EMAClave: An Efficient Memory Authentication for RISCV Enclaves

Omais Pandith*, Rafail Psiakis*, Johanna Toivanen*
*Secure Systems Research Center
Technology Innovation Institute, Abu Dhabi, UAE
*{omais.shafi@tii.ae,rafail.psiakis@tii.ae,johanna.toivanen@tii.ae}

***Abstract*—Enclave technologies have been a common solution for secure execution in the recent times. Most of the prior designs proposed for RISCV enclaves do not evaluate any performance overheads associated with encryption and integrity of data. The recent competing scheme proposed for RISCV enclaves and several other competing schemes for other architectures use an integrity tree to prevent against replay attacks, ensuring the integrity of the data. However, the downside of such approaches are multiple memory accesses required for traversing an integrity-tree and an additional storage overheads. In this paper, we propose *EMAClave* to ensure integrity protection using a modified bloom filter and additional hardware modifications. Moreover, we also prevent cache side channel attacks by proposing an intelligent cache allocation technology when secure and non-secure applications are running together. We show that we are able to outperform the recent competing scheme *Penglai* by around 16.1%.**

I. Introduction

In modern computing systems, trusted execution environments like ARM Trustzone [1] and Intel SGX [2] have become commonplace for secure execution. Given the interconnected nature of today's world and the large number of applications being created to run on such secure hardware, security has emerged as a top-tier design criterion. Open source RISCV processors have gained popularity, and this trend is expected to continue. Enclave solutions have recently been attempted to be ported to RISCV processors and the research community has put out numerous ways to address the problems [3]–[6].

To ensure there is no tampering of the data, most of the prior approaches build a tree of counters to ensure the integrity of the data blocks. Unfortunately, such architectures require a lot of resources to store, compute, and manage counters. One k-bit counter needs to be used for each 64-byte data block, building a tree of counters. Reading new data blocks or changing the contents of existing data blocks frequently require traversing this tree. This is an extremely expensive operation both in terms of time and storage overhead. Given how difficult it is to maintain the state required to guarantee confidentiality and integrity, current research [3], [7], [8]has focused heavily on finding ways to reduce this overhead. In addition to this, most of the existing designs [3], [4], [6], [9] provide different mechanisms to prevent against cache side channel attacks. However, such designs either do not evaluate/support the encryption and integrity along with cache side channel prevention mechanisms [4], [6] or the design takes into account the side channel prevention only for critical sections of the code [3] which might not always be the case.

In this work, we propose a technique, *EMAClave* where we do not use integrity trees at all. To the best of our knowledge, it is the first work that proposes an orthogonal approach for integrity verification. We propose to use a novel bloom filter based integrity verification where we maintain a bloom filter per core inside the processor that stores the hashes of the block data at a particular timestamp. Bloom filters have significantly low overhead compared to integrity trees as there are no tree traversals in the Bloom filter. Sadly, Bloom filters cannot be used to ensure the integrity of all the blocks of an application due to space constraints. We therefore propose an intelligent way of capturing the most frequent and the far blocks by doing some modifications in the the lower level caches. This ensures the integrity of our hot blocks using the bloom filter stored on-chip. We further propose a cache allocation mechanism for secure and non-secure applications based on the frequent and non-frequent blocks of the applications. We show that *EMAClave* reduces the space and the performance overheads for a suite of RV8 and MiBench benchmarks by 75% and 16.1% respectively compared to the most recent competing scheme *Penglai* [3] proposed for RISCV processors. Let us summarize the key contributions of our paper.

The key contributions of this paper are as follows:

1) We propose an orthogonal approach of verifying the integrity of the data by using a bloom filter based approach, supported by hardware modifications to keep track of the hot cache blocks of the application. We evaluate our design for RISCV processors.
2) We propose a cache allocation mechanism to prevent cache side channel attacks. This mechanism intelligently allocates the Last Level Cache (LLC) lines for secure and non secure applications, ensuring performance while secure and non-secure applications are run together. By combining bloom filter verification and cache allocation mechanism, we show the performance improvement of

979-8-3503-4859-0/DATE24/© 2024 EDAA

around 16.1% compared to recent competing scheme [3] with the space reduction of around 75%.

