Crystalor: Recoverable Memory Encryption Mechanism with Optimized Metadata Structure

Rei Ueno*
Tohoku University
Sendai, Japan
rei.ueno.a8@tohoku.ac.jp

Hiromichi Haneda
Tohoku University
Sendai, Japan
hiromichi.haneda.r5@dc.tohoku.ac.jp

Naofumi Homma
Tohoku University
Sendai, Japan
naofumi.homma.c8@tohoku.ac.jp

Akiko Inoue
NEC Corporation
Kawasaki, Japan
a_inoue@nec.com

Kazuhiko Minematsu
NEC Corporation
Kawasaki, Japan
k-minematsu@nec.com

ABSTRACT

This study presents an efficient recoverable memory encryption mechanism, named Crystalor. Existing memory encryption mechanisms, such as Intel SGX integrity tree, offer neither crash consistency nor recoverability, which results in attack surfaces and causes a non-trivial limitation of practical availability. Although the crash consistency of encrypted memory has been studied in the research field of microarchitecture, existing mechanisms lack formal security analysis and cannot incorporate with metadata optimization mechanisms, which are essential to achieve a practical performance. Crystalor efficiently realizes provably-secure recoverable memory encryption with metadata optimization. To establish Crystalor with provable security and practical performance, we develop a dedicated universal hash function PXOR-Hash and a microarchitecture equipped with PXOR-Hash. Crystalor incurs almost no latency overhead under the nominal operations for the recoverability, while it has a simple construction in such a way as to be compatible with existing microarchitectures. We evaluate its practical performance through both algorithmic analyses and system-level simulation in comparison with the state-of-the-art ones, such as SCUE. Crystalor requires 29–62% fewer clock cycles per memory read/write operation than SCUE for protecting a 4 TB memory. In addition, Crystalor and SCUE require 312 GB and 554 GB memory overheads for metadata, respectively, which indicates that Crystalor achieves a memory overhead reduction of 44%. The results of the system-level simulation using the gem5 simulator indicate that Crystalor achieves a reduction of up to 11.5% in the workload execution time compared to SCUE. Moreover, Crystalor achieves a higher availability and memory recovery several thousand times faster than SCUE, as Crystalor offers *lazy recovery*.

KEYWORDS

Memory encryption; Crash consistency; Crash window problem; Parallelizable authentication tree; Secure computer architecture

*The present affiliation of Rei Ueno is Kyoto University, Japan, and his present email address is ueno.rei.2e@kyoto-u.ac.jp.

1 INTRODUCTION

1.1 Background

Memory encryption is an essential security primitive for modern computers. Owing to extensive attacks on main memory, including the cold boot attack, Rowhammer, and RAMbleed [26, 35, 38], which pose real threats, memory encryption is becoming increasingly relevant in a wide range of computers. Trusted execution environment (TEE) mechanisms, such as Intel software guard extension (SGX) and ARM secure encrypted virtualization (SEV), support main memory (DRAM) encryption for realizing resource isolation, remote attestation, *etc.* Moreover, non-volatile and persistent memories (NVMs), such as MRAM [17], have been deployed and have attracted substantial attention. In several recent systems, NVMs have been deployed as main memory in addition to or instead of DRAM for higher performance, lower power consumption, and larger capacity, such as NVDIMM and Intel Optane Persistent Memory [2]. Attacks on main memory are more serious and realistic for NVM than DRAM owing to the non-volatility. With the advances in large-scale (persistent) memories, a high demand exists for the development of efficient memory encryption mechanism for the security of modern and future computers.

Conventional security notions. Memory encryption realizes the confidentiality and/or authenticity (integrity) of main memory data based on symmetric cryptography, including encryption, message authentication code (MAC), and authenticated encryption (AE) [23]. In [5], Avanzi *et al.* classified and formalized memory protection into three levels:

- L1: confidentiality only,
- L2: confidentiality and integrity,
- L3: confidentiality, integrity, and replay protection.

For example, AMD–SEV employs AES–XEX for memory encryption [1], corresponding to L1 security. As well, the Optane module is equipped with a 256-bit AES–XTS engine for data confidentiality without authenticity. However, authenticity is currently considered to be essential for protecting main memory, as some practical attacks have been reported that exploit the lack of (full) authenticity of (deterministically) encrypted memory [11, 41, 42, 50, 51, 70]. Here, replay attack, which involves a data move in the time domain (*i.e.*, copy-and-paste of data from past timing), is sophisticated manipulation/forgery. Replay attack cannot be prevented by the simple use of

encryption and MAC (*i.e.*, L2 security), as the manipulated data had been originally valid. Hence, many studies have been devoted to the realization of L3 security, which considers the strongest adversary counteracted by memory encryption (*i.e.*, an active but non-invasive hardware attacker $\mathcal{A}^{\mathrm{HW}}_{\mathrm{active}}$ [5]), from both cryptographic and microarchitectural perspectives [14, 15, 21, 23, 24, 32]. For example, Intel SGX memory encryption, based on the SGX integrity tree (SIT), achieves L3 security [14].

Memory authentication trees. For an L3-secure memory encryption, Merkle tree [47, 48] (in combination with an encryption of leaf nodes) and parallelizable authentication tree (PAT) [27] have been developed for protection against extensive attacks with a realistic computational latency. SIT is a popular PAT instance [14, 23, 24]. Each leaf node consists of encrypted data, a counter value (*i.e.*, nonce), and a verification tag, whereas each intermediate/root node consists of a nonce and a tag to verify its child node as security metadata. When reading (resp. storing) data in a leaf node, all nodes on the path from the leaf to root nodes are verified (resp. updated). Such a tree structure enables real-time processing of verification and update because its path verification is far faster than the MAC computation for the entire memory. In addition, the root node, which consists of only several hundred bits, is stored on-chip, as the attacker supposedly cannot manipulate on-chip data. Thus, the root node acts as a root of trust to preclude replay attacks.

Crash consistency. The aforementioned L3 security has been considered as a sufficiently strong goal in the plain model; however, *it is insufficient for the practice of memory encryption.* For sound availability, memory data should guarantee *crash consistency*: the data are not broken after a crash (*e.g.*, sudden power-off/blackout) occurs [46]. For this purpose, memory controller in CPU would equip a write pending queue (WPQ) as an asynchronous DRAM refresh (ADR) domain to persist data to be stored in main/persistent memory [62]. For encrypted memory based on Merkle tree or PAT, however, it is mandatory to guarantee the consistency of all (meta)data, including intermediate nodes. If a tag verification (on intermediate nodes) fails, the related data (leaf node) becomes unavailable because it may have been manipulated. However, it is non-trivial to guarantee crash consistency of the entire tree because a path is to be updated serially in practice (although the computation may be parallelized [27]). This is called *crash window* problem [30, 46]: there must exist a moment when a path is inconsistent (*i.e.*, some nodes are updated while others are not yet). Thus, it is challenging to efficiently guarantee consistency against crashes whenever the system operates.

PAT vs. Merkle tree. In PAT, the MAC tags on a path can be updated in parallel. PAT enables algorithmically lower latency than Merkle tree because path update of Merkle tree is not parallelizable, while crash consistency may be easier for Merkle tree than PAT [19, 78]. In this study, we develop an efficient recoverable mechanism of PAT, which is promising to improve memory encryption.

Attack on availability. The security (*i.e.*, confidentiality and integrity) of PAT has been proven [27, 32] in the adversarial model in which the attacker (intentionally) causes crashes and queries inconsistent trees. However, the crash consistency suggests an additional security issue with respect to availability; that is, the existing crash consistency mechanisms (see Section 5) can neither correct the errors (not owing to crash) nor recover the memory with integrity. For example, DRAM bits sometimes flip accidentally due to soft-error effects [61][1]. In addition, a malicious attacker may mount a type of denial-of-service (DoS) attack by injecting error(s) into the tree nodes *via*, for example, Rowhammer [35] and physical side-channels, which makes the related data unavailable. Thus, the L3 memory encryption significantly degrades availability against malicious attackers and faults, and it is quite important to develop how to maintain the availability of encrypted memory.

Additional security notion and our motivation. Given the above observations, we introduce yet another security level for memory encryption beyond [5]:

- L4: confidentiality, integrity, replay protection, and recoverability.

An encrypted memory is recoverable if the memory data can be securely recovered and against manipulation and error (except for leaf nodes) in addition to crash. In particular, the L4 security considers the adversary as in the L3 security (*i.e.*, $\mathcal{A}^{\mathrm{HW}}_{\mathrm{active}}$), but achieves a higher availability and resilience against faults and DoS attacks than L3 secure memory encryption. The goal of this study is to develop an efficient and recoverable memory encryption mechanism that achieves L4 security for the practice of memory encryption. Here, an L4-secure memory encryption should realize recovery from any manipulation/errors of intermediate nodes and detect those of leaf nodes, as the leaf node contains data unrelated to the tree structure. Leaf node recoverability should be maintained by, for example, an error-correcting code (ECC) and *tagged/colored memory* [18, 25, 39, 52, 60]. Note that the crash consistency is *not* equivalent to recoverability, as a crash-consistent (but not recoverable) memory encryption may not support a recovery but only supports detection in cases of manipulation and errors.

1.2 State-of-the-art and its limitations

Crash consistency has been studied in the research field of microarchitecture (see Section 5), but little formal and provable security analysis has been conducted. In HPCA 2023, Huang and Hua presented a state-of-the-art crash recovery mechanism for L3-secure PAT, named *shortcut update (SCUE)* [30]. SCUE exploits a simple property of naïve PAT metadata; that is, a parent node counter is always equal to the sum of the child node counters. SCUE requires little overhead when applied to a PAT-based secure memory, and would be currently the most efficient for L4 memory encryption.

However, SCUE has two major limitations in terms of provable security and compatibility with existing optimization mechanisms. SCUE is based on a property of counters in a naïve PAT *without optimization.* Meanwhile, several PAT structural optimizations using advanced counter mechanisms to compress security metadata have been presented, which significantly reduce the memory encryption overhead. *Split Counter (SC)* is the most typical advanced counter mechanism [71], followed by, *e.g.*, *Vault* [64] and *Morphable Counters* [59]. As the metadata overhead has a significant impact on

[1]Some server-grade memories are equipped with an error-correcting code (ECC) of single error correction and double error detection (SECDED) to tolerate them, while it is not for consumer-level devices (*e.g.*, laptop and smartphone). Note that it offers neither correction/detection capabilities nor security against malicious attackers.

the latency of memory read/write operations owing to the limited bandwidth and cache size, the adoption of such optimizations is crucial for efficient memory encryption. However, SCUE cannot incorporate such structural optimizations using advanced counter mechanisms, which do not preserve the aforementioned simple property of counters. Other major crash consistency mechanisms are also incompatible with these optimizations [77].

Indeed, *combination of recoverable memory encryption and such optimization techniques have rarely been studied and exploited to date, and a mismatch exists between them.* In addition, as the SCUE does not support a lazy recovery (see Section 3.5), the availability should be maintained in a more efficient manner. Owing to the advantages of optimization techniques, it is important for the deployment of secure main memory to develop a recoverable memory encryption mechanism for PAT compatible with optimizations.

1.3 Our contributions

We propose a recoverable memory encryption mechanism, named *Crystalor*, which stands for *CRYptographically Secure Tree-based Authentication with Leaf-Only Recovery.* Crystalor efficiently supports the recoverability with almost no latency overhead and is designed to be securely instantiated with any memory authentication tree, including optimized PAT. Crystalor enables an L4-secure memory encryption on the basis of an L3-secure PAT, while our L4-secure memory encryption is compatible with structural optimizations. Meanwhile, as Crystalor incurs almost no latency overhead for the recoverability, our L4-secure memory encryption achieves as efficient performance as that of L3-secure ones.

Our core concepts are twofold. First, Crystalor does *not* guarantee the consistency/recovery of the entire tree but guarantees the integrity and recovery of only leaf nodes (*i.e.*, payload data) using a distinct verification tag, named recovery tag. Second, at a recovery, Crystalor relinquishes the tree except for leaf nodes and creates a new tree from the leaf nodes with resilience against replay attacks. In Section 3, we present our key observations: (1) an almost universal (AU) hash function [13] is sufficient for provably secure recovery tag verification, and (2) it is possible to build an efficient AU hash function with two properties, namely rate-1 (*i.e.*, one block cipher call to process one input block) and incremental update capability. Accordingly, we develop a (computational) AU hash function named PXOR-Hash, which is tailor-made for our purpose. PXOR-Hash is designed similarly to PXOR-MAC, which is used in the state-of-the-art PAT named ELM [32]; however, PXOR-Hash achieves another security goal/level that we set and has a construction different from PXOR-MAC, which yields efficient implementation. In addition, the recoverability of Crystalor is based on the cryptographic protection, which enables *lazy recovery.* The L4-secure memory encryption based on Crystalor achieves a higher availability than conventional ones thanks to the lazy recovery.

