SoK: Cryptographic Protection of Random Access Memory — How Inconspicuous can Hardening Against the most Powerful Adversaries be?

Roberto Avanzi[1], Ionuț Mihalcea[2], David Schall[3], Héctor Montaner[4], and Andreas Sandberg[2]

[1]Arm Germany, GmbH, roberto.avanzi@arm.com and Caesarea Rothschild Institute, University of Haifa, Israel, roberto.avanzi@gmail.com
[2]Arm Limited, UK, ionut.mihalcea@arm.com, andreas.sandberg@arm.com
[3]School of Informatics, University of Edinburgh, United Kingdom, david.schall@ed.ac.uk
[4]Graphcore, Cambridge UK, hector.montaner@outlook.com

***Abstract*—Confidential Computing is the protection of data in use from access or modification by any unauthorized agent, including privileged software. For example, in Intel SGX and TDX, AMD SEV, and Arm CCA this protection is implemented via access control policies. Some of these architectures also include memory protection schemes relying on cryptography, to protect against physical attacks.**

We review and classify such schemes, from academia and industry, according to models of adversaries with varying capabilities, necessitating different protection levels. The building blocks are encryption, integrity, and anti-replay primitives. We discuss these primitives, consider their possible combinations, and evaluate the performance impact of the resulting schemes. We present a framework for the performance evaluation in a simulated system. To understand the best and worst case overhead, systems with varying load levels are considered.

We propose new solutions to further reduce the performance and memory overheads of such technologies. We show that advanced counter compression techniques make it viable to store counters used for replay protection in a physically protected memory. By repurposing some ECC bits to store integrity tags, we achieve hitherto unattained performance while providing confidentiality, integrity, and replay protection.

1. Introduction

Cloud computing promises to increase efficiency and drive down cost for users. Such services co-locate multiple mutually untrusted tenants in the same data center and sometimes even the same physical machines. Compared to traditional on-premises solutions, users of cloud computing face two additional threats. First, hostile tenants may try to exploit bugs in the hypervisor or *access control* mechanisms to impact the confidentiality, integrity, or availability of co-located virtual machines. Second, the service provider or its contractors may try to gain access to customer data.

Similar threats exist in client devices, such as phones, which have evolved into smart terminals and identity providers. Like in a data center, adversaries may use co-located untrusted code or even have physical access to the device. Use cases such as secure payments, secure identification, and software anti-piracy rely on strong confidentiality and integrity guarantees. These are often provided in separate components, e.g., SIM cards, USB tokens, or TPMs. Consolidating their functionality onto the main *System-on-a-Chip (SoC)* enables new use cases while reducing total costs.

AMD SEV [1], Arm CCA [2], Intel's Client [3] and Scalable SGX [4], and Intel TDX [5] move towards this goal by providing managed access control mechanisms. Some even include protection against adversaries with physical access to the system. For instance, Intel's Client SGX implements a *Memory Encryption Engine (MEE)* [3] that provides confidentiality, as well as integrity and protection against replay attacks. Such strong security guarantees can be very costly in terms of performance and storage. For this reason, AMD SEV, Intel TDX, and Scalable SGX (the latter two sharing the same memory protection scheme) provide weaker guarantees in exchange for better performance.

The question that we answer in this study is: *What cryptographic technologies are available to protect the contents of data-in-use in RAM against an adversary with physical access to the system, and what are their memory overheads and performance costs?*

The starting point is a review of the techniques documented in the scientific and technical literature. Even though we cite several architectures for implementing *Trusted Execution Environments (TEEs)*, the scope of this paper does not address aspects such as OS and Hypervisor support, I/O, virtualization, attestation and IPC mechanisms. We focus on solutions for cryptographic memory protection that are entirely implemented within the SoC package limits.

In real-world applications, understanding the cost of a solution is crucial. Area and power constraints limit the viable options, but relaxing them can be justified by strong market requirements. On the other hand, solutions with high performance penalties and memory overheads risk being rejected without further consideration of their merits. For this reason, *we focus mainly on performance penalty and memory overheads*. We also propose new methods to further reduce these costs.

Our performance evaluation uses the industry-standard SPEC 2017 [6] benchmark suite running on the `gem5` simulator [7], [8]. We use the entire benchmark suite and do not pick just a few benchmarks.

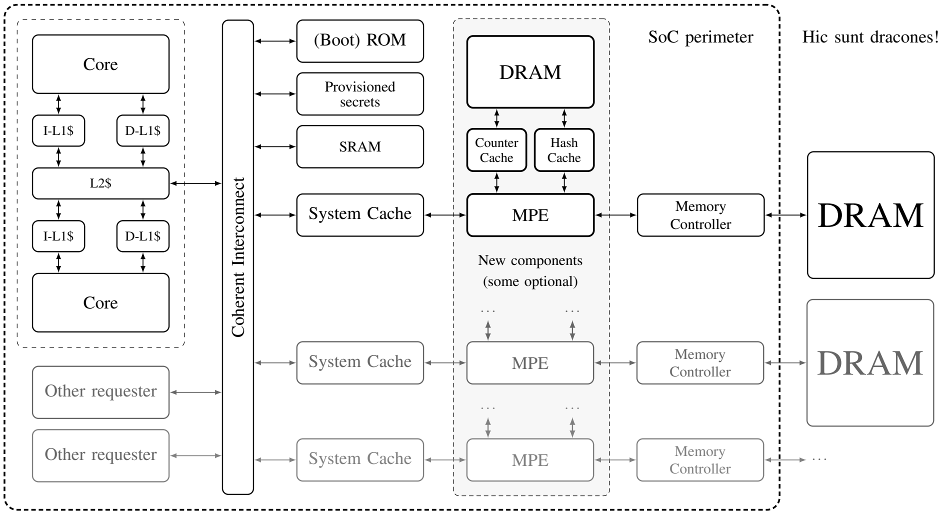


Figure 1: Simplified system level view of a SoC with Memory Protection Engine(s). DMC is the Dynamic Memory Controller.

This work also fills a gap in the literature, as there are only very few papers surveying the subject. The paper [9], published in 2013, contains a thorough survey of memory encryption techniques until its publication, but its performance data is taken from the surveyed papers. Its abstract, states *"To date, little practical experimentation has been conducted, and the improvements in security and associated performance degradation has yet to be quantified."* Ten years later, this sentence still holds true. For instance, the papers [10] and [11] are more recent, but the comparisons are very limited. In the papers [12], [13] and [14] the performance evaluation is performed on a large set of benchmarks, but the comparisons are only performed against the baseline methods they improve upon, and not in general.

Outline. In Section 2 we provide a summary of our results. Section 3 contains background material, such as: the models of the adversaries; a discussion of the memory protection levels; cryptographic parameters; and a treatment of memory integrity structures. The latter is where we obtain the largest performance improvements. (The cryptographic primitives are described instead in Appendix B.) Section 4 describes the actual benchmarks and discusses how these support the claims in Section 2. In Section 5 we conclude.

Acknowledgments. Parts of Ionuț Mihalcea's work for this paper was performed in fulfillment of the requirements for an M.Sc. degree [15]. Ionuț wishes to thank his academic supervisor Prof. Konstantinos Markantonakis, and his line manager at Arm, Paul Howard, for their steady support.

David Schall's work was done during two internships at Arm Research and Arm's Architecture and Technology Group, respectively. Part of the work performed during the first internship is documented in his Master's Thesis [16].

The authors wish to thank Matthias Boettcher, Mike Campbell, Yuval Elad, Wendy Elsasser, Charles Garçia-Tobin, Alexander Klimov, Kazuhiko Minematsu, Jason Parker, Prakash Ramrakhyani, Gururaj Saileshwar, Andrew Swaine, Peter Williams and Nicholas Wood for many interesting discussions on the subject of this paper.

2. Summary

Cryptographic memory protection relies on the following technologies: (i) Encryption; (ii) Authentication; and (iii) Replay protection structures. Among the replay protection structures we consider also the physical protection of *a relatively small amount* of memory, such as including it in the SoC. We exclude the application of this approach to the entire physical memory, because cost and thermal considerations make it impractical for general purpose client and server SoCs.

There are only a few meaningful combinations of these technologies. They are added sequentially to access control, forming four increasingly robust *Protection Levels:*

L0: Access Control only;
L1: L0 + Memory encryption;
L2: L1 + Memory integrity; and
L3: L2 + Protection against replay attacks.

While one can imagine use cases for integrity only without encryption, we are not aware of any such scheme.

We implement Protection Levels L1 to L3 in the *Memory Protection Engine (MPE)*, an IP block sometimes known as *Memory Encryption Engine (MEE)*, e.g., in SGX. As depicted in Fig. 1, in a typical SoC the MPE sits between the main interconnect (or a system cache) and a memory controller. It can optionally have its own caches, and even access to a physically secure private DRAM to store metadata.

As a starting point for choosing the components used to implement each Protection Level, we first review the state-of-the-art. Table 1 outlines cryptographic memory protection in various *Trusted Execution Environments (TEEs)*. While the TEE list is not exhaustive (a more complete list is given in [17]), the list of primitives and structures is comprehensive, except for some deprecated methods. (These technologies are detailed in Section 3.4 and Appendix B.)

We obtain the following groups of alternatives:

1. The AES vs. a lightweight cipher suitable for memory encryption. We use QARMA-128, cf. Appendix B.1. QARMA [18] is a *Tweakable Block Cipher (TBC)*: Beside the secret key and a text, a TBC accepts a *third* input known as a *tweak*, which is used together with the key to select the permutation computed by the cipher. Unlike the key, the tweak may be controlled by an adversary. TBCs simplify the design of modes of operation. One of their first applications has been to memory encryption [9].
2. *Direct encryption*, where a plaintext block is input to the cipher to compute its ciphertext, vs. *CounTeR mode (CTR) encryption*, where the encryption of successive counter values results in a *keystream* which is XOR-ed to the plaintext (Cf. Appendix B.3 for more details).
3. Various *Message Authentication Code (MAC)* algorithms to safeguard memory integrity, such as Carter-Wegman *Universal Hash Functions (UHFs)* [19] (for instance, encrypted linear functions of the message), encrypted checksums of the plaintext, or *Parallel MAC (PMAC)* [20] (see Appendix B.2 for a discussion of the options).
4. The choice of 32 b vs. 64 b MACs for the integrity tags.
5. Different the sizes of the caches used by the MPE, as well as on-chip memory to store MACs or counters.
6. Optionally repurposing some ECC bits to store MACs.
7. Different sizes of the memory regions protected by one MAC. This is obtained both varying the cache line size and letting a single MACs cover multiple cache lines.
8. Synchronous vs. asynchronous integrity verification.
9. Integrity *counter trees* (see Section 3.4 for more details) where the nodes (which in this paper always fit in one cache line) contain from 8 to 256 highly compressed counters. The nodes can contain such a high number of counters because the system ensures that their most significant bits are equal. This common part is stored only once in the node, while the least significant bits of each counter are stored individually. In this paper for the first time we show the advantages of counters split into *three* parts.

We run the benchmarks with different loads on the memory subsystem. To our knowledge, this is the first evaluation of this type. We also randomize the internal state of the system structures to simulate the more realistic performance characteristics of a not-freshly booted system.

The main two results are the following ones:

R1 Nearly-transparent strong memory protection is possible with current technology, for both client and server systems and in most conditions.

R2 Data structures and the organization of integrity trees play a first-order concern when considering overall *performance*. Cryptographic primitives significantly affect area and power where light-weight block ciphers significantly outperform the AES, but their impact on performance is major only in L1 and L2 schemes, whereas in L3 schemes it is minor.

More results and observations follow:

R3 The performance of encryption methods that are based on *direct encryption* methods, such as L1 and L2 schemes, is very sensitive to the latency of the cipher. Moving from the AES to QARMA brings a significant reduction in performance loss. (Cf. Section 4.3.)

R4 Regards to the previous claim, the additional read latency due to encryption has a greater impact on performance than the write latency. (Cf. Section 4.10.)

R5 Lightweight ciphers can reduce area and thus power usage (cf. Section 4.12). This makes ciphers like QARMA suitable for CTR encryption, even if overall performance is barely affected by cipher latency.

R6 Using 32 b MACs in place of 64 b ones halves their memory requirements, which is significant. However, MAC memory accesses have poor spatial locality, and the impact on performance is marginal (Cf. Section 4.3.)

R7 Small MAC caches have a minor effect on performance. In general, MAC caches are not major performance factors. Counter caches are more effective than the hash caches. The relative improvements due to caching increase with the load of the system. (Cf. Section 4.4.)

R8 Similarly, using longer cache lines (i.e., 128 B instead of 64 B) does not necessarily improve overall performance significantly. However, it halves the memory used by the MACs and enables more aggressive metadata packing in the counter trees. (Cf. Section 4.5.)

R9 While asynchronous integrity verification improves performance, it is security risk as the system may speculate on potentially corrupted data. (Cf. Section 4.6.)

R10 If we store *MACs in repurposed ECC bits* (short: MirE) the performance of L2 and L3 schemes has a major improvement — the same applies if the MACs are stored in an internal memory. (Cf. Section 4.8.)

R11 Incremental MACs, each covering multiple cache lines, have a detrimental effect on performance. The optimization of compressing the plaintext to store MACs, whenever possible, together with the payload [13], which serves to reduce the number of memory accesses, cannot be used: Compressibility is a side-channel revealing properties of the data, defeating the purpose of confidentiality protection [37], [38]. (Cf. Section 4.9.)

R12 Increasingly higher arity counter trees offer major and progressive reduction in both memory overhead and performance penalties, despite the complexity of their structure and implementation. (See Table 2 for the memory overheads.) However, as the arity of such integrity trees increases, with the counter group size staying constant, the system must re-encrypt memory or regenerate integrity nodes increasingly often. Using 3-way split counters is instrumental in reducing the cost of these *Read-Modify-Writes (RMWs)* operations. (Cf. Sections 4.3, 4.8 and 4.11.)

R13 The most striking finding is that the smaller trees, in fact just their leaf level, are compact enough to be stored in a physically secure, on-chip of in-package memory, that is relatively small with respect to the total RAM, i.e.,

Table 1: Selected documented TEEs and cryptographic memory protection schemes (the heading AC means Access Control).

