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, 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*—There is a trend towards providing stronger isolation between mutually untrusted processes running on the same computer system. For example, Intel SGX, AMD SEV, and Arm CCA use access control to protect programs from hostile peer and higher privileged software. Some of these technologies include cryptographic memory protection, such as encryption and integrity checks, to protect against physical attacks.**

We review the main technologies that ensure confidentiality and integrity of main memory. Our review covers both academic proposals and deployed technology. We classify these technologies according to models of adversaries with varying capabilities and group them according to their protection levels. To understand the best and worst case overhead, we evaluate these technologies on a system where the benchmark suite is running in isolation *and* on a heavily loaded system.

We additionally propose new solutions to further reduce the performance and memory overheads of such protection. For example, we show that advanced counter compression techniques make it viable to store counters used for replay protection in a physically protected memory. This memory is just 1:256 of the total off-chip memory. By repurposing some ECC bits to store integrity tags, we can achieve hitherto unattained performance while providing full confidentiality, integrity, and replay protection. In a representative server system running the industry-standard SPEC 2017 benchmark suite, we achieve a 1.96% performance overhead if the benchmarks run in isolation and 3.28% when the memory system is fully saturated.

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, which 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.

Solutions such as AMD SEV [1], Arm CCA [2], and Intel SGX [3] and TDX [4] move towards this goal by providing architectural mechanisms to reduce the impact of hypervisor exploits and malicious operators. Some even include protection against adversaries with physical access to the system. For instance, Intel SGX implements a *Memory Encryption Engine (MEE)* [3] that provides confidentiality, as well as integrity and replay protection using an integrity tree. Such strong confidentiality and integrity guarantees can be very costly in terms of performance and storage. Other technologies such as AMD SEV and Intel TDX opt to provide weaker confidentiality and integrity guarantees to offer improved performance.

In this paper, we review the state of the art techniques used to provide confidentiality and integrity guarantees in hardware. We cover the techniques used to protect off-chip memories (e.g., DRAM) from an adversary with physical access to the system. We then propose new solutions to further reduce performance and memory overheads while maintaining the highest level of integrity and confidentiality.

In order to systematically classify and compare existing techniques, we define different protection levels based on a taxonomy of adversaries with varying technical capabilities. Suitable technical mitigations are then deployed to implement each protection level. This lets us reason about the trade-off between security guarantees and performance.

The evaluation uses the industry-standard SPEC 2017 [5] benchmark suite running on the gem5 simulator [6], [7]. We use the entire benchmark suite and do not pick just a few benchmarks. The most striking result is that advanced counter compression makes it viable to store counters in a physically secure memory that is relatively small with respect to the total RAM, i.e. 1:128 or 1:256. This enables implementations of memory replay protection with very low performance penalties, especially if some ECC bits are repurposed to

store integrity tags: Performance penalties smaller than 5% can be attained even under heavy bus contention.

To our knowledge, this is also the first evaluation with various degrees of memory bus loading and with randomization of the internal state state to simulate the more realistic performance of a not-freshly booted system.

2. Systematization of the problem

2.1. Definitions

The software-accessible volatile memory attached to a memory controller is viewed as an array of blocks. The size of these blocks corresponds to the cache line size of the *last level cache*, which is usually a system cache. Due to their direct correspondence to cache lines, we call these blocks cache lines even when stored in off-chip memory.

If a scheme provides *integrity*, it associates an *integrity tag* with one or more cache lines. Commonly, the integrity tag is a *Message Authentication Code (MAC)*.

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 can be achieved by associating a counter with each cache line and including it in the computation of the function.

Here, an *on-chip* component is defined as a physically secure block in the same package as the processing elements.

2.2. Problem statement and protection levels

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

To properly answer this question we need to group the technologies according to the adversaries they are meant to defend against, avoiding artificial distinctions between adversaries, for instance between adversaries capable of running software on the target device and adversaries that can mount RowHammer attacks [8]. Classifying adversaries primarily on how they access the target devices and their resources (i.e. budget) results in the cleanest classification. Complexity and cost of the attack are a secondary concern.

2.2.1. Level 1: Basic memory encryption. *This level provides only memory confidentiality. This level provides spatial uniqueness, but neither temporal uniqueness nor integrity verification.*

It uses a direct memory encryption method, i.e. the plaintext is passed directly through the encryption function to compute the ciphertext. No metadata is stored.

This level is meant to defeat adversaries that can only run software on the target and manipulate external interfaces. Beside software exploitation, the adversaries can also mount RowHammer attacks. However, in general integrity violations are only a partial concern, as they can arguably be made less effective by deploying memory encryption. We exclude access pattern and ciphertext side channels as these variants require more sophisticated techniques.

This basic Protection Level is deployed in many commercial systems, a notable example being AMD's SEV [1].

It can be argued that careful access memory allocation and alignment that physically separates mutually untrusted components could be used to prevent RowHammer attacks. This approach works in theory, but it would require extensive re-engineering of existing system software and would not be practical in most real-world scenarios.

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

2.2.2. Level 2: Encryption and integrity verification. *This level extends Level 1 with integrity tags, in order to detect memory corruption. but does not provide any temporal uniqueness. Adversaries can still mount replay attacks.*

Level 2 targets adversaries with physical access to the complete system containing the protected components. The adversaries have access to exposed interfaces and communication buses but they do not have the capabilities to access on-chip communication interfaces. They mainly perform passive attacks, e,g,: physical side-channel analysis in close proximity, contact or connection to the target device; eavesdropping the content of external RAM, either at runtime via memory bus probing, chip or module interposition; abuse of DMA channels; or cold-boot attacks.

Because of the similarity of the involved techniques, these countermeasures are also effective against limited active adversaries, which may only be able to corrupt individual memory locations. In order to defeat targeted replay of the memory together with the integrity tags, more sophisticated countermeasures are required (see Level 3 below). It can be argued that this distinction is arbitrary, but in reality it is made necessary by the difference in complexity and cost of both attacks and *countermeasures*. In other words, the system designer and integrator will decide whether to accept risks arising from specific adversaries following a proper risk assessment. That is, this ends up being a business decision and as such is outside the scope of this paper. Still, an adversary capable of active HW attacks may choose to perform passive attacks instead, at least initially, for reasons of *detectability*. A passive attack has the advantage of being less likely to trigger errors that may betray the perpetrator.

The Intel TDX *Multi-Key Total Memory Engine with Integrity (MKTMEi)* [4] is a L2/MirE solution, where MirE means *MACs in repurposed ECC bits*. We found no documentation on error correction in a TDX system, but the 28b MAC field size suggests the following: Four instances of a $(255, 247)$ Hamming code are used, truncated to $(143, 135)$ to cover 128 bits and 7 bits of the MAC each — with the remaining 4 bits of the effective 576 bits in each cache line

used for parity. This very configuration is proposed in [9]. CSI:Rowhammer [10] is a L2/MirE solution as well.

2.2.3. Level 3: Encryption and replay protection. *This level includes* Replay protection*, which is any form of integrity protection that is capable of detecting not only memory corruption, but also replay of memory contents including associated metadata.*

In addition to the capabilities of the previous adversaries, the adversaries addressed in this Level also perform *active* attacks, e.g.: blocking, corrupting, replaying memory transactions, or even injecting new ones [11]. See also [12]–[14].

The Intel SGX *Memory Encryption Engine* [3], ELM [15], and the memory protection mechanism of Apple's Secure Enclave [16] (starting with the A11 and S4 SoCs) are L3 solutions. SYNERGY [17] is a L3/MirE solution.

2.2.4. Out of scope adversaries. Out of scope for the research described in this paper are adversaries that can mount highly invasive attacks at the chip or package level that require considerable experience, resources, and time to succeed. Examples of such attacks range from micro-probing attacks [18] to actual chip reverse engineering and editing using a Focused Ion Beam Microscope [19]–[21].

Address scrambling has been deployed to thwart adversaries that reverse engineer software properties from access patterns. Such schemes however provide a static scrambling per boot session, and an attacker can still detect address reuse: if the program is known, this is all they need to mount the attack. It can be argued that address scrambling can make RowHammer attacks more difficult, but so does also data scrambling. Therefore we do not consider these adversaries.

2.3. System level view of the technical solution

To implement the above Protection Levels, we introduce a *Memory Protection Engine (MPE)*. This is not a new idea: all cryptographic memory protection designs use such a HW block, sometimes known as a *Memory Encryption Engine (MEE)*. As depicted in Fig. 1, the MPE sits between the main interconnect (or a system cache) on one side and a memory controller on the other side in a typical SoC. It can optionally have caches, internal buffers (not depicted), and even access to a physically secure DRAM (e.g., on-chip or in the same package) to store metadata. *The memory protection schemes studied in this paper are implemented in a MPE.*

The fact that we functionally represent the MPE as a separate block sitting between system cache and memory controller does not preclude other designs. For example, a MPE could be part of the memory controller or a wrapper around the system cache. A MPE would normally be associated either with each memory channel, and thus reside between on-chip interconnect and external memory, or with each core cluster (or core) that needs memory protection, and thus reside between those cores and the on-chip interconnect. We do not consider the latter design because it is uncommon in the literature and it becomes a bottleneck when the associated core or core cluster is generating considerable traffic. In such a design, data is already encrypted when stored in the system cache; integrity and anti-replay information may be generated either by the core-private MPE or only upon eviction from the system cache. Unlike core-private MPEs, MPEs associated with each memory channel benefit from large system caches and memory interleaving to reduce bandwidth saturation risks. This said, we acknowledge that some *secure cores* may need a private MEE.