The performance advantage of Crystalor is evaluated by algorithmic analyses and a system-level simulation. Our results reveal that, for protecting a 4 TB memory as an example, Crystalor achieves 29–62% latency reduction per memory read/write in the algorithmic level and a 44% reduction in memory overhead compared to SCUE owing to the compatibility with SC. The system-level simulation using the gem5 simulator [9] shows that Crystalor achieves at most 11.5% lower latency than SCUE. In addition, Crystalor achieves a recovery that is several thousand times faster (*i.e.*, a recovery cost reduction by 90–99.9%) than SCUE owing to metadata structural optimization and lazy recovery, under same condition/assumption.

Table 1: Notations in this paper

Symbol	Meaning	Symbol	Meaning
K, L	Secret keys	β	Tree arity
n	Block length	d	Tree depth
E_K	Block cipher (AES)	ℓ	Bit length of leaf node
F	AU function	k	Degree of SC
D	Input to F	b	# input blocks to MAC
$M[i]$	ℓ-bit plaintext of i-th leaf	$\mathrm{addr}[i]$	address of i-th node
$C[i]$	ℓ-bit ciphertext of i-th leaf	$\mathrm{ctr}[i]$	i-th counter
$T[i]$	Tag of i-th node	$\mathrm{ctr}_{\mathrm{Ma}}[i]$	i-th major counter of SC
$N[i]$	Nonce of i-th node	$\mathrm{ctr}_{\mathrm{mi}}[i][j]$	j-th minor counter of i-th major counter

1.4 Paper organization

Section 2 introduces the (persistent) memory encryption and secure memory. Section 3 outlines the proposed crash recovery mechanism Crystalor, and explains its hardware architecture and operation. Section 4 demonstrates the algorithm-level evaluation and a system-level simulation. Section 5 briefly introduces existing studies related to this paper. Finally, Section 6 concludes this paper.

2 PRELIMINARIES

Notations used in this paper are summarized in Table 1.

2.1 System and threat models of memory encryption

Figure 1 shows a system model with memory encryption, which is simplified, but follows many existing studies and is based on a formalization in [5, 23]. We omitted CPU core(s) to focus on our interest. The model separates the memory system into two areas: the on-chip trusted area and the off-chip untrusted area. On-chip data are assumed to be secure and trustable; that is, any attacker can *neither* eavesdrop nor manipulate on-chip data. In contrast, he/she can perform arbitrary eavesdropping and manipulation of off-chip data. The goal of memory encryption is to realize the confidentiality and integrity of memory data. The on-chip area contains cache(s), memory protection hardware, SRAM, and a memory controller containing a WPQ[2]. The SRAM stores the secret key and root nonce of PAT. The memory protection hardware performs cryptographic computation (*i.e.*, encryption/decryption and MAC). The memory controller employs a WPQ as an ADR domain to buffer data from the cache to memory until it is written, which guarantees the data persistency at the time of cache data replacement/flush. This threat model is compatible with many existing studies on memory encryption [21, 23, 30] and corresponds to the strongest adversary covered by memory encryption (*i.e.*, $\mathcal{A}^{\mathrm{HW}}_{\mathrm{active}}$ formalized in [5]).

It is assumed that the attacker can eavesdrop and manipulate arbitrary data in the memory, including the leaf and intermediate nodes of PAT. The goals of memory encryption are its confidentiality and integrity, while the motivation of recoverability is securely

[2]Memory protection hardware and the SRAM may be considered as parts of memory controller as extension.

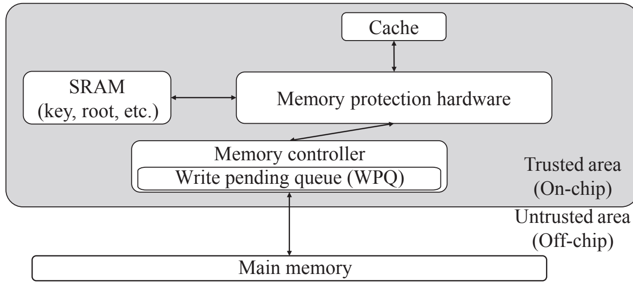


Figure 1: System model of encrypted memory (CPU core(s) are omitted).

maintaining the availability of encrypted memory against crushes, memory errors (*e.g.*, manipulation and accidental bit-flips), and adversarial DoS attacks targeting memory data. Crystalor can realize recovery against any crashes and any manipulation/errors on intermediate nodes. Crystalor does not realize recovery from manipulation/errors on leaf nodes, while Crystalor can detect any manipulation/errors on leaf nodes. We stress that any mechanism cannot always recover data if encrypted data (*i.e.*, contents unrelated to the tree structure) is compromised. Our proposed mechanism can incorporate with ECC(s) and tagged/colored memory [18, 25, 39, 52, 60], which offer an error correction in encrypted data. However, their error-correction capability is necessarily bounded, and attacker may manipulate leaf node(s) beyond it. Here, although compromised leaves are no longer available, PAT suppresses the unavailability against manipulation/error as payload data encryption is divided into small leaf nodes. In contrast, one manipulation/error makes all related memory data unavailable unless recoverability. Thus, the recoverability of PAT is essential for practical memory encryption, in addition to ECCs and tagging/coloring for memory. Note that there exist other types of DoS attack on the system such as system shutdown by privileged user. As our scope is memory protection, we do not consider such DoS attacks targeting outside memory, which should be countered by other means/layer.

Remark 2.1 (Side-channel attacks). We do not consider on-chip side-channel attacks as in previous studies. For example, power/EM analyses on memory encryption hardware may steal the secret key [37]. Microarchitectural attacks represented by Flush+Reload and Prime+Probe [45, 74], which are exploited by Spectre and Meltdown [36, 43], may also be a threat to the confidentiality of payload data. These side-channel attacks are attempts to eavesdrop *on-chip* data. Hence, they are beyond our scope: protection of *off-chip* data against eavesdropping and manipulation. On-chip side-channel attacks should be countered by different means, such as masking and cache randomization [12, 55, 69] (while an authentication tree named MEAS claims a power/EM side-channel security [67]).

2.2 Symmetric cryptography for memory encryption

Existing works on memory encryption (*e.g.*, [15]) have frequently employed AES encryption. For confidentiality, the counter-mode AES has been utilized owing to its parallelizability and random accessibility. For integrity, a MAC has been commonly employed. Note that replay attacks cannot be prevented by simple use of MAC, because they use a valid triple of ciphertext, nonce, and tag.

For both encrypting and verifying payload data efficiently, an AE can be utilized [56] instead of a composition of the counter-mode AES encryption and a MAC [8]. For efficient memory encryption, AE should have two desirable properties: block-level parallelizability[3] and rate-1. A parallelizable AE can encrypt several blocks in parallel, which allows for low-latency implementation using multicore or pipelining. Rate-1 relates to the number of AES calls to complete the encryption and tag computation. A rate-1 AE (*e.g.*, [58]) has a latency of almost half of a composition of the counter-mode AES and a MAC. The use of AE significantly improves the performance of memory encryption, where payload data essentially requires both confidentiality and integrity.

Parallelizability and rate-1 are also desirable for MAC to achieve the integrity of intermediate nodes. In addition, certain MACs have another desirable property known as incrementality [7]. When a data block is changed, an incremental MAC can update the tag with $O(1)$ calls of the underlying cryptographic primitive if old data, nonce, and tag are available, whereas non-incremental MAC, such as CMAC [16], requires $O(m)$ primitive calls, where m is the number of input blocks. Namely, the updated tag of incremental MAC can be computed only from an old tag, old data block, and new data block, as well as old and new nonces. The number of blocks updated in a MAC computation is usually one in memory encryption; hence, incremental MAC significantly improves its performance.

2.3 Parallelizable authentication tree (PAT)

PAT has been employed for memory encryption to realize (1) real-time processing and (2) protection against replay attacks with a minimal overhead of on-chip memory. Figure 2 depicts an overview of PAT with an arity of two as an example. PAT encrypts the leaf node using an AE and verifies intermediate nodes using a nonce-based MAC. The i-th intermediate of PAT consists of $(N[i], T[i])$, where $N[i]$ and $T[i]$ are its nonce and tag, respectively. The i-th leaf node of PAT consists of a tuple $(N[i], T[i], C[i])$, where $C[i]$ is the ciphertext of payload data. The nonce of each node is given by a concatenation of its address $\mathtt{addr}[i]$ and counter $\mathtt{ctr}[i]$ as $N[i] = \mathtt{addr}[i] \parallel \mathtt{ctr}[i]$, where $\cdot \parallel \cdot$ denotes the bit concatenation[4]. The tag of an intermediate node verifies its child node counters as the MAC input. In both verification and update, the tag of each node in a path is computed in parallel as its computation does not depend on the computation results of any lower-level nodes, which indicates the node-level parallelizability of PAT.

ELM. ELM is a state-of-the-art PAT proposed by Inoue *et al.* [32]. ELM is optimized for low latency and scalability to large memory. For this purpose, Inoue *et al.* introduced an AE and incremental MAC named Flat-OCB and PXOR-MAC, respectively. In addition, ELM unifies some computations shareable with Flat-OCB and PXOR-MAC among the entire tree. ELM has a lower latency and less memory overhead than SIT [14], while both schemes use AES-128 and have an equivalent provable security reduction to AES.

[3]Block-level parallelizability is a property of the mode and MAC, whereas node-level parallelizability is of the authentication tree.
[4]Note that $\mathtt{addr}[i]$ does not need storing because it is implicitly determined from its address [14, 32].

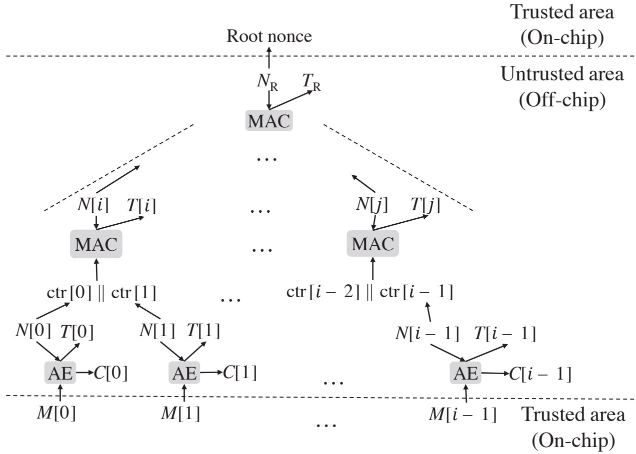


Figure 2: Overview of binary PAT. $M[i]$**,** $C[i]$**,** $N[i]$**, and** $T[i]$ **denote plaintext (*i.e.,* payload data), ciphertext, nonce, and tag of** i**-th node, respectively, where** $N[i]$ **consists of address addr**$[i]$ **and counter ctr**$[i]$**.** N_R **and** T_R **are root nonce and tag, respectively. Leaf node is defined as** $(N[i], T[i], C[i])$**, whereas other nodes are defined as** $(N[i], T[i])$**.**

Notably, several major previous studies on memory encryption (*e.g.*, [21, 30]) employed a classical HMAC, which has a larger latency owing to its serial structure and does not have incrementality. The MAC of ELM (and SIT) is significantly faster than HMAC.

2.4 Split Counter (SC)

SC is the foremost advanced counter mechanism for PAT structural optimization [71], which compresses the tree size (*i.e.*, suppresses the memory overhead for metadata) [4, 53, 54]. This contributes to reduction of the amount of communication between memory and CPU as well as lower latency of AE/MAC computation.

Figure 3 illustrates an overview of an SC-based PAT with an arity of two. An SC-based tree splits a counter into major and minor counters. Minor counter is unique to a node, while major counter is shared by several nodes. In computing AE for a leaf node, the nonce is determined by a concatenation of its address, its major counter, and its minor counter (*e.g.*, $\mathsf{addr}[0] \parallel \mathsf{ctr}_{\mathsf{Ma}}[0] \parallel \mathsf{ctr}_{\mathsf{mi}}[0][0]$ in Figure 3) and the input (*i.e.*, data to be encrypted/decrypted and verified) is the payload data (*e.g.*, $M[0]$). In computing MAC of an intermediate or root node, the nonce is determined by its address, its own major counter, and its own minor counter (*e.g.*, $\mathsf{addr}[i] \parallel \mathsf{ctr}_{\mathsf{Ma}}[i] \parallel \mathsf{ctr}_{\mathsf{mi}}[i][0]$) and the input (*i.e.*, data to be verified) is a concatenation of the major counter(s) and all corresponding minor counters of its child node (*e.g.*, $\mathsf{ctr}_{\mathsf{Ma}}[0] \parallel \mathsf{ctr}_{\mathsf{mi}}[0][0] \parallel \mathsf{ctr}_{\mathsf{mi}}[0][1]$). In updating a node, its minor counter is incremented. Here, if a minor counter overflows, then all minor counters that share a major counter are reset to zero, and the major counter is incremented. Thus, the SC significantly reduces the total bit length of the counters, thereby maintaining the uniqueness of the nonce.

