog [10], 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. Hardware approaches, however, are not feasible as they require substantial hardware changes leading to high area and cost overheads.

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

Fig. 1a 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 Fig. 1b, 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 [2]. 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. To provide bounds checking protection, we recompile the application programs along with SoftBound which adds additional instructions to perform checking on each memory instruction.

IV. Evaluation Methodology

The default architectural parameters used for evaluation are shown in Table II. The baseline system models (1) single issue, in-order, and (2) single issue, out-of-order processors. The DRAM size that an application can access is $4GB$, denoted by "effective DRAM size". The hashing engine takes a 512-bit block (i.e. one cache line) as input and computes a 128-bit 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

TABLE II: 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
L1-I/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
DRAM BW/Latency	10 GBps/100 cycles

can be fed four 128-bit child hashes to compute their 128-bit 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 [8].

All experiments are performed using the core, cache hierarchy, and memory system models implemented within the Graphite simulator [11]. Memory integrity checking is faithfully modeled and integrated into Graphite. We simulate the following configurations:

1) **Baseline** is a vanilla system without any memory integrity or bounds checking capability.
2) **MI** implements Merkle tree based memory integrity checking on top of the baseline system.
3) **BC** uses SoftBound to add bounds checking on top of the baseline system.
4) **MI_BC** implements both Merkle tree based memory integrity verification and SoftBound based bounds checking to provide a holistic multi-factor authentication framework.

We test two SPEC [12] benchmarks (A-STAR, BZIP), four benchmarks from MIBENCH [13] (CRC, DIJKSTRA, SHA, and Q-SORT), four benchmarks from MACHSUITE [14], (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.

V. 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.

Instruction Count – Fig. 2 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 DIJKSTRA and MATMUL increases to $2.96\times$ and $8.12\times$, respectively. On the other hand, 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.

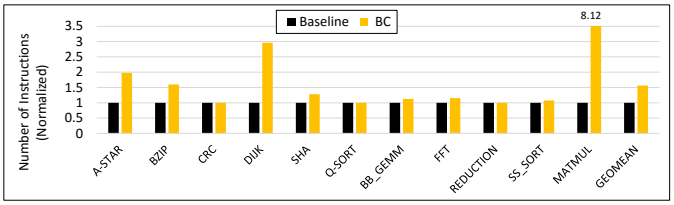


Fig. 2: Number of instructions of baseline and bounds checking. The results are normalized to in-order insecure baseline.

DRAM Accesses – Fig. 3 shows the number of DRAM accesses of the simulated configurations for the 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. 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.

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 $4\times$, however, this access count does not have any significant impact on system performance, as evident from Fig. 4. 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, $\approx 77\%$ of the total DRAM accesses are made to verify memory integrity.

Completion Time – Fig. 4 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. This is evident from the results of CRC, SHA, BB-GEMM, and SS-SORT. Each

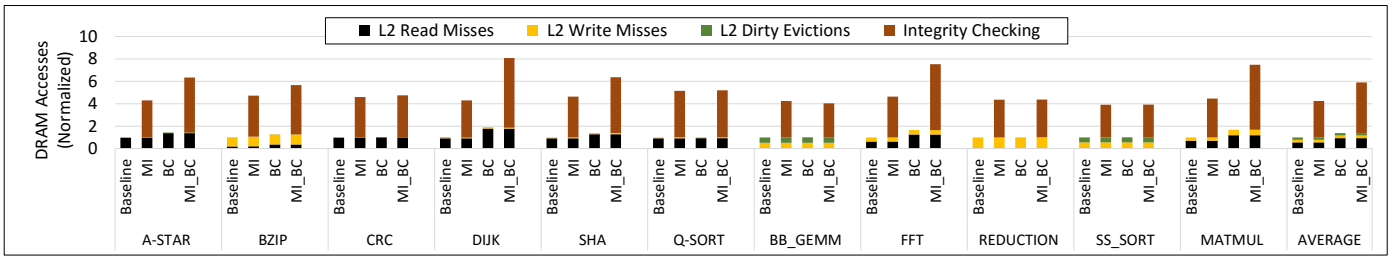


Fig. 3: DRAM accesses breakdown of the evaluated schemes. The results are normalized to in-order insecure baseline.

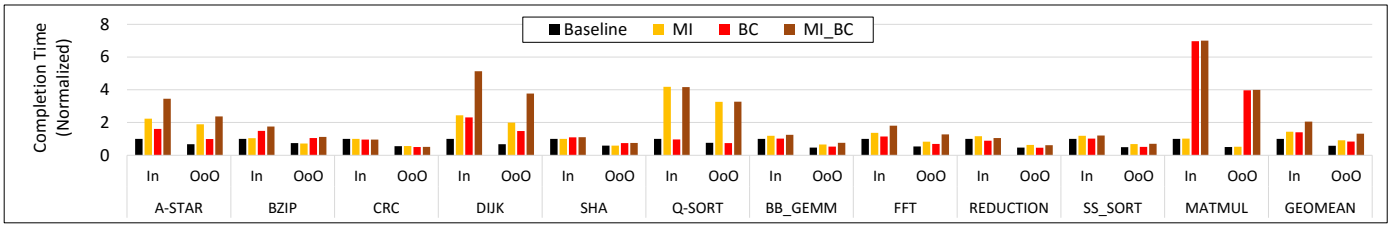


Fig. 4: Completion time of the evaluated schemes under in-order and out-of-order processors. The results are normalized to in-order insecure baseline.

of these workloads shows a slowdown of $< 25\%$ in MI_BC under the in-order processor. The memory-bound workloads, on the other hand, 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.

VI. Conclusion

Attacks on memory pose serious challenges in designing secure embedded systems. Solutions that provide complete memory safety come with substantial architectural modifications and overheads, making them infeasible for embedded systems. In this paper, we propose M-MAP: a holistic memory authentication framework for complete memory safety that implements hardware based memory integrity verification along with software based bounds checking in order to keep a balance between hardware modifications and performance. M-MAP implemented on top of a lightweight out-of-order processor 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] 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*.

[2] 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*.

[3] 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 Int. Symposium on Computer Architecuture*, 2014.

[4] 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.

[5] 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 *Proc. of the 41st ACM SIGPLAN-SIGACT Symp. on Principles of Prog. Lang.*, 2014.

[6] 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, 2015.

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

[8] 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, 2003.

[9] S. K. Haider, M. Ahmad, F. Hijaz, A. Patni, E. Johnson, M. Seita, O. Khan, and M. van Dijk, "M-map: Multi-factor memory authentication for secure embedded processors," Cryptology ePrint Archive, Report 2015/831, 2015.

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

[11] 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.

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

[13] 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.

[14] 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.