II. Background

A. Counter Mode Memory Encryption

The majority of processors implement the counter mode encryption [10], which is the most widely used encryption technique. The two main benefits are that it successfully hides the decryption latency and increases system security because a separate counter is kept for each block. The block cipher AES receives the counters as an input and uses the key K_{ctr} as the second input to create the *one-time pad* (OTP). By executing an XOR operation between the OTP and the data, the data block is decrypted or encrypted when it is received or transferred off-chip to the main memory (see Figure 1).

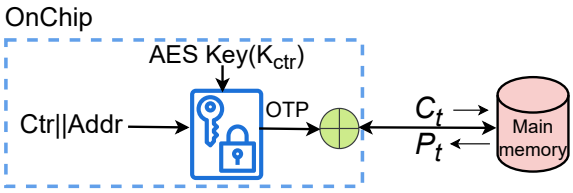

Fig. 1: Counter Mode encryption

Assume P_t is the plain text, C_t is the ciphertext,A is the address of the cache block and Ctr as the counter, we have:

$$OTP = AES(A||Ctr, K_{ctr}) \tag{1}$$

$$P_t = C_t \oplus OTP \tag{2}$$

Since the key can be easily created if the two different cache blocks contain the identical data, the OTP generated for each cache block must be unique. To ensure that the encrypted data is varied, the counter is increased for each cache line written. Although counters can be stored in main memory, previous works [3], [7], [8], [10]–[12] suggest a caching mechanism to limit the number of times these counters are accessed from memory. This mechanism also enables the generation of the OTP in parallel with the fetch operation from main memory.

B. Data Integrity

Even though the entire memory is encrypted, an eavesdropper could still alter the data. This characteristic is referred to as *data integrity* violation. To prevent and recognize these occurrences, each block of the cache line has a message authentication code (MAC) attached to it. The MAC is a 64-bit field that is often created from plain text using a one-way hash function. When the data are retrieved from main memory, the encrypted data must first be decrypted before the MAC can be computed. The MAC that was read from the main memory is compared with the MAC that was computed.The victim would be able to recognize the integrity violation right away if the MACs did not match.

Although the MAC can guarantee data integrity, it is unable to recognize *replay attacks*. Replay attacks allow the attacker to silently replace the $\langle Data, MAC, Counter\rangle$ tuple in the memory with previously accepted values. Each level of MACs ensures the integrity of the level above by having many levels in memory. Integrity trees are frequently used in works for many architectures [3], [7], [8], and they are constructed on top of the encryption counters. The MACs defending the integrity of the encryption counters are created using the counters from the parent counter (level above), which is co-located with the encryption counter cache lines. This continues all the way to the secure chip's root, which is saved. This basis of confidence guarantees the integrity of the data.

III. Related Work

We shall refer to Table I specifically in this section; it summarizes the related work in the chronological order. Majority of the secure enclave technologies proposed for RISCV provide different protection mechanisms for handling different type of attacks. From the software side, *Keystone* [9] provides isolation among different enclaves by using PMP (Physical Memory Protection) registers. It further ensures with the use of scratchpad memory to store most of the data inside the chip for prevention against the data attacks. On the hardware side, *Sanctum* [4] offers robust security guarantees against an eavesdropping software threat model including cache timing and memory access pattern attacks. However it does not evaluate any encryption and integrity support for RISCV processors. Similarly, *TimberV* [5] proposes a new tagged memory architecture providing flexible and efficient isolation of code and data on small embedded systems. The tag isolation is provided with a memory protection unit to isolate individual processes, while maintaining low memory overhead. *CURE* [6] offers different types of enclaves, ranging from sub-space enclaves, over user-space enclaves, to self-sustained kernel-space enclaves which can execute privileged software. CURE's protection mechanisms are based on new hardware security primitives on the system bus, the shared cache and the CPU.