Let l_{ctr} be the bit length of the counter without SC. Let l_{Ma} and l_{mi} be those of the major and minor counters of an SC-based PAT, respectively. Typically, l_{ctr}, l_{Ma}, and l_{mi} are 64, 56, and 8,

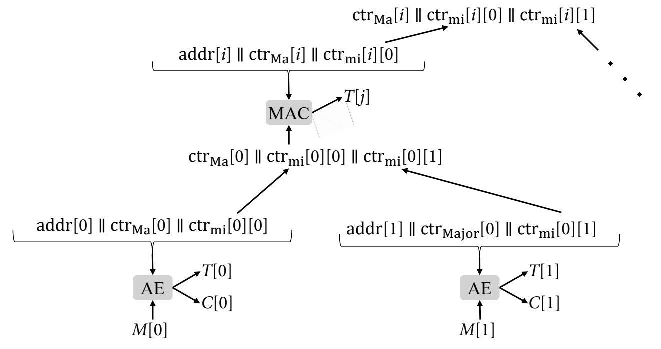


Figure 3: Example of SC-based binary PAT, where $\mathsf{ctr}_{\mathsf{Ma}}[i]$ **is** i**-th major counter, and** $\mathsf{ctr}_{\mathsf{mi}}[i][j]$ **is** j**-th minor counter sharing** $\mathsf{ctr}_{\mathsf{Ma}}[i]$**. MAC input of parent node consists of major counter(s) and all minor counters of child nodes.**

respectively [31, 71]. If k nodes share a major counter, SC reduces the counter size from kl_{ctr} to $l_{\mathsf{Ma}} + kl_{\mathsf{mi}}$.

2.5 Shortcut update (SCUE)

SCUE is a state-of-the-art recoverable memory encryption mechanism with PAT presented by Huang and Hua in HPCA 2023 [30]. In SCUE, the counter of the nonce in PAT is *incremented by one* whenever the node is updated and never decreases under nominal operation (without any reset nor overflow). Its security against replay relies on the fact that a replay attacker can decrease a counter of a node by replacing the nonce and tag in the past but cannot increase it, yet no formal security analysis has been conducted.

The proposals of SCUE include (i) the efficient integrity verification of leaf nodes and (ii) the reconstruction of intermediate nodes from the leaf nodes. The basic concepts underlying SCUE are that (i) the root counter should be always equivalent to the sum of all leaf counters and (ii) a parent node counter is always equivalent to the sum of its child node counters, unless any manipulation/errors. These facts are apparent as the counter represents the number of node updates. After a crash, the integrity of leaf node counters is verified using the MAC with their tags consistently stored in memory. Assuming the security of MAC, the attacker cannot perform any forgery except for replay. Then, SCUE recovers the intermediate nodes before the crash in a bottom-up manner. Each parent node counter value is determined as the sum of its child node counters, termed as *dummy counter*, because they are always equivalent unless manipulation. Finally, to detect a replay of leaf nodes, the SCUE checks the equivalence between the on-chip root counter and the top dummy counter (*i.e.*, sum of leaf counters). The root counter is manipulation-free because it is on-chip, and a replay decreases a counter but cannot increase it, indicating that the sum of leaf counters must be fewer than the root counter if replayed.

Incompatibility of SCUE with SC. The SCUE is essentially based on the fact that the sum of the leaf (resp. child node) counters is always equivalent to the root (resp. their parent node) counter. However, it does not hold for SC-based PAT. In an SC-based PAT, all minor counters sharing the same major counter are reset to zero

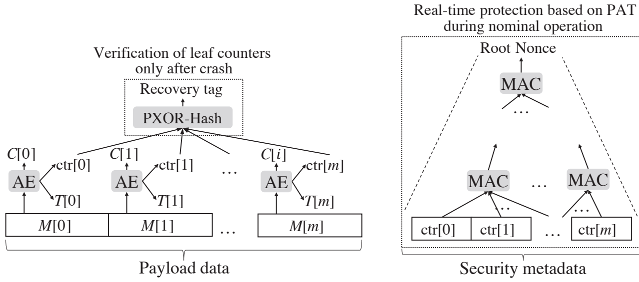


Figure 4: Secure memory based on PAT and Crystalor.

when a minor counter overflows. Here, values of other minor counters sharing an identical major counter are not preserved but are discarded regardless of the value; namely, dummy counter would not be computable. Hence, SCUE can neither verify the integrity of leaf nodes nor reconstruct intermediate nodes of SC-based PAT; thus, SCUE cannot work with SC.

3 PROPOSED MECHANISM: CRYSTALOR

3.1 Basic concept of Crystalor

Figure 4 depicts the proposed L4-secure memory encryption named Crystalor based on PAT. Crystalor distinctly provides crash recoverability and security against crashes, while PAT solely provides confidentiality and integrity *under nominal operation without a crash*. The basic concepts of Crystalor are to use a distinct *recovery tag*, which verifies the leaf node counters *only after a crash or PAT verification error* but can be updated with almost no overhead under nominal operations, and to construct a new tree with a proven resilience against replay following the recovery tag verification.

Our idea is based on the fact that, if we can verify the leaf nodes regarding replay attacks without intermediate node consistency, the intermediate node is no longer required. Thus, we disregard the entire tree consistency, which incurred a non-negligible overhead in most existing schemes, and relinquish the intermediate and root nodes. While the integrity of leaf node is verified by AE except for replay attacks, Crystalor verifies leaf node counters using the recovery tag stored on-chip against replay attacks. The recovery tag verification is only performed after a crash or verification failure, while PAT provides integrity under nominal operation. This indicates that the recovery tag verification does not require real-time processing. In contrast, the tag update requires real-time processing because it should be performed whenever storing data. Thus, we present an incremental universal hash [13] named PXOR-Hash, which is tailor-made for an efficient and optimal realization of such recovery tag. PXOR-Hash is designed similarly to PMAC (and PXOR-MAC); however, PXOR-Hash and PMAC achieve different security goals/levels owing to the difference in their contexts (see Section 3.2), which improves the efficiency and latency of PXOR-Hash compared to PMAC. Importantly, PXOR-Hash verifies any data regardless of its structure, which enables the integrity verification of PAT with structural optimizations (in contrast to SCUE).

The proposal of Crystalor includes how to rebuild the entire tree (*i.e.*, intermediate nodes) from the verified leaf nodes. Recovery of an SC-based PAT is impossible because minor counter values at overflow are discarded, which causes uncertainty on the intermediate counters. In contrast to existing mechanisms, Crystalor creates a new tree, where resilience against replay attacks is proven.

3.2 Recovery tag verification using PXOR-Hash

AE can verify leaf nodes (*i.e.*, payload data) with a nonce consisting of its address and counter. Here, as we use an implicit (*i.e.*, physical) address for the nonce, we can detect a forgery, including splicing, but not a replay attack on a leaf node. To detect a replay, we should verify the integrity of counters using a recovery tag stored on-chip securely. As the recovery tag is verified only after a crash, only its update requires real-time processing (whereas its verification does not). The requirements of recovery tags for security and practical performance are as follows.

Requirement 1 (Security). *Let F denote a function that computes an n-bit recovery tag from input D, which consists of leaf counter blocks. For any adversary with practical resources, the probability of finding a collision on F (i.e. a distinct pair D and D' such that $F(D) = F(D')$) is negligible in n.*

Requirement 2 (Incremental update). *Assume that old tag and old input blocks are available. If one input block is changed, then the new tag can be computed with $O(1)$ calls of symmetric cryptographic primitive. Note that this assumption is generally true in our context because old data remains on-chip before the update.*

Requirement 3 (Fast recovery). *The tag can be computed and verified with $m + O(1)$ calls of symmetric cryptographic primitive (i.e., rate-1), where m is the input length.*

Requirement 1 is crucial as it directly represents a forgery of a recovery tag given an input (*i.e.*, leaf node counters). The function F is either keyed or unkeyed. In the former case, we assume that the adversary does not know the (random) key, owing to the availability of on-chip key register. Here, if F is keyed, Requirement 1 is equivalent to requiring F to be an almost universal (AU) hash function (See Definition 1) [13]. AU hash functions have been extensively studied, which can be efficiently constructed owing to the secret key dependency, compared to one-way hash functions such as SHA-2 or SHA-3. For a keyed function $F : \mathcal{K} \times \mathcal{X} \to \mathcal{Y}$ where $\mathcal{K}$ is the key space, we write F_K to denote $F(K, \cdot)$.

Definition 1 (AU hash function). *Let $F : \mathcal{K} \times \mathcal{D} \to \{0, 1\}^n$ be a function for a key $K \in \mathcal{K}$ and plaintext $D \in \mathcal{D}$. The function F is an ϵ-AU hash function if $\Pr[K \leftarrow \mathcal{K} : F_K(D) = F_K(D')] \leq \epsilon$ holds for any D and $D' \in \mathcal{D}$ such that $D \neq D'$.*

We remark that a full-fledged (nonce-based) MAC will also work; however, an AU hash function is sufficient for our purpose. This is because, in our architecture, the recovery tag is stored in the on-chip trusted/secure area, where *the adversary in Requirement 1 cannot see nor manipulate it*. This feature is crucial because a collision is usually easy to find if the output of the AU hash is visible to the adversary. A nonce is unnecessary because *each leaf node counter is never repeated under nominal operation*. If the recovery tag was stored off-chip or the plaintext of F could take the same value, we would need to employ a conventional MAC, or add a nonce as a new input of F and employ nonce-based MAC. However, this will increase the computational cost or latency compared to an AU hash function. Based on these observations, we develop an AU hash function PXOR-Hash, which fulfills these three requirements.

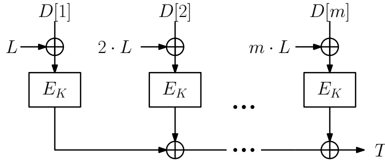


Figure 5: Block diagram of PXOR-Hash.

Although an AU hash function could be built using algebraic operations (*e.g.*, GHASH in GCM), such constructions have difficulties in incremental updates for large inputs. Instead, we adopt a computational variant of AU hash function, which is a simplified version of PMAC (*i.e.*, the sum of input-masked AES). This enables incremental updates for large inputs and provable security guarantee based on the symmetric primitive that we use (namely, AES) [57].

Construction of PXOR-Hash (for Requirement 3). Let $E_K(\cdot)$ denote a block cipher encryption using a secret key K (typically, E_K is AES encryption). Let $D[1], D[2], \ldots, D[i], \ldots, D[m]$ be input data blocks to be verified, where m is the number of data blocks. In the context of Crystalor, each $D[i]$ consists of counters for a leaf node nonce (see below). Figure 5 and Figure 6 show the block diagram and algorithmic description of PXOR-Hash, respectively. The tag of PXOR-Hash is computed as

$$T = E_K(L \oplus D[1]) \oplus E_K(2 \cdot L \oplus D[2]) \oplus \cdots \oplus E_K(i \cdot L \oplus D[i]) \oplus \cdots \oplus E_K(m \cdot L \oplus D[m]),$$

where $L = E_K(0)$, the operator $\oplus$ denotes a bit-wise XOR, and the multiplication is over $\mathbb{F}_{2^n}$ (n is the block length of E_K). Note that L can be pre-computed and stored in on-chip memory in advance to remove its latency. Obviously, the tag is computed with m calls of E_K, which means rate-1.

Incremental update (for Requirement 2). Given an old tag T, let us consider updating the i-th block $D[i]$ to $D'[i]$. The new tag is expressed as

$$T' = E_K(L \oplus D[1]) \oplus E_K(2 \cdot L \oplus D[2]) \oplus \cdots \oplus E_K(i \cdot L \oplus D'[i]) \oplus \cdots \oplus E_K(m \cdot L \oplus D[m]). \quad (1)$$

Using the old tag T and old data $D[i]$, the new tag T' is equivalently computed as

$$T' = T \oplus E_K(i \cdot L \oplus D'[i]) \oplus E_K(i \cdot L \oplus D[i]), \quad (2)$$

which requires only two E_K calls, whereas the naïve computation in Equation (1) requires m calls.

Concrete realization. For the leaf node verification, we compute and store a recovery tag T using PXOR-Hash on-chip, where the inputs are $\mathrm{ctr}[1] \parallel \mathrm{ctr}[2] \parallel \cdots \parallel \mathrm{ctr}[m]$. We use AES-128 for a block cipher, and 64-bit counters (without SC). The input data are expressed as $D[i] = \mathrm{ctr}[2i] \parallel \mathrm{ctr}[2i+1]$ for each $1 \le i \le m$. If the $2i$- or $(2i+1)$-th node is updated, then $D[i]$ is also updated using the computation in Equation (2). Also, if we use SC, the i-th input data block is given by all node counters related to the i-th major counter; that is, $D[i] = \mathrm{ctr_{Ma}}[i] \parallel \mathrm{ctr_{mi}}[i][0] \parallel \mathrm{ctr_{mi}}[i][1] \parallel \cdots \parallel \mathrm{ctr_{mi}}[i][k]$, where k is the number of minor counters. If a leaf node related to $\mathrm{ctr_{Ma}}[i]$ and $\mathrm{ctr_{mi}}[i][1], \mathrm{ctr_{mi}}[i][2], \ldots, \mathrm{ctr_{mi}}[i][k]$ is updated, then $D[i]$ is updated. Note that the address is not required to be input to PXOR-Hash because PXOR-Hash can detect a change in the block order[5]. If a crash occurs, Crystalor first verifies the leaf node using the AE and then detects a replay attack by comparing the on-chip recovery tag and the tag computed by Equation (1).