General purpose software solutions can be envisioned as well. For instance, memory management could be delegated to a trusted piece of software which pages encrypted memory in and out of a dedicated, on-chip, scratch-pad memory. In these cases, however, performance is usually not a strong requirement, and is expected to be poor.

It is likely that the full design space is not knowable, and there may be further meaningful realizations of memory protection that are unknown to us.

2.4. Building blocks

The following types of technologies are used to implement the various protection levels: (i) Memory encryption primitives and modes; (ii) Authentication primitives; (iii) Integrity and replay protection structures; and (iv) Physical mechanisms to protect *a relatively small amount* of memory from tampering, such as including it in a tamper-proof package. The first groups are reviewed in Section 3.

Note that we exclusively consider solutions that need the security perimeter to be no larger than the physical package of the SoC. Hence, "smart memory" [22] is out of scope.

2.5. Cost indicators

For real-world applications it is very important to know how *expensive* a solution to a problem is. The two principal cost indicators are the performance penalty and the memory overhead. Area and power constraints restrict which solutions can be considered for viability, but relaxing these constraints can sometimes be justified in the presence of a strong market requirement. On the other hand, a solution that impacts performance or memory availability too heavily will face major acceptance hurdles. For this reason, *we focus mainly on performance penalty and memory overhead.*

Since power consumption of a circuit is roughly linear in both its area and the time it is active, the MPE's area (including caches and on-chip memory) and the performance penalty are the main factors determining its energy cost.

3. Background

We summarize the main technologies used in this paper.

Memory encryption primitives. Block ciphers are the most commonly used primitives for memory encryption. Stream ciphers have a long initial latency which makes them unsuitable in this context.

In *direct encryption*, the block cipher is applied blockwise to the plaintext to generate the ciphertext. In *CounTeR*

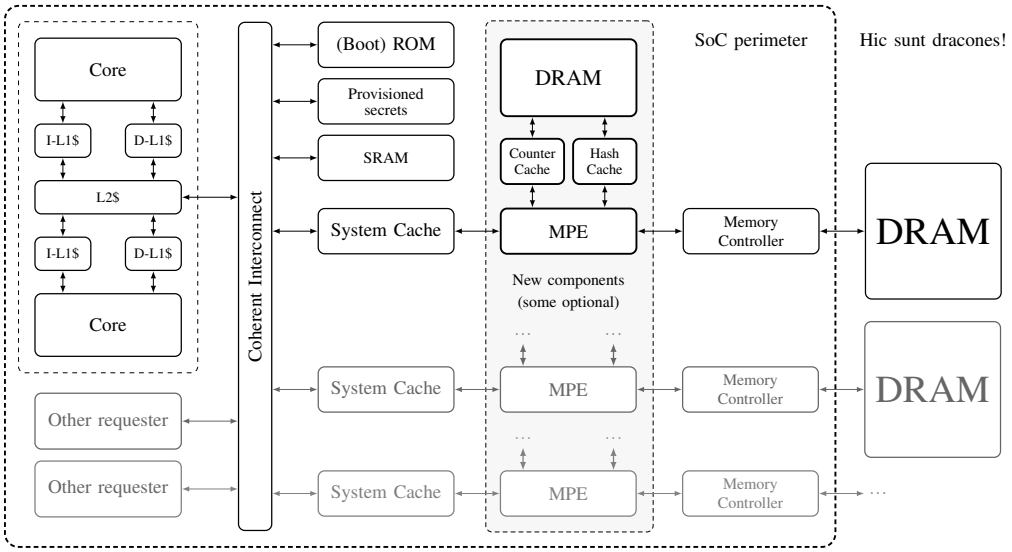

Figure 1: Simplified system level view of a SoC with Memory Protection Engine(s) (DMC is the Dynamic Memory Controller

mode (CTR) encryption, the encryption of successive values of a counter is used to generate a *keystream*, which is then XOR-ed with the plaintext to obtain the ciphertext.

We only consider block ciphers with a block size of 128 bits. The selected block ciphers are the `AES` [23] and the *Tweakable Block Cipher (TBC)* `QARMA` [24]. In addition to the secret key and a text, TBCs accept a *third* input known as a *tweak*. The tweak is used together with the key to select the permutation computed by the cipher. Unlike the key, the tweak may be assumed to be controlled by an adversary. TBCs simplify the design of modes of operation. One of their first applications has been in memory encryption [25] where they perform better than non-tweakable ciphers.

Other candidates (e.g. `PRINCE` [26]) either have similar latencies, are not tweakable, or have a shorter block size.

Authentication primitives. Standard hash functions such as `SHA-2` [27] or `SHA-3` [28] can be turned into MACs but the resulting schemes are very slow and not parallelizable.

Encrypted *Universal Hash Functions (UHFs)* [29] are a better choice. UHFs admit fully parallelizable constructions, such as multi-linear functions of the input computed over a binary Galois field, as used in `SGX` [30]. We note that if a cache is available for UHF-based MACs, the cached values need not be encrypted: The universal hashes are encrypted only when evicted from the cache, and the cached hashes can be verified more efficiently.

Apple's Secure Enclave [16] uses a CMAC [31] to compute integrity tags. This method is not parallelizable and has a high latency but the use case does not need very high throughput. This approach is however unsuitable for general purpose use cases that require high bandwidth and low latency. Instead, we evaluate TBC-based *Parallel MACs (PMACs)* [32]. PMACs are more expensive than encrypted UHFs because the text is first processed by encryption instead of Galois multiplications, but they they can be used for error detection and correction beside integrity, cf. [10], [17], [33].

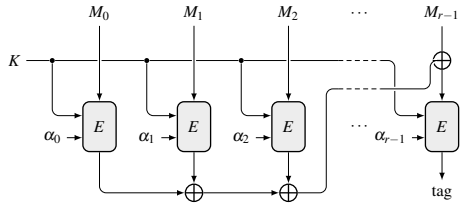

Figure 2: PMAC computed with a TBC for the cases where freshness is not implemented

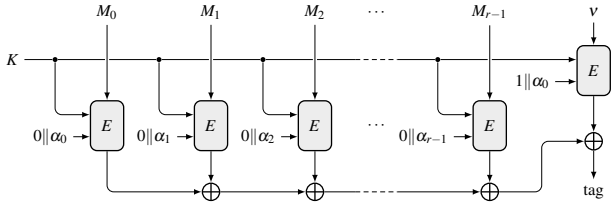

Figure 3: PMAC computed with a TBC for the cases where freshness information is available

The computation of PMACs is depicted in Figs. 2 and 3. 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 [15].

We do not consider encrypted checksums of the plaintext despite being used in Rogaway's *Offset Codebook mode (OCB)* [32]. OCB requires freshness to be provided and all practical systems we are aware of use CTR encryption instead in this case. The main reason for using CTR mode is usually the reduced read latency since the block cipher latency can be hidden behind the off-chip memory access. Due to the one-to-one correspondence between ciphertext and plaintext bits, data encrypted using CTR is malleable. Such systems must therefore include a MAC for integrity.

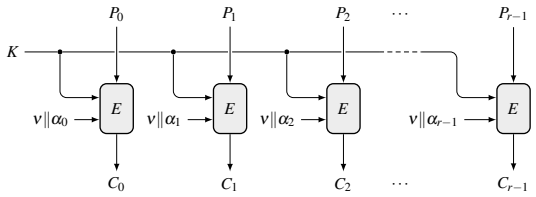

Figure 4: Tweaked ECB mode

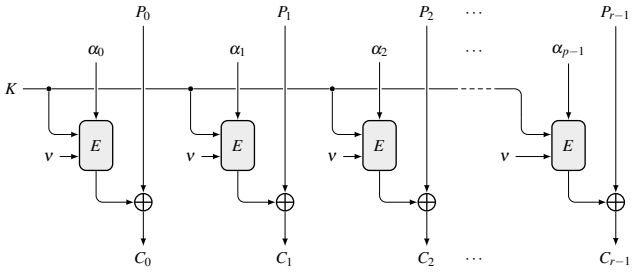

Figure 5: CounTeR in Tweak mode

Modes of operation. For memory encryption, many authenticated encryption modes of operation can be simplified somewhat because the payloads are of fixed length and usually a multiple of the underlying cipher's block length.

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 [32]. 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 [15] which is similar to OCB [32]. In Flat-ΘCB mode, a truncated CTR-encrypted checksum of the plaintext is used to define an *Authenticated Encryption* mode. 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. 4, and an XEX construction is not needed.