System	Year	Level	AC	Encryption	Authentication	Structure	References
Hall and Jutla's PAT	2002	L3	N	Unspecified	Unspecified	Counter Tree	[21]
AEGIS	2003	L3	N	AES-CBC	Incremental hash	Merkle Tree (MT)	[22], [23]
AEGIS (alt. design)	2003	L3	N	"OTP"	MD5/SHA-1	Log Hashes	[24]
Yan et al.	2006	L3	N	AES-GCM	GMAC	Split Counter Tree	[25]
SecureMe	2007	L3	N	AES-CTR	SHA-1/HMAC	Bonsai MT	[26]
Bastion	2010	L3	Y	AES-ECB	AES-CMAC	MT	[27]
Intel's Client SGX1/SGX2	2013	L3	Y	AES-CTR	Encrypted UHF	Counter Tree	[28]
AMD-SEV-SNP	2016	L1	Y	AES-XEX	None	None	[1], [29], [30]
SYNERGY	2018	L3	N/A	AES-GCM	GMAC	Bonsai MT, MACs in ECC bits	[31]
Apple's Secure Enclave	2020	L3	Y	AES-XEX	AES-based CMAC	Bonsai "Integrity tree"	[32]
Intel TDX, Scalable SGX	2020	L2	Y	AES-XEX	Reduced SHA-3	MACs in ECC bits	[4], [5]
Keystone	2020	L1–L3	Y	AES128, in an unspecified mode	Unspecified	Secure paging of on-chip memory to external RAM	[33]
Arm CCA	2021	L0	Y	Optional	Optional	Optional	[2]
PENGLAI	2021	L3	Y	Unspecified	Unspecified	Dynamically allocated MT	[34]
ELM	2022	L3	Y	Flat-ΘCB (OCB)	Flat-ΘCB and PXOR-MAC	Counter Tree	[35]
CSI:RowHammer	2023	—	N/A	Optional	PMAC	MACs in ECC bits	[36]

1:128 or 1:256. This enables L3 schemes with very low performance penalties. Combined with MirE, they lead to a performance hit of just 3.32% even under extreme bus contention. (Cf. Sections 4.7, 4.8, 4.10 and 4.11.)

3. Background

3.1. Definitions

Following the Arm terminology [2], a *Realm* is a process domain that is isolated from other process domains through policies enforced by a small *Trusted Computing Base (TCB)*.

The *Software (SW)*-accessible volatile, external memory, connected to a memory controller, is seen as an array of blocks. These blocks match the *Last Level Cache*'s cache line size and are thus also called *Cache Line (cache line)*.

An encryption or authentication function is said to provide *spatial uniqueness* if, when computed on equal inputs, but written to different locations, it results in different outputs. This is achieved by including the *Physical Address (PA)* of the encrypted or authenticated cache line in the computation.

An encryption or authentication function provides *temporal uniqueness* (*freshness*) when repeated writes of the same plaintext to the same location result in different outputs. This is achieved by including a counter in its computation.

In what follows a mode (of operation) is a general purpose encryption mode of operation. A Memory Encryption (ME) mode is understood to be an encryption mode of operation with plaintext and ciphertext having the size as a cache line, and no associated data.

An *on-chip* component is defined as a physically secure block in the same package as the processing elements. In this case the package shall be *tamper-averting*, i.e., a package that is either tamper-proof/resistant, or tamper-evident/detecting, where issues are handled by the TCB.

3.2. Adversaries

To adequately answer the question posed in the Introduction, we categorize technologies based on the adversaries they defend against. The adversaries are distinguished according to their access to the target, and their resourcefulness.

Before defining the adversaries, a few critical remarks need to be done. Cryptographic memory protection cannot address most side channels, including all those that exploit physical effects: These are thus out of scope. The exclusion applies to the access-pattern side channel as well: Adversaries can reverse engineer software properties or elicit secrets from access patterns, but the only generic and provably effective mitigation would involve *Oblivious RAMs* (ORAM) [39]. However, ORAMs carry prohibitive performance penalties. The same applies to SW exploitation, timing attacks and micro-architectural side-channels. For all these threats, mitigations should be applied to SW as needed.

Denial-of-Service attacks on Realms must in general be accepted, since user-space services can, for instance, always deny resources, including scheduling, to Realms.

We can now define the following Adversaries:

- Adversary $\mathscr{A}^{\text{SW}}$ can run SW on the target, and provide inputs to it, including through external interfaces.
- The Adversary $\mathscr{A}^{\text{HW}}_{\text{passive}}$ has physical access to the system that contains the target, including its internals, but does not have the capabilities to access on-chip communication interfaces. $\mathscr{A}^{\text{HW}}_{\text{passive}}$ can interpose chips and modules for the sole purpose of monitoring transactions.
- $\mathscr{A}^{\text{HW}}_{\text{active}}$, also performs *active* attacks, e.g., blocking, corrupting, replaying or injecting transactions on the memory bus [40] or other interfaces.
- $\mathscr{A}^{\text{HW}}_{\text{invasive}}$ can mount highly invasive attacks at the chip or package level. Examples range from micro-probing attacks [41] to actual chip reverse engineering and editing using a Focused Ion Beam Microscope [42]. *Note that $\mathscr{A}^{\text{HW}}_{\text{invasive}}$ is out of scope for the research described in this paper, as the proper defenses are HW countermeasures.*

SW and HW-capable adversaries are independent. The HW adversaries form a hierarchy $\mathscr{A}^{\text{HW}}_{\text{passive}} \subsetneq \mathscr{A}^{\text{HW}}_{\text{active}} \subsetneq \mathscr{A}^{\text{HW}}_{\text{invasive}}$.

3.3. Protection Levels

We provide detailed definitions of the Protection Levels. Table 1 shows how some documented solutions map to them.

For each protection level we also list the technologies used to implement it, which are taken from options described in Section 2, Table 1. For more details about these technologies, we refer the reader to Appendix B.

We also assume that all algorithms are parallelized wherever possible.

3.3.1. L0: Access control. Access control policies to implement *reverse sanbdboxing* are the first line of defense against $\mathscr{A}^{\mathrm{SW}}$. However, RowHammer attacks (and microarchitectural side channels) have significantly increased the power of $\mathscr{A}^{\mathrm{SW}}$, enabling them to bypass reverse sandboxing.

Physically separating memory rows of different process domains through access control and precise memory allocation policies could theoretically prevent RowHammer attacks. However, this approach requires complex system software changes and is impractical in real-world scenarios.

We do not discuss the implementation of L0.

From here on, we assume that appropriate access control policies are in place to stop unauthorized agents within *the SoC, but not to prevent RowHammer attacks.*

3.3.2. L1: Memory encryption. *This level provides spatial uniqueness, but not temporal uniqueness.*

Interest in L1 is driven by confidentiality requirements and to make attacks that depend on memory corruption (for instance RowHammer) more difficult. For this reason, L1 must use direct encryption with a cipher that enjoys a strong diffusion property, i.e., any input change induces a flip of each output bit with likelihood 1/2.

In general, protection against $\mathscr{A}^{\mathrm{SW}}$ is very limited, as is against $\mathscr{A}^{\mathrm{HW}}_{\mathrm{passive}}$ since the latter can detect ciphertext repeats. Also, note that attacks on the integrity of a system may still cause SW to reveal its contents, therefore this scheme alone does guarantee confidentiality. Only full replay protection (L3) thwarts the particular attack just mentioned. Warm-boot and cold-boot attacks [43] are properly mitigated. Note that the same arguments apply also to L2.

Address scrambling (a very lightweight encryption mechanism of the PA to permute the memory layout) may also be somewhat effective against RowHammer. It is deployed in some devices like smart cards for the purpose of mitigating side channel attacks. Note that since these schemes are usually static per boot session, address reuse can be detected: this is often all an adversary needs to mount an attack. Hence, it should be considered only as an additional *defense-in-depth* measure and not as a complete mitigation per se.

Implementation aspects. If AES is the chosen primitive, a cache line is encrypted using the *XOR, Encrypt, and XOR (XEX)* construction [20], with the PA as the tweak, as in AMD SEV, TDX, and Apple's secure enclave. The chosen low-latency block cipher for memory encryption is QARMA-128 (as explained in Appendix B.1). QARMA-128 is used in a Tweaked *Electronic Codebook (ECB)* mode as in Fig. 14a, with the PA as tweak.

3.3.3. L2: Encryption and integrity verification. *This level extends L1 with* integrity tags*, to detect memory corruption. It does not provide any temporal uniqueness, hence it must rely on a direct encryption method. An integrity tag is usually a MAC. Adversaries can still mount replay attacks.*

L2 targets $\mathscr{A}^{\mathrm{HW}}_{\mathrm{passive}}$. It is also partly effective against $\mathscr{A}^{\mathrm{HW}}_{\mathrm{active}}$, if they only corrupt individual memory locations or have a limited time budget. To defeat targeted replay of the memory together with the integrity tags, more countermeasures are required (see Level L3 below).

This distinction within $\mathscr{A}^{\mathrm{HW}}_{\mathrm{active}}$, though seemingly arbitrary, is necessary due to varying complexities and costs not only of the attacks but also of the *countermeasures*. System designers can assess threats and make business decisions about accepting specific risks. Similarly, active Adversaries might opt for keeping their attacks passive at least initially, to avoid detection and to collect data for cryptanalysis.

MirE: MACs in repurposed ECC bits. If ECC memory is available, storing the MACs in (part of) the ECC bits eliminates the need to reserve normal memory for the MACs, and significantly reduces memory traffic. It is an important part of the Intel TDX design. Note that MACs are still accessible to a HW capable adversary.

MirE raises the question of the performance impact of using ECC memory. Reported penalties are smaller than 0.5% [44], stemming from increased traffic and additional processing in the DRAM controller. On servers, ECC bits, if not repurposed, are used for error detection, hence memory access times are not affected. In other cases, ECC memory impact is negligible compared to the baseline, so we do not evaluate it as a separate configuration.

Implementation aspects. The same encryption techniques are used as for L1. For Intel TDX the MAC is computed using truncated SHA-3, with the latency assumed to be comparable to AES-128. In any other MirE scheme, following [36], the tag is computed using QARMA$_5$-64-σ_0. *Note that not all the ECC bits need to be repurposed for a MAC: these bits may contain both a shorter ECC and a MAC.*

If the MACs are not stored in repurposed ECC bits, hashing is done by a *multilinear* UHF [19] at 32 or 64 bits. Note that these MACs are actually kept as unencrypted *hashes* while on-chip, which speeds up verification, and we encrypt them block-wise when they are evicted from the hash cache groups. For instance, four 32 b hashes are encrypted as a single 128 b block. This enhances system robustness and security against corruption and replay attacks. In schemes with freshness (i.e., L3), the freshness data of the hashes that are encrypted together must be joined to form the common tweak for the hash block encryption.

3.3.4. L3: Encryption, integrity, and replay protection. With respect to L2, this level fully mitigates also against

$\mathscr{A}^{\mathrm{HW}}_{\mathrm{active}}$, by providing replay protection: In order to replay a cache line together with its counter and MACs the adversary either must successfully perform cryptanalysis or wait for a counter repeat. Note that in some variants, the counters themselves may be hidden to the adversary. More information about these data structures is found in Section 3.4.

Implementation aspects. The freshness information is included in the encryption and in the tag computation. A *CounTeR mode (CTR)* encryption mode is used with both `AES` (following AEGIS, the method by Yan et al., and SGX) and `QARMA`, except with *Encryption for Large Memory (ELM)*, which uses Flat-ΘCB. The anti-replay technologies are described in the next subsection.

3.4. Memory integrity structures

A table of hashes or MACs protects against memory corruption, but it is not sufficient against replay attacks, unless the table is itself protected. This can be achieved by storing it in a tamper-averting memory or by covering it with a structure such as a *Merkle Tree (MT)* [45]. MT nodes can be cached [23] to speed up verification.

If freshness-based encryption is used, we can protect the memory by just protecting the counters, for instance with a *Bonsai Merkle Tree*, i.e., a MT protecting the counter table [26]. A different method in the *counter tree* (a refactoring of Hall and Jutla's *Parallelisable Authentication Tree (PAT)* [46]) also used in SGX [3]. A node of the counter tree is called a *Counter Group (CG)*. A CG contains a counters, which correspond to the a children of the node. The counters in a leaf, resp. non-leaf CG are one-to-one with a cache lines, resp. children CGs, A MAC is computed on every node and it is either stored dedicated table, along with the MACs of the data cache lines. or in the node's cache line along with the counters. Since the latter approach has better performance, for simplicity we consider only it. The MAC of a CG is computed on the a counters in the node and the parent counter. Before a node is evicted, its parent counter is first incremented and the node's MAC is recomputed.

The *split counters* optimization [25] replaces a group of a counters with a group consisting of a single *major counter* and $a' > a$ smaller, *minor counters*, associated with that major counter. A *logical counter* in this scheme is defined as the concatenation of a minor counter and its associated major counter. Each node (a data cache line or a CG) is associated with a logical counter. The increased arity (for instance, from $a = 8$ to $a' = 64$) reduces both storage overhead for counters and tree depth. When a minor counter overflows, the common major counter is ticked to ensure that values do not repeat. Since this changes the values of all the logical counters associated with that major counter, all the sibling nodes need to be refreshed. For data cache lines this means that they are re-encrypted, and for both types of nodes the MACs need to be recomputed. All minor counters in the group are reset to zero at this point to reduce the frequency of minor counter overflows.

Despite these RMWs, split counter trees bring a major performance improvement over monolithic counters. We introduce here 3-way split counters (with major, middle, and minor counters) to both increase arity *and* reduce RMWs.

Instead of using full trees, two optimizations can be done.

[LoC] One option is storing the data cache line counters in an in-package tamper-averting DRAM (an SRAM would be too large) which is MPE private (i.e., invisible to the rest of the system and outside adversarial control). We call this solution LoC which stands for *Leaves-on-Chip*. In fact, if we store the leaf nodes in a physically protected memory, such as on-chip, then we do not need to compute any other nodes from the original tree. LoC is sometimes mentioned in the literature only to be dismissed as unviable because of the large overhead.

[BoC] A less expensive version of the LoC solution consists of keeping the leaf nodes in external memory and store the level immediately above on chip. We call this tree arrangement BoC for *Branches-on-Chip*. Similarly to LoC, the system needs no further levels of the tree to ensure the integrity of the tree. This idea seems new.

