Software-Based Off-Chip Memory Protection for RISC-V Trusted Execution Environments

Gui Andrade
guiand@berkeley.edu
UC Berkeley

Dayeol Lee
dayeol@berkeley.edu
UC Berkeley

David Kohlbrenner
dkohlbre@berkeley.edu
UC Berkeley

Krste Asanović
krste@berkeley.edu
UC Berkeley

Dawn Song
dawn@berkeley.edu
UC Berkeley

Abstract

We present a software-based memory protection for RISC-V enclaves. Our system provides confidentiality and integrity guarantees for the enclave pages when an attacker can arbitrarily read or write to external memory. Unlike hardware-based implementations such as Memory Encryption Engine (MEE) in Intel SGX, our software-based implementation requires no additional security-specific hardware. We use instead only a small on-chip scratchpad as our trusted memory region. This results in a portable and highly adaptable solution, applicable to primarily embedded contexts. Our approach is implemented as a module for Keystone, which is an open-source framework for RISC-V enclaves.

1 Introduction

Many kinds of hardware-software systems require some measure of secret keeping, even from the user in possession of the hardware. Game systems and TVs implement copy protection schemes; the Apple iPhone secures user data such that even law enforcement cannot access it; car manufacturers (strive to[13]) secure safety-critical equipment from tampering.

In all of these scenarios, the adversary has non-trivial *physical* access to the device. Thus, signals running from an SoC package to off-chip DRAM may be tapped by a sufficiently determined attacker [4], rendering off-chip memory unsuitable for storing security-critical secrets such as DRM decryption keys.

In this paper, we demonstrate defenses against such adversaries in the context of a Trusted Execution Environment (TEE). We propose, implement, and evaluate a scheme for encryption and integrity protection of all Secure Enclave accesses to DRAM.

There are several existing systems to accomplish similar goals, the most well known being Intel's SGX and Memory Encryption Engine (MEE). These solutions are high performance and have significant benefits, but require dedicated additional hardware to support. We instead propose leveraging the Keystone TEE framework's model, where applications in enclaves may handle their own paging entirely from software to add support for encryption and integrity

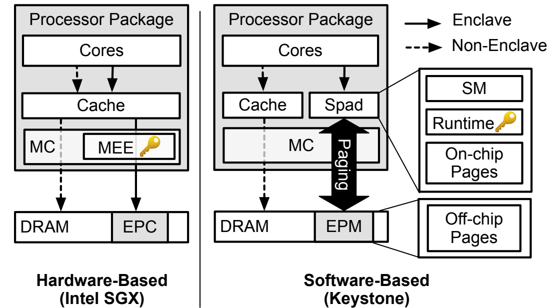

Figure 1. Our software-based off-chip memory protection compared to the hardware-based one in Intel SGX. Intel SGX relies on hardware extension of the memory controller (MC), whereas our approach relies on an on-chip scratchpad (Spad) and the runtime software inside the enclave.

protection. A similar approach, but for full operating systems and lacking integrity protection, was proposed and evaluated in Chen et al [1]. Our system is applicable to cases where an application is expected to rarely grow in size past an available scratchpad memory, but cannot be guaranteed not to. In these cases, the costs of software encryption and integrity protection are well amortized across the lifetime of the application and avoid the costs associated with extensive hardware additions.

2 Background

Trusted Execution Environments (TEEs) are a wide-spread security mechanism used for establishing trust in remote systems. TEEs provide strong security guarantees (confidentiality and integrity) for applications running inside a TEE provided partition, often called an *enclave*. TEEs are expected to protect enclaves against adversarial privileged software (including an OS), other enclaves, and in some cases adversaries with physical access.

2.1 Keystone

This paper describes a scheme analogous to those commonly implemented in security extensions to commodity CPUs[5][8], but implemented completely in software for RISC-V. This

memory encryption and integrity scheme is implemented within Keystone, an open-source TEE framework for RISC-V systems developed at UC Berkeley[10]. One of the notable advantages RISC-V brings to Keystone is allowing each enclave to handle its own page faults entirely in-enclave.

Keystone leverages RISC-V's Machine-mode and Physical Memory Protection features to operate decoupled from, and effectively transparent to, a host operating system. It provides a lean Security Monitor (SM) which executes in M-mode, handling management of enclaves and context switching between the host OS and enclaves as requested. Keystone also provides a modular enclave Supervisor-mode runtime to handle interrupts, memory management, and to shim essential Linux syscalls. This runtime is generally small, and supports only a few needed features for the enclaved application. Runtimes are customized and provided by the enclave application developer, and are not part of the trusted Security Monitor.

Our proposed memory protection scheme operates as an additional set of optional modules for the Keystone Eyrie runtime. We use the existing paging module to determine when and which pages must be evicted from the on-chip memory or retrieved from DRAM. Without our additional protections, any sensitive data in these pages would otherwise be visible to some physical adversaries.