All the above hardware designated techniques provide different types of security mechanisms, however none of them provide support for encryption and integrity. However, *Penglai* [3] provides the support for encryption and integrity of the data with the help of mountable integrity trees where the counters are arranged in the form of the tree. The root-of-trust and the sub-tree roots are stored inside the chip and the sub-trees are stored in the main memory.

If there is an access to any node of the sub-tree, the mount table stored inside the chip is checked for the sub-tree root and the corresponding sub-tree is mounted from the main memory to the on chip. Even though the mounting operation is significantly fast, however, we still need to check the hashes of all the sub-tree nodes for the integrity before fetching from the main memory. They are basically trying to save storage by keeping only frequent counters mounted inside the on chip tree and bringing the less frequent counters at run time, however it still has significant performance overheads in terms of the checking of the integrity of the nodes that are fetched at run time. In addition, it has significant storage overheads also

Paper	Venue	Remarks	Cache side channel	Encryption Support	Integrity Support
Sanctum [4]	Usenix Security'16	Protection against cache timing and memory access pattern attacks	✓	✗	✗
TimberV [5]	NDSS'19	Cache tagging mechanism for the isolation	✗	✗	✗
KeyStone [9]	EuroSys'20	PMP protection using a set of PMP registers supported	✓	✗	✗
CURE [6]	Usenix Security'21	Multiple level enclave support	✓	✗	✗
Penglai [3]	OSDI'21	Mountable integrity trees for replay attacks	✓	✓	✓
EMAClave	-	Tracks the frequency of the blocks and uses the modified bloom filter to store hashes for integrity	✓	✓	✓

TABLE I: Summary of Prior Work

as we need to maintain the most frequent counters and the intermediate hashes on-chip and additionally, we need a mount table that keeps track of the sub-tree roots. Moreover, it uses an integrity tree counters with more bits which leads to the increase in the number of levels of the tree, thus more space overhead.

In our design, *EMAClave*, we provide the same level of security using a small modified bloom filter and some additional changes in the caches. In addition, we propose a very intelligent way of cache allocation for preventing against side channel attacks when secure and non-secure applications are running together with a good performance. *To the best of our knowledge*, this is the first work that proposes a modified bloom filter along with the intelligent cache allocation technology for the RISCV enclaves, ensuring maximum performance and the same level of security as provided by the recent competing schemes.

Key Takeaways: Most of the prior works do not provide any encryption and integrity hardware support for RISCV enclaves. The recent competing scheme proposes to use a mountable integrity trees for the integrity of the data, however there are performance and additional storage overheads associated with it. The aim of this paper is to mitigate such overheads for RISCV enclaves.

IV. Implementation

A. Overview

Majority of the state-of-the-art approaches for RISCV enclaves do not provide support for encryption and integrity of the data. However, the recent competing scheme maintain a mountable integrity tree of counters to prevent against replay attacks. There are performance and additional storage overheads associated with their approach. To mitigate such overheads, we propose *EMAClave* where we use a modified bloom filter to ensure the integrity of the data along with some changes in the caches. We further propose a cache allocation technique for prevention against cache side channel by splitting between the secure and non-secure application intelligently, ensuring performance of both the applications is balanced.

B. Maintaining FrequentMissCount and Timestamp

To monitor the frequency of the blocks, we divide the blocks into two types - *far blocks* and *near blocks*. Typically far blocks are the blocks that are much away from the caches and the vice-versa for the near blocks. During the initial warm-up stage, we basically check the average access latency of L1 cache for each data block. The blocks which have higher latencies are classified into *far blocks* while as the blocks with the lower latencies are classified into *near blocks*. We further quantify the extend of the farness/nearness of the blocks by maintaining a 4-bit *FrequentMissCount* in the caches. If the count value is larger, then those blocks are considered to be as the most frequent blocks. We use the LSB bits of the the tag to maintain such information (1 bit to distinguish between the far and near block, 4-bits to count the frequency of the block used).