3.4.1. Memory overhead comparison. In Table 2 we compare the memory overheads of various integrity tree implementations, including the new very high arity trees introduced in this paper. We assume that a MAC can cover up to 4 cache lines. When multi-cache line MACs are used, each cache line is encrypted individually and is associated with its own counter. Evicting a cache line from the last level cache will not require the re-encryption of adjacent cache lines. For completeness, we also include the *Tamper-Evident Counter (TEC)* tree [47] in the table. It has a large memory overhead, and requires a wide encryption mechanism with a very high latency. This makes it unattractive for practical deployment.

3.4.2. Additional structures. We do not evaluate the *Isolated Tree with Embedded Shared Parity* (ITESP) [14] separately. One of its configurations packs 32 counters in a 64 B cache line where the size of minor counters is 4b, and the freed 128 bits are used to store two 64 b parity/integrity fields, each covering 16 cache lines. We speculate that its performance for a single Realm should be just slightly worse than a 64-ary 64 B split counter groups L3 scheme with MirE, since no MAC table is kept. The closest benchmark that we perform is L3 with 128-ary 128 B split counter groups with 3b minors and MirE. The main benefits of ITESP emerge when multiple Realms run concurrently, a configuration not supported by our setup, because each Realm would have its own integrity tree and metadata cache.

For completeness' sake we mention *Log Hashes* [24]. Log Hashes maintain an incremental hash of a Realm's entire memory by adding the hashes of all cache lines in it. The hash of a cache line is computed on the concatenation of the contents of the line, its address, and a secret key. The Log Hash is updated with each memory write, by subtracting the contribution of the old contents, and adding that of new contents. Verification of the memory occurs only when

Table 2: Memory Overhead of Various Types of Integrity Trees.

		Overhead	
Type of Tree	*cache line size:*	*64 B*	*128 B*
Merkle Tree with $a = 4$, resp. 8		33.3%	16.7%
Monolithic Counter Tree with embedded MAC, $\ell_c = 56$			
• $\ell_H = 64$; $n = 1$; $a = 8$, resp. 16		26.8%	12.9%
• $\ell_H = 32$; $n = 1$; $a = 8$, resp. 16		20.5%	9.79%
• $\ell_H = 32$; $n = 2$; $a = 8$, resp. 16		17.4%	8.23%
• $\ell_H = 32$; $n = 4$; $a = 8$, resp. 16		15.8%	7.45%
Split Counter Tree (SCT) with embedded MAC, $\ell_c = 64$			
• $\ell_H = 64$; $n = 1$; $\ell'_c = 6$, resp. 7		14.1%	7.04%
• $\ell_H = 32$; $n = 1$; $\ell'_c = 6$, resp. 7		7.84%	3.91%
• $\ell_H = 32$; $n = 2$; $\ell'_c = 6$, resp. 7		4.71%	2.34%
• $\ell_H = 32$; $n = 4$; $\ell'_c = 6$, resp. 7		3.15%	1.57%
• $\ell_H = 32$; $n = 1$; $\ell'_c = 3$		7.04%	3.52%
• $\ell_H = 32$; $n = 2$; $\ell'_c = 3$		3.91%	1.95%
• $\ell_H = 32$; $n = 4$; $\ell'_c = 3$		2.35%	1.17%
PAT with $a = 8$, resp. $a = 16$		28.6%	13.3%
TEC tree with $a = 8$, resp. $a = 16$		42.9%	20.0%
128-ary 3-way SCT / MirE, $\ell_H = 32$		—	0.78%
256-ary 3-way SCT / MirE, $\ell_H = 32$		—	0.39%

Legend: *ℓ_H, ℓ_c, and ℓ'_c are the bit lengths of a hash or MAC; of a monolithic or major counter; and a minor counter, respectively. a is a counter groups's arity, i.e., the number of its monolithic or minor counters; and n is how many cache lines a MAC covers.*

the Realm interacts externally. Log Hashes are well-suited only for long-running tasks with minimal I/O, where their performance impact can be negligible. They are unsuitable for general applications and remain unimplemented in practice.

3.5. Cryptographic parameters

To ensure long-term confidentiality, *encryption keys should be at least 128 b long*. Shorter keys are not used in any currently deployed or recently proposed memory protection scheme. Sometimes longer keys are an option, for instance 256 b keys for Intel's TDX, but we posit that this does not offer increased practical security and only increases latency: Indeed, a proper complexity analysis of quantum-computer-assisted key search against AES-128 proves it is secure even against adversaries with access to a large-scale quantum computer [48]. Deployed technologies such as Intel's SGX and TDX, and AMD's SEV use the AES in modes that need two independent keys, or even AES-256. QARMA-128 and QARMAv2-128 allow the use of 256-bit keys as well.

Encryption block sizes must be at least 128 b, to reduce the likelihood of any attack that exploits ciphertext collisions.

Authentication keys should be at least 128 b long as well.

Note that only the TCB and no SW environment may set any key, and SW will only manage process identities.

We posit that a length of 32 b (or even 28 b) is sufficient for both data and counter group MACs, to deter Adversaries that simply want to corrupt memory, for instance with RowHammer attacks. This is, in fact, one of the main reasons to deploy a L2 scheme. The TCB must destroy (i.e., internally invalidate and overwrite) the key or tweak associated with the address where an integrity violation occurred — and possibly other internal information. The target process will no longer be able to execute, and the information in it will be lost to the adversaries. It is essential that the TCB responds so to integrity violations before giving back control to the operating system or the hypervisor. Otherwise, to make just one example, an $\mathscr{A}^{\mathrm{HW}}_{\mathrm{active}}$ adversary with the ability to run privileged SW would be able to brute force a short MACs.

If the chosen authentication primitive produces a longer MAC than needed, the output is simply truncated.

In L3 schemes, an Adversary may attempt to replace a cache line together with its MAC. To do this without triggering an integrity fault, they wait until the counter associated with the target cache line repeats. If the counters are sufficiently long, the attack cannot succeed. For this reason, monolithic counters must be at least 64 b long (it can be argued that 56 bits suffice). The minimal aggregated length of a major and a minor counter (or major plus middle plus minor) shall also be 64 b. If an Adversary wants to replay a cache line together with its MAC and counter, they will similarly have to either guess the embedded MAC or wait that the parent counter repeats.

For Merkle Trees the minimal hash length is 128 b, to ensure that attacks have a time complexity of at least 2^{64}.

3.6. On the design space

In Fig. 1 we have depicted the MPE as a separate block between system cache and memory controller, but this is far from the only option: it can be implemented as part of the memory controller or a wrapper around the system cache. Typically, a MPE is linked to a memory channel, but it can also be private to a core, and thus reside upstream of the on-chip interconnect. In such a design, the MPE is a bottleneck, whereas MPEs associated with memory channels benefit from memory interleaving, reducing bandwidth saturation risks. However, private MPEs are suitable for *secure cores*, like software-defined TPMs.

Some pure SW solutions work as follows: At boot, a part of a cache is *address locked* in order to keep the TCB in it (and effectively reducing its size). All memory reads/writes to external memory are then trapped to this code to augment them with encryption and integrity support. Performance is clearly severely impacted in a such a SW-based solution, as examples like [49], [50] show. A different, less secure, approach [51], [52] keeps most of the RAM encrypted except for a few recently used pages, which are re-encrypted once they have been idle for a sufficiently long time.

Recall that we only consider solutions contained in the SoC package. This excludes "smart memory" [53] or the CXL.memory *Integrity and Data Encryption (IDE)* scheme [54]. Such devices require logic for attestation, secure link setup, and encryption, involving cryptographic engines in every memory module if not every chip, so it would be more expensive, hardware-wise, than a MPE-based solution. CXL is however suitable for disaggregated memory configurations, covering transport between compute and memory nodes.

The breadth of the subject and constant developments (cf. Table 1) imply that the full design space is likely not knowable. The present work represents just a snapshot.

4. Benchmarking plan, results, and discussion

4.1. Benchmarking environment and methodology

It would be impractical to implement several thousands of combinations of technologies in silicon for the purpose of evaluating them. A solution to this problem lies in prototyping, i.e., the creation of an approximate implementation of the desired features, which can thus be tested and benchmarked. Very accurate models can be created even without implementing all details. For instance, the latencies of cryptographic primitives can be derived from actual implementations and inserted as delays into the simulation.

The prototypes used in this paper are built in the gem5 simulator [7], [8]. gem5 allows engineers to build SW versions of HW components typically included in computer systems. It abstracts the interfaces between components, which can be combined flexibly. It provides approximate timing models for many processor cores.

The modeled CPU core is an Arm Cortex A72 with a 2 GHz frequency and a 1 GHz system frequency. The cache hierarchy includes L1-I (48 KiB, LRU replacement policy, 3-way set associative, 1 cycle latency) and L1-D (32 KiB, LRU replacement policy, 2-way, 1 cycle latency) caches, and a unified L2 cache (1MiB, tree-PLRU replacement policy, 16-way, 5 cycles latency). The memory is 16 GiB DRAM in a dual-rank DDR4 DIMMs. The MPE-private caches are 4-way set associative with an LRU replacement policy.

We assume that the SoC is implemented in a 7 nm process. We take the latencies from [18], for instance 15.76 ns for a pipelined implementation of AES-128, 4.8 ns for $\texttt{QARMA}_{11}$-128-σ_1 and 2.2 ns for $\texttt{QARMA}_5$-64-σ_0. Note that implementation, process, libraries all affect the crypto block's latency, but system and CPU clocks do not. We assume we reuse the IP blocks from [18] with their own clocks, thus with the exact same performance characteristics. This is a reasonable assumption since this is how hard macros are used in practice. The above latency of $\texttt{QARMA}_5$-64-σ_0 is also used in [36], and essentially for the same purpose as ours.

Our evaluation uses the SPEC 2017 [6] benchmark suite. Detailed software models such as gem5 increase execution time by several orders of magnitude: a typical SPEC benchmark can take around a month to run [55]. To facilitate rapid prototyping, we use the SimPoint [56] methodology, which is well understood in academia and industry. It uses clustering to find representative regions that serve as a proxy for the whole application. The results are finally combined using weighted averages, that reflect the regions' importance to the overall application. Up to 10 SimPoints of 30 million instructions from each benchmark are simulated in place of several billions of instructions. (Regarding reproducibility, including all details needed to re-generate our SimPoints would be impractical.)

An alternative approach would have been to run the entire benchmarks, as opposed to SimPoints, in parallel on a large distributed cloud. This unfortunately does not work in practice since the longest running workloads would have taken weeks to months to run to completion while providing few or no benefits compared to SimPoints. The quicker turnaround, less than an hour to run all SPEC 2017 on a big-enough cluster, is in fact instrumental when exploring a vast space of optimizations.

Regardless of how the simulation is performed, we may ask ourselves about the impact on systems that include context switches, virtual memory swap, and any type of I/O. These aspects are very difficult to emulate. In fact, benchmarking in such a context seems absent from the literature on cryptographic memory protection. However, we can observe that (i) The additional memory used for metadata is not visible to the operating system and will be unaffected by paging and similar operations; and (ii) It can be argued that context switches, paging, and general I/O are affected by the performance penalties on memory accesses only in a minor way: context switch code and data can reside in pinned memory, and the timing of disk, network operations is dominated by media which are orders of magnitude slower than physical RAM. Speaking in particular of context switches, consider a CPU-intensive task running on a 128-core shared machine with about 500 active user sessions. There are 70 unique users on the machine, many of them running a full GNOME environment, with a 15-min average load level of 65 (which is very high). We observe less than a handful of context switches per second per core. Any cold start effect after the context switch would be in the noise since warming all the caches take just a few million instructions (roughly a few milliseconds).

Therefore, any performance penalty we present here is likely an upper bound to the real-world one.

4.2. Selection of the benchmarking sets

All MPE configurations span a vast multidimensional space. Exhaustively evaluating them all is clearly infeasible, not to speak of the difficulties of properly presenting the data. Hence, we explore the design space in various stages, each consisting of a *set* of runs of the benchmark suite. Each set focuses on some previous configurations and expands the parameter space *where we expect that it has some noticeable impact*. Some schemes, such as L1 schemes, do not carry over to the successive sets because they do not have implementation parameters beyond the encryption primitive.

We use shorthands to describe the various configurations:

$$\textsf{Level}\,/\,\texttt{Cipher}\,/\{\textsf{additional technologies}\}\,/\\ /\,\text{MAC length}\,/\,\text{cache line length}\ .$$

The optional "additional technologies" may include: counter representation (mono or split) and arity, Leaves or Branches on Chip (LoC or BoC), or the use of MACs in Repurposed ECC bits (MirE).

The default cache line length is 64 B, unless the counter groups are on chip, in which case it is 128 B. The default MAC length is 56–64 b.

"{AMD} SME" is equivalent to L1/AES/GFmul/CL64B, "{Intel} TDX" to L2/AES/MirE/28b/CL64B, and "{Intel} SGX" is based on Client SGX, i.e., L3/AES/mono-8/56b/CL64B. LoC always implies counters are split. The shorthand L3/LoC denotes a version of L3 that uses LoC, and thus no integrity tree. Similarly, L3/BoC is a L3 solution with the leaf counters off chip and the next level on chip, also without a full tree. L3 *without* BoC or LoC denotes a replay-protection-capable scheme based on an integrity tree and *no counters on-chip*.

4.2.1. Simulation of system load.

The benchmarks are first run on an *unloaded* system, where the current benchmark is the only running task.

We want an upper bound for the performance degradation in a fully *loaded* system, with up to hundreds of processes running on dozens of processing elements, all sharing the bandwidth of the memory subsystem, such as in a cloud server. Directly simulating such a system is very complex and impractical. We instead inject synthetic traffic upstream of the MPE, but after the L2 cache. We do not include a L3 cache in the system to simulate the extreme situation where the latter has been completely swamped by traffic coming from other requesters or clusters of requesters. The question is, how much extra traffic we must inject.

Therefore, we measure the effective memory latency of the system with various levels and schemes of memory protection, and we observe that the latency starts to degenerate catastrophically for most of them between 8 and 10 GiB/s. For instance, a SGX-like L3 MPE covering the entire memory starts to degrade if more than 8 GiB/s of traffic is injected. We take this value as the traffic for a fully-loaded system and halve it, i.e., 4 GiB/s for the partially-loaded system.

The simulated traffic consists of 75% reads and 25% writes of entire cache lines (64 B or 128 B). The access pattern is a mix of cache-line-aligned linear and random accesses. The linear accesses are sequential, and the random ones are at randomly generated addresses, both across the whole reserved range. The traffic generator alternates 100 µs of simulated time of linear accesses with 200 µs of random accesses, for as long as the workload is running.