In CTR encryption with a TBC, the counter and PA are used as tweak and text respectively (cf. Fig. 5) to generate the keystream. When not using a TBC, the counter and PA are concatenated and then encrypted.

Memory integrity structures. A table of hashes or MACs is sufficient to protect against memory corruption.

Protection against replay attacks requires that the table is either physically protected in an on-chip memory or with a tree structure such as a Merkle Tree [34]. Merkle Tree nodes can be cached [35] to speed up verification.

If counter-based encryption is used, we can protect memory by recursively protecting just the counters as follows: a set of a of counters and an *embedded* MAC form a node called a counter group, which has the same size as a cache line. Each counter group has a children, which can be either cache lines of data or child counter groups. Each counter in a counter group is associated with one child. The embedded MAC is computed on the a counters in the same node and the parent counter. For data (leaf) nodes, the MAC is not embedded. It is instead stored in a separate table. Before a counter group, or a data cache line is evicted, its parent counter is first incremented and the counter group's or cache line's MAC is recomputed. Such a *counter tree* is for instance used in Intel's SGX [3]. Counter trees are in fact just an in-memory reorganization of Hall and Jutla's *Parallelisable Authentication Tree (PAT)* [36].

With the *split counters* optimization [37] a group of a counters is replaced by 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 counter group) 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 value of all of 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 *Read-Modify-Write (RMW)* operations, split counter trees bring a major performance improvement over monolithic counters. As a further optimization, we introduce 3-way split counters (with major, middle, and minor counters) to increase arity and reduce RMWs at the same time.

In Table 1 we compare the memory overheads of various integrity tree implementations. We assume that a MAC can cover 1, 2, or 4 cache lines. When multi-cache line MACs are used, each cache line is still encrypted individually and is associated with its own counter. Hence, 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 [38] 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.

Cryptographic parameters. In the choice of parameters such as key and MAC lengths, the fundamental difference between encryption and authentication is that the encryption parameters must provide long term confidentiality whereas authentication only needs to deter an adversary. A system can monitor unrecoverable integrity violations and take corrective actions (e.g., destroy sensitive state and shut down) if such events are detected. Hence, we recommended that:

- Encryption keys should be at least 128b long. We note that a proper complexity analysis of quantum-computer-

Table 1: Memory Overhead of Various Types of Integrity Trees at 32b and 64b security levels

Type of Tree *cache line size:*	Overhead *64B*	*128B*
Merkle Tree with $a = 4$, resp. 8	33.3 %	16.7 %
Monolithic Counter Tree with embedded MAC, $\ell_c = \ell_h = 56$		
• $\ell_H = 64$; $n = 1$; $a = 8$, 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 with embedded MAC, $\ell_c = \ell_h = 56$		
• $\ell_H = 64$; $n = 1$; $\ell'_c = 6$, 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 %

Legend: *$\mathscr{L}_{CL}$, ℓ_H, ℓ_h, ℓ_c, and ℓ'_c are the bit lengths of a cache line; a data hash or MAC; of a embedded hash value or MAC; of a monolithic or major counter; and a minor counter, respectively. a is the arity of a counter group, i.e. the number of its monolithic or minor counters; and n is the number of cache lines a MAC covers.*

assisted key search against AES-128 proves AES-128 post-quantum secure [39]. Deployed technologies such as Intel's SGX and TDX, and AMD's SEV use AES-128 in modes that need two independent 128-bit keys.

- Encryption block sizes must be at least 128b;
- For Merkle Tree the required hash length is 128b;
- Authentication keys should be at least 128b long;
- Data MACs should be at least 32b long (28b in a MirE configuration); and
- Monolithic counters must be at least 56b long. The minimal aggregated length of a major and a minor counter (or major plus middle plus minor) is also 56b.

Intel's SGX uses 56b MACs and 56b counters. With these parameters, a successful replay attack on its memory of would need both the counter and the MAC to be repeated, with time $2^{56} \times O(2^{56/2}) = O(2^{84})$. Since, however, the purpose of memory integrity is to *deter* these attacks, we note that halving the data MACs memory requirement (and using, say, 28-32b MACs) is sufficient: with 64b worth of counters and 32b data MACs, corruption has likelihood 2^{-32} and replay needs time $2^{64} \times O(2^{32/2}) = O(2^{80})$.

4. Setup of the study

4.1. Scope of the comparisons

Depending on the level, several variants of the involved technologies can be combined. We summarize the variants we compare in the following list. The entries marked with † contain new contributions in this paper, while ⋆ denote variations not hitherto compared to each other.

1) Use of the AES-128 or QARMA-128 ciphers;
2) Size of the MACs (32b or 64b);⋆
3) Counter trees: monolithic, or split;
4) Various choices for the size of counter and hash caches.
5) Use of on-chip memory for hashes and/or counters;†
6) Repurposing of ECC bits for data MAC storage;
7) Synchronous or asynchronous integrity checking;
8) Use of single MACs covering multiple cache lines, with cached incremental hashing;†
9) Arity variations in the counter groups;
10) We consider both 64B and 128B cache line sizes;⋆ and
11) We run the benchmarks as the only running tasks, i.e. on an *unloaded* system, and also under extreme memory bus contention, i.e. on a *loaded* system.⋆

We assume that all algorithms are parallelized wherever possible (i.e., there are sufficient instances of the relevant building blocks to attain the lowest possible latency of the whole scheme). This is not a significant restriction in our study. As we shall see, the fastest L2 and L3 schemes use CTR encryption and have low sensitivity to the latency of the encryption and authentication primitives (for instance, the performances of the variants with AES and QARMA are quite close). Hence, their performance even on pipelined implementations would be similar.

4.2. Technologies used for each level

We list the technologies used to implement the protection levels defined in Section 2.4.

L1 If AES-128 is the chosen encryption primitive, a cache line is encrypted using the XEX construction, with the PA as the tweak. If QARMA-128 is chosen, it is used in Tweaked *Electronic Codebook (ECB)* mode as in Fig. 4, with the PA as tweak.

L2 The same encryption modes are used as for L1. Hashing is done by a *multi-linear* UHF [29] at 32 or 64 bits. The hashes are encrypted block-wise when they are evicted from the hash cache in cache line-sized groups. For the security and reliability implications, cf. Remark 4.1.

L3 First of all, this level provides freshness over L2. The freshness information must be included in the tag computation. A CTR encryption mode is used with both AES and QARMA, except for ELM, which uses FLAT-OCB. Replay protection is provided as well, by including the counters in the tag computation and preventing the adversary from tampering with the counter groups. This is commonly achieved by using an integrity tree, but there are other options, as follows:

LoC One such option is storing the counters in a in-package tamper proof DRAM (an SRAM would be too large) which is MPE private (i.e., invisible to the rest of the system and outside adversarial control). This solution is denoted by LoC which stands for *Leaves-on-Chip*. The names comes from the fact that if we store the leaf nodes on chip, we do not need to compute any other nodes from the original tree, since the leaf nodes are protected by virtue of where they are physically stored.

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 need no further levels of the tree to ensure the integrity of the tree.

MirE *MACs in repurposed ECC bits*. This eliminates the need to reserve normal memory for the MACs. This significantly reduces memory traffic since it eliminates the separate memory transactions used to fetch them. Note that MACs are still accessible to a HW capable adversary. Hence, freshness information, if available, *must* still enter the MAC computation. Following [10], the tag is computed using $\texttt{QARMA}_5$-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.*

MirE raises the question of the performance impact of using ECC memory. Because of expanded traffic and extra processing in the DRAM controller, there are penalties which have been reported as smaller than 0.5% [40]. On servers, schemes that do not repurpose the ECC bits will still be using them for error detection and correction, so memory access time will not vary. In all other cases, we consider the impact of ECC memory to be so small as to not significant change the performance relative to baseline. Hence, we do not evaluate ECC memory as a separate configuration.

Remark 4.1. A new idea we introduce in this paper applies to the cases where UHF-based MACs reside in external RAM. We keep them as *hashes* when cached on-chip which speeds up verification w.r.t. caching MACs. When a dirty block is evicted from the hash cache, we *encrypt the groups of hashes directly* before writing them to memory. For instance, four 32b hashs are encrypted as a single 128b block. Corrupting one hash will corrupt all hashes in the group with high probability, increasing the detectability of any corruption, and with it both security and robustness of the system. If freshness is available, the (middle and) minor counters associated with the hashs in the same block are grouped and concatenated to their common major counter to form a tweak.

4.3. 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 that 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 [6], [7]. `gem5` allows engineers to build software versions of hardware components typically included in computer systems. `gem5` also helps abstract away the interfaces between components, which can thus be combined programmatically and configured at run-time. It includes approximate timing models for several processor cores.