Algorithm TagGen(D, K, L)
1 $T \leftarrow 0^n$
2 **for** $i = 1$ to m
3 $\quad T \leftarrow E_K(i \cdot L \oplus D[i]) \oplus T$
4 **return** T

Algorithm Verify(D, K, L, T)
1 $T' \leftarrow$ TagGen(D, K, L)
2 **if** $T = T'$
3 $\quad$ **return** $\top$
4 **else**
5 $\quad$ **return** $\perp$

Algorithm Update$(D[i], D'[i], i, K, L, T)$
1 $T \leftarrow E_K(i \cdot L \oplus D[i]) \oplus T$
2 $T \leftarrow E_K(i \cdot L \oplus D'[i]) \oplus T$
3 **return** T

Figure 6: PXOR-Hash algorithms, where $D = (D[1], D[2], \ldots, D[m])$ and $D'[i]$ is new data to be updated.

Security of PXOR-Hash (for Requirement 1). As mentioned previously, PXOR-Hash is a computational AU hash function, or more precisely, an almost XOR-universal (AXU) hash function. An AXU hash function is an AU hash function. Note here that, for a leaf node with $\mathrm{ctr_{Ma}}[i]$ and $\mathrm{ctr_{mi}}[i][j]$, i and j are implicit inputs to PXOR-Hash (that is, the input order of the major and minor counters, which represents the node address). As PXOR-Hash can detect a swap of bits/blocks, Crystalor is secure against splicing. Thus, PXOR-Hash can detect any manipulation on leaf node counters, if the collision probability is negligible.

Integrity will be lost if the securely stored hash value (*i.e.*, T) and output of PXOR-Hash with a modified (forged) input collide. Concretely, the probability for each forgery attempt is at most $4m/2^n$ when $m \le 2^{n-2}$, where $n = 128$ and m is the number of input blocks, assuming that the underlying AES is computationally secure (*i.e.*, a pseudorandom permutation). Hence, the collision probability is negligible in practice if $m \ll 2^{n-2} = 2^{126}$. For example, even for a very large memory of 1 P bits, the collision probability is less than 2^{-83}, which is practically negligible. The collision probability of PXOR-Hash can be obtained by analyzing (the message hashing part of) PMAC [57]. Originally the collision probability was at most $m^2/2^n$ [57]; Minematsu and Matsushima [49, Lemma 2] improved it to $4m/2^n$, assuming that $m \le 2^{n-2}$. These proofs considered doubling-based masks, which means that the i-th input mask is $2^i \cdot L$, where "2" denotes the generator of the field $\mathrm{GF}(2^n)$. This differs from $i \cdot L$ in PXOR-Hash as we adopted it for hardware suitability. However, the proof of [49, Lemma 2] is applicable to our case with trivial changes. Hence, we omit the proof here.

If a stronger security bound is required, which may occur when m is even larger, or if the block size is smaller, we can use stronger methods, such as (the message hashing part of) PMAC with multiple masks [22] and TBC-based PHASH [57]. The former could

[5] This is because $i \cdot L \ne i' \cdot L$ holds for any distinct i and i' (up to $2^{128} - 1$). To change the block order of D, attacker needs to find i and i' such that $E_K(i \cdot L \oplus D[i']) = E_K(i' \cdot L \oplus D[i])$, where $i \cdot L \ne i' \cdot L$ for any i and i'. This is infeasible without K and L due to the pseudorandomness of AES. From a provable security perspective, each block encryption process $Y[i] = E_K(i \cdot L \oplus D[i])$ implements an encryption of a tweakable block cipher (TBC) [44] based on AES and is proven secure if AES is a pseudorandom permutation [57]. This implies that, if $i \ne i'$, TBC outputs (irrespective of $D[i]$ and $D[i']$) are indistinguishable from independent random values.

Table 2: Comparison of incremental primitives

		PXOR-Hash (Ours)	PXOR-MAC [32]	PMAC [57]
Incremental update	# AES calls	**2**	4	4
	Depth	**1**	1	2
Verification	# AES calls	$\boldsymbol{m}$	$m+1$	$m+1$
	Depth	**1**	1	2
Inverse freeness		✓	✓	-

be instantiated low-latency block ciphers such as Prince [10], and the latter could be instantiated with a low-latency TBC such as QARMA [3]. Both methods further reduce the contribution of input length in the collision bound.

Comparison of PXOR-Hash with other incremental primitives. PXOR-Hash is a straightforward derivative of PMAC to implement an incremental AU hash function. Our finding is that an incremental AU hash (thus PXOR-Hash) applied for the whole data, not taking the nonce, suffices for recovery tag generation. Table 2 displays the number of AES calls and depth of PXOR-Hash, PXOR-MAC, and PMAC to process an m block input, which are major incremental primitives. Depth means the number of serial AES calls. Inverse freeness here means that incremental update does not require decryption. PXOR-MAC is nonce-based and achieves forgery resistance in the plain model. However, as mentioned, only collision resistance (*i.e.*, AU) is sufficient for the security of Crystalor. In Table 2, PXOR-Hash halves the cost of an incremental update than PXOR-MAC and PMAC by specializing in Crystalor, which yields significant advantages (*e.g.*, reductions of latency and energy consumption to process recovery tag) during nominal operation.

3.3 Memory recovery by constructing new tree

To date, PAT recovery after a crash has been realized by reconstructing intermediate nodes (*i.e.*, nonce counters) using a redundancy or the relation between parent and child nodes. For example, Anubis uses a shadow table to preserve the node addresses under updates [77]. SCUE reconstructs intermediate nodes from leaf nodes in a bottom-up manner, owing to the consistency between the sum of child node counters and a parent node counter. These existing methods cannot work with SC because minor counter values are discarded and reset when an overflow occurs.

Here, intermediate nodes are not payload data but are only used for verifying leaf nodes regarding a replay attack. In other words, the intermediate nodes are unnecessary if we can verify the leaf nodes in another way (*e.g.*, the recovery tag of PXOR-Hash). Therefore, Crystalor relinquishes the old tree, except for the leaf node, and constructs a new tree. However, if an old counter is used in the new tree, it is exploited by a replay attack, resulting in a feasible forgery. Thus, we should construct a new tree with counter values greater than the old ones for resilience against replay.

We derive an upper bound of the number of updates of a node from its child node counters and propose its use as the new counter value. Consider a case in which k leaf nodes share a major counter and the minor counter bit length is l. A parent node has a major counter $\mathrm{ctr_{Ma}}[i]$ and a minor counter $\mathrm{ctr_{mi}}[i][j]$ for each $1 \le j \le k$. For a PAT with arity of β, it has β child nodes with major counters $\mathrm{ctr_{Ma}}[i']$ and minor counters $\mathrm{ctr_{mi}}[i'][j']$ ($1 \le i' \le \beta/k$ and $1 \le j' \le k$). After a crash, Crystalor computes the maximum possible number of the parent nodes (*i.e.*, an upper bound), which is denoted by $\mathrm{ctr_{ub}}[i][j]$, as

$$\mathrm{ctr_{ub}}[i][j] = \sum_{i'=1}^{\beta/k} \left(\mathrm{ctr_{Ma}}[i'] \left(k(2^l - 1) + 1 \right) + \sum_{j'=1}^{k} \mathrm{ctr_{mi}}[i'][j'] \right).$$

Crystalor then computes the major and j-th minor counter values of the i-th intermediate node as

$$\mathrm{ctr_{Ma}}[i] = \sum_{j=1}^{k} \left\lfloor \frac{\mathrm{ctr_{ub}}[i][j]}{2^l} \right\rfloor, \tag{3}$$

$$\mathrm{ctr_{mi}}[i][j] = \mathrm{ctr_{ub}}[i][j] \bmod 2^l, \tag{4}$$

respectively, where $\lfloor \cdot \rfloor$ is the floor function. All counters of intermediate and root nodes are computed bottom-up by repeating this computation from the leaf nodes. This is based on a fact that the upper bits of nonce are shared as a major counter while the lower l bits are unique to each minor counter.

We prove Theorem 1 to validate the security of the recovery of Crystalor against replay attack.

Theorem 1. *Let $ctr_{Ma}[i]$ and $ctr_{mi}[i][j]$ be the i-th parent node major and minor counter values computed by Equations* (3) *and* (4)*, respectively. Any new tree is resistant to replay attacks.*

Proof. First, we consider a single crash. Let c be the number of updates from the previous reset of minor counters until the next reset (*i.e.*, major counter increment). It always holds $2^l \le c \le k(2^l - 1) + 1$, because c is the minimum if only one node is updated (and the others are not updated at all), whereas c is the maximum if each of the k minor counters has the maximum value (*i.e.*, $2^l - 1$). Let $u_{i'}$ be the total number of updates of nodes sharing the i'-th major counter, which is bounded above as

$$u_{i'} \le \mathrm{ctr_{Ma}}[i'] \left(k(2^l - 1) + 1 \right) + \sum_{j'=1}^{k} \mathrm{ctr_{mi}}[i'][j'],$$

because $\mathrm{ctr_{Ma}}[i']$ denotes the number of minor counter resets. Let $u_{i,j}$ be the number of updates of a parent node with $\mathrm{ctr_{mi}}[i][j]$, which is bounded above as

$$u_{i,j} \le \sum_{i'=1}^{\beta/k} u_{i'} = \mathrm{ctr_{ub}}[i][j].$$

This indicates that $\mathrm{ctr_{ub}}[i][j]$ is greater than or equal to the number of updates of the node (*i.e.*, the true value of the parent counter ever before). The equality holds if c is the maximum value whenever and wherever the minor counter resets or if any minor counter reset has not occurred (*i.e.*, $\mathrm{ctr_{Ma}}[i'] = 0$ for all i'). Thus, for all $1 \le j \le k$, the parent node is updated at most $\mathrm{ctr_{ub}}[i][j]$ times. As $2^l \le c \le k(2^l - 1) + 1$ also holds for the parent node, the number of updates of the parent major counter must be less than $\sum_{j=1}^{k} \left\lfloor \mathrm{ctr_{ub}}[i][j]/2^l \right\rfloor$, which indicates that the new major counter value never appears before the crash. Thus, its replay is impossible.

Next, we consider multiple-crash cases, where the counter values are given by Equations (3) and (4) in past. If a leaf node counter is incremented, a corresponding $\mathrm{ctr_{ub}}[i]$ always has a greater value than the previous state, because it is monotonically increasing in terms of both $\mathrm{ctr_{Ma}}[i']$ and $\mathrm{ctr_{mi}}[i'][j']$. This implies that either or

both $\mathrm{ctr}_{\mathrm{Ma}}[i]$ and $\mathrm{ctr}_{\mathrm{mi}}[i][j]$ are greater than any previous state. In addition, if a child node is updated, its parent node counter increases accordingly; however, its increase amount is not as great as the number of updates, as aforementioned. Thus, the new counter values determined by Equations (3) and (4) are always new, which guarantees resistance against replay attacks. □

The integrity of leaf nodes is verified by AE, excluding replay attacks. The recovery tag verification detects the replay attacks on leaf nodes. In addition, Theorem 1 states that the counter values of the new tree are always greater than values before the crash, which indicates the resistance of the new tree to replay attacks. Thus, Crystalor provides both crash recoverability and integrity against any manipulation attacks. Note that, in a recovery operation, the new tree construction and leaf node/tag verification should be carefully executed so as to avoid replay attacks (see Section 3.5).

3.4 Hardware architecture

Figure 7 displays the hardware architecture of Crystalor for `L4` memory encryption [32], in which we employ ELM. Crystalor utilizes dedicated hardware components to compute and update the recovery tag apart from memory protection hardware for ELM computation, as Crystalor operates distinctly and independently of PAT. The dedicated hardware consists of an SRAM to store and update the recovery tag (Recovery TAG register) in addition to the secret key of PXOR-Hash key (KEY register). PXOR-Hash hardware consists of pipelined AES encryption hardware, which can process multiple update transactions in parallel in the most efficient manner. The recovery tag is stored in both the SRAM and cache (Recovery TAG register and cache) to improve the tag computation speeds. This dedicated hardware operates at every timing of leaf node update (*i.e.*, storing encrypted payload data to memory) to simultaneously and consistently update/store the recovery tag to Recovery TAG register and cache. In Figure 7, the recovery tag is always updated on-chip but is not disclosed to the off-chip memory. This is mandatory for security to prevent any manipulation attack on these data. In other words, as the recovery tag is securely processed and stored, it does not require as strong protection as MAC, which leads to an efficient implementation of PXOR-Hash.