Additionally, for each block of data, we keep a 64-bit timestamp in the lower level cache.. The cache block is 512 bits in size, but in our studies, we discovered that 98.1% of cache blocks can be compressed using the C-Pack compression engine [13], to fewer than 448 bits. Therefore, for blocks that are compressible, we retain the 64-bit timestamp together with their contents (the timestamp will be the least significant bits), and for blocks that are not compressible, we retain the timestamp in main memory. The compressible blocks are distinguished using the sixth LSB bit (=1) of the tag.

C. EMAClave Design

We now understand the design of *EMAClave* (refer to Figure 2) to ensure the protection against replay attacks. We use a modified *Bloom filter* [14] based authentication mechanism to prevent against any replay attack. A bloom filter is a data structure that is used to test whether an element belongs to the set. It is a N bit vector where initially all the bits are set to 0. A set of k different hash functions are defined, each of which maps or hashes some set element to one of the N array positions, generating a uniform random distribution. The addition of any element in the bloom filter refers to setting the corresponding hash function entry to 1. However, in our case we do not just set the entry to 0 or 1, instead we store the hash values in the bloom filter as we shall discuss next.

In our *EMAClave* design, we use the bloom filter along with the hardware modifications discussed in the forementioned

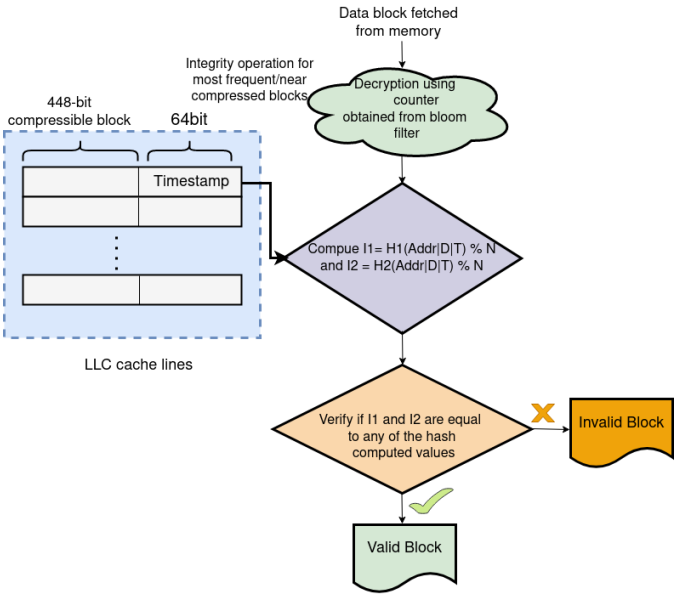

Fig. 2: Integrity verification using Bloom filter based authentication

sections for the integrity of the data. The first step in running an application in a secure environment is to load the code and data into the caches from memory. We initially do a warmup and monitor the blocks based on the *FrequentMissCount* and the far/near blocks. For the blocks which are far and the counts are high, we populate the entries of the bloom filter and the entries are maintained at the block level. We use two hashing functions to map into the entries of the bloom filter. For every cache block, we compute the hash of the data D concatenated with the timestamp T and the address $Addr$ of the data block as:

$$I1 = H1(Addr||D||\mathcal{T}) mod\ N$$
$$I2 = H2(Addr||D||\mathcal{T}) mod\ N \quad (3)$$