The processor is modeled as an approximated Arm Cortex A72-like core, with a 2GHz frequency and a 1GHz system frequency. The cache hierarchy includes L1-I (48KiB, LRU replacement policy, 3-way set associative, 1 cycle latency) and L1-D (32KiB, 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 16GiB 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 7nm process. Thus, we can re-use the latencies from [24], for instance a latency of 15.76ns for a pipelined implementation of `AES`-128, of 4.8ns for $\texttt{QARMA}_{11}$-128-σ_1 and 2.2ns for $\texttt{QARMA}_5$-64-σ_0. This latency of $\texttt{QARMA}_5$-64-σ_0 is also used in [10].

Our evaluation uses the SPEC 2017 [5] 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 [41]. To enable rapid prototyping and analysis, we use the SimPoint [42] methodology. This technique uses clustering to find a number of representative regions that serve as a proxy for the whole application. After simulating these regions, their results are combined using a set of weights that signify how important a region is to the application as a whole. Instead of sequentially simulating several billions of instructions per benchmark, we simulate 10 SimPoints of 30 million instructions from each benchmark. This both decreases the number of instructions we need to simulate and improves parallelism.

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 of 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, any type of I/O. These aspects are very difficult to emulate. In fact, benchmarking in such a context seems absent from the literature. However, we can observe that (i) The increased memory footprints of the solution with smallest memory overhead (1:128 and less) should not have a noticeable effect on paging; 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. *Therefore, any performance penalty we present here is likely an upper bound to the real-world one.*

5. Benchmarking Plan, Results, and Discussion

All MPE configurations span a vast multi-dimensional space. Exhaustively evaluating them all is clearly infeasible, not to speak of the difficulties of properly presenting the data. For this reason, we have 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 (including adding some new techniques) *where we expect that it has some noticeable impact*. Initial sets of runs start with commonly used parameters (such as tree arity and cache sizes), and the subsequent sets explore what happens when they are varied.

We use shorthands to describe the various configurations:

Level / {additional technologies} / Cipher /
/ cache line length / MAC length .

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

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

"{Intel} TDX" is equivalent to L2/AES/MirE, "{Intel} SGX" to L2/AES/mono, and "{AMD} SME" to L1/AES. LoC always implies counters are split. The shorthand L3/LoC is used to denote the 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*.

General Remarks. Without memory protection, our benchmarks run 14.1%, resp. 9.5% slower on a loaded system with 64B, resp. 128B cache lines than on an unloaded system. Changing the cache line length from 64B to 128B 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.

Unloaded vs. Partially loaded vs. Loaded Systems. All benchmark runs are first run on an *unloaded* system, where the current benchmark is the only running task.

We then 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. This traffic amounts to 8 GiB/s. It is obtained from the measurements reported in Fig. 6 and it corresponds to the point where the latency of the memory subsystem just starts to diverge for a SGX-like L3 MPE covering the entire memory while handling mostly linear traffic. We assume that MAC verification is synchronous because, following the discussion of the benchmark runs in the next section, this will be the most likely implementation. The simulated traffic is a mix of linear and random accesses. We do not add a L3 cache to the system, in order to simulate the extreme situation where the latter has been completely swamped by traffic coming from other requesters or clusters of requesters.

Fig. 6 suggests that we can expect the performance penalty to depend on the load of the system. For this reason, we evaluate performance for an unloaded, partially loaded, and unloaded system. Beyond the loaded configuration we expect a catastrophic deterioration of performance, both with and without the MPE. At this point a cloud provider would typically migrate VMs or start new machines in order to balance load and meet overall performance targets. This would bring also the MPE penalties back under control.

For this reason, we also consider a *partially loaded* system where the injected traffic is 4 GiB/s instead of 8 GiB/s.

A note on short minor counters. In order to make our simulations as realistic as possible, in all split counter runs we initialize the counters to random values. This way the benchmarks will not be advantaged w.r.t. real world software, or other benchmarks that write less often to the same locations, by the fact that they start with zero counters that are not expected to overflow for a longer time. In a real world setting, the user cannot expect that the integrity trees will be initialized in a favorable way. In fact, this configuration choice magnified the performance difference between 3-way and 2-way split counters of the same arity, highlighting the advantage of 3-way split counters.

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

5.1. 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 (e.g., AMD SME), L1/QARMA, L2/AES, L2/AES/MirE (e.g., Intel TDX), L2/QARMA, L2/QARMA/MirE, and ELM with both monolithic and split counters, SGX, L3/QARMA/split – 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 32b and 64b MACs in selected cases – possibly shortened to 28b, resp. 56b, to fit them in the data structures – as for instance SGX uses 56b MACs while TDX uses 28b MACs, so choosing this size is a common question that implementors face.

For SGX, hash encryption is CTR as described by Intel [3]. We use this method for the SGX split counters variant (L3/AES/split) as well, and in any L3 scheme where

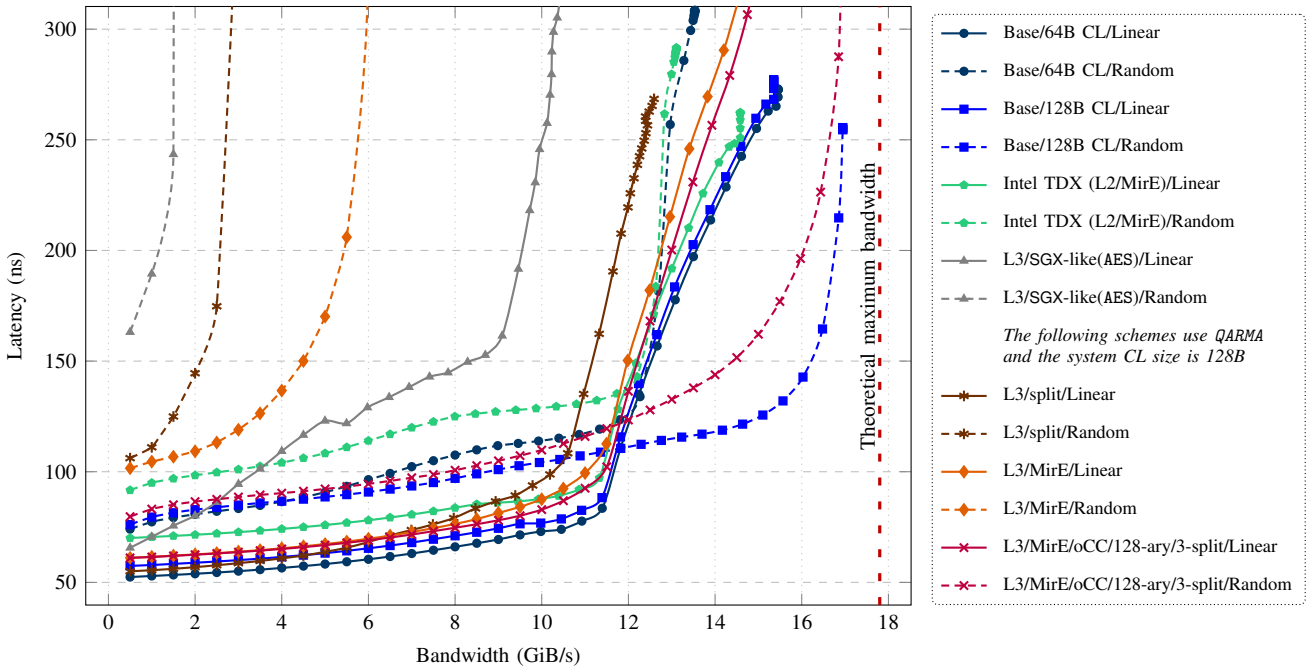

Figure 6: Bandwidth/latency plot with various MPEs and without, for linear or random synthetic traffic

the published architecture prescribes its use. In all other cases, data MACs are replaced by 32b long hashes which are directly encrypted in groups of four upon eviction.

The ELM method follows [15] except when QARMA is used, in which case the *XOR and Encrypt (XE)* constructions are replaced by simply feeding nonces and separation fields as the tweak to QARMA, as well as using $\texttt{QARMA}_5$-64-σ_0 to generate the keystream values to encrypt the tags.

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

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

A first look at the results in Fig. 7 shows that:

- In the basic variants, performance penalties increases with the protection levels.
- The performance of L1 and L2 schemes improves if we replace the AES with QARMA because of the latter's lower latency. A minor improvement occurs even for L3 because the CTR keystream generation, while it can be performed in parallel with a memory fetch, still affects write latency to the point that it bears effect.
- Split counter trees are superior to monolithic trees in memory overhead (see Table 1) and performance.
- A small hash cache has a minor effect on performance. This is perhaps the first third-party explanation of the rationale for not including one in SGX.
- ELM has a higher performance penalty than SGX, having the encryption primitive on the critical path.
- As expected, using 64b MACs results in slightly worse performance than using 32b MACs.

For the remainder of the evaluation, we assume that MACs are 32 bits long and directly encrypted in groups of four except with SGX, MirE, or otherwise explicitly indicated.

5.2. Set 2: Impact of MPE Cache Size

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 4KiB, 16KiB, and 64KiB; and counter cache sizes are 16KiB, 64KiB, 256KiB, and 1MiB. We selected these sizes since we expect this to be a reasonable range when implemented in SRAM. The presented results use QARMA for encryption as the AES results display an identical pattern.