Crystalor requires on-chip SRAM for storing the 128-bit recovery tag and PXOR-Hash keys (K and L). The SRAM overhead is 384 ($= 128 \times 3$) bits in total. Note that PXOR-Hash provides sufficient security even for long inputs (e.g., a collision probability of $4m/2^n \approx 2^{-89}$ for 4TB memory); thus, storing only one recovery tag on-chip is sufficient. Crystalor also utilizes a 128-bit on-chip cache for Recovery TAG cache and a round-based AES encryption engine [65, 66] for PXOR-Hash, which has a sufficiently low latency regarding the latency of ELM computation. In addition, it is implemented with less than 15 K GE, far smaller than ELM hardware [32].

The remaining parts are similar to the existing ones, with some modifications for Crystalor. The ELM hardware includes a 952-bit cache for storing the ELM secret key and precomputable intermediate values, and use an $(\ell+64)$-bit non-volatile register to protect AE computation against crashes for the leaf node persistency (see Section 3.5). We employ an on-chip WPQ using an ADR to persist data during store operation [63], namely, to guarantee the consistency of the leaf nodes with recovery tag. Note that the intermediate nodes are discarded at a crash; hence, they do not require persistency; thus, MAC outputs (*i.e.*, intermediate nodes) are computed and updated with background processing and are directly written to memory without WPQ. We also utilize atomic persistency mechanism(s) and hardware redo logging as in existing methods.

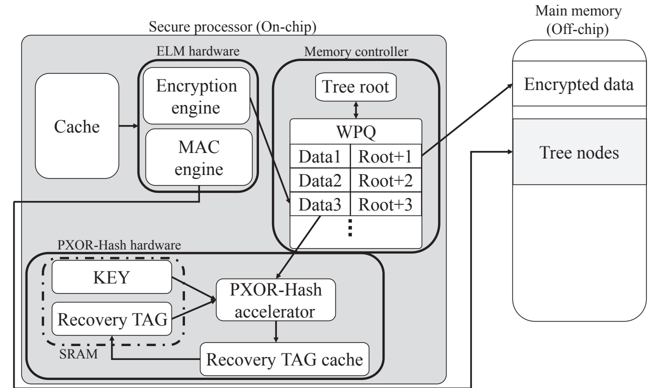


Figure 7: Crystalor hardware architecture.

Remark 3.1 (Combination with other mechanisms). Figure 7 shows the simplest construction without any optimization mechanism. Advanced counter mechanisms such as SC, Vault, and Morphable counters and sophisticated mechanisms to handle metadata (*e.g.*, *Lelantus* [76]) are applicable. The mechanism(s) for atomic data persistency can be also adopted/combined [28, 33, 34, 68]. As PXOR-Hash operates distinctly from PAT, such optimization mechanisms for PAT can be readily incorporated together.

Performance overhead. The latency overhead of Crystalor during nominal operation solely depends on the computational cost of the recovery tag update. As mentioned in Section 3.2, the recovery tag update is completed within only two AES encryption calls for new and old $D[i]$'s. The two AES encryptions are performed in parallel using pipelined AES hardware, which requires significantly smaller latency of ELM update. Therefore, the latency overhead is negligible and has no impact on the system performance, as Crystalor and PAT operate distinctly. Crystalor incurs little overhead in the memory controller owing to its simplicity.

3.5 Operations

We describe the store and recovery operations of ELM–Crystalor. Read operation requires no Crystalor operation.

Store operation. For security and recoverability (covering a crash during AE encryption), we must simultaneously and consistently update the leaf node counter in the memory and the on-chip recovery tag, which is realized by the following steps:

1. Store operation to the memory is issued. It would correspond to a cache data replacement (*i.e.*, cache miss) or an explicit instruction to guarantee the data persistency at a timing, such as `clflush` and `clwb` in x86 architecture.
2. (**ELM computation**) Payload data are encrypted by AE. The MAC tags of the corresponding intermediate/root

nodes are computed in the background, and the corresponding counters are incremented on-chip. When AE encryption starts, a busy frag (one-bit on-chip NVM) is raised. AE inputs (*i.e.*, payload data and nonce) are preserved in non-volatile register until the result is moved to WPQ, for recovery from a crash during AE encryption (*i.e.*, the leaf node consistency).
3. (**PXOR-Hash computation**) The new recovery tag is computed from the old tag, old data, and new data. This step should be computed in parallel with Step 2.
4. (**Write to WPQ**) The AE encryption result and nonce are moved to WPQ. The busy flag is put down when completed.
5. (**Recovery tag update**) Recovery TAG cache and registers are updated.
6. (**Store data**) The WPQ data are stored in memory. The store operation is completed.

Note that Steps 4 and 5 should be synchronously executed in parallel for consistency between the recovery tag and leaf counters. Data can be updated in an atomically persistent manner, with the aid of some mechanisms for this purpose (*e.g.*, [28, 33, 34, 68]).

Recovery. This is executed after a crash or upon a verification failure (due to manipulation or fault). Crystalor securely recovers the memory in an identical manner for both cases as follows.

1. (**AE status check**) If the busy flag has been raised, the AE encryption and PXOR-Hash update are performed using the payload data and nonce preserved in the non-volatile register and the result is stored to the memory through the WPQ, according to the redo logs.
2. (**New tree construction**) The counter values of intermediate and root nodes are derived in a bottom-up manner according to Section 3.3. The MAC tag of each node is also computed from the counters and addresses.
3. (**Recovery tag verification**) The recovery tag of PXOR-Hash is computed from all leaf node counters, and then the computed tag is compared to the on-chip Recovery TAG register value. If these are equivalent, the nominal operation restarts; otherwise, Crystalor gives an error signal.

The new tree constructed at Step 2 can detect any manipulation on leaf nodes except for replay during a crash, while Step 3 detects any replay of leaf counters; thus, all manipulations can be detected by their combination. Here, we should load the leaf counter *only once* to compute both the new tree and recovery tag, which guarantees that the leaf counter values used in computing them are identical (*i.e.*, not tampered/replayed), as the attacker cannot tamper with on-chip data. If we loaded the leaf counter *twice distinctly* for computing the new tree and recovery tag, the attacker could insert replays during the interval between two loads, resulting in a feasible forgery[6]. Therefore, execution of Steps 2 and 3 should be parallel using ELM and PXOR-Hash hardware to prevent any replay. Namely, after a leaf value is input to the new tree computation, then the leaf value should be directly sent to on-chip PXOR-Hash hardware to compute the recovery tag. Note that large on-chip SRAM is not required for this because PXOR-Hash/MAC is computed by sequentially summing AES encryption results.

Lazy recovery. Crystalor can detect any counter replay solely by the recovery tag verification, whereas SCUE computes AE/MAC for leaf nodes[7]. Hence, upon a reboot, Crystalor does *not* require verifying the AE leaf node for protection because it will be verified before it is actually used. In other words, the leaf node AE verification can be omitted at the time of recovery, and the verification is completed lazily and concurrently during nominal operation after the recovery. Thus, Crystalor does not have to read the leaf node data nor compute AE at the reboot time. This yields a significant reduction in the recovery cost (as evaluated in Section 4.3) compared to the non-lazy recovery because the leaf node occupies a major part of encrypted memory. Such a lazy strategy was adopted in some previous studies [77], and is covered by the provable security of PAT. An attacker in a practical use scenario of memory encryption, who can manipulate off-chip data and trigger crashes, cannot bypass the recovery tag verification and PAT verification simultaneously; thus, their combination detects any replay.

Remark 3.2 (Tradeoff of lazy recovery and intermediate layer consistency). In the SCUE paper [30], they utilized the consistency of the lowest *intermediate* nodes (but not leaf nodes at the actual lowest layer of PAT). This allows for a lazy recovery without computing AE/MAC for leaf nodes, although guaranteeing the consistency of intermediate layer(s) would be a non-negligible cost during nominal operation. Meanwhile, Crystalor offers a more efficient recovery if utilizing such a consistency of higher layer, because Crystalor can compute the recovery tag using nonces (counters) of the intermediate layer nodes instead of leaf nodes in this case. Generally, if lower w layers are guaranteed to be consistent at a crash, Crystalor should verify inputs of the $(w + 1)$-th lowest layer nodes using the recovery tag, while SCUE recovery would require the AE/MAC verification of the w-th lowest layer nodes. The height of layer to be verified is crucial for the efficiency of the recovery because the number of nodes is exponentially greater for lower layer. For a fair and sound comparison, the evaluation in Section 4.3 assumed $w = 1$ for both Crystalor and SCUE; that is, only leaf nodes (*i.e.*, the lowest layer of PAT) are guaranteed to be consistent, as our hardware architecture.

4 PERFORMANCE EVALUATION

4.1 Algorithm-level evaluation

We assume to utilize AE and MAC hardware presented in [32] for the ELM hardware in this evaluation, which is based on an unroll-pipelined AES architecture that computes one-block per clock cycle with a latency of 10 clock cycles. For ELM with depth of d, we assume to use one AE and d MAC hardwares for a parallelized computation of path update/verification. We consider a typical SC parameter [31, 53]: the lengths of major and minor counters (*i.e.*,

[6]Namely, a replay attack was feasible if the attacker could insert a leaf node replay for the timing of load for the new tree computation but the non-tampered counter was loaded for the recovery tag computation.

[7]Upon a crash, an attacker can insert a replay data without SCUE detected by incrementing another leaf node counter, such that the sum of leaf node counters is preserved. If the replayed node is loaded before detecting the incremented counters, the replay is not detected. Thus, entire leaf node AE verification is mandatory to detect such a replay with a counter increment, implying the insecurity of lazy recovery for SCUE. Hence, SCUE requires verifying the leaf node AE at the timing of recovery.

l_{Ma} and l_{mi}) are 56 and 8 bits, respectively, and the number of nodes sharing a major counter (*i.e.*, k) is 8.

SC was originally proposed as an optimization method to reduce the metadata size (*i.e.*, memory overhead). The reduction of metadata size also contributes to a latency reduction because the length of data to be verified by MAC and the amount of communication between memory and CPU are reduced. We derive the relation between the covered region size and ELM parameters to calculate the (optimal) latency of ELM for a given covered region size. Subsequently, we can select an optimal parameter that covers the region with the minimum latency. Note here that the advantages of SC directly represent the supremacy of Crystalor over SCUE.

Latency and covered region size. Let b denote the number of input blocks to MAC and let ℓ denote the bit length of a leaf node. Here, b is derived from the tree arity β as $b = \beta/2$ and $b = \beta/8$ without and with SC, respectively. Without SC, an intermediate node has $2b$ child nodes because a counter is represented by 64 bits. This indicates that the tree has $2^d b^d$ leaf nodes. If SC is applied, an intermediate node can have $8b$ nodes because an input block $D[i]$ consists of a 64-bit major counter and 8-bit minor counters of eight child nodes. This indicates that the tree with SC has $2^{3d} b^d$ leaf nodes. For given b and ℓ, ELM can cover a region of $2^d b^d \ell$ and $2^{3d} b^d \ell$ bits without and with SC, respectively. Meanwhile, the update latencies of Flat-OCB and PXOR-MAC hardware in [32] are $14 + \ell/128$ and $12 + b$ clock cycles, respectively. Note that the metadata size does not include bits to indicate the address because it is implicit. Table 3 reports the latency and covered region size for different values of b and ℓ without and with SC, where the latency means $\min(14 + \lceil \ell/128 \rceil, 12 + b)$ as the bottleneck. From Table 3, we confirm that SC significantly reduces the latency to cover a given region. For example, for $d = 5$ and 7, to cover a 4 TB region, ELM without SC requires at least 78 and 30 clock cycles for the update, whereas ELM with SC requires only 30 and 22 clock cycles, respectively. Thus, SC reduces the latency overhead by 38 and 8 cycles for $d = 5$ and 7 (*i.e.*, 62% and 29%), respectively.