The value of N is chosen experimentally and is set to 4096 entries.The mapped entries are initially set as 0. Once the hash of the data blocks are mapped to the bloom filter using the above hash functions, we store either $H1(Addr||D||\mathcal{T})$ or $H2(Addr||D||\mathcal{T})$ in the bloom filter at both the entries indexed by $I1$ and $I2$. This way we are ensuring we store the two hash values for the same data block. Note that we use two hashing functions in the hardware - SHA256 [15] and PRESENT [16] for the hash generation. In addition, it is to be noted that we use the LSB 16-bits of the hash generated to be stored in the bloom filter and this is also used as the counter for the counter mode encryption. Therefore, the total storage overhead for the modified bloom filter would be 8KB per core (16bits * 4096 entries) which is significantly lesser than the on-chip counter caches (32KB) needed to be maintained for the integrity trees. This is because we do not need to store the intermediate counters like in integrity trees and additionally, we have the intelligent way of caching for only most frequently used blocks. The timestamp which we maintain will be crucial if an attacker launches a replay attack by transmitting outdated data. As the timestamp will be different for the older block, the hash of the data D concatenated with the timestamp T and the address $Addr$ will not match the entry of the Bloom filter. In order to eventually save capacity on-chip, it is important to note that we only maintain this Bloom filter for the most frequently used cache blocks.

D. Cache Partitioning for Cache Side Channel Attacks

To prevent against cache side channel attacks, different cache partitioning mechanisms have been proposed in the literature. We follow the similar approach, however with the objective to have a better performance. The recent competing scheme *Penglai* uses the Cache locking strategy to lock the blocks of the secure region. However, it is assuming the cache locking only for the critical sections of the code. It does not take into account if the secure application has a bigger working set.

We use the cache partitioning strategy based on the compressed and the frequent blocks. Basically we give preference to the cache blocks which have compression bit as 1 and the *FrequentMissCount* is high at the same time. This will help us ensuring we have the frequent blocks along with their timestamps residing in the cache most of the times. Suppose we have two applications running, one as secure and other an non-secure. Our cache partitioning strategy will allocate the cache based on the ratio of frequent/compressed blocks and the rest non-frequent blocks for the secure region. Similarly for the non-secure region, the cache partitioning will allocate the cache based on the ratio of frequent and the non-frequent blocks. The ratio between the two is the *percentage of allocation* given to secure region. We update the cache allocation percentage after every 10 million instructions.

E. Hardware Modifications

EMAClave requires a few hardware modifications that are needed to be incorporated in our final design. They are: ❶ Maintaining a 8KB Bloom filter per core ❷ A C-Pack compression engine ❸ Two hashing units.

V. Evaluation

A. Experimental Setup

We simulate the benchmarks from RV8 [17] and MiBench [18] benchmark suites on a cycle-accurate architectural simulator, Tejas [19], which has been rigorously validated with native multi-core hardware. We use SPIKE emulator [20] to generate the traces to be given as input to the Tejas simulator. We use Cacti6.0 [21] to derive the latencies of all our memory structures. We simulate a 2-core chip with each core having a private data cache. We also have a 2 MB shared L2 cache with directory based cache coherence. Table II shows our simulator's configuration setup. We use a 32KB counter cache to store counters for all the previous designs [3], [7], [8].

Parameter	Value	Parameter	Value
Cores	2	Frequency	2.4 GHz
Private L1 i-cache, d-cache			
Latency	3 cycles	Block size	64
Associativity	8	Size	32 kB
Shared L2 cache			
Size	2MB	Block size	64
Associativity	16	Latency	8 cycles
Main Memory Latency	100 cycles		

TABLE II: Simulation parameters

B. Comparison Using Bloom filters for Integrity

1) Performance Comparison: In this section, we compare our scheme *EVAClave* with the recent competing scheme, *Penglai* which implements the mountable integrity tree. In addition, we implement the *VAULT* [7] and *MorphableCtr* [8] integrity trees which are implemented for SGX enclaves. Each benchmark is simulated first for 10 million instructions (warmup stage) and then with a complete simulation for 500 million instructions. Figure 3 shows the performance comparison among all the schemes. Let us try to understand the reason for the different deltas among the schemes.