We confirm the expected significant performance gains with larger MPE caches, the counter cache having a higher effect than the hash cache. 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). The counter cache would intuitively have an access pattern that is similar to the hash cache. However, the reach of the counter cache is bigger since counters tend to be smaller (this is especially true 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.

The improvement gets more significant as the load of the system increases (see Fig. 9).

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.

5.3. Set 3: Impact of the Cache Line Length

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

It is assumed that counter group and cache line sizes are equal. The results of Set 3 are combined with those of Set 4 in Fig. 10.

Our results show the relative impact of memory protection is comparable across systems with 64 B cache lines and 128 B cache lines and depends on factors such as load and integrity verification strategy. A large benefit of using wider cache lines is that it effectively cuts the storage overhead of MACs in half and enables more aggressive metadata packing in counter groups.

Since we already know that our reference system without an MPE performs 1.4% to 5.5% better with 128 B cache lines, we expect that that using to 128B cache lines, at least for the system cache, is generally beneficial in a system with a MPEs. It is worth noting that having shorter cache lines in coherent caches closer to the CPU is still possible and may be beneficial for multi-threaded workloads but studying this is outside of the scope of this paper.

5.4. 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.

L1 is out of scope as it does not offer integrity, so we consider only L2 and L3. The results are displayed in Fig. 10.

Whereas in the unloaded case asynchronous MAC verification does not significantly improve performance, in the loaded case the speedup is non-negligible. As the memory bus approaches saturation, decoupling decryption and MAC verification significantly reduces the performance penalty by letting the CPU use data before it has been verified.

The use of 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 potentially extract sensitive information. Mitigating this issue introduces significant complexity which would be detrimental the integrity of the system. In the following, we assume synchronous MAC verification since this is the trade-off we expect in future implementations.

5.5. Set 5: Use of on-Chip Memory for L3

Going beyond caching as explored in Set 2, we explore how a secure MPE-private on-chip memory affects the performance the MPE.

As MACs/hashes have a larger memory overhead than counter groups, we do not expect an implementation with on-chip hashes and off-chip counter groups. For brevity, we do not consider the latter case.

The results in Fig. 11 confirm that relieving the contention on the memory bus between data and metadata reduces performance penalties in the unloaded and partially loaded cases. A somewhat surprising result is the LoC configuration where counter tree leaves are store on-chip is only marginally better than not using a dedicated on-chip memory. This can be explained by the effectiveness of the counter cache. While leaf nodes have poor temporal locality, the temporal locality increases 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. As a consequence, the behavior similar to LoC since the cache hit terminates the tree walk.

When storing 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 5.6, it can be approximated by repurposing ECC bits for MAC storage.

There is a small performance difference between AES and QARMA in an unloaded system, that substantially decreases when the system is under load. The smaller difference in a loaded system can be explained by the fact that the increased memory latency effectively hides the cipher latency which can occur in parallel with the data and metadata accesses.

5.6. Set 6: Impact of Repurposing ECC Bits, 3-way Split Counters, and Large Counter Caches

The deployment of Intel TDX MKTMEi [4] and [9] 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 their corresponding data.

We focus on L3 with and without MirE, since L2/MIRE schemes are reported in Fig. 7. We expect that MirE implementations are optimized for performance and to reduce storage overhead. For that reason, we focus on 128B cache lines which enable much more efficient counter group packing. With MirE, a hash cache is not needed since MACs and data are fetched in the same memory transaction. In this case, we compute MACs using the PMACs algorithm.

In addition to classic 2-way split counters, *we introduce 3-way split high-arity 128B counter groups*. The purpose of this optimization is to pack more minor counters into the same counter group while keeping the amount of RMW operations under control. The minor counters in the 256-ary counter groups cannot be longer than 3 b, and the major counter does not need to be larger than 64 b, which means that we can fit 32×6 b middle counters. To quantify the impact of this optimization, we evaluate the configuration with and without middle counters. The 3-way split counter groups configurations we evaluate are:

- 128B cache lines and counter groups with: 128 × 7b minor, 8 × 8b middle, and 1 × 64b major counters; This results in a memory overhead of 1:128.
- 128B cache lines and counter groups with: 256 × 3b minor, 32 × 6b middle, and 1 × 64b major counters; This results in a memory overhead of 1:256.

In [9], [43] "delta encoded" split counters with rebasing are used together with methods to accommodate a limited number of larger minor counters in a counter group to reduce the amount of RMWs. We skip these optimisations since our 3-way split counter groups (cf. also Section 5.9) perform better, by nearly eliminating any RMW overhead.

The results (Fig. 12) show that combining LoC and MirE provides the highest protection level at very low performance overhead. In fact, this is the only combination of techniques that can yield nearly negligible performance penalties on a loaded or partially loaded system. Middle counters are instrumental in getting the best performance out of the high-arity counter groups, which would otherwise incur in very large RMWs overheads. The resulting schemes perform even better than L1 direct encryption. This can be explained by L1 schemes having the cipher latency on the critical path to external memory, while L3 and variants hide the cipher latency behind the the off-chip memory latency.

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. 12 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.

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 slow, but dense, DRAM can be used for these caches.

5.7. 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 the storage overhead, we focus this investigation on using 128 B cache lines. This configuration already reduce metadata storage requirements by a factor of two compared to 64 B cache lines. We test both L2 and L3 configurations, L3/LoC, and L3/BoC, with a MAC covering 1, 2, or 4 cache lines. These runs are reported only with QARMA-128 as the encryption cipher, since the performance differences are caused only by the increased memory traffic, and we can therefore expect configurations with AES to follow the same pattern. Multiple-cache line MACs effectively reduce storage overheads, but at a significant performance cost, as shows in Fig. 8.

An alternative way to store MACs would be to use plaintext compression to fit a MAC inside together with the data in the same block. Indeed, the performance of an incremental hashing scheme may be improved somewhat [44]. However, this approach comes with significant drawbacks. First, we need a fallback mechanism to store MACs when the cache line cannot be sufficiently compressed to fit a MAC in the same block. Second, an attacker capable of monitoring memory transactions would be able to infer properties of the data just by observing its compressibility [45], [46]. The latter completely defeats the purpose of a MPE in the first place. Thus, we have decided not to consider it.

5.8. Set 8: Detailed Breakdown of the Performance in 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 and L1/QARMA, with 64B cache lines;
- Intel TDX/64B CLs (i.e. L2/AES/MirE);
- L2/QARMA/64B CLs, with off-chip 64b MACs and MirE;
- Intel SGX (i.e. L3/AES/56b MACs);
- L3/QARMA/LoC/MirE, with 128- and 256-ary 3-way split counter groups.

The performance of the individual SPEC2017 benchmarks (cf. Figs. 13 to 18) shows a few expected results, namely that some applications such as omnetpp, mcf, and bwaves suffer significantly more than average under most MPE configurations. Increasing integrity tree arity by means of split counters is key for an initial reduction of the penalties, but it is only with 3-way LoC and MirE that L3 penalties can be pushed to be smaller than 5% for most benchmarks. The only difference between unloaded and loaded systems is the degree of amplification of the performance penalties.

5.9. Set 9: Impact of RMW Operations

All split counter methods need, as said before, to perform some batches of RMW operations to re-encrypt data or re-compute some embedded MACs whenever a major counter is incremented. These are expensive operations and we need to understand their impact on performance.

Hence, we compare the performance of an MPE with a hypothetical one 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, simulating instead their timing delays.

This gives an upper bound on the actual time spent performing RMW operations. We select the L3 schemes with high-arity split counters. For these schemes we report the