Beyond 8 GiB/s, we expect a cloud provider to counter performance deterioration by migrating VMs to other machines to balance load and meet overall performance targets. This would also bring MPE penalties back under control.

4.2.2. Baseline performance.

Without memory protection, our benchmarks run on a loaded system 14.1% slower than on an unloaded system with 64 B cache lines, and 9.5% slower with resp. 128 B cache lines. Changing the cache line length from 64 B to 128 B results in an average speedup of 1.4% in an unloaded system and 5.5% in a loaded system.

In all cases, runs are always compared to the baseline (unloaded) with the same cache line size.

4.2.3. Initialization of short minor counters.

When a piece of software starts to run, in a real-world setting any minor/middle counter will have assumed, because of previous processes, essentially random values. If all minor/middle counters are initialized with zero values before running a benchmark, the latter is put at an advantage, since the minor/middle counters will take longer to overflow, and the number of RMWs may be underestimated. The use of SimPoints may even amplify this bias. Therefore, in order to make our simulations as realistic as possible, in all split counter runs we initialize the counters to random values. This configuration choice magnifies the performance difference between 2-way and 3-way split counters of the same arity, highlighting the superiority of the latter.

We now report and discuss the results of all the runs.

4.3. Set 1: state-of-the-art

We start with the state-of-the-art and some simple variations thereof to get an initial overview of the relative performance merits of the deployed or proposed technologies. We compare L1/AES/CL64B (e.g., AMD SME), L1/QARMA/CL64B, L2/AES/32b/CL64B, L2/AES/MirE/28b/CL64B (e.g., Intel TDX), L2/QARMA/32b/CL64B, L2/QARMA/MirE/32b/CL64B, and ELM with both monolithic and split counters, SGX, L3/QARMA/split-64/32b/CL64B — all with and without a hash cache if not fixed by the manufacturer's architecture, since some architectures have a hash/MAC cache while other ones, such as SGX, avoid it. We also compare 32 b and 64 b MACs in selected cases — shortened to 28 b, resp. 56 b, in TDX, resp. SGX.

Note that SGX here is not a full implementation of Intel's Client SGX architecture, but only of its encryption, integrity, and anti-replay features, the latter expanded to the whole memory. For SGX, hash encryption is CTR as described by Intel [3]. We use this method for the SGX-like variant with AES-256 (L3/AES256/mono-8/56b/CL64B) as well. In all other cases, data MACs are replaced by 32 b long hashes which are directly encrypted in groups of four upon eviction.

Note that TDX includes also Scalable SGX.

The ELM method follows [35] except when QARMA is used. With QARMA the XEX constructions are replaced by simply feeding nonces and separation fields as the tweak to QARMA, as well as using QARMA$_5$-64-σ_0 to generate the *One-Time Pads (OTPs)* to encrypt the tags.

Note that monolithic counter trees are 8-ary, resp. 16-ary with 64 B, resp. 128 B cache lines. For 2-way split counters, minor counters are always 6, resp. 7 bits long, and the arity is therefore 64, resp. 128.

For schemes with freshness, the counter cache is 64 KiB as in SGX to level the comparisons.

These principles apply to every successive set as well, except where explicitly indicated otherwise.

The results, as detailed in Fig. 2, support Results **R2**, and **R6**. Also, ELM has worse performance than SGX, having the encryption primitive on the critical path.

The latencies of AES-256, AES-128, and QARMA-128 are 21.99 ns, 15.67 ns and 4.80 ns, respectively, and they are strictly correlated to the corresponding performance penalties of a L1 scheme on an unloaded system: 7.93%, 6.37%, and

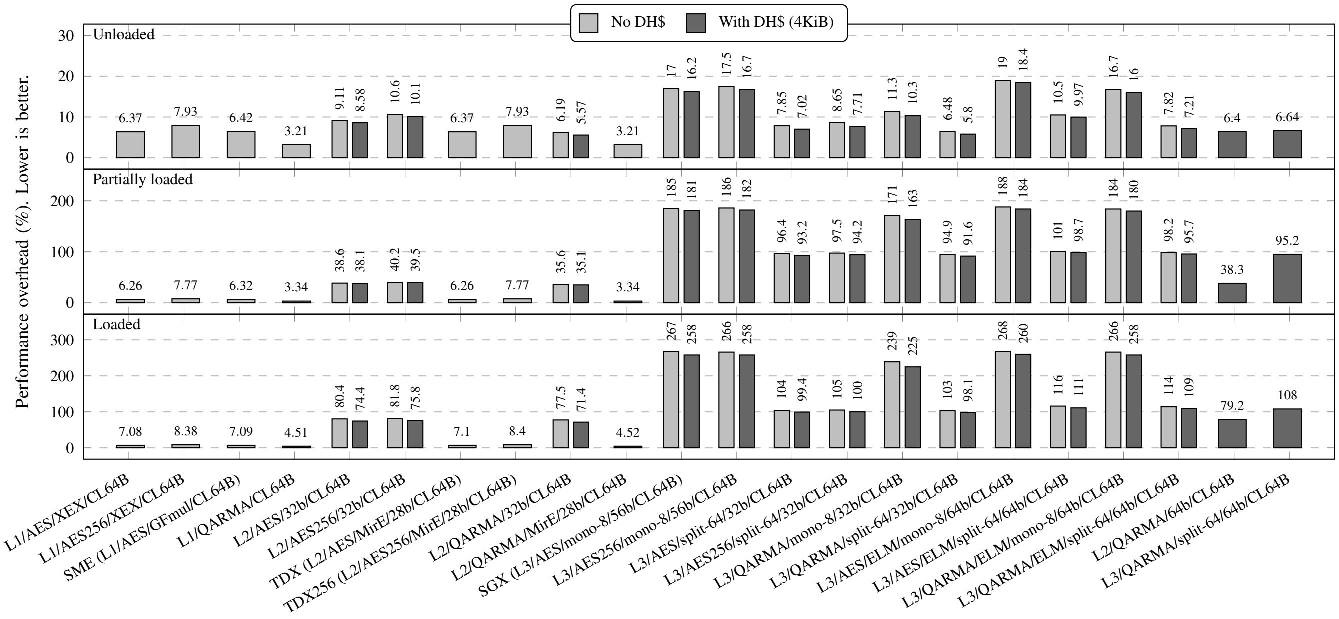


Figure 2: Set 1 (Section 4.3). Comparison of base levels and state-of-the-art.

3.21%. This holds also for varying loads and L2/MirE schemes. For L2/non-MirE and L3 schemes, the difference becomes less significant as the slowdown due to traffic contention between data and medatada increases. This proves Results **R3**.

*For the remainder of the evaluation, because of Result **R6**, for simplicity's sake we shall assume that MACs are 32 bits long and directly encrypted in groups of four except with SGX, MirE, or otherwise explicitly indicated. Similarly, since split counters perform better than monolithic counters (this goes towards Result **R12**), we shall assume that L3 configurations will make use of split counters.*

For brevity,in Sets 2, 3 and 4 we leave out AES from the comparison as it has an identical memory access pattern and similar results to using QARMA-128.

4.4. Set 2: Impact of MPE cache sizes

The goal here is to understand the impact of the sizes of the two MPE caches, namely the hash and counter caches.

L1 does not need caches, so we only consider L2 and L3.

The hash cache sizes we evaluate are 4 KiB, 16 KiB, and 64 KiB; and counter cache sizes are 16 KiB, 64 KiB, 256 KiB, and 1 MiB. We expect these sizes to be within a reasonable range when implemented as SRAM. The presented results are based on the L2/QARMA/32b/CL64B and L3/QARMA/split-64/32b/CL64B configurations (i.e., 32 b MACs, 64 B cache lines, and 64-ary split counters for L3).

These results, displayed in Fig. 3 support Result **R7**. The small benefit of the hash cache can mostly be attributed to spatial locality (most temporal locality has already been exploited by normal data caches). Intuitively, the access patterns of the counter and the hash cache should be similar. However, the reach of the counter cache is bigger since counters are smaller when using split counters and nodes closer to the root cover a large amount of address space which makes them more likely to be reused.

Starting with Set 3, the MPE has a 16KiB hash cache and a 256KiB counter cache. Level L3 uses split counters, unless explicitly indicated otherwise, or with SGX.

4.5. Set 3: Impact of the cache line length

Another fundamental piece of information is how the choices of 64 B and 128 B cache lines affects L2 and L3 performance: Doubling the cache line size will halve the memory overheads, but at least in theory the coarser memory granularity may negatively affect performance.

This set comprises L2/QARMA/32b and L3/QARMA/split/32b with 64 B and 128 B cache lines. Counter group and cache line sizes are always equal which implies that L3 split counter configurations have arity 64 in the 64 B case and 128 in the 128 B case.

The results of Set 3 are combined with those of Set 4 in Fig. 4. They prove Result **R8**. Since we already know that our reference system without a MPE performs 1.4% to 5.5% better with 128 B cache lines, we expect that using to 128 B cache lines, at least for the system cache, is generally beneficial in a system with a MPE.

We acknowledge that changing the cache line size for the coherent cache system might be a major undertaking. However, there are important cases where it is feasible and reasonably non-intrusive. For example, inclusive last-level caches (LLCs) could store and perform writebacks of pairs of 64 B cache lines while still performing coherence on the individual lines. Similarly, LLCs outside the coherent domain (system caches) may use 128 B cache lines while the coherent caches use 64 B cache lines. Both options make the effective cache line size 128 B from the point of the MPE.

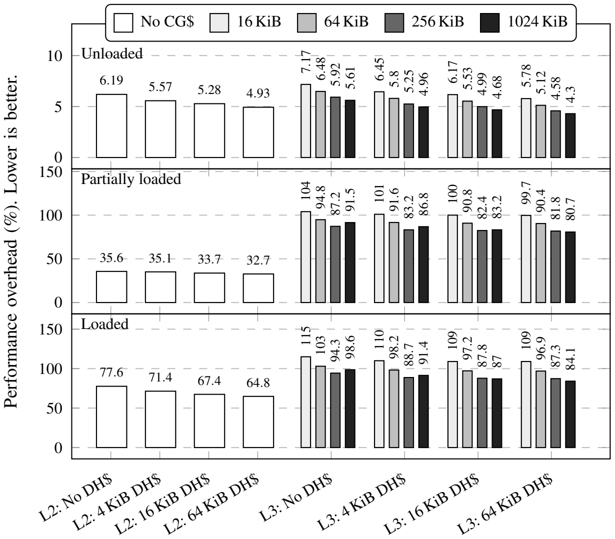


Figure 3: Set 2 (Section 4.4). Impact of MPE cache sizes on L2/QARMA/32b/CL64B (32 b MACs,64 B cache lines) and L3/QARMA/split-64/32b/CL64B (64-ary split counters).

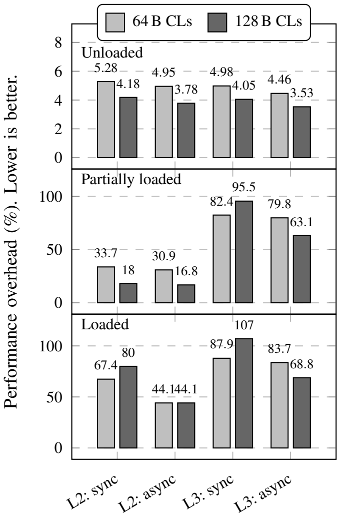


Figure 4: Sets 3 and 4 (Sections 4.5 and 4.6). Impact of cache line size and asynchronous MAC verification.

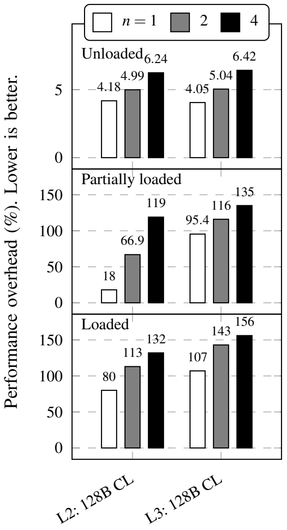


Figure 5: Set 7 (Section 4.9). Impact of incremental MACs.

4.6. Set 4: Asynchronous MAC verification

So far we have assumed that integrity tags are verified synchronously. In principle, asynchronous verification can improve performance by releasing data to the CPU before its corresponding MACs has been fetched from memory and verified. Therefore, we assess how synchronous verification improves overall performance over asynchronous verification.

We test only L2 and L3, as they offer integrity. We reuse the configurations of Set 3. The results are shown in Fig. 4.

One would be tempted to implement asynchronous MAC verification, as it can become quite effective, especially under increasing memory system load. However, asynchronous verification comes with a significant drawback. Since the CPU is speculating on MAC verification being successful, adversaries have a window of opportunity where the CPU is using data under their control and mount an attack. Mitigating this issue introduces significant complexity which would be detrimental the integrity of the system. This is Result **R9**.

From here, we only use synchronous MAC verification.

4.7. Set 5: Use of on-chip memory for L2 and L3

Going beyond caching as explored in Set 2, we explore the impact of secure MPE-private on-chip memory.

Since MACs have a larger memory overhead than counters, we do not expect schemes with on-chip hashes and off-chip counters. Hence, we ignore such a configuration.

Fig. 6 results confirm that relieving the memory bus contention between data and metadata improves performance.

The BoC configuration only marginally outperforms the schemes that do not rely on on-chip memory. This is explained by considering a system without on-chip memory: Temporal locality is poor for leaf nodes, but it improves closer to the root of the tree as each node corresponds to a large memory space. This makes it likely that integrity verification encounters a cache hit at the level just below the leaf level. Therefore, performance is similar to BoC.

With all metadata on chip, the performance is close to the baseline. This may not be realizable in practice. However, as we shall see in Section 4.8, it can be approximated by repurposing ECC bits for MAC storage.

For this set of runs we kept the AES to show that for L3 the performance is similar to QARMA. However, on an unloaded system, AES and QARMA show a slight performance gap. This gap decreases as the system load increases, due to the fact the cipher latency becomes proportionally smaller than the increasing memory access latency.