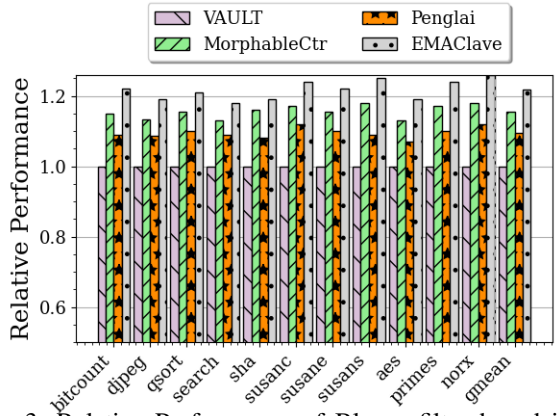

Fig. 3: Relative Performance of Bloom filter based integrity vs Integrity tree based.

We observe that *EMAClave* performs better than *VAULT* by 22.1%. This is because *VAULT* uses the variable arity tree which leads to the increased levels in the tree that leads to extra memory accesses which eventually adds to the performance degradation. *Penglai* performs even better than *VAULT* because it keeps the only frequent blocks in the cache and mounts the new nodes at run time leading to less memory accesses. Similarly, *MorphableCtr* performs even better than both of them primarily because it uses a k-bit counter based on whether the block is used or not. However, the downside of it is that it only keeps a binary decision - a higher bit counter if the block is used and a lower bit counter if it is not used at all. Our proposed design, *EMAClave* performs better than *MorphableCtr* and *Penglai* by 8.2% and 13% respectively. This is because it does not have any additional overheads of the tree traversals that are needed in these schemes. It just maintains a small bloom filter that is able to ensure the integrity with a very less performance overhead.

2) ED2 Comparison: We now compare the Energy-Delay (ED^2) values among the different schemes. Figure 4 shows the comparison of ED^2 values among the schemes. We note that *EMAClave* has the lowest ED^2 value among all competing systems, differing from *MorphableCtr* and *Penglai* by around 15% and 22%, respectively. This is due to the fact that *EMAClave* performs notably better than earlier schemes in terms of both performance and storage.

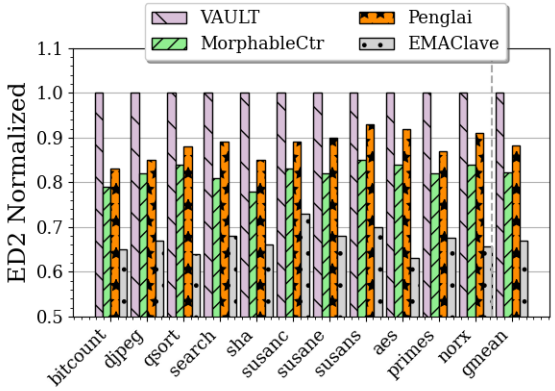

Fig. 4: ED^2 Comparison

C. Comparison Using Cache Allocation

In this Section, we study the performance of using our intelligent cache allocation technology against the Cache locking mechanism proposed by *Penglai*. For a 2-core setup, we run a secure and non-secure application together on separate cores and found that using our Cache allocation technology, we are able to outperform Cache line locking mechanism by around 5.3% (refer to Figure 5). This is because for a Cache line locking, the secure cache lines are locked which affects the non-secure applications running. Instead, in our design, we monitor the cache blocks based on the average L1 latencies and intelligently allocate the cache for secure and non-secure applications, leading to an increase in the performance.