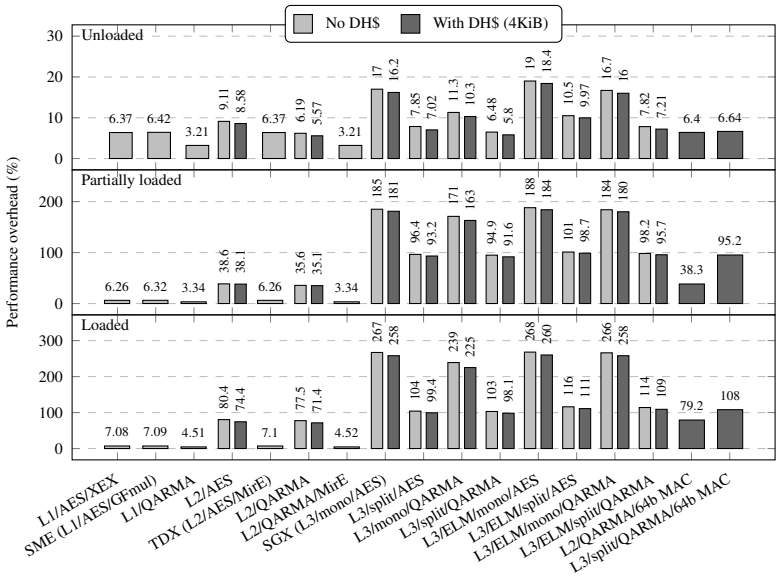

Figure 7: Set 1: Comparison of base levels and state of the art

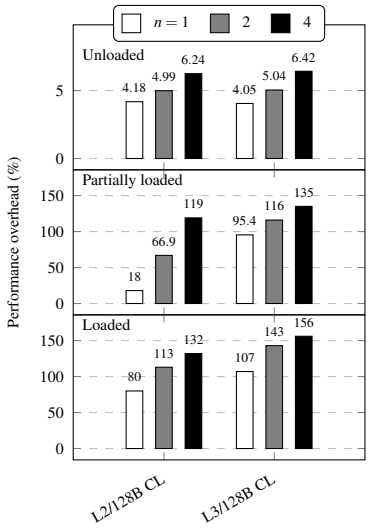

Figure 8: Set 7: Impact of incremental MACs

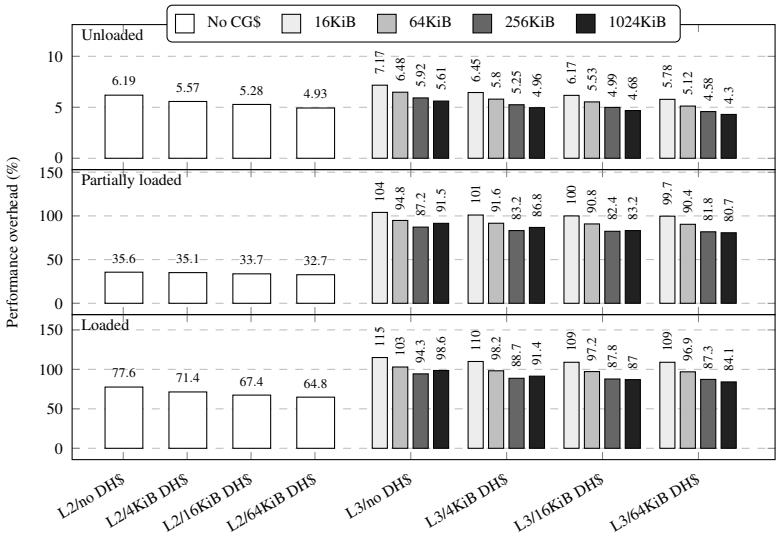

Figure 9: Set 2: Impact of MPE cache sizes;
the memory encryption cipher is QARMA-128; cache lines are 64B

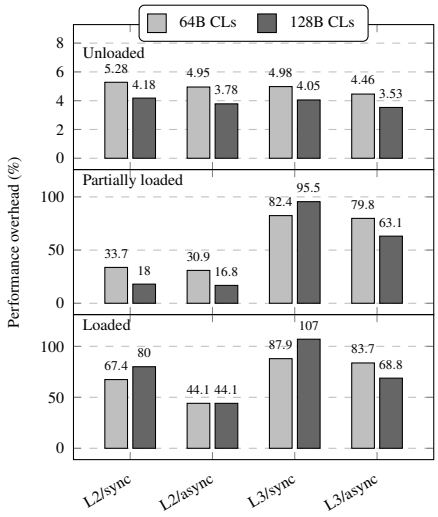

Figure 10: Set 3 and **Set 4**: Impact of cache line size and asynchronous MAC verification

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 results are shown in Figs. 17 and 18. 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 counter groups) carries a significant performance penalty, but the use of middle counters brings the performance close to the ideal case where RMWs are "free".

Note that the performance penalties and the proportion of time spent doing RMWs increase with the load. This suggests that further research to RMWs may benefit loaded systems. However, even in this case the penalties with 256-ary, 3-way split counter groups are smaller than with a direct encryption L2/AES/64B CLs/MirE scheme as in TDX.

5.10. A remark on area and power

The area of the MPE mostly comprises cryptographic circuits, caches, and any internal DRAM for counter storage (if present). The control circuitry, and possibly Galois multipliers, are small in comparison to the block ciphers implementations.

There is extreme variability in the parameters. For instance, in order to squeeze the maximum performance from schemes based on direct encryption, such as L1 and L2 schemes, the implementer may use several encryption blocks in parallel for encryption and for the integrity PMAC.

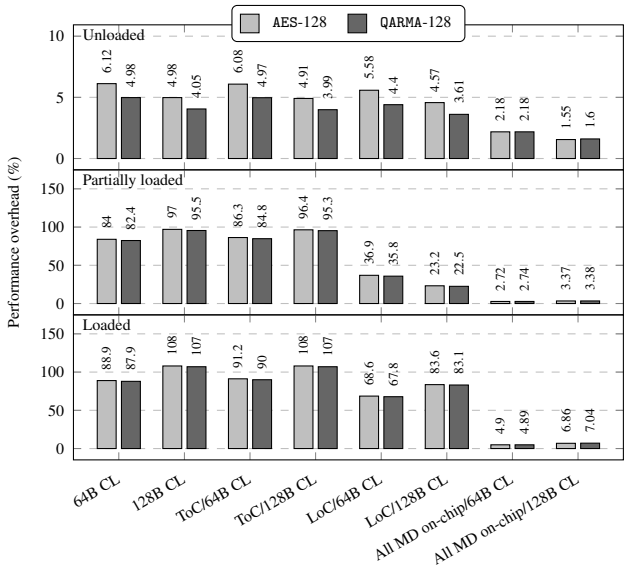

Figure 11: Set 5: L3; Impact of storing metadata on-chip

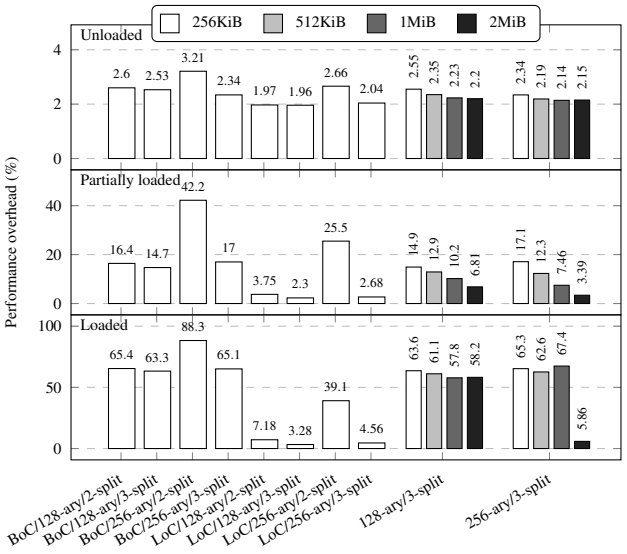

Figure 12: Set 6: Impact of repurposing ECC bits for MACs and of large counter caches on L3/MirE schemes

Sacrificing some latency, pipelined implementations can be used to save area. As described in [24], the implementations of AES and of QARMA may vary a lot. For a single MPE we estimate between ≈50KGE (gate equivalents) for a pipelined encryption circuit based on QARMA, optimized for area, and ≈ 800K GE for 8 parallel instances if optimized for latency in order to encrypt a whole 64 B cache line at the same time. AES implementations are much bigger with optimized implementations exceeding 17K GE *per round* [47]. A full latency-optimized implementation can be ≈ 170KGE, and 8 such blocks in parallel would use an area of ≈ 1.3MGE. A pipelined QARMA circuit and a fully parallelized AES circuit would furthermore have comparable total latency.

Integrity tag computation and verification can re-use the same blocks used for encryption, or much smaller ones based on $\texttt{QARMA}_5$-64-σ_0, as in [10].

We recall from Section 5.6 that the MPE caches can be built from DRAM, with the area roughly being one transistor per bit. A 4 KiB cache is about 16 KGE, and a 256 KiB cache is roughly 1 MGE. We can expect that if resources are balanced, the caches will be between 1/3 and 1 times the area of the encryption blocks.

Even one large MPE per memory channel represents a minor, but not negligible, amount of area for a modern SoC, that can include a few to several billion transistors. Architects and implementers need to carefully consider the various trade-offs. The cost of DRAM memory included in the package or module to store entire counter tables is minor with respect to the total memory of the whole system, but it cannot be ignored, especially since a tamper proof or detecting multi-chiplet design bears its own additional costs.

Besides these considerations, it is infeasible to provide area estimates for all configurations. In general, the energy consumption of an MPE seems to be a minor contribution to the total power envelope. Still, designs like QARMA help bring the latter further down, as the most energy consuming components in the MPE are the encryption primitives: a single pipelined implementation of QARMA-128 can process eight blocks in the same time as eight parallel AES-128 blocks, and this can make a significant difference in CTR based systems which are less sensitive to cipher latency.

6. 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. We unified the treatment of different protection levels according to models of the targeted adversaries.

The large number of possible configurations results in a vast set of independent choices, corresponding to different prices in term of performance penalty, memory overhead, and hardware cost. The lack of an absolute metric to combine these three costs into one rating makes it challenging to provide recommendations for each use case. Therefore, the extensive set of benchmarking runs we present should be used as a guidance for further investigations. This said, we can provide some rough guidance.