4.8. Set 6: Impact of repurposing ECC Bits, 3-way split counters, and large counter caches

The deployment of Intel TDX's Multi-Key Total Memory Engine with Integrity (MKTMEi) [5] and [57] suggests that using ECC bits for tags may be an acceptable trade-off for real-world deployments. This is essentially an approximation of storing MACs on-chip since the ECC bits are stored out-of-band and fetched in parallel with the data.

We consider both L2 and L3 configurations, with and without MirE. We expect that MirE implementations are optimized for performance and to reduce storage overhead. For that reason, except for L1/QARMA/MirE/CL64B, we focus on 128 B cache lines which enable much more efficient counter packing. With MirE, a hash cache is not needed since MACs and data are fetched in the same memory transaction, and the MAC algorithm if PMACs.

In addition to classic 2-way Split *Counter Groups (CGs)*, *we introduce high-arity 3-way Split CGs*, which we define only in the length of 128 B, with 128 and 256 logical counters per node. The purpose of this optimization is to keep the

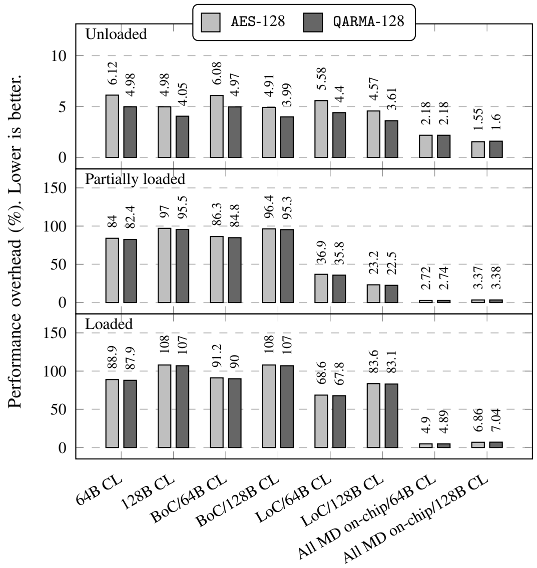


Figure 6: Set 5 (Section 4.7). L3: Impact of storing metadata on-chip.

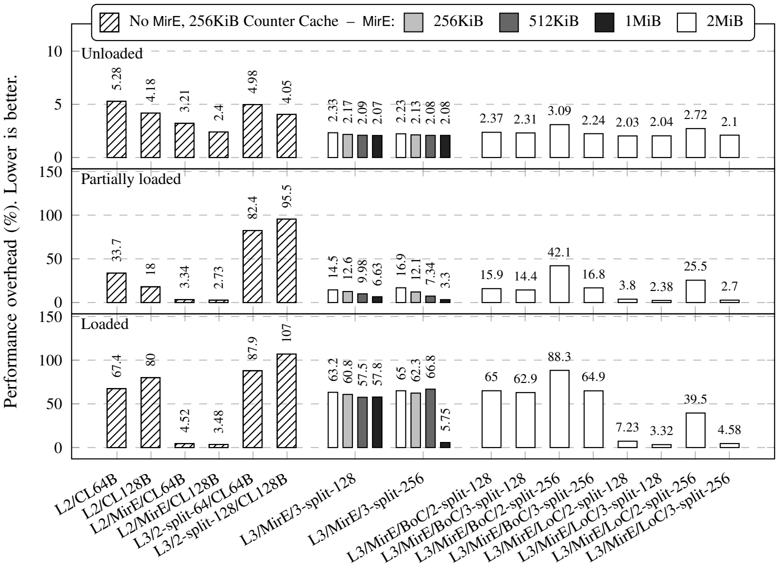


Figure 7: Set 6 (Section 4.8). {L2|L3}/QARMA/MirE. Impact of repurposing ECC bits for MACs and large counter caches. For arity 128 or 256 the CL is always 128 B.

amount of RMW operations under control, with one variant also increasing the density of the CGs. To quantify the impact of this optimization, we evaluate the configuration with and without middle counters. We evaluate the following 3-way split CGs configurations for 128 B cache lines and CGs and without embedded MACs:

- 128 × 7 b minor, 8 × 8 b middle, and 1 × 64 b major counters, with a memory overhead of 1:128;
- 256 × 3 b minor, 32 × 6 b middle, and 1 × 64 b major counters, with a memory overhead of 1:256;

If MACs must be embedded in the counter group, for the 128-ary tree the lengths of the major and middle counters would be reduced to 48 and 6 bits, and for the 256-ary tree the middle counters would 5 bits long – in both cases with 32 b embedded MACs.

In [12], [57] "delta encoded" split counters with rebasing are used together with methods to accommodate a limited number of larger minor counters in a CG to reduce the amount of RMWs. We skip these optimizations since our 3-way split CGs (cf. also Section 4.11) perform better, by nearly eliminating any RMW overhead.

The data (Fig. 7) supports Results **R10** and **R12**. Middle counters are instrumental in getting the best performance out of the high arity CGs, which would otherwise incur in very large RMWs overheads. This demonstrates Result **R13**. The resulting schemes perform even better than L1 schemes, where the cipher is on the critical path to external memory, while in L3 the cipher is computed in parallel to off-chip memory accesses.

For a 16 GiB protected memory, the BoC configuration needs 256 KiB of on-chip storage. An alternative to the BoC configuration would be to use that memory for a counter cache. The 512 KiB configuration in Fig. 7 corresponds to this configuration since the baseline counter cache size is 256 KiB. In such cases, the larger cache normally performs on-par with BoC in an unloaded system and slightly better under load. This can be explained by two effects. First, the cache approximates BoC since the level just above the leaf level is very likely to be resident in the cache. Second, unused branch nodes can be replaced by useful leaf nodes which improves efficiency. On a fully loaded system, LoC performance is reached in practice only when the cache is large enough to cover the tree working set of the running applications. In the case of SPEC 2017, this typically happens between 1 MiB and 2 MiB of cache.

4.9. Set 7: Impact of incremental MACs

If we cannot store MACs in the ECC bits or on-chip, there is another option for reducing their storage overhead: to compute them incrementally over multiple cache lines.

Since the goal is to reduce storage overhead, we focus this investigation on 128 B cache lines. These already reduce metadata storage requirements by a factor of two compared to 64 B cache lines. We test both L2 and L3 configurations with a MAC covering 1, 2, or 4 cache lines. These runs are reported in Fig. 5 only with QARMA-128 encryption, since the performance differences are caused only by the increased memory traffic. In fact, AES results follow the same pattern. These measurements prove Result **R11**.

4.10. Set 8: Breakdown of selected configurations

To better understand the behavior of the MPE, we select a few interesting configurations and show all individual benchmarks in the suite:

- AMD SEV (L1/AES/GFmul/CL64B) and L1/QARMA/CL64B;
- Intel TDX (L2/AES/MirE/28b/CL64B);

- L2 with (L2/QARMA/MirE/28b/CL64B) and without (L2/QARMA/64b/CL64B) MirE;
- Intel SGX (L3/AES/mono-8/56b/CL64B);
- 128- and 256-ary 3-way split CGs (L3/QARMA/LoC/3-split-128/MirE/28b/CL128 and L3/QARMA/LoC/3-split-256/MirE/28b/CL128).

The SPEC2017 benchmarks (cf. Figs. 8 to 13) exhibit some expected results: certain tasks, like omnetpp, mcf, and bwaves experience more significant performance impact across most MPE configurations.

Fig. 8 supports the claim in Result **R4**. The two XEX schemes L1/AES/GFmul/CL64B and L1/AES/XEX/CL64B differ only in the computation of the tweaking mask. In the first case it is performed via Galois multiplications, which we highly optimize for speed, resulting in a latency of 0.55 ns in the chosen process. In the second case AES encryption is used instead. We recall that AES-128 latency is 15.76 ns. Thus, on the write path, the latency is, roughly one, resp. two AES instances, while on the read path is it always one AES instance. Despite the significant difference on the write path, the penalties are almost exactly the same.

4.11. Set 9: Impact of RMW operations

All split counter methods need, as already mentioned, to perform some batches of RMW operations to re-encrypt data or re-compute some embedded MACs whenever a minor, resp. middle counter overflows. These are expensive operations and we want to understand their impact on performance.

We compare the performance of L3 MPEs against hypothetical ones where the RMW operations have zero cost, i.e., are instantaneous. This is achieved by simply skipping them: such an experiment is possible because the simulated MPE does not actually perform cryptographic operations, inserting instead timing delays in their places. This gives an upper bound on the actual time spent in the RMW operations.

For the 128- and 256-ary split counter schemes, we report the performance with 3-way split counters, the performance with 2-way split counters by omitting the middle counters, and the performance with skipped RMWs. The selected combinations are the ones in Set 8 with RMWs.

The results are shown in Figs. 12 and 13. We notice that the impact of RMWs is not always negligible. Using 2-way split counters with 3b minors (L3/QARMA/LoC/MirE with 256-ary CGs) carries a significant performance penalty, but the use of middle counters brings the performance close to the ideal case where RMWs are “free”.

The performance penalties and the proportion of time spent doing RMWs increase with the load. However, even at full load, the performance penalty with 256-ary, 3-way split CGs is smaller than with a direct encryption L2/AES/64B CLs/MirE scheme as in TDX.

This Set of runs proves Results **R1**, **R12**, and **R13**.

4.12. Remarks on area and power

Power consumption of a circuit is roughly a linear function of both its area and the time it is active.[1] Thus, the MPE's total area and the performance penalty are the main factors determining its energy cost.

The area of the MPE mostly consists of arithmetic circuits, caches, and any internal DRAM (if present). In comparison, the control circuitry has negligible area.

Not only is estimating areas for all configurations impractical, but also implementations can vary greatly. For direct encryption schemes like L1 and L2, implementing multiple encryption blocks in parallel maximize performance, but area can be saved by sacrificing some of that performance using pipelined designs. An area-optimized implementation of QARMA-128 (with 256-bit keys) is roughly $\approx$ 50 KGE for a single pipelined block [18]. Latency-optimized AES implementations exceed 17 KGE per round [58], hence the area for a single instance is $\approx$ 160 KGE and for eight parallel blocks $\approx$ 1.3 MGE. Note, also, that a pipelined QARMA circuit and a fully parallelized AES circuit would have comparable total latency — and this would deliver similar performance and security to a L1/L2 scheme, while having different areas.

Integrity can re-use the encryption blocks, or smaller ones like QARMA$_5$-64-σ_0, as in [36].

Remark 4.1. Unlike normal CPU caches, the speed of the MPE caches is not critical: counters and hashes just need to be available to the MPE before the data from RAM. This implies that, instead of SRAM, slower, bur denser, DRAM can be used for these caches.

We recall from Remark 4.1 that the MPE caches can be built from DRAM. A DRAM memory cell uses a capacitor and transistor, or in some cases two transistors. The area can thus be capped by two transistors per bit, with a minor amount of control logic. A 4 KiB cache is thus about 64 KGE, and a 256 KiB cache is roughly 4 MGE.

With these numbers at hand, we see that, for modern SoCs with billions of transistors, a single large MPE per memory channel is a small but not negligible cost. Although an additional 1:128 or 1:256 of in-package or in-module DRAM might seem a minor costs, when compared to the total system memory, it cannot be disregarded, especially considering the added expenses of tamper-averting designs. Architects and implementers must weigh all the trade-offs.

5. Conclusions

We performed a thorough survey and evaluation of the available technologies for the cryptographic protection of memory contents, together with some previously not considered variants. The numerous possible configurations have each their performance penalty, memory overhead, and hardware cost. The lack of an absolute metric to combine these three costs into one rating makes it very challenging to provide recommendations for each use case. This said, we have enough data to provide some rough guidance.

[1] To be more precise, power consumption is the sum of dynamic power, that depends on switching current, and static power, that depends on leakage current, and thus on power gating.

If only confidentiality is needed, L1 schemes can perform very efficiently, and we recommend the use of a lightweight, high-security encryption primitive (e.g., QARMA) in a direct mode. If integrity protection is required, but replay attacks are out of scope, L2 schemes with a short MAC can be made very efficient by using ECC bits to store the MACs.

In what follows, we only consider L3 memory protection: nearly-transparent strong memory protection is possible with current technology, but the hardware costs may be prohibitive.

Server SoCs are expensive, with multiple cores and memory channels. Current systems can address a few terabytes of physical memory. The high total system costs allow us to make an argument for counter-based encryption with high arity counter groups stored in on-chip DRAM. The additional cost for counter group storage would be *relatively* minor (1:128 or 1:256 of the external memory). We observe that placing, say, 64 GiB or more of DRAM in a module close to the main SoC is feasible for client devices today. The same technology could be used to place a large tamper-averting memory in a server SoC package, to be used as a large counter cache. When combined with MirE, it would enable the highest level of memory protection at a lower performance impact than all currently deployed schemes without replay protection.

It can be argued that the area budget for such a large memory should rather be used for a system cache, which benefits the *whole* system and reduces the traffic routed through the MPE. Such a cache could also be dynamically re-partitioned between system and MPE. This would rely on an analysis of the traffic and of the impact of the partitioning that goes beyond the scope of this paper.

If these approaches are not possible, storing the integrity tree off-chip and using MirE still provides good performance when combined with a large counter cache.

On client devices, memories usually lack ECC, making MirE not applicable. However, for use cases such as security modules and business oriented containers, memory bus saturation is less of a concern. We thus expect performance penalties to be contained, usually in line with unloaded systems, and we recommend the use of high arity split counter trees in a dynamically allocated carve-out.

Future work includes contributing our MPE model to the gem5 project which we hope will stimulate future research and enable studies for specific workloads and configurations.

Appendix

1. Additional Results

In Figs. 8 to 13 we collect selected detailed benchmarking results for the **Set 8** and **Set 9** runs.

2. Cryptographic Primitives

2.1. Memory encryption primitives. RAM is commonly encrypted using a block cipher: the long initial latency of stream ciphers makes them unsuitable for the purpose.

For simplicity, we only consider block ciphers with a block size of 128 bits: smaller block sizes are used only for smart cards and small embedded devices, and longer blocks are uncommon. The selected block ciphers are the AES [59] and QARMA [18], where the second is chosen as a representative of lightweight ciphers. The latencies of most suitable lightweight ciphers are similar (e.g., PRINCE [60]) or worse (for instance SKINNY [61]). To estimate performance penalties for these ciphers, readers can interpolate between our AES and QARMA results. A revised version of QARMA, QARMAv2 [62], has been introduced. Its latency is nearly equal to QARMA's, so we do not consider it as a separate configuration option.