Table 3: Latency and covered region of ELM with and without SC, where b is number of input blocks (corresponding to tree arity), ℓ is bit length of AE, and d is tree depth

				Covered region [Byte]					
				ELM w/o SC			ELM with SC		
b	ℓ	Update†	Verify	$d = 3$	$d = 5$	$d = 7$	$d = 3$	$d = 5$	$d = 7$
4	512	21	18	33 K	2 M	134 M	2 M	2 G	2 T
	1,024	25	22	66 K	4 M	268 M	4 M	4 G	4 T
	2,048	33	30	131 K	8 M	537 M	8 M	8 G	9 T
	4,096	49	46	262 K	17 M	1 G	17 M	17 G	18 T
	8,192	81	78	524 K	34 M	2 G	34 M	34 G	35 T
8	512	22	20	262 K	67 M	17 G	17 M	69 G	281 T
	1,024	25	22	524 K	134 M	34 G	34 M	137 G	563 T
	2,048	33	30	1 M	268 M	69 G	67 M	275 G	1 P
	4,096	49	46	2 M	537 M	137 G	134 M	550 G	2 P
	8,192	81	78	4 M	1 G	274 G	268 M	1 T	5 P
16	512	30	28	2 M	2 G	2 T	134 M	2 T	36 P
	1,024	30	28	4 M	4 G	4 T	268 M	4 T	72 P
	2,048	33	30	8 M	9 G	9 T	537 M	9 T	144 P
	4,096	49	46	17 M	17 G	18 T	1 G	18 T	288 P
	8,192	81	78	34 M	34 G	35 T	2 G	35 T	576 P
32	512	46	44	17 M	69 G	281 T	1 G	70 T	5 E
	1,024	46	44	34 M	137 G	563 T	2 G	141 T	9 E
	2,048	46	44	67 M	275 G	1 P	4 G	281 T	18 E
	4,096	49	46	134 M	550 G	2 P	9 G	563 T	37 E
	8,192	81	78	268 M	1 T	5 P	17 G	1 P	74 E
64	512	78	76	134 M	2 T	36 P	9 G	2 P	590 E
	1,024	78	76	268 M	4 T	72 P	17 G	5 P	1 Z
	2,048	78	76	537 M	9 T	144 P	34 G	9 P	2 Z
	4,096	78	76	1 G	18 T	288 P	69 G	18 P	5 Z
	8,192	81	78	2 G	35 T	576 P	137 G	36 P	9 Z
128	512	142	140	1 G	70 T	5 E	69 G	72 P	76 Z
	1,024	142	140	2 G	141 T	9 E	137 G	144 P	151 Z
	2,048	142	140	4 G	281 T	18 E	275 K	288 P	302 Z
	4,096	142	140	9 G	563 T	37 E	550 K	576 P	604 Z
	8,192	142	140	17 G	1 P	74 E	1 P	1 Z	1 Y

† "Update" actually means "Verify then Update," because PAT requires tag verification always before update for provable security [14, 32].

Metadata size. We evaluate the contribution of SC to reducing the memory overhead for storing metadata. Without SC, the metadata size is $112 \sum_{i=0}^{d} \beta^i - 56$, whereas with SC, it is $72 \sum_{i=0}^{d} \beta^i - 56 \sum_{i=0}^{d-1} \beta^i$, according to [31]. For example, to cover a 4 TB region with a tree of $b = 4$, ELM without and with SC has an overhead of 554 GB and 312 GB for storing the metadata, respectively, which indicates a 44% reduction of the overhead by SC. Thus, SC significantly improves the memory encryption performance.

Additional latency due to minor counter overflow. When a major counter is incremented (*i.e.*, a minor counter overflows), SC requires the re-computation of tags related to the major counter. If the j-th node of i-th major counter overflows, then the major counter $\mathrm{ctr}_{\mathrm{Ma}}[i]$ is incremented, and the minor counters $\mathrm{ctr}_{\mathrm{mi}}[i][j]$ for all j ($1 \le j \le k$) are reset to zero. We should recompute the tag of nodes for all j, as its nonce counter $\mathrm{ctr}_{\mathrm{Ma}}[i] \parallel \mathrm{ctr}_{\mathrm{mi}}[i][j]$ is updated. This means that $k-1$ MAC/AE updates accompany a minor counter overflow. The system-level simulation for the performance evaluation should regard the latency due to minor counter overflow. Nevertheless, the latency overhead by minor counter overflow is not critical as its frequency is low. On average, it incurs less than one clock cycle latency per store operation.

Remark 4.1 (Tree depth and hardware resource). The tree depth d is a parameter that exploits tradeoffs between a *hardware resource* (*i.e.*, the number of MAC engines) and covered region/latency, while b and ℓ optimal in terms of latency are determined systematically for a given covered region. In other words, for an optimal fixed b and ℓ, we can enlarge the covered region size by increasing d, using $d - 1$ parallel MAC engines. Conversely, we can reduce the latency for a fixed covered size by increasing d. Thus, the significance of covered region size and improvement by SC depend on d.

4.2 System-level simulation

We performed system-level simulations using the gem5 simulator [9] for the validation. We simulated a CPU with an encrypted main memory (NVM) as same as previous studies, while it is applicable to standard DRAM as well. In this simulation, we evaluated

the memory encryption mechanisms with ELM, which is the state-of-the-art and achieves the highest performance among PATs[8]. We evaluated the proposed and existing methods as follows:

- Insecure: Memory without any security mechanism.
- ELM without SC: ELM not using SC (not recoverable).
- ELM with SC: ELM using SC (not recoverable).
- ELM–SCUE: ELM with SCUE [30] (SC is inapplicable).
- ELM–ASIT: ELM with Anubis [77] (SC is inapplicable).
- ELM–Crystalor (this work): ELM with Crystalor, to which SC is applied.

Insecure and ELMs (not recoverable) were the baselines to evaluate the overhead of PAT and crash recoverability, respectively. We determined the latency according to the memory capacity (*i.e.*, 4 TB) and Table 3. When store operation, all schemes are assumed to perform the path verification before update for the sake of provable security (against replay attacks) [27, 32]. For ease, feasibility, and reproducibility of the experiment, we employed several simplifications for the simulation, similarly to some existing studies. We virtually inserted the latency of ELM to read and store operations according to Table 3, while we omitted the simulation of security metadata packets. For ELM with SC, to evaluate the latency about minor counter overflow, we employed an apportionment, in which we assumed that the writings to memory were uniformly distributed, calculated the expected latency due to the minor counter overflow, and added the rounded-up value to the latency in Table 3. Note that the assumption of uniform distribution was used only for the estimation of minor counter overflow cost, but the workload execution times were evaluated using their actual memory accesses. These simplifications were applied to all of the above methods, which enabled a fair and sound comparison.

We employed a benchmarking workload set, which has been commonly used in many previous studies as a de facto standard (*e.g.*, [29, 40, 72, 79]). The workloads include random insertions of data to a hash table (HT), binary search tree (BST), red-black tree (RBT), and queue (Queue), each of which has a distinct memory access pattern. To analyze the difference, we simulated the workloads with data sizes of 64, 512, 1,024, and 4,096 bytes.

Results. Figure 8 reports the normalized workload execution times of the gem5 simulation, in which Insecure is the baseline. We did not evaluate $d = 3$, as ELM with $d = 3$ without SC cannot cover a 4 TB region with a practical latency (this demonstrates the significance of SC). ELM–Crystalor exhibits almost the same performance as ELM with SC. As well, ELM–SCUE and ELM without SC are the almost same. As Crystalor and SCUE incur no latency overhead under nominal operation, the performances of ELM–Crystalor and ELM–SCUE depend solely on the tree parameter. In contrast, ASIT incurs a non-trivial latency overhead to verify and update the shadow table. Comparing ELM with and without SC (*i.e.*, ELM–Crystalor and ELM–SCUE), the performance gain by SC is more significant when the data size is larger. This is because the reduction in latency in reading and storing (*i.e.*, verifying and updating) memory data is more dominant and visible as the numbers of data read and write increase for a larger data size. In addition, the improvement in execution time by SC is more significant when $d = 5$ than $d = 7$, because the reduction ratio of latency by SC is larger when $d = 5$ for covering a 4 TB region. Thus, we confirm that ELM–Crystalor can reduce the workload execution time by at most 11.5% compared to the state-of-the-art mechanism (*i.e.*, ELM–SCUE).

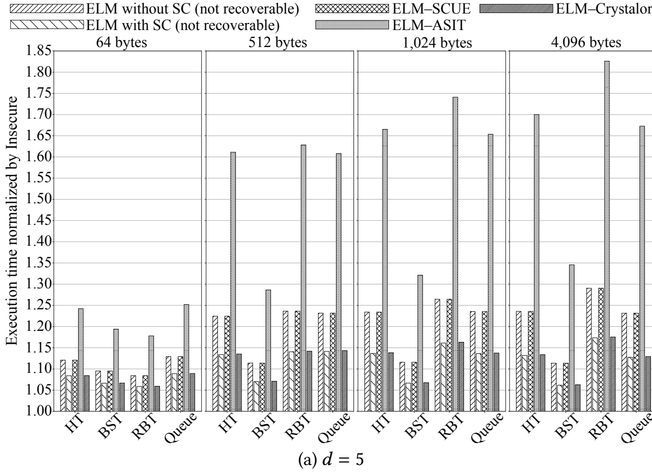
[8]Some previous studies (*e.g.*, [30]) utilized a classical HMAC, which is assumed to require 40, 80, or 160 clock cycles for Verify and Update. However, its concrete realization/implementation was not mentioned, and the number of input blocks to AE/MAC, which actually determines the latency, was not considered. Thus, its practical validity is unclear. We employ the ELM-style evaluation to determine the clock cycles for a fair, modern, and practical performance comparison. Our results are based on the in-depth evaluation of latency in the ELM paper, which considers a concrete cryptographic hardware implementation and the number of input blocks to AE/MAC, while previous studies did not. Note that HMAC is not optimal in terms of latency and is not incremental, while PXOR-MAC in ELM was proposed for an optimized latency [32].

Table 4: Simulation conditions

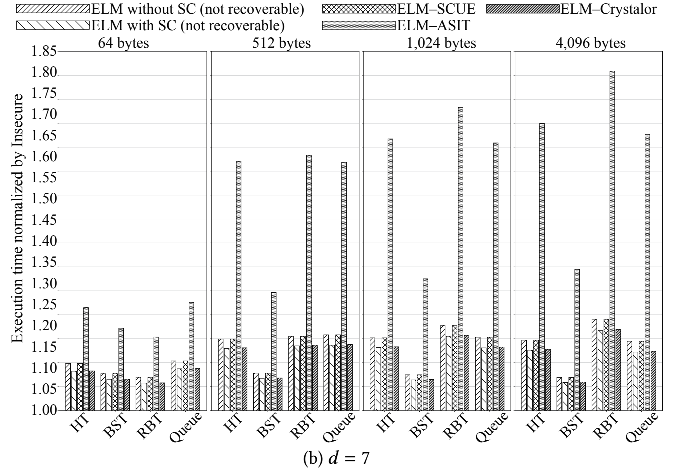
CPU and caches	
CPU core	One core, out-of-order, 2.4 GHz
L1 instruction cache	32 KB, 8-way, 2 cycles
L1 data cache	64 KB, 8-way, 2 cycles
L2 cache	32 KB, 8-way, 2 cycles
Metadata cache	256 kB with cache line 64 byte
Memory controller and main memory (NVM)	
WPQ size	8 entries
Memory latency	Read 50 ns and Write 150 ns
Memory size (covered region)	4 TB
ELM (d = 5), to which SC is applied	
Update and verify latency	30 and 28 cycles
Tree parameters	$b = 16$ and $\ell = 1{,}024$
ELM (d = 5), to which SC is inapplicable/not applied	
Update and verify latency	78 and 76 cycles
Tree parameters	$b = 64$ and $\ell = 1{,}024$
ELM (d = 7), to which SC is applied	
Update and verify latency	22 and 20 cycles
Tree parameters	$b = 8$ and $\ell = 1{,}024$
ELM (d = 7), to which SC is inapplicable/not applied	
Update and verify latency	30 and 28 cycles
Tree parameters	$b = 16$ and $\ell = 1{,}024$

Improved scalability for larger memory. Regarding the tree depth d, the performance gain by the proposed method is greater for a shallower tree (*i.e.*, $d = 5$ in this experiment). Recall that d is a parameter that exploits tradeoffs between a hardware resource (*i.e.*, the number of MAC engines) and a covered region. As the cover region size in the experiment is fixed at 4 TB, the size is relatively larger for $d = 5$, and the latency overhead by PAT-based protection is larger for $d = 5$. The SC compresses the tree/metadata size more effectively when protecting a larger memory. Hence, the performance gain by SC (and the proposed method) is greater for $d = 5$. More quantitatively, for a given arity β, the use of SC can reduce the number of input blocks to PXOR-MAC to $1/4$ in the used parameter. This indicates that the use of SC reduces the latency of PXOR-MAC asymptotically by $1/4$ for a larger β. Thus, the use of SC (and ELM-Crystalor) reduces the latency of the memory read/write by up to $1/4$ when protecting a larger memory. The experimental results on $d = 5$ and 7 indicate an improved scalability.