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 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 allows

us to makes an argument for CTR encryption with high arity counter groups in on-chip DRAM. The additional cost for counter group storage would be *relatively* minor (1:128 or 1:256 of the external memory). 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 would likely also less expensive than basing the protection of local memory on the `CXL.memory` *Integrity and Data Encryption (IDE)*.

If there is no budget for a large on-chip counter memory, a big counter cache may still provide almost the same benefit. However, it can be argued that such a budget should rather be spent on the system caches, which benefits the *whole* system and reduces the traffic routed through the MPE. This also suggests the idea of dynamically partitioning a common cache between system and MPE. Dynamically reconfiguring it would be straight-forward to implement in hardware, but it should rely on an analysis of the traffic and of the impact of the partitioning that goes beyond the scope of this paper.

We observe that placing 64 GiB or more of DRAM in a module close to the main SoC is feasible for client devices today. Hence, one can could imagine making such memories tamper proof and using them as a general purpose cache in a server system equipped with up to several terabytes of memory – instead of keeping all counter groups on chip. If this approach is 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.

We finally observe that data structures and the organization of integrity trees play a first-order concern when considering overall *performance*. Cryptographic primitives only make a small difference for performance but significantly affect area and power where light-weight block ciphers significantly outperform classic block ciphers.

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

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] 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

[5] 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

[6] 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

[7] 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

[8] M. Seaborn and T. Dullien, "Exploiting the DRAM RowHammer bug to gain kernel privileges," Talk at Black Hat 2015, 2016, https://www.blackhat.com/us-15/briefings.html.

[9] 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

[10] 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.

[11] 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

[12] A. Barenghi, L. Breveglieri, I. Koren, and D. Naccache, "Fault Injection Attacks on Cryptographic Devices: Theory, Practice, and Countermeasures," *Proc. IEEE*, vol. 100, no. 11, pp. 3056–3076, 2012, doi:10.1109/JPROC.2012.2188769

[13] E. Blass and W. Robertson, "TRESOR-HUNT: Attacking CPU-Bound Encryption," in *28th Annual Computer Security Applications Conference, ACSAC 2012*, R. H. Zakon, Ed. ACM, 2012, pp. 71–78, doi:10.1145/2420950.2420961

[14] L. Zussa, J.-M. Dutertre, J. Clédiere, B. Robisson, and A. Tria, "Investigation of timing constraints violation as a fault injection means," in *27th Conference on Design of Circuits and Integrated Systems (DCIS), Avignon, France*, November 2012, pp. 1–6.

[15] 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

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

[17] 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

[18] 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

[19] 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

[20] B. Shakya, N. Asadizanjani, D. Forte, and M. M. Tehranipoor, "Chip Editor: Leveraging Circuit Edit for Logic Obfuscation and Trusted Fabrication," in *Proceedings of ICCAD 2016*, F. Liu, Ed. ACM, 2016, doi:10.1145/2966986.2967014

[21] S. Herschbein, S. Tan, R. Livengood, and M. Wong, "Focused Ion Beam (FIB) for Chip Circuit Edit and Fault Isolation," in *ISTFA 2021: Tutorial Presentations*, ser. International Symposium for Testing and Failure Analysis, November 2021, pp. h1–h113, doi:10.31399/asm.cp.istfa2021tph1

[22] S. Aga and S. Narayanasamy, "InvisiMem: Smart Memory Defenses for Memory Bus Side Channel," in *Proceedings of the 44th Annual International Symposium on Computer Architecture, ISCA 2017*. ACM, 2017, pp. 94–106, doi:10.1145/3079856.3080232

[23] 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

[24] 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

[25] 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

[26] 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

[28] ——, "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

[29] 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

[30] 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

[31] 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

[32] 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

[33] 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

[34] 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

[27] 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

[35] 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

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

[37] 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

[38] 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 *Cryptographic Hardware and Embedded Systems - CHES 2007, 9th International Workshop, Proceedings*, ser. Lecture Notes in Computer Science, P. Paillier and I. Verbauwhede, Eds., vol. 4727. Springer, 2007, pp. 289–302, doi:10.1007/978-3-540-74735-2_20

[39] S. Jaques, M. Naehrig, M. Roetteler, and F. Virdia, "Implementing Grover Oracles for Quantum Key Search on AES and LowMC," in *Advances in Cryptology - EUROCRYPT 2020 - 39th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Proceedings, 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

[40] 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/

[41] 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.

[42] 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

[43] 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

[44] 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

[45] 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

[46] 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

[47] 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

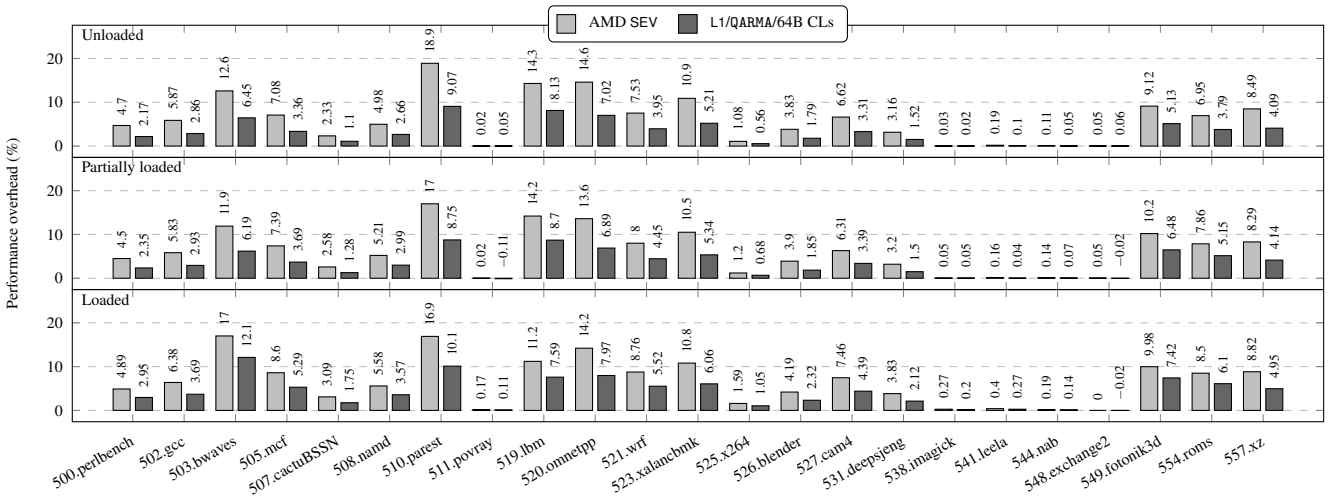

Figure 13: Set 8: AMD SEV (i.e. L1/AES/64B CLs) vs. L1/QARMA/64B CLs

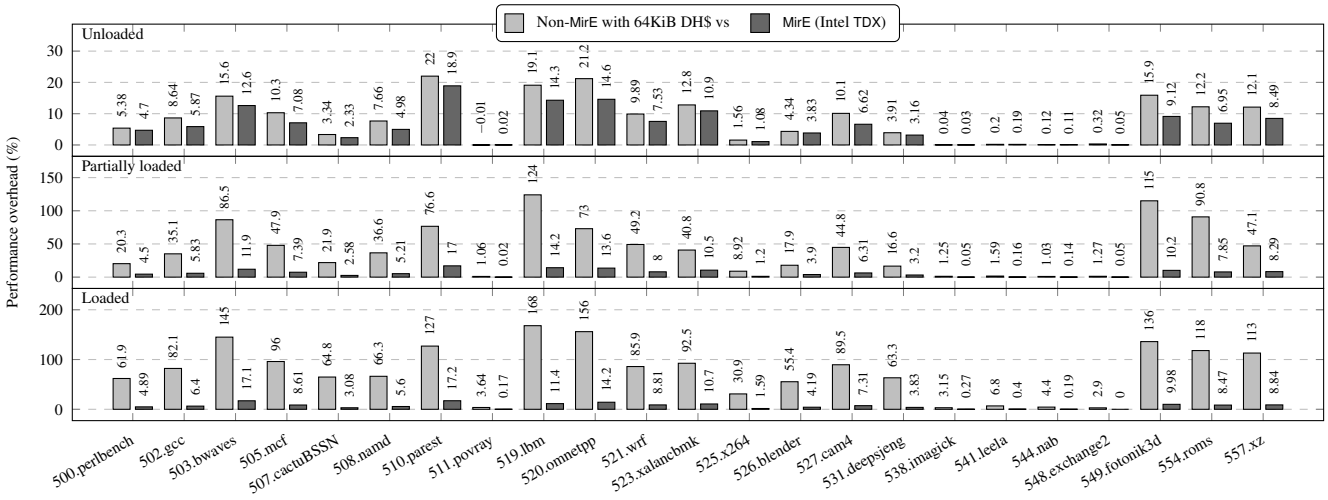

Figure 14: Set 8: L2/AES with and without MirE (64B cache lines)