Apart from the AES, we do not consider non twekable block ciphers. The reason is that as using such ciphers (even lightweight examples such as MIDORI-128 [63]) in concrete modes would require constructions that lead to increased latency anyway. We also do not consider ciphers with block sizes that make them less suitable for memory encryption: For instance SPEEDY, [64] has a block size of 192 bits, and ASCON [65], which can be used in a tweaked mode such as *Masked Even-Mansour (MEM)* [66], 320 bits.

2.2. Authentication primitives. Standard hash functions such as SHA-2 [67] or SHA-3 [68] can be turned into *Message Authentication Codes (MACs)*, but the resulting schemes are very slow and not parallelizable.

Carter-Wegman Hashes [19], i.e., encrypted *Universal Hash Functions (UHFs)*, are a better choice. UHFs admit fully parallelizable constructions, such as multilinear functions of the input computed over a binary Galois field, as used in SGX [69]. We note that if there is a MAC cache, it is the *not-yet-encrypted* UHF values that are cached: they are thus verified more efficiently.

Apple's Secure Enclave [32] uses a CMAC [70] to compute integrity tags. CMAC can not be made parallel and has a high latency, but their use case does not need very high throughput. It is however unsuitable for general usage requiring high bandwidth and low latency. Instead, we evaluate *Tweakable Block Cipher (TBC)*-based *Parallel MACs (PMACs)* [20]. PMACs are more expensive than encrypted UHFs, but they can be used for error detection and correction beside integrity, cf. [31], [36], [71]. The computation of PMACs is depicted in Figs. 14c and 14d. Such constructions can easily be made *incremental* where, upon a write, only the part of the message that has changed needs to be recomputed. A variant for non-TBCs, called PXOR-MAC is described in [35].

Encrypted checksums of the plaintext as in Rogaway's *Offset Codebook mode (OCB)* [20] are an inexpensive way to compute integrity tags. However, such schemes suffer from two drawbacks. First, they require freshness, and with freshness all practical systems we are aware of use *CounTeR mode (CTR)* encryption with a UHF-based MACs: CTR mode is superior since the block cipher computation is performed in parallel with a ciphertext fetch from external memory, thus reducing the decryption critical path to just one XOR. Second, the encrypted checksum of the plaintext

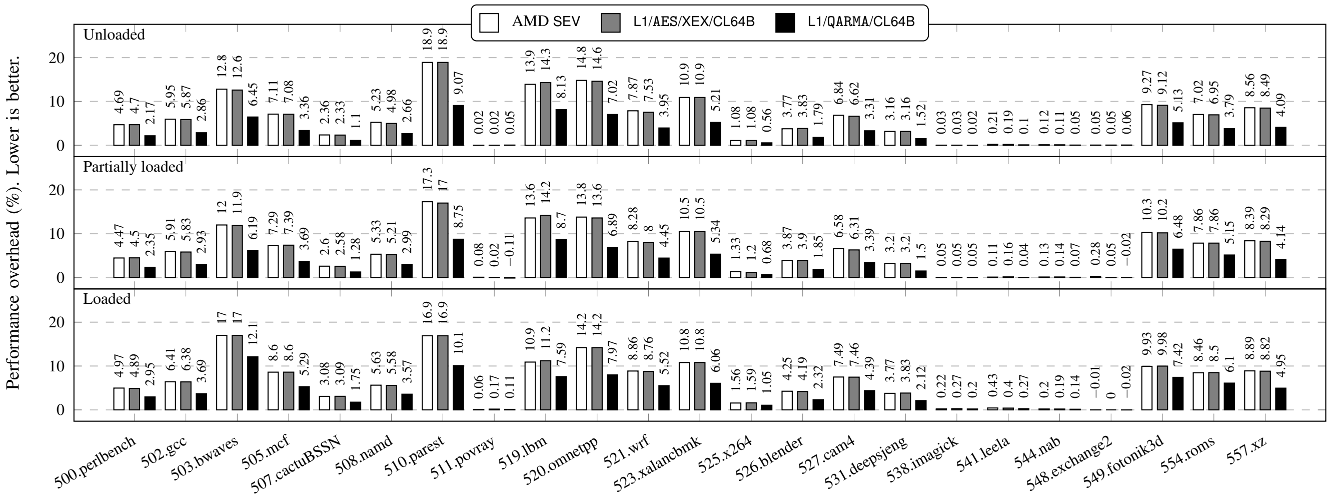


Figure 8: Set 8 (Section 4.10). Comparison of AMD SEV (L1/AES/GFmul/CL64B), L1/AES/XEX/CL64B, and L1/QARMA/CL64B.

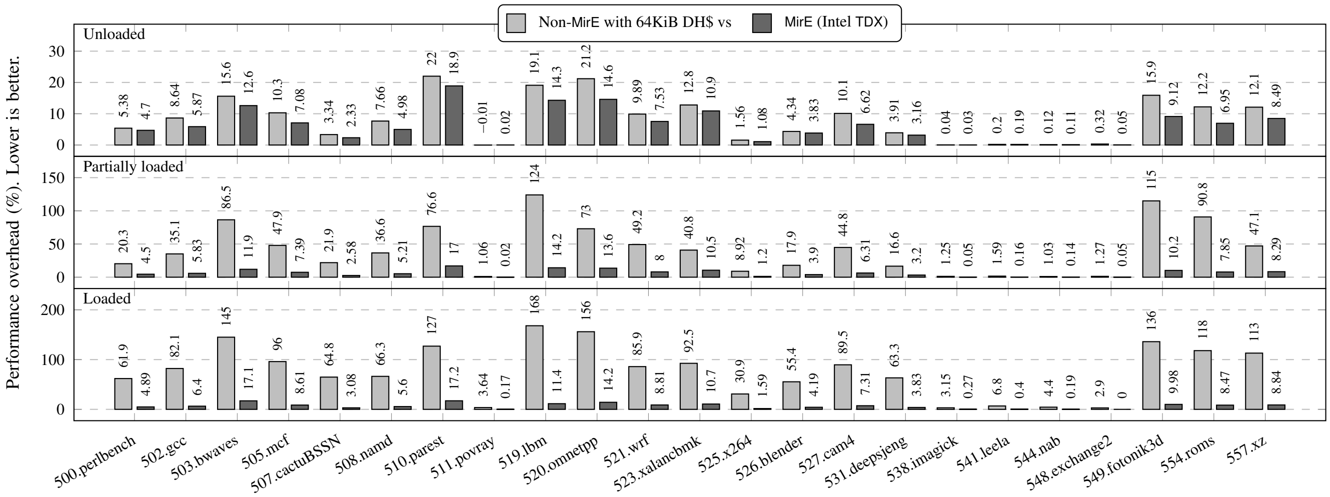


Figure 9: Set 8 (Section 4.10). L2 impact of MirE: L2/AES/32b/CL64B vs. L2/AES/MirE/28b/CL64B (e.g., Intel TDX).

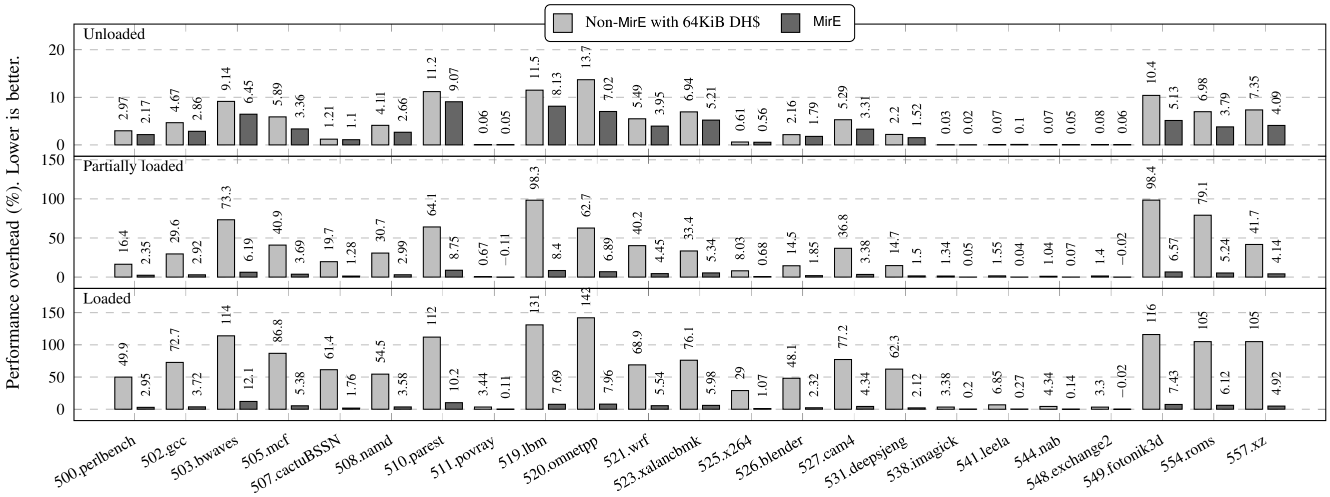


Figure 10: Set 8 (Section 4.10). L2 impact of MirE when using QARMA: L2/QARMA/32b/CL64B vs. L2/QARMA/MirE/28b/CL64B.

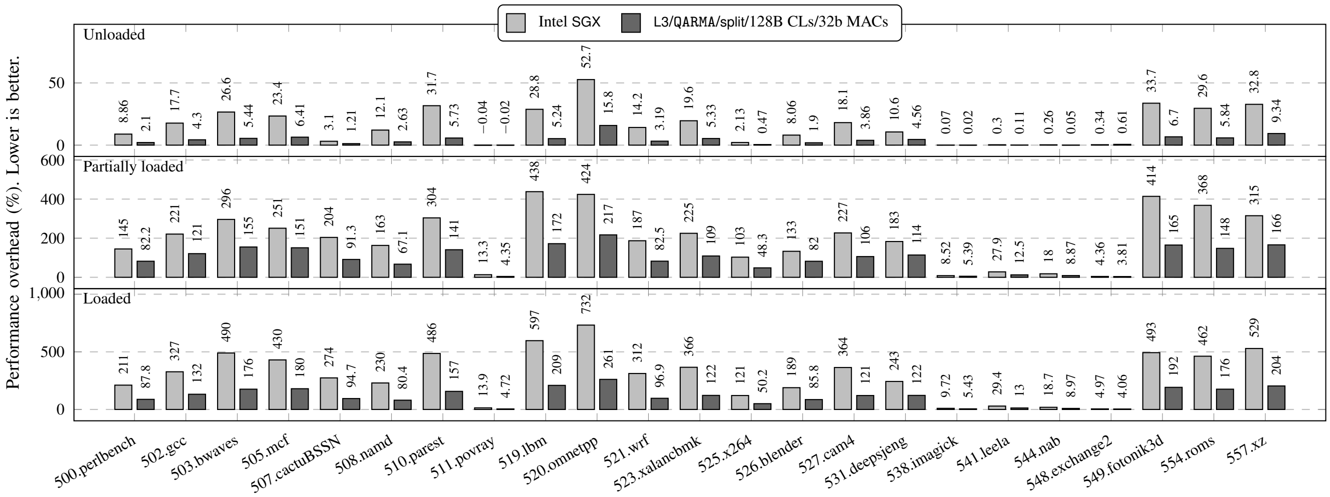


Figure 11: Set 8 (Section 4.10). Impact of split counters: L3/AES/mono-8/56b/CL64B (Intel SGX) vs. L3/QARMA/split-128/32b/CL128B.

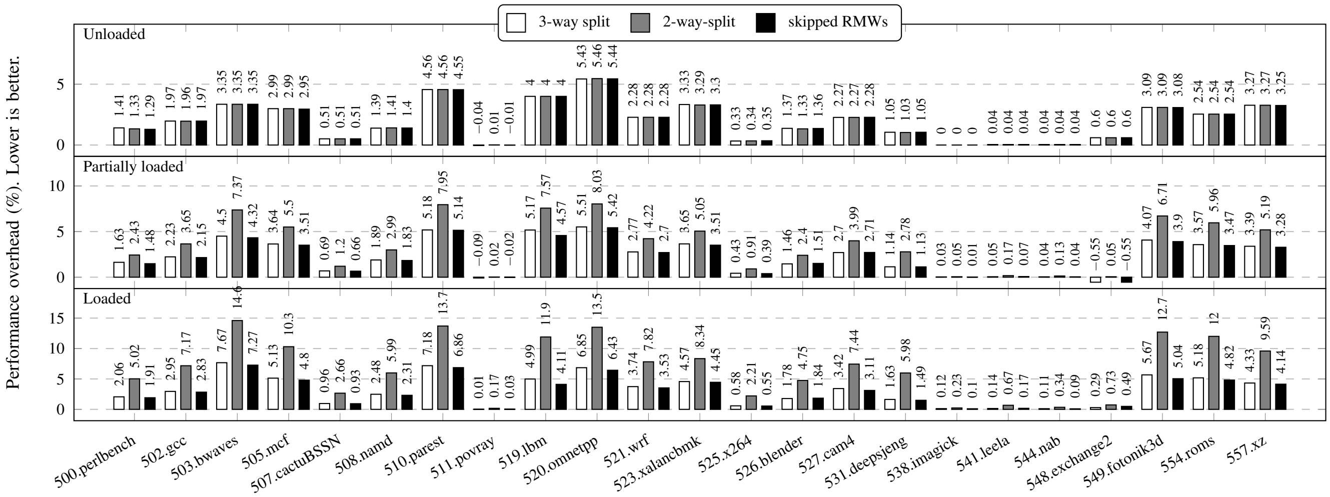


Figure 12: Sets 8 and **9** (Sections 4.10 and 4.11). L3/QARMA/MirE/split-128/LoC/28b/CL128B with 3-way and 2-way split counters, and without RMWs.