On other CPUs and workloads. We used a simple CPU and common benchmarking workloads in the simulation. The latency overhead of memory encryption is mainly incurred by main memory accesses. In other words, the number of cache misses would be highly related to the latency overhead. This would indicate that, for example, a larger (resp. smaller) cache and a better (resp. worse) cache replacement policy would mitigate (resp. deteriorate) the latency overhead while other factors (*e.g.*, pipeline stages and the



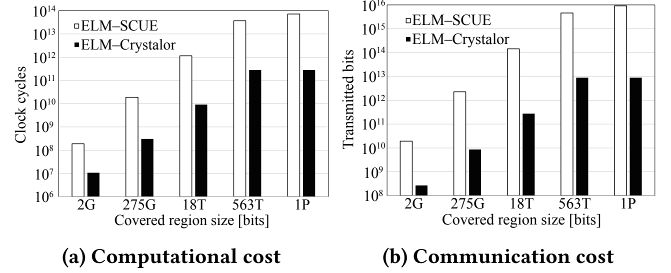
(a) $d = 5$



(b) $d = 7$

Figure 8: Simulated execution times normalized by Insecure, where proposed method is right-most bin for each workload.

number of CPU cores) would have a less impact. In addition, some other benchmarking workloads have been frequently employed in the microarchitectural research field (*e.g.*, SPEC). The workloads in our evaluation would require non-trivial memory-access costs, while some other workloads would be intensive rather in, for example, arithmetic operations, which would render the difference in latency overheads smaller. Thus, changes of CPU construction and workload would have an impact on the magnitude of advantage of ELM–Crystalor. Meanwhile, the evaluation results would be basically consistent for other CPU constructions and workloads, as ELM–Crystalor offers faster store and load at the algorithm level.



(a) Computational cost **(b) Communication cost**

Figure 9: Estimation of recovery costs.

4.3 Recovery cost estimation

Lazy recovery cost of Crystalor. We here assume that only leaf nodes are guaranteed to be consistent (see Remark 3.2). Consider an ELM–Crystalor with SC, whose major and minor counters are 56 and 8 bits, respectively. The recovery time of Crystalor is evaluated by the number of AES calls for PXOR-Hash of recovery tag verification and PXOR-MAC of a new tree construction. To protect an M-bit memory, the number of AES calls in PXOR-Hash is $M/8\ell$, while the leaf node verification by Flat-OCB is not required at the time of recovery as mentioned in Section 3.5. In addition, for arity of β, a PXOR-MAC computation requires $1 + \beta/8$ AES calls, while a new tree construction is realized with $\sum_{i=1}^{d} \beta^{i-1}$ PXOR-MAC computations. This indicates that the new tree construction requires $(1+\beta/8) \sum_{i=1}^{d} \beta^{i-1}$ AES calls in total. Moreover, a new tree construction requires computations of new counter value $\sum_{i=1}^{d} \beta^{i-1}$ times. Recall that $M = \beta^d \ell$. Thus, Crystalor recovery is realized with $\beta^d/8 + (1 + \beta/8) \sum_{i=1}^{d} \beta^{i-1}$ AES calls and $\sum_{i=1}^{d} \beta^{i-1}$ new counter computations, which corresponds to the number of intermediate nodes (including root node). In addition, Crystalor requires to read $8\beta^d$ bits of leaf node metadata from memory, while Crystalor writes $16(4 + \beta) \sum_{i=2}^{d} \beta^{i-1}$ bits of metadata to memory. Owing to the lazy recovery, the costs are independent of leaf node bit length (*i.e.*, ℓ).

Recovery cost of SCUE. For comparison, we consider ELM–SCUE recovery with a 64-bit counter. Its recovery cost is evaluated by the number of counter-summings and PXOR-MAC computations. The equivalence check between the root counter and the sum of leaf node counters requires $\sum_{i=1}^{d} \beta^{i-1}$ counter-summings. The tree recovery performs one counter-summing and one PXOR-MAC computation per node. A PXOR-MAC requires $1 + \beta/2$ AES calls and $\sum_{i=1}^{d} \beta^{i-1}$ intermediate nodes exist. The leaf node AE verification requires $\beta^d(\lceil \ell/128 \rceil + 1)$ AES calls. Thus, the computational cost is $2 \sum_{i=1}^{d} \beta^{i-1}$ counter-summings and $\beta^d(\lceil \ell/128 \rceil + 1) + (1 + \beta/2) \sum_{i=1}^{d} \beta^{i-1}$ AES calls. Meanwhile, SCUE reads $\beta^d \ell + 64\beta^d$ bits from memory and it writes $64(1 + \beta) \sum_{i=2}^{d} \beta^{i-1}$ bits to memory.

Evaluation result. Figure 9a and Figure 9b report the computational cost (*i.e.*, the number of clock cycles) and the communication cost between CPU and memory for some covered region sizes, respectively. Here, the throughputs of the counter-summing, new counter computation, and one block AES encryption are assumed to be one per clock cycle [14, 32]. In Figure 9b, we evaluated by the number of transmitted bits, in which the writing cost was tripled as in Table 4. Crystalor achieved a reduction of 90–99.9% of recovery costs from SCUE, owing to the lazy recovery. While the computational and traffic costs of the leaf node AE verification is a major part of SCUE (*i.e.*, $\beta^d(\lceil \ell/128 \rceil + 1)$ AES calls and $\beta^d \ell$ bits read, respectively), Crystalor does not require them. Thus, we confirm the advantage of Crystalor in recovery cost as well as the performance.

5 RELATED STUDIES

Crash consistency of encrypted memory. Note that some crash consistency studies listed here did not address the crash window problem (See [30]). In [75], Mao *et al.* presented *Osiris*, which recovers the tree only from counter values by exploiting error-correcting code bits equipped with memory. In [73], Yang *et al.* presented *cc-NVM*, which caches flushed counter values in WPQ and employs a MAC, in contrast to Osiris. In [6], Awad *et al.* presented *Triad-NVM* for the recovery of Merkle tree, which reconstructs the tree from flushed nodes. In [77], Zubair and Awad presented *Anubis*. It uses a *shadow table* in memory, which contains information on cached nodes and identifies and recovers non-updated nodes. In [72], Yang *et al.* presented *ShieldNVM*, which introduces an epoch-based mechanism to aggressively cache the metadata with the consistency preserved. In [19], Freij presented *Persistent Level Parallelism (PLP)*, which realizes an atomically persistent update of Merkle tree using a pipeline that propagates the updates. In [29], Huang and Hua presented *STAR* to achieve a reduction in the write overhead and fast recovery. It exploits the SIT lazy scheme and instant persistency for modifications in the cache. In [20], Freij *et al.* presented *Bonsai Merkle Forest*, which divides a Merkle tree into subtrees to efficiently address the crash window problem of Merkle tree awith non-volatile metadata caches.

Advanced mechanisms for metadata optimization. The SC, which is the pioneering advanced counter mechanism [71], compresses and optimizes the tree/metadata structure by splitting noncecounters into major and minor counters. In [64], Taassori *et al.* presented *Vault*, which adjusts the tree arity to reduce the frequency of counter overflow and improve the capacity of the covered region. aIn [59], Saileshwar *et al.* presented an improved counter representation compared to SC, named *Morphable counters*. It caches more counters in a line and reduces the cost of counter overflows.

Summary. Exiting crash consistency mechanisms incur a non-negligible latency overhead during nominal operation (except for SCUE), and do not work with SC-like optimizations. Crystalor efficiently realizes crash recovery while fully exploiting optimizations. It is an important future work to evaluate a combination of Crystalor with other optimization techniques than SC.

6 CONCLUSION

This study presented Crystalor, an L4-secure memory encryption mechanism. Crystalor incurs almost no latency overhead under nominal operation and achieves an efficient recovery. Although existing mechanisms (*e.g.*, SCUE) are incompatible with structural optimizations, Crystalor fully exploits its advantages and offers the same security and recoverability. We algorithmically and experimentally confirmed that Crystalor has a significant advantage over conventional mechanisms in terms of the memory overhead and execution time/latency, with a reduced recovery cost. At the algorithmic level, for protecting a 4 TB memory with ELM, Crystalor requires 29–62% fewer clock cycles per memory read/write operation than SCUE, while Crystalor and SCUE require 312 GB and 554 GB memory overheads for storing metadata, respectively (namely, Crystalor achieves a 44% reduction of memory overhead). We performed a system-level simulation using the gem5 simulator. We confirmed that Crystalor achieves a reduction in the workload execution time by at most 11.5% from SCUE. Moreover, Crystalor offers a lazy recovery owing to its cryptographic protection, which achieved a recovery that is several thousands faster than SCUE.

ACKNOWLEDGMENTS

We are grateful to anonymous shepherd and reviewers for their care and feedbacks to improve this paper. We also thank Dr. Shinya Takamaeda for his useful advice and comments. This work has been partially supported by JSPS Kakanhi Grant No. 19H21526 and 23K18457, and JST CREST No. JPMJCR19K5.

REFERENCES

[1] 2023. AMD Secure Encrypted Virtualization (SEV). https://www.amd.com/en/developer/sev.html. (2023). Visited in September 2023.

[2] 2023. Intel Optane Technology. https://www.intel.com/content/www/us/en/architecture-and-technology/optane-technology/optane-for-data-centers.html. (2023). Visited in September 2023.

[3] Roberto Avanzi. 2017. 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. Symmetric Cryptol.* 2017, 1 (2017), 4–44.

[4] Roberto Avanzi, Subhadeep Banik, Orr Dunkelman, Hector Montaner, Prakash Ramrakhyani, Francesco Regazzoni, and Andreas Sandberg. 2020. Protecting Memory Contents on ARM Cores. Real World Crypto (RWC). (2020). https://rwc.iacr.org/2020/slides/Avanzi.pdf

[5] Roberto Avanzi, Andreas Sandberg, Ionut Mihalcea, David Schall, and Héctor Montaner. 2022. SoK: Hardware-Supported Cryptographic Protection of Random Access Memory. Cryptology ePrint Archive, Paper 2022/1472. (2022).

[6] Amro Awad, Mao Ye, Yan Solihin, Laurent Njilla, and Kazi Abu Zubair. 2019. Triad-NVM: Persistency for Integrity-Protected and Encrypted Non-Volatile Memories. In *ISCA*. 104–115.

[7] Mihir Bellare, Oded Goldreich, and Shafi Goldwasser. 1995. Incremental Cryptography and Application to Virus Protection. In *STOC*. 45–56.

[8] Mihir Bellare and Chanathip Namprempre. 2008. Authenticated Encryption: Relations among Notions and Analysis of the Generic Composition Paradigm. *J. Cryptol.* 21 (2008), 469–491.

[9] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. 2011. The Gem5 Simulator. *SIGARCH Comput. Archit. News* 39, 2 (2011), 1–7.

[10] Julia Borghoff, Anne Canteaut, Tim Güneysu, Elif Bilge Kavun, Miroslav Knezevic, Lars R. Knudsen, Gregor Leander, Ventzislav Nikov, Christof Paar, Christian Rechberger, Peter Rombouts, Søren S. Thomsen, and Tolga Yalçin. 2012. PRINCE - A Low-Latency Block Cipher for Pervasive Computing Applications - Extended Abstract. In *ASIACRYPT 2012*. 208–225.

[11] Robert Buhren, Shay Gueron, Jan Nordholz, Jean-Pierre Seifert, and Julian Vetter. 2017. Fault Attacks on Encrypted General Purpose Compute Platforms. In *CODASPY '17*. 197–204.

[12] Federico Canale, Tim Güneysu, Gregor Leander, Jan Thoma, Yosuke Todo, and Rei Ueno. 2023. SCARF: A Low-Latency Block Cipher for Secure Cache-Randomization. In *USENIX Security '23*. 1937–1954.

[13] J. Lawrence Carter and Mark N. Wegman. 1979. Universal classes of hash functions. *J. Comput. System Sci.* 18, 2 (1979), 143–0154.

[14] Victor Costan and Srinivas Devadas. 2016. Intel SGX explained. Cryptology ePrint Archive, Paper 2016/086. (2016).

[15] Morris J. Dworkin. 2001. *SP 800-38A 2001 edition. Recommendation for block cipher modes of operation: Methods and techniques.* Technical Report. National Institute of Standards & Technology.

[16] Morris J. Dworkin. 2005. *SP 800-38B. Recommendation for Block Cipher Modes of Operation: The CMAC Mode for Authentication.* Technical Report.

[17] Tetsuo Endoh, Hiroaki Honjo, Koichi Nishioka, and Shoji Ikeda. 2020. Recent Progresses in STT-MRAM and SOT-MRAM for Next Generation MRAM. In *VLSI Technology*. 1–2. https://doi.org/10.1109/VLSITechnology18217.2020.9265042

[18] Ali Fakhrzadehgan, Yale N. Patt, Prashant J. Nair, and Moinuddin K. Qureshi. 2022. SafeGuard: Reducing the Security Risk from Row-Hammer via Low-Cost Integrity Protection. In *HPCA*. 373–386.

[19] Alexander Freij, Shougang Yuan, Huiyang Zhou, and Yan Solihin. 2020. Persist Level Parallelism: Streamlining Integrity Tree Updates for Secure Persistent Memory. In *MICRO*. 14–27.