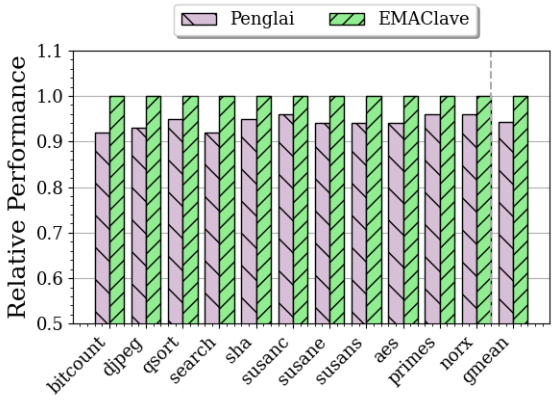

Fig. 5: Relative Performance using cache allocation.

D. Combined Performance using Bloom filter and cache allocation

We now study the performance of our design when bloom filter based integrity supported is merged with the cache allocation technology proposed by us in the paper. We compare the performance with the *Non-Secure* design that does not have any security features provided. We also compare with *Penglai* which is the most recent competing scheme providing support for majority of the security features for RISCV enclaves. Figure 6 shows the performance of *EMAClave* compared to *Non-Secure* and *Penglai* design. It is obvious that the *Non-Secure* design will have the best performance as it does not include any security features. However, we observe from the Figure 6 that *EMAClave* performs better than *Penglai* by around 16.1%. This is because we use a bloom filter for integrity compared to the tree that has the additional overhead of traversing the tree nodes. Additionally, our design proposes a very intelligent cache allocation technology, ensuring the performance is not degraded for the real setup when secure and non-secure applications are running together.

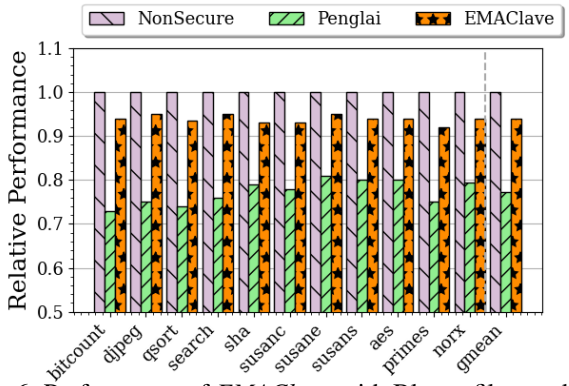

Fig. 6: Performance of *EMAClave* with Bloom filter and cache allocation.

VI. Conclusion

In the race of safeguarding systems against physical threats, the security off-chip main memory is a paramount concern, especially when they are used by secure enclave technologies. Competing RISC-V enclave approaches protect the memory by introducing integrity trees, increasing memory access requirements for integrity-tree traversal and associated storage overhead. To address these challenges, this paper presented *EMAClave*, an innovative solution that ensures data integrity using a bloom filter and integrated hardware enhancements. We showed that EMAClave effectively mitigates cache side-channel vulnerabilities through an intelligent cache allocation mechanism, exhibiting notable performance improvements — outperforming the recent competing scheme *Penglai* by approximately 16.1%.

References

[1] S. Pinto and N. Santos, "Demystifying arm trustzone: A comprehensive survey," *ACM Computing Surveys (CSUR)*, vol. 51, no. 6, pp. 1–36, 2019.

[2] V. Costan and S. Devadas, "Intel sgx explained.," *IACR Cryptology ePrint Archive*, vol. 2016, no. 086, pp. 1–118, 2016.

[3] E. Feng, X. Lu, D. Du, B. Yang, X. Jiang, Y. Xia, B. Zang, and H. Chen, "Scalable memory protection in the {PENGLAI} enclave," in *15th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 21)*, pp. 275–294, 2021.

[4] V. Costan, I. Lebedev, and S. Devadas, "Sanctum: Minimal hardware extensions for strong software isolation," in *25th USENIX Security Symposium (USENIX Security 16)*, pp. 857–874, 2016.

[5] S. Weiser, M. Werner, F. Brasser, M. Malenko, S. Mangard, and A.-R. Sadeghi, "Timber-v: Tag-isolated memory bringing fine-grained enclaves to risc-v.," in *NDSS*, 2019.