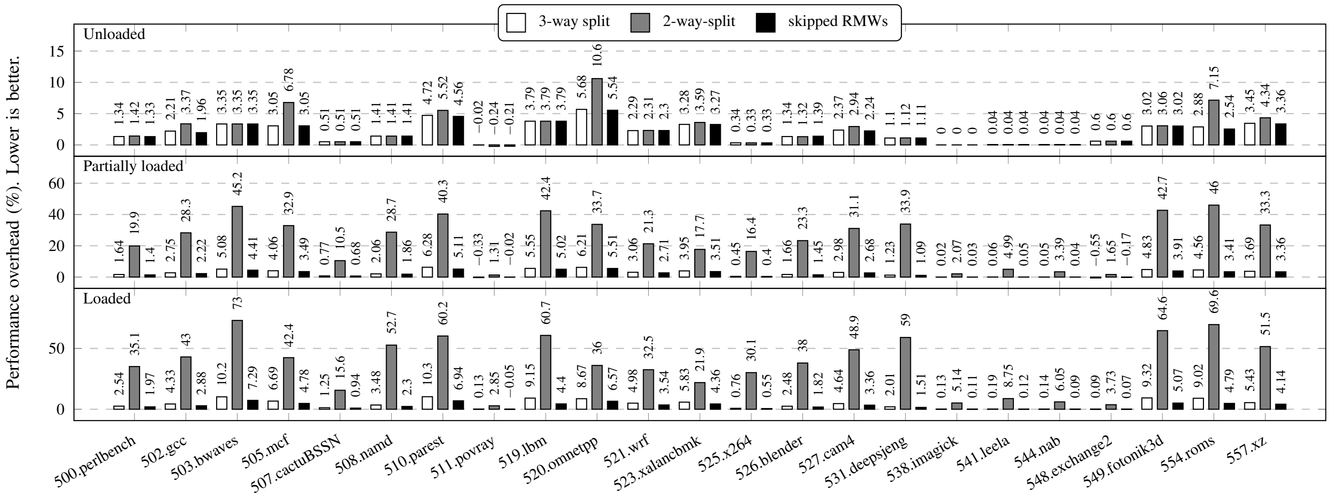
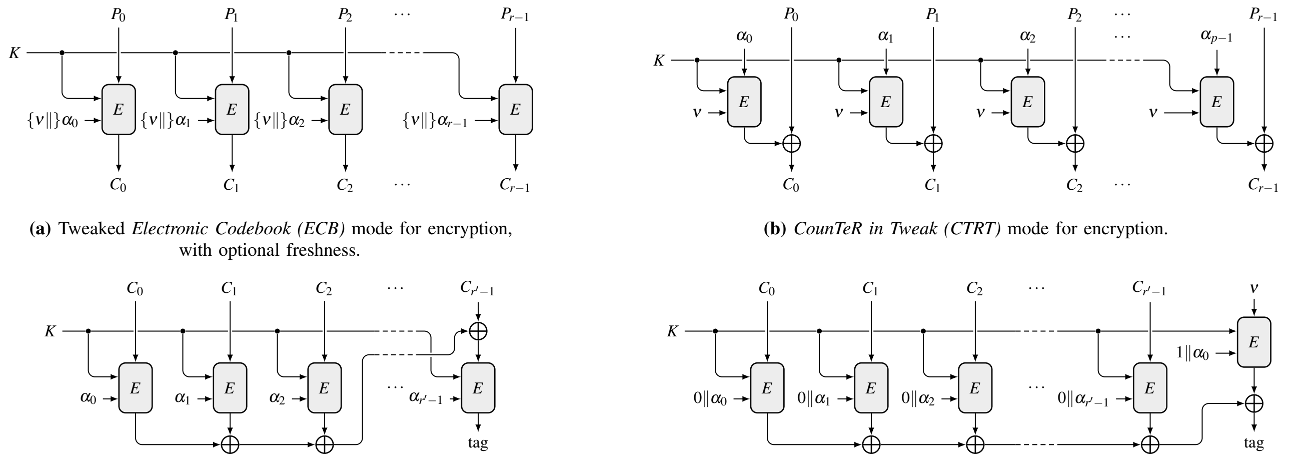


Figure 13: Sets 8 and **9** (Sections 4.10 and 4.11). L3/QARMA/MirE/split-256/LoC/28b/CL128B with 3-way and 2-way split counters, and without RMWs.



(a) Tweaked *Electronic Codebook (ECB)* mode for encryption, with optional freshness.

(b) *CounTeR in Tweak (CTRT)* mode for encryption.

(c) *Parallel MAC (PMAC)* for authentication computed with a TBC for the cases where freshness is not implemented.

(d) PMAC for authentication computed with a TBC for the cases where freshness information is available.

Figure 14: Encryption and authentication methods designed around a *Tweakable Block Cipher (TBC)*.They show how freshness can lead to shorter critical paths. Notation: E is a TBC, the two inputs on the left side of the block being the key (above) and the tweak; $P = P_0\|\cdots\|P_{r-1}$, resp. $C = C_0\|\cdots\|C_{r-1}$ is the partition of a plaintext, resp. ciphertext in blocks of equal size; α_i is the *Physical Address (PA)* of the i-th block; and v a nonce. If freshness is available, both encryption and authentication algorithms use it, and they share the same nonce. The TBC used for authentication may have a smaller block size than the encryption TBC, in which case $r \neq r'$.

needs to be verified after decryption, potentially worsening overall latency.

2.3. Modes of operation. For memory encryption, many (authenticated encryption) modes of operation can be simplified somewhat because the length of the payload is a fixed multiple of the underlying cipher's block length.

Some older schemes, such as Bastion [27], use the block cipher in *Electronic Codebook (ECB)* mode, but the lack of spatial uniqueness keeps plaintext patterns in the ciphertext, therefore modes that provide spatial uniqueness are necessary.

For direct encryption, spatial uniqueness is achieved by using the PA as the tweak. With a non-TBC, the latter is used in the *XOR, Encrypt, and XOR (XEX)* construction [20], which is just the XTS mode of operation [72] for a message whose length is a multiple of the block size. XEX is defined as $C_i = E_K(P_i \oplus M_i) \oplus M_i$. In other words, a tweak-derived *mask* is added to the input and the output of the cipher. The first mask M_0 is derived by encrypting the tweak, and the successive masks M_i for $i \geq 1$ are obtained by multiplying the first mask by a fixed sequence of values. Using a single finite field element γ we can put $M_i = \gamma^i \cdot M_0$. Inoue et al. introduce a Flat-ΘCB mode [35] which is similar to OCB [20]. They define the L3 scheme *Encryption for Large Memory (ELM)* using Flat-ΘCB mode for data and PXOR-MAC to authenticate counter groups.

With a TBC, the PA (concatenated with freshness if provided) of each block is used directly as a tweak, cf. Fig. 14a, and a XEX construction is not needed.

In CTR encryption with a TBC, the counter and PA are used as tweak and text respectively (cf. Fig. 14b) to generate the keystream. When not using a TBC, the counter and PA are concatenated and then encrypted.

References

[1] D. Kaplan, J. Powell, and T. Woller, "AMD Memory Encryption White Paper," April 2016. [Online]. Available: https://www.amd.com/system/files/TechDocs/memory-encryption-white-paper.pdf

[2] D. P. Mulligan, G. Petri, N. Spinale, G. Stockwell, and H. J. M. Vincent, "Confidential Computing - a brave new world," in *Proceedings of SEED 2021*. IEEE, 2021, pp. 132–138, doi:10.1109/SEED51797.2021.00025

[3] S. Gueron, "A Memory Encryption Engine Suitable for General Purpose Processors," *IACR Cryptol. ePrint Arch.*, 2016. [Online]. Available: http://eprint.iacr.org/2016/204

[4] S. Johnson, R. Makaram, A. S. to ni, and V. S. la ta, "Supporting Intel® SGX on multi-socket platforms," August 2020, Technical Report.

[5] Intel, "Intel® Trust Domain Extensions White Paper," August 2021. [Online]. Available: https://www.intel.com/content/www/us/en/developer/articles/technical/intel-trust-domain-extensions.html

[6] J. Bucek, K. Lange, and J. von Kistowski, "SPEC CPU2017: Next-Generation Compute Benchmark," in *Companion of the 2018 ACM/SPEC ICPE*, K. Wolter, W. J. Knottenbelt, A. van Hoorn, and M. Nambiar, Eds. ACM, 2018, pp. 41–42, doi:10.1145/3185768.3185771

[7] N. L. Binkert, B. M. Beckmann, G. Black, S. K. Reinhardt, A. G. Saidi, A. Basu, J. Hestness, D. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. S. B. Altaf, N. Vaish, M. D. Hill, and D. A. Wood, "The gem5 Simulator," *SIGARCH Comput. Archit. News*, vol. 39, no. 2, pp. 1–7, 2011, doi:10.1145/2024716.2024718

[8] J. Lowe-Power, A. M. Ahmad, A. Akram, M. Alian, R. Amslinger, M. Andreozzi, A. Armejach, N. Asmussen, S. Bharadwaj, G. Black, G. Bloom, B. R. Bruce, D. R. Carvalho, J. Castrillón, L. Chen, N. Derumigny, S. Diestelhorst, W. Elsasser, M. Fariborz, A. F. Farahani, P. Fotouhi, R. Gambord, J. Gandhi, D. Gope, T. Grass, B. Hanindhito, A. Hansson, S. Haria, A. Harris, T. Hayes, A. Herrera, M. Horsnell, S. A. R. Jafri, R. Jagtap, H. Jang, R. Jeyapaul, T. M. Jones, M. Jung, S. Kannoth, H. Khaleghzadeh, Y. Kodama, T. Krishna, T. Marinelli, C. Menard, A. Mondelli, T. Mück, O. Naji, K. Nathella, H. Nguyen, N. Nikoleris, L. E. Olson, M. S. Orr, B. Pham, P. Prieto, T. Reddy, A. Roelke, M. Samani, A. Sandberg, J. Setoain, B. Shingarov, M. D. Sinclair, T. Ta, R. Thakur, G. Travaglini, M. Upton, N. Vaish,

I. Vougioukas, Z. Wang, N. Wehn, C. Weis, D. A. Wood, H. Yoon, and É. F. Zulian, "The gem5 Simulator: Version 20.0+," *CoRR*, vol. abs/2007.03152, 2020, doi:10.48550/arXiv.2007.03152

[9] M. Henson and S. Taylor, "Memory Encryption: A Survey of Existing Techniques," *ACM Comput. Surv.*, vol. 46, no. 4, pp. 53:1–53:26, 2013, doi:10.1145/2566673

[10] S. Mofrad, F. Zhang, S. Lu, and W. Shi, "A comparison study of intel SGX and AMD memory encryption technology," in *Proceedings of the 7th International HASP@ISCA 2018 Workshop, Los Angeles, CA, USA, June 02-02, 2018*, J. Szefer, W. Shi, and R. B. Lee, Eds. ACM, 2018, pp. 9:1–9:8, doi:10.1145/3214292.3214301

[11] K. Suzaki, K. Nakajima, T. Oi, and A. Tsukamoto, "TS-Perf: General Performance Measurement of Trusted Execution Environment and Rich Execution Environment on Intel SGX, Arm TrustZone, and RISC-V Keystone," *IEEE Access*, vol. 9, pp. 133 520–133 530, 2021, doi:10.1109/ACCESS.2021.3112202

[12] G. Saileshwar, P. J. Nair, P. Ramrakhyani, W. Elsasser, J. A. Joao, and M. K. Qureshi, "Morphable Counters: Enabling Compact Integrity Trees For Low-Overhead Secure Memories," in *Proceedings of the 50th IEEE/ACM MICRO, 2018*. IEEE Computer Society, 2018, pp. 416–427, doi:10.1109/MICRO.2018.00041

[13] M. Taassori, A. Shafiee, and R. Balasubramonian, "VAULT: Reducing Paging Overheads in SGX with Efficient Integrity Verification Structures," in *Proceedings of ASPLOS 2018*, X. Shen, J. Tuck, R. Bianchini, and V. Sarkar, Eds. ACM, 2018, pp. 665–678, doi:10.1145/3173162.3177155

[14] M. Taassori, R. Balasubramonian, S. Chhabra, A. R. Alameldeen, M. Peddireddy, R. Agarwal, and R. Stutsman, "Compact leakage-free support for integrity and reliability," in *Proceedings of the 47th ISCA*. IEEE, 2020, pp. 735–748, doi:10.1109/ISCA45697.2020.00066

[15] I. Mihalcea, "Prototyping Memory Integrity Tree Algorithms for Internet of Things Devices," Master's thesis, Information Security Group, Royal Holloway University of London, UK, 2022.

[16] D. H. Schall, "Evaluation and Optimization of Memory Encryption and Integrity Protection," Master's thesis, University of Kaiserslautern, Department of Electrical Engineering and Information Technology, Microelectronic Systems Design Research Group, 2019.

[17] M. Schneider, R. J. Masti, S. Shinde, S. Capkun, and R. Perez, "Sok: Hardware-supported trusted execution environments," *CoRR*, vol. abs/2205.12742, 2022, doi:10.48550/arXiv.2205.12742

[18] R. Avanzi, "The QARMA Block Cipher Family – Almost MDS Matrices over Rings with Zero Divisors, Nearly Symmetric Even-Mansour Constructions with Non-Involutory Central Rounds, and Search Heuristics for Low-Latency S-Boxes," *IACR Trans. on Symmetric Cryptology*, vol. 2017, no. 1, pp. 4–44, 2017, doi:10.13154/tosc.v2017.i1.4-44

[19] L. Carter and M. N. Wegman, "Universal Classes of Hash Functions," *J. Comput. Syst. Sci.*, vol. 18, no. 2, pp. 143–154, 1979, doi:10.1016/0022-0000(79)90044-8

[20] P. Rogaway, "Efficient Instantiations of Tweakable Blockciphers and Refinements to Modes OCB and PMAC," in *ASIACRYPT 2004, Proceedings*, 2004, pp. 16–31, doi:10.1007/978-3-540-30539-2_2

[21] W. E. Hall and C. S. Jutla, "US Patent US US7451310 B2: Parallelizable authentication tree for random access storage, filed Dec. 2,2002," http://www.google.com/patents/US7451310, November 2008.