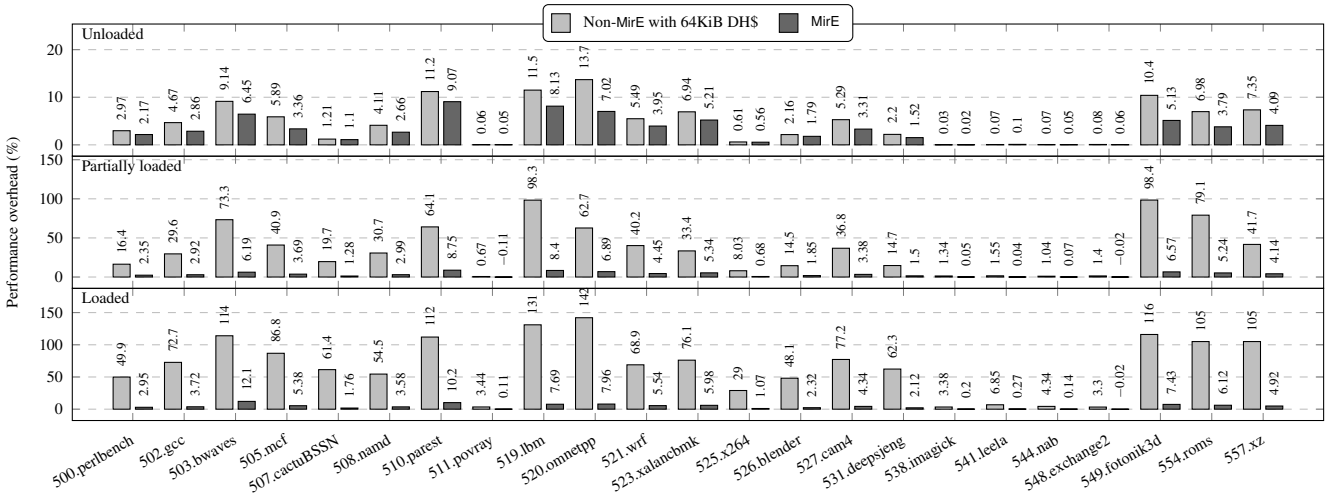

Figure 15: Set 8: L2/QARMA with and without MirE (64B cache lines)

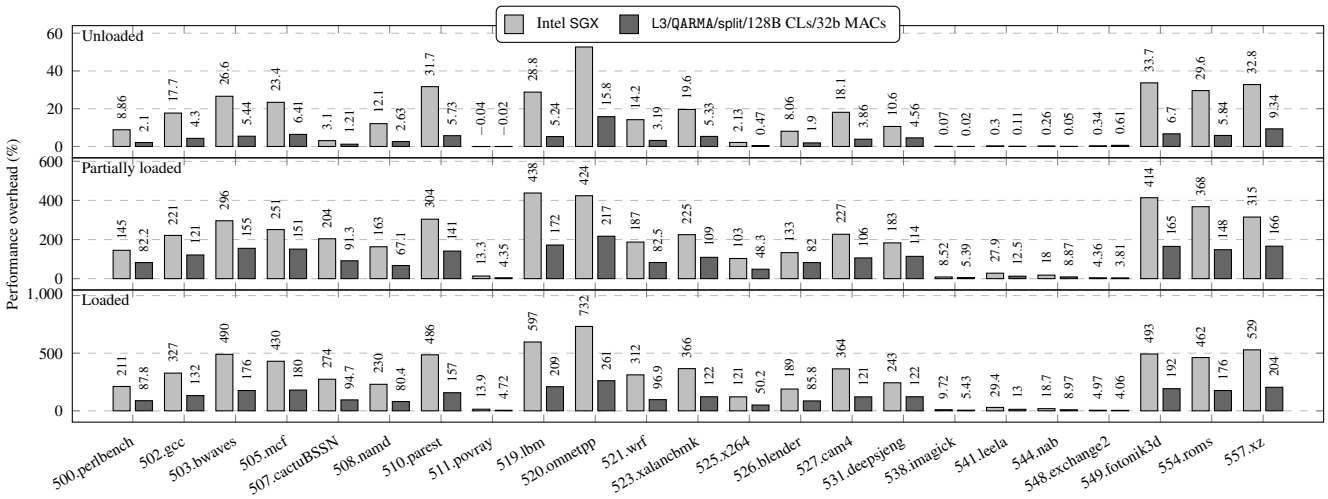

Figure 16: Set 8: Intel SGX/64B CLs (L3/AES/56b MACs/64B cache lines) vs. L3/QARMA/split/128B cache lines/32b MACs

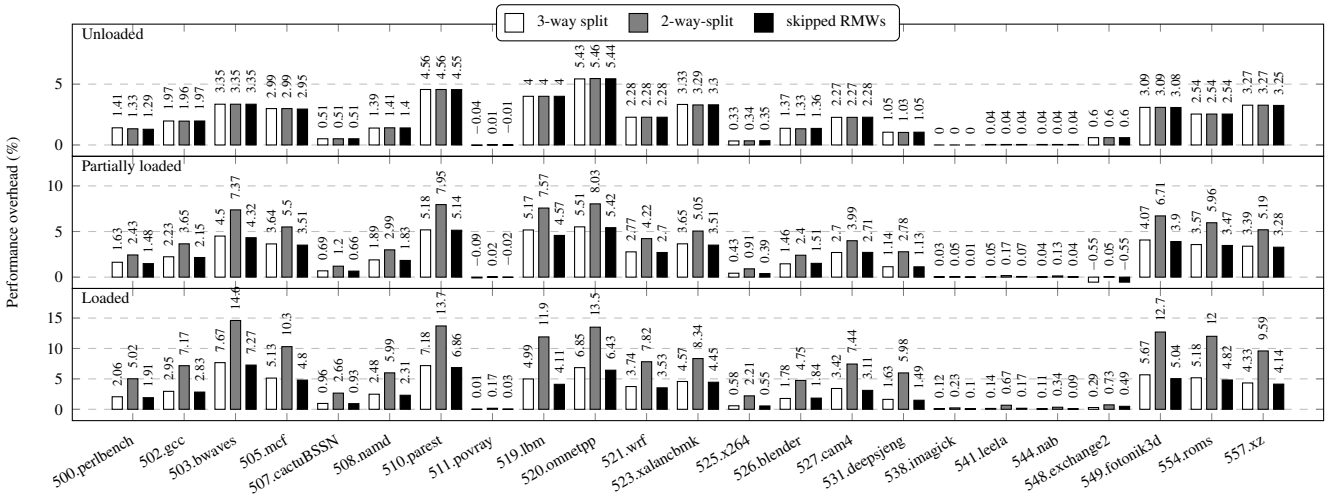

Figure 17: Set 8 and **Set 9**: L3/MirE/QARMA/LoC/128-ary – runs with 3-way and 2-way split counters, and with no RMWs

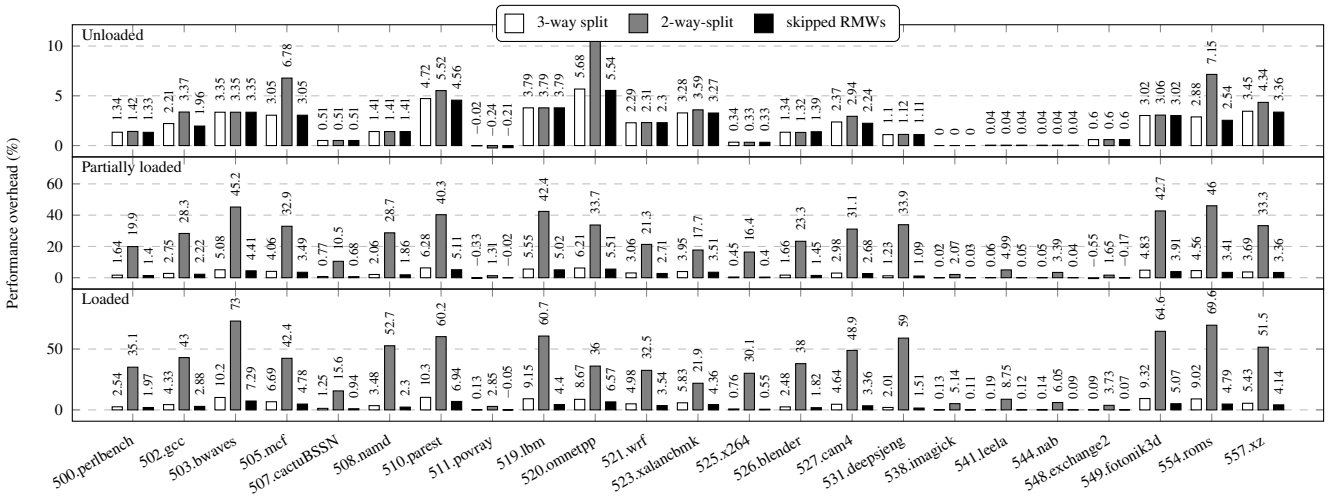

Figure 18: Set 8 and **Set 9**: L3/MirE/QARMA/LoC/256-ary – runs with 3-way and 2-way split counters, and with no RMWs