[6] R. Bahmani, F. Brasser, G. Dessouky, P. Jauernig, M. Klimmek, A.-R. Sadeghi, and E. Stapf, "{CURE}: A security architecture with {CUstomizable} and resilient enclaves," in *30th USENIX Security Symposium (USENIX Security 21)*, pp. 1073–1090, 2021.

[7] M. Taassori, A. Shafiee, and R. Balasubramonian, "Vault: Reducing paging overheads in sgx with efficient integrity verification structures," in *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 665–678, ACM, 2018.

[8] G. Saileshwar, P. Nair, P. Ramrakhyani, W. Elsasser, J. Joao, and M. Qureshi, "Morphable counters: Enabling compact integrity trees for low-overhead secure memories," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 416–427, IEEE, 2018.

[9] D. Lee, D. Kohlbrenner, S. Shinde, K. Asanović, and D. Song, "Keystone: An open framework for architecting trusted execution environments," in *Proceedings of the Fifteenth European Conference on Computer Systems*, pp. 1–16, 2020.

[10] W. Shi, H.-H. Lee, M. Ghosh, and C. Lu, "Architectural support for high speed protection of memory integrity and confidentiality in multiprocessor systems," in *Proceedings. 13th International Conference on Parallel Architecture and Compilation Techniques, 2004. PACT 2004.*, pp. 123–134, IEEE, 2004.

[11] C. Yan, D. Englender, M. Prvulovic, B. Rogers, and Y. Solihin, "Improving cost, performance, and security of memory encryption and authentication," in *ACM SIGARCH Computer Architecture News*, vol. 34, pp. 179–190, IEEE Computer Society, 2006.

[12] B. Rogers, S. Chhabra, M. Prvulovic, and Y. Solihin, "Using address independent seed encryption and bonsai merkle trees to make secure processors os-and performance-friendly," in *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 183–196, IEEE Computer Society, 2007.

[13] X. Chen, L. Yang, R. P. Dick, L. Shang, and H. Lekatsas, "C-pack: A high-performance microprocessor cache compression algorithm," *IEEE transactions on very large scale integration (VLSI) systems*, vol. 18, no. 8, pp. 1196–1208, 2009.

[14] H. Song, S. Dharmapurikar, J. Turner, and J. Lockwood, "Fast hash table lookup using extended bloom filter: an aid to network processing," *ACM SIGCOMM Computer Communication Review*, vol. 35, no. 4, pp. 181–192, 2005.

[15] S. Gueron, S. Johnson, and J. Walker, "Sha-512/256," in *2011 Eighth International Conference on Information Technology: New Generations*, pp. 354–358, IEEE, 2011.

[16] A. Bogdanov, L. R. Knudsen, G. Leander, C. Paar, A. Poschmann, M. J. Robshaw, Y. Seurin, and C. Vikkelsoe, "Present: An ultra-lightweight block cipher," in *International workshop on cryptographic hardware and embedded systems*, pp. 450–466, Springer, 2007.

[17] RV8, "Rv8 benchmark suite." https://github.com/michaeljclark/rv8-bench.

[18] MiBench, "Mibench benchmark suite." https://github.com/embecosm/mibench.

[19] S. R. Sarangi, R. Kalayappan, P. Kallurkar, S. Goel, and E. Peter, "Tejas: A java based versatile micro-architectural simulator," in *Power and Timing Modeling, Optimization and Simulation (PATMOS), 2015 25th International Workshop on*, pp. 47–54, IEEE, 2015.

[20] RISCV, "Spike: Riscv emulator." https://github.com/riscv-software-src/riscv-isa-sim.

[21] N. Muralimanohar, R. Balasubramonian, and N. P. Jouppi, "Cacti 6.0: A tool to model large caches," *HP laboratories*, pp. 22–31, 2009.