[22] G. E. Suh, D. E. Clarke, B. Gassend, M. van Dijk, and S. Devadas, "AEGIS: architecture for tamper-evident and tamper-resistant processing," in *Proceedings of the 17th Annual International Conference on Supercomputing, ICS 2003*, U. Banerjee, K. Gallivan, and A. González, Eds. ACM, 2003, pp. 160–171, doi:10.1145/782814.782838

[23] B. Gassend, G. E. Suh, D. E. Clarke, M. van Dijk, and S. Devadas, "Caches and Hash Trees for Efficient Memory Integrity Verification," in *Proceedings of HPCA'03*, 2003, pp. 295–306, doi:10.1109/HPCA.2003.1183547

[24] G. E. Suh, D. E. Clarke, B. Gassend, M. van Dijk, and S. Devadas, "Efficient Memory Integrity Verification and Encryption for Secure Processors," in *Proceedings of the 36th Annual International Symposium on Microarchitecture*, 2003, pp. 339–350, doi:10.1109/MICRO.2003.1253207

[25] C. Yan, D. Englender, M. Prvulovic, B. Rogers, and Y. Solihin, "Improving Cost, Performance, and Security of Memory Encryption and Authentication," in *Proceedings of ISCA 2006*, 2006, pp. 179–190, doi:10.1109/ISCA.2006.22

[26] 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 MICRO-40, 2007*, 2007, pp. 183–196, doi:10.1109/MICRO.2007.44

[27] D. Champagne and R. B. Lee, "Scalable architectural support for trusted software," in *Proceedings of HPCA 2010*, 2010, pp. 1–12, doi:10.1109/HPCA.2010.5416657

[28] F. McKeen, I. Alexandrovich, A. Berenzon, C. V. Rozas, H. Shafi, V. Shanbhogue, and U. R. Savagaonkar, "Innovative instructions and software model for isolated execution," in *Proceedings of HASP 2013*, 2013, p. 10, doi:10.1145/2487726.2488368

[29] AMD, "Secure Encrypted Virtualization API Version 0.24," April 2020, Technical Report.

[30] ——, "AMD SEV-SNP: Strengthening VM isolation with integrity protection and more," January 2020, Technical Report.

[31] G. Saileshwar, P. J. Nair, P. Ramrakhyani, W. Elsasser, and M. K. Qureshi, "SYNERGY: Rethinking Secure-Memory Design for Error-Correcting Memories," in *Proceedings of HPCA 2018*. IEEE Computer Society, 2018, pp. 454–465, doi:10.1109/HPCA.2018.00046

[32] Apple Inc., "Secure Enclave," 2020. [Online]. Available: https://support.apple.com/en-gb/guide/security/sec59b0b31ff/web

[33] D. Lee, D. Kohlbrenner, S. Shinde, K. Asanovic, and D. Song, "Keystone: an open framework for architecting trusted execution environments," in *Proceedings of EuroSys '20*, A. Bilas, K. Magoutis, E. P. Markatos, D. Kostic, and M. I. Seltzer, Eds. ACM, 2020, pp. 38:1–38:16, doi:10.1145/3342195.3387532

[34] 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 OSDI, July 14-16, 2021*, A. D. Brown and J. R. Lorch, Eds. USENIX Association, 2021, pp. 275–294. [Online]. Available: https://www.usenix.org/conference/osdi21/presentation/feng

[35] A. Inoue, K. Minematsu, M. Oda, R. Ueno, and N. Homma, "ELM: A Low-Latency and Scalable Memory Encryption Scheme," *IEEE Trans. Inf. Forensics Secur.*, vol. 17, pp. 2628–2643, 2022, doi:10.1109/TIFS.2022.3188146

[36] J. Juffinger, L. Lamster, A. Kogler, M. Lipp, M. Eichlseder, and D. Gruss, "CSI:Rowhammer – Cryptographic Security and Integrity against Rowhammer," in *Proceedings of IEEE S&P '23*, 2023.

[37] J. Kelsey, "Compression and Information Leakage of Plaintext," in *Proceedings of FSE 2002*, ser. Lecture Notes in Computer Science, J. Daemen and V. Rijmen, Eds., vol. 2365. Springer, 2002, pp. 263–276, doi:10.1007/3-540-45661-9_21

[38] M. Schwarzl, P. Borrello, G. Saileshwar, H. Müller, M. Schwarz, and D. Gruss, "Practical Timing Side Channel Attacks on Memory Compression," *CoRR*, vol. abs/2111.08404, 2021, doi:10.48550/arXiv.2111.08404

[39] O. Goldreich, "Towards a theory of software protection and simulation by oblivious rams," in *Proceedings of STOC, 1987*, A. V. Aho, Ed. ACM, 1987, pp. 182–194, doi:10.1145/28395.28416

[40] M. A. Khelif, J. Lorandel, O. Romain, M. Regnery, D. Baheux, and G. Barbu, "Toward a hardware Man-in-the-Middle attack on PCIe bus," *Microprocess. Microsystems*, vol. 77, 2020, doi:10.1016/j.micpro.2020.103198

[41] S. Skorobogatov, “How microprobing can attack encrypted memory,” in *Proceedings of DSD 2017*, H. Kubátová, M. Novotný, and A. Skavhaug, Eds. IEEE Computer Society, 2017, pp. 244–251, doi:10.1109/DSD.2017.69

[42] R. Torrance and D. James, “The State-of-the-Art in IC Reverse Engineering,” in *Proceedings of CHES 2009*, ser. Lecture Notes in Computer Science, C. Clavier and K. Gaj, Eds., vol. 5747. Springer, 2009, pp. 363–381, doi:10.1007/978-3-642-04138-9_26

[43] J. A. Halderman, S. D. Schoen, N. Heninger, W. Clarkson, W. Paul, J. A. Calandrino, A. J. Feldman, J. Appelbaum, and E. W. Felten, “Lest we remember: cold-boot attacks on encryption keys,” *Commun. ACM*, vol. 52, no. 5, pp. 91–98, 2009, doi:10.1145/1506409.1506429

[44] M. Bach, “ECC and REG ECC Memory Performance,” May 2014. [Online]. Available: https://www.pugetsystems.com/labs/articles/ECC-and-REG-ECC-Memory-Performance-560/

[45] R. C. Merkle, “Protocols for Public Key Cryptosystems,” in *Proceedings of the 1980 IEEE S&P*. IEEE Computer Society, 1980, pp. 122–134, doi:10.1109/SP.1980.10006

[46] W. E. Hall and C. S. Jutla, “Parallelizable Authentication Trees,” in *Proceedings of SAC 2005*, 2005, pp. 95–109, doi:10.1007/11693383_7

[47] R. Elbaz, D. Champagne, R. B. Lee, L. Torres, G. Sassatelli, and P. Guillemin, “TEC-Tree: A Low-Cost, Parallelizable Tree for Efficient Defense Against Memory Replay Attacks,” in *Proceedings of CHES 2007*, 2007, pp. 289–302, doi:10.1007/978-3-540-74735-2_20

[48] S. Jaques, M. Naehrig, M. Roetteler, and F. Virdia, “Implementing Grover Oracles for Quantum Key Search on AES and LowMC,” in *Proceedings of EUROCRYPT 2020, Part II*, ser. Lecture Notes in Computer Science, A. Canteaut and Y. Ishai, Eds., vol. 12106. Springer, 2020, pp. 280–310, doi:10.1007/978-3-030-45724-2_10

[49] P. Colp, J. Zhang, J. Gleeson, S. Suneja, E. de Lara, H. Raj, S. Saroiu, and A. Wolman, “Protecting data on smartphones and tablets from memory attacks,” in *Proceedings of ASPLOS 2015*, Ö. Özturk, K. Ebcioglu, and S. Dwarkadas, Eds. ACM, 2015, pp. 177–189, doi:10.1145/2694344.2694380

[50] T. Matsumoto, R. Miyachi, J. Sakamoto, M. Suzuki, D. Watanabe, and N. Yoshida, “RAM encryption mechanism without hardware support,” *J. Inf. Process.*, vol. 28, pp. 473–480, 2020. [Online]. Available: https://doi.org/10.2197/ipsjjip.28.473, doi:10.2197/ipsjjip.28.473

[51] P. A. H. Peterson, “Cryptkeeper: Improving security with encrypted RAM,” in *Proceedings of IEEE HST 2010*, 2010, pp. 120–126, doi:10.1109/THS.2010.5655081

[52] J. Götzfried, T. Müller, G. Drescher, S. Nürnberger, and M. Backes, “Ramcrypt: Kernel-based address space encryption for user-mode processes,” in *Proceedings of AsiaCCS 2016*, X. Chen, X. Wang, and X. Huang, Eds. ACM, 2016, pp. 919–924, doi:10.1145/2897845.2897924

[53] S. Aga and S. Narayanasamy, “InvisiMem: Smart Memory Defenses for Memory Bus Side Channel,” in *Proceedings of ISCA 2017*, 2017, pp. 94–106, doi:10.1145/3079856.3080232

[54] CXL Consortium, “Compute express link™ resource library,” 2019. [Online]. Available: https://www.computeexpresslink.org/resource-library

[55] A. Sandberg, “Understanding Multicore Performance: Efficient Memory System Modeling and Simulation,” Ph.D. dissertation, Uppsala University, Disciplinary Domain of Science and Technology, Mathematics and Computer Science, Department of Information Technology, Division of Computer Systems, Uppsala, Sweden, 2014.

[56] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder, “Automatically Characterizing Large Scale Program Behavior,” in *ACM SIGPLAN Notices, vol. 37 (Proceedings of ASPLOS-X, 2002)*, K. Gharachorloo and D. A. Wood, Eds. ACM Press, 2002, pp. 45–57, doi:10.1145/605397.605403

[57] S. F. Yitbarek and T. M. Austin, “Reducing the Overhead of Authenticated Memory Encryption Using Delta Encoding and ECC Memory,” in *Proceedings of DAC 2018*. ACM, 2018, pp. 1–35, doi:10.1145/3195970.3196102

[58] R. Ueno, N. Homma, S. Morioka, N. Miura, K. Matsuda, M. Nagata, S. Bhasin, Y. Mathieu, T. Graba, and J. Danger, “High Throughput/Gate AES Hardware Architectures Based on Datapath Compression,” *IEEE Trans. Computers*, vol. 69, no. 4, pp. 534–548, 2020, doi:10.1109/TC.2019.2957355

[59] J. Daemen and V. Rijmen, “AES and the Wide Trail Design Strategy,” in *EUROCRYPT 2002*, ser. Lecture Notes in Computer Science, L. R. Knudsen, Ed., vol. 2332. Springer, 2002, pp. 108–109, doi:10.1007/3-540-46035-7_7

[60] J. Borghoff, A. Canteaut, T. Güneysu, E. B. Kavun, M. Knezevic, L. R. Knudsen, G. Leander, V. Nikov, C. Paar, C. Rechberger, P. Rombouts, S. S. Thomsen, and T. Yalçin, “PRINCE — A Low-Latency Block Cipher for Pervasive Computing Applications - Extended Abstract,” in *ASIACRYPT 2012*, ser. Lecture Notes in Computer Science, X. Wang and K. Sako, Eds., vol. 7658. Springer, 2012, pp. 208–225, doi:10.1007/978-3-642-34961-4_14

[61] C. Beierle, J. Jean, S. Kölbl, G. Leander, A. Moradi, T. Peyrin, Y. Sasaki, P. Sasdrich, and S. M. Sim, “The SKINNY family of block ciphers and its low-latency variant MANTIS,” in *Proceedings of CRYPTO 2016, Part II*, ser. Lecture Notes in Computer Science, M. Robshaw and J. Katz, Eds., vol. 9815. Springer, 2016, pp. 123–153, doi:10.1007/978-3-662-53008-5_5

[62] R. Avanzi, S. Banik, O. Dunkelman, M. Eichlseder, S. Ghosh, M. Nageler, and F. Regazzoni, “The QARMAv2 family of tweakable block ciphers,” *IACR Transactions on Symmetric Cryptology*, no. 3, pp. 25–73, Sep. 2023, doi:10.46586/tosc.v2023.i3.25-73

[63] S. Banik, A. Bogdanov, T. Isobe, K. Shibutani, H. Hiwatari, T. Akishita, and F. Regazzoni, “Midori: A Block Cipher for Low Energy,” in *Proceedings of ASIACRYPT 2015, Part II*, ser. Lecture Notes in Computer Science, T. Iwata and J. H. Cheon, Eds., vol. 9453. Springer, 2015, pp. 411–436, doi:10.1007/978-3-662-48800-3_17

[64] G. Leander, T. Moos, A. Moradi, and S. Rasoolzadeh, “The SPEEDY family of block ciphers engineering an ultra low-latency cipher from gate level for secure processor architectures,” *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, vol. 2021, no. 4, pp. 510–545, 2021, doi:10.46586/tches.v2021.i4.510-545

[65] C. Dobraunig, M. Eichlseder, F. Mendel, and M. Schläffer, “Ascon v1.2: Lightweight authenticated encryption and hashing,” *J. Cryptol.*, vol. 34, no. 3, p. 33, 2021, doi:10.1007/s00145-021-09398-9

[66] R. Granger, P. Jovanovic, B. Mennink, and S. Neves, “Improved masking for tweakable blockciphers with applications to authenticated encryption,” in *Proceedings of EUROCRYPT 2016, Part I*, ser. Lecture Notes in Computer Science, M. Fischlin and J. Coron, Eds., vol. 9665. Springer, 2016, pp. 263–293, doi:10.1007/978-3-662-49890-3_11

[67] NIST, “FIPS PUB 180-4 – Secure Hash Standard,” National Institute of Standards and Technology, Gaithersburg, MD, United States, Tech. Rep., Mar. 2012. [Online]. Available: https://csrc.nist.gov/publications/detail/fips/180/4/final

[68] ——, “FIPS PUB 202 – SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions,” National Institute of Standards and Technology, Gaithersburg, MD, United States, Tech. Rep., Aug. 2015. [Online]. Available: https://csrc.nist.gov/publications/detail/fips/202/final

[69] S. Gueron, “Memory Encryption for General-Purpose Processors,” *IEEE Secur. Priv.*, vol. 14, no. 6, pp. 54–62, 2016, doi:10.1109/MSP.2016.124

[70] T. Iwata and K. Kurosawa, “OMAC: One-Key CBC MAC,” in *FSE 2003, Revised Papers*, ser. Lecture Notes in Computer Science, T. Johansson, Ed., vol. 2887. Springer, 2003, pp. 129–153, doi:10.1007/978-3-540-39887-5_11

[71] R. C. Huang and G. E. Suh, “IVEC: Off-Chip Memory Integrity Protection for Both Security and Reliability,” in *Proceedings of ISCA 2010*, A. Seznec, U. C. Weiser, and R. Ronen, Eds. ACM, 2010, pp. 395–406, doi:10.1145/1815961.1816015

[72] IEEE, “IEEE standard for cryptographic protection of data on block-oriented storage devices 1619–2018,” January 2019. [Online]. Available: http://ieeexplore.ieee.org/servlet/opac?punumber=4493431