[20] Alexander Freij, Huiyang Zhou, and Yan Solihin. 2021. Bonsai Merkle Forests: Efficiently Achieving Crash Consistency in Secure Persistent Memory. In *MICRO*.

1227–1240.
[21] Blaise Gassend, G. Edward Suh, Dwaine Clarke, Marten van Dijk, and Srinivas Devadas. 2003. Caches and Hash Trees for Efficient Memory Integrity Verification. In *HPCA 2003*.
[22] Peter Gazi, Krzysztof Pietrzak, and Michal Rybár. 2016. The Exact Security of PMAC. *IACR Trans. Symmetric Cryptol.* 2016, 2 (2016), 145–161.
[23] Shay Gueron. 2016. A Memory Encryption Engine Suitable for General Purpose Processors. Cryptology ePrint Archive, Paper 2016/204. (2016).
[24] Shay Gueron. 2016. Memory Encryption for General-Purpose Processors. *IEEE Security Privacy* 14, 6 (2016), 54–62.
[25] Richard H. Gumpertz. 1983. Combining tags with error codes. In *ISCA*. 160–165.
[26] J. Alex Halderman, Seth D. Schoen, Nadia Heninger, William Clarkson, William Parl, Joseph A. Calandrino, Ariel J. Feldman, Jacob Appelbaum, and Edward W. Felten. 2009. Lest we remember: cold-boot attacks on encryption keys. *Communication of the ACM* 52, 5 (2009), 91–98.
[27] W. Eric Hall and Charanjit S. Jutla. 2006. Parallelizable Authentication Trees. In *SAC 2006*. 95–109.
[28] Xijing Han, James Tuck, and Armo Awad. 2021. Dolos: Improving the Performance of Persistent Applications in ADR-Supported Secure Memory. In *MICRO*. 1241–1253.
[29] Jianming Huang and Yu Hua. 2021. A Write-Friendly and Fast-Recovery Scheme for Security Metadata in Non-Volatile Memories. In *HPCA 2021*. 359–370.
[30] Jianming Huang and Yu Hua. 2023. Root crash consistency of SGX-style integrity trees in secure non-volatile memory systems. In *HPCA 2023*. 152–164.
[31] Akiko Inoue, Kazuhiko Minematsu, Maya Oda, Rei Ueno, and Naofumi Homma. 2020. ELM: A Low-Latency and Scalable Memory Encryption Scheme. Cryptology ePrint Archive, Paper 2020/1374. (2020). Preliminary and long version of a paper with same title.
[32] Akiko Inoue, Kazuhiko Minematsu, Maya Oda, Rei Ueno, and Naofumi Homma. 2022. ELM: A Low-Latency and Scalable Memory Encryption Scheme. *IEEE Trans. Inf. Forensics Security* 17 (2022), 2628–2643.
[33] Jungi Jeong, Chang Hyun Park, Jaehyuk Huh, and Seungryoul Maeng. 2018. Efficient Hardware-Assisted Logging with Asynchronous and Direct-Update for Persistent Memory. In *MICRO*. 520–532.
[34] Arpit Joshi, Vijay Nagarajan, Stratis Viglas, and Marcelo Cintra. 2017. ATOM: Atomic Durability in Non-volatile Memory through Hardware Logging. In *HPCA*. 361–372.
[35] Yoongu Kim, Ross Daly, Jeremie Kim, Chris Fallin, Ji Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu. 2014. Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors. In *ISCA 2014*. 361–372.
[36] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. 2019. Spectre Attacks: Exploiting Speculative Execution. In *IEEE S&P 2019*. 1–19.
[37] Paul Kocher, Joshua Jaffe, and Benjamin Jun. 1999. Differential Power Analysis. In *CRYPTO 1999*. 388–397.
[38] Andrew Kwong, Daniel Genkin, Daniel Gruss, and Yuval Yarom. 2020. RAMBleed: Reading Bits in Memory Without Accessing Them. In *IEEE S&P*. 695–711.
[39] Lukas Lamster, Martin Unterguggenberger, David Schrammel, and Stefan Mangard. 2023. HashTag: Hash-based Integrity Protection for Tagged Architectures. In *USENIX Security '23*. 2797–2814.
[40] Mengya Lei, Fan Li, Fang Wang, Dan Feng, Xiaomin Zou, and Renzhi Xiao. 2022. SecNVM: An Efficient and Write-Friendly Metadata Crash Consistency Scheme for Secure NVM. *ACM Trans. Archit. Code Optimization* 19, 1 (2022), 1–26.
[41] Mengyuan Li, Luca Wilke, Jan Wichelmann, Thomas Eisenbarth, Radu Teodorescu, and Yinqian Zhang. 2022. A Systematic Look at Ciphertext Side Channels on AMD SEV-SNP. In *IEEE S&P 2022*. 337–351.
[42] Mengyuan Li, Yinqian Zhang, Huibo Wang, Kang Li, and Yueqiang Cheng. 2021. CIPHERLEAKS: Breaking Constant-time Cryptography on AMD SEV via the Ciphertext Side Channel. In *USENIX Security '21*. 717–732.
[43] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. 2018. Meltdown: Reading Kernel Memory from User Space. In *USENIX Security '18*. 973–990.
[44] Moses D. Liskov, Ronald L. Rivest, and David A. Wagner. 2011. Tweakable Block Ciphers. *J. Cryptol.* 24, 3 (2011), 588–613.
[45] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B Lee. 2015. Last-Level Cache Side-Channel Attacks are Practical. In *IEEE S&P 2015*. 244–256.
[46] Sihang Liu, Aasheesh Kolli, Jinglei Ren, and Samira Khan. 2018. Crash Consistency in Encrypted Non-volatile Main Memory Systems. In *HPCA 2018*. 310–323.
[47] Ralph C. Merkle. 1979. Method of providing digital signatures. US4309569A. (1979). https://patents.google.com/patent/US4309569.
[48] Ralph C. Merkle. 1988. A Digital Signature Based on a Conventional Encryption Function. In *CRYPTO '87*. 369–378.
[49] Kazuhiko Minematsu and Toshiyasu Matsushima. 2007. New Bounds for PMAC, TMAC, and XCBC. In *FSE*. 434–451.
[50] Mathias Morbitzer, Manuel Huber, Julian Horsch, and Sascha Wessel. 2018. SEVered: Subverting AMD's Virtual Machine Encryption. In *EuroSec 2018*. 1–6.
[51] Mathias Morbitzer, Sergej Proskurin, Martin Radev, Marko Dorfhuber, and Erick Quintanar Salas. 2021. SEVerity: Code Injection Attacks against Encrypted Virtual Machines. In *IEEE S&P Workshops*. 444–455.
[52] Pascal Nasahl, Robert Schilling, Mario Werner, Jan Hoogerbrugge, Marcel Medwed, and Stefan Mangard. 2021. CrypTag: Thwarting Physical and Logical Memory Vulnerabilities Using Cryptographically Colored Memory. In *ACM ASIACCS*. 200–212.
[53] Qi Pei and Seunghee Shin. 2021. Efficient Split Counter Mode Encryption for NVM. In *ISPASS 2021*. 93–95.
[54] Qi Pei and Seunghee Shin. 2021. Improving the Heavy Re-encryption Overhead of Split Counter Mode Encryption for NVM. In *ICCD*. 425–432.
[55] Moinuddin K. Qureshi. 2018. CEASER: Mitigating Conflict-Based Cache Attacks via Encrypted-Address and Remapping. In *MICRO*. 775–787.
[56] Phillip Rogaway. 2002. Authenticated-Encryption with Associated-Data. In *ACM CCS*. 98–107.
[57] Phillip Rogaway. 2004. Efficient Instantiations of Tweakable Blockciphers and Refinements to Modes OCB and PMAC. In *ASIACRYPT 2004*. 16–31.
[58] Phillip Rogaway, Mihir Bellare, and John Black. 2003. OCB: A Block-Cipher Mode of Operation for Efficient Authenticated Encryption. *ACM Trans. Inf. Syst. Secur.* 6, 3 (2003), 365–403.
[59] Gururaj Saileshwar, Prashant J. Nair, Prakash Ramrakhyani, Wendy Elsasser, Jose A. Joao, and Moinuddin K. Qureshi. 2018. Morphable counters: enabling compact integrity trees for low-overhead secure memories. In *MICRO*. 416–427.
[60] Gururaj Saileshwar, Prashant J. Nair, Prakash Ramrakhyani, Wendy Elsasser, and Moinuddin K. Qureshi. 2018. SYNERGY: Rethinking Secure-Memory Design for Error-Correcting Memories. In *HPCA 2018*. 454–465.
[61] Bianca Schroeder, Eduardo Pinheiro, and Wolf-Dietrich Weber. 2009. DRAM errors in the wild: a large-scale field study. *ACM SIGMETRICS Performance Evaluation Review* 37, 1 (2009), 193–204.
[62] SNIA. 2014. NVDIMM Messaging and FAQ. (2014). https://www.snia.org/sites/default/files/NVDIMM%20Messaging%20and%20FAQ%20Jan%2020143.pdf
[63] Solid State Storage Initiative. 2014. NVDIMM Messaging and FAQ. (Jan. 2014).
[64] Meysam Taassori, Ali Shafiee, and Rajeev Balasubramonian. 2018. VAULT: Reducing Paging Overheads in SGX with Efficient Integrity Verification Structures. In *ASPLOS*. 665–678.
[65] Rei Ueno, Sumio Morioka, Naofumi Homma, and Takasumi Aoki. 2016. A High Throughput/Gate AES Hardware Architecture by Compressing Encryption and Decryption Datapaths—Toward Efficient CBC-mode Implementation. In *CHES*. 538–558.
[66] Rei Ueno, Sumio Morioka, Noriyuki Miura, Kohei Matsuda, Makoto Nagata, Shivam Bhasin, Yves Mathieu, Tarik Graba, Jean-Luc Danger, and Naofumi Homma. 2020. High Throughput/Gate AES Hardware Architectures Based on Datapath Compression. *IEEE Trans. Comput.* 69, 4 (2020), 534–548.
[67] Thomas Unterluggauer, Mairo Werner, and Stefan Mangard. 2019. MEAS: memory encryption and authentication secure against side-channel attacks. *J. Cryptogr. Eng.* 9 (2019), 137–158.
[68] Xueliang Wei, Dan Feng, Wei Tong, Jingning Liu, and Liuqing Ye. 2020. MorLog: Morphable Hardware Logging for Atomic Persistence in Non-Volatile Main Memory. In *ISCA 2020*. 610–623.
[69] Mario Werner, Thomas Unterluggauer, Lukas Giner, Michael Schwarz, Daniel Gruss, and Stefan Mangard. 2019. ScatterCache: Thwarting Cache Attacks via Cache Set Randomization. In *USENIX Security '19*. 675–692.
[70] Luca Wilke, Jan Wichelmann, Mathias Morbitzer, and Thomas Eisenbarth. 2020. SEVurity: No Security Without Integrity: Breaking Integrity-Free Memory Encryption with Minimal Assumptions. In *IEEE S&P 2020*. 1483–1496.
[71] Chenyu Yan, D. Englender, M. Prvulovic, B. Rogers, and Yan Solihin. 2006. Improving Cost, Performance, and Security of Memory Encryption and Authentication. In *ISCA*. 179–190.
[72] Fan Yang, Youmin Chen, Haiyu Mao, Youyou Lu, and Jiwu Shu. 2020. ShieldNVM: An Efficient and Fast Recoverable System for Secure Non-Volatile Memory. *ACM Trans. Storage* 16, 2 (2020), 1–31.
[73] Fan Yang, Youyou Lu, Youmin Chen, Haiyu Mao, and Jiwu Shu. 2019. No Compromises: Secure NVM with Crash Consistency, Write-Efficiency and High-Performance. In *DAC 2019*. 1–6.
[74] Yuval Yarom and Katrina Falkner. 2014. FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack. In *USENIX Security '14*.
[75] Mao Ye, Clayton Hughes, and Amro Awad. 2018. Osiris: A Low-Cost Mechanism to Enable Restoration of Secure Non-Volatile Memories. In *MICRO*. 403–415.
[76] Jian Zhou, Amro Awad, and Jun Wang. 2020. Lelantus: fine-granularity copy-on-write operations for secure non-volatile memories. In *ISCA 2020*. 597–607.
[77] Kazi Abu Zubair and Amro Awad. 2019. Anubis: ultra-low overhead and recovery time for secure non-volatile memories. In *ISCA*. 157–168.
[78] Kazi Abu Zubair, Sudhanva Gurumurthi, Vilas Sridharan, and Amro Awad. 2021. Soteria: Towards Resilient Integrity-Protected and Encrypted Non-Volatile Memories. In *MICRO*. 1214–1226.
[79] Pengfei Zuo, Yu Hua, and Yuan Xie. 2019. SuperMem: Enabling Application-transparent Secure Persistent Memory with Low Overheads. In *MICRO*. 479–492.