3 Threat Model

We consider an adversary who has physical access to our device, and some level of capability to interact with off-chip components, notably DRAM. There are two primary threats to consider when defending against a physical adversary with DRAM access.

Simpler is the 'cold-boot' attacker [9], who has the capability to halt the device and examine all DRAM content. They do not, critically, have the ability to modify this DRAM content and then continue execution of the system. For defending against this adversary all that is required is confidentiality for all data residing in DRAM.

More complex is an active attacker who has the capability to mutate all DRAM content during execution. This requires significantly more hardware investment and expertise than the cold-boot adversary. In this case, the defender must ensure both confidentiality and integrity for all DRAM memory content. Note that we do not guarantee the ability to recover from adversarial mutation of DRAM content, simply that we can detect it and halt safely.

Classical software adversaries (other processes, enclaves, compromised operating systems, etc.) are handled by the standard Keystone TEE guarantees, and are out of scope for this paper.

Cache-based side-channel attacks are not applicable in this model, as we repurpose the shared L2 cache into a scratchpad. Side-channel attacks based on coarse-grained

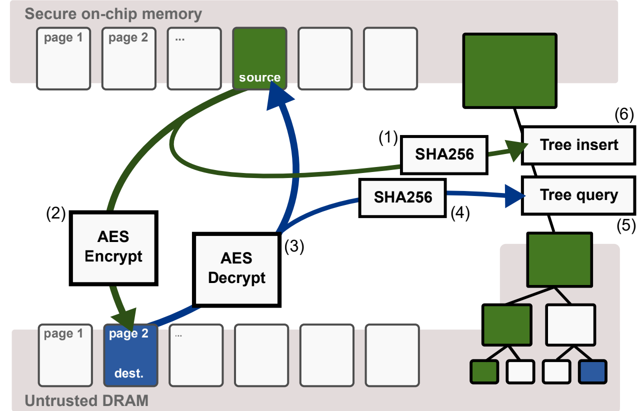

Figure 2. Dataflow for our protection scheme during a page swap, including both encryption and integrity protection.

data-dependent memory access patterns or timing are out of scope of our work. We also assume the target is able to tolerate Denial of Service attacks: at any time the adversary may simply turn off the device or have a compromised OS stop scheduling the enclave to execute.

4 Software-Based Memory Protection Design

Our design uses several distinct components; a generic paging system, an integrity tree, and page encryption. While we implement all components as software modules in Keystone, these can be integrated and accelerated independently with hardware extensions where available.

When the system's on-chip secure memory is exhausted, it uses Keystone's enclave self-paging system to manage pages being swapped from on-chip to DRAM and back. These pages, when stored in DRAM, are encrypted using AES with a per-enclave key and integrity protected using hashes stored in a Merkel-tree style construction for efficiency.

Figure 2 shows the complete set of operations that occur on a page swap between on-chip memory and DRAM. In the case of an enclave application page-fault due to the accessed page being stored in off-chip DRAM: we hash (1) and encrypt (2) the page leaving the scratchpad, both using a counter value unique to each page (not shown), and then insert it into off-chip storage. The incoming page is then decrypted (3), hashed (4), integrity checked against the tree (5), and finally the outgoing page's integrity information is stored in the tree (6).

4.1 Hardware Requirement

We assume a standard RISC-V platform which is capable of running a Keystone-based [10] TEE system. Keystone provides basic memory isolation using RISC-V's physical memory protection (PMP), as well as remote attestation and enclave management. We assume that the hardware has an

authenticated root of trust and is running a Keystone security monitor (SM), which is trusted machine-mode software.

The only non-standard hardware requirement for our off-chip memory protection is a small on-chip scratchpad memory, mapped to a physical address range that is not accessible by supervisor or user software. Such a scratchpad can be instantiated from an M-mode configurable Loosely Integrated Memory (LIM) such as the LIM available on SiFive's FU540 SoC. Existing hardware-based approaches [5, 8] to memory encryption use extensions in microarchitectural components such as memory controller to store needed on-chip state. Those extensions can protect the off-chip memory transparently from the software by encrypt or decrypt the cache lines on-the-fly, and by initiating additional memory transactions if needed.

As our solution is software based, we can use a generic scratchpad rather than dedicated on-chip memory, split into partitions as needed for different components. Thus, the Keystone Security Monitor as well as the enclave under protection must be loaded entirely into the available scratchpad memory.

4.2 Enclave Self-Paging

Keystone allows each enclave to manage its own virtual memory mapping by completely releasing the memory management unit (MMU) to the enclave during execution. The Keystone Eyrie runtime resides exclusively in on-chip memory during execution, and is responsible for paging the application content into and out of the scratchpad. On boot up on an enclave, the runtime queries the SM for a DRAM storage region to use as a swap space.

Whenever a page fault occurs, this paging mechanism will evict a page from on-chip memory to the backing store, and optionally load a demanded page to the same physical address. In pseudocode:

Algorithm 1 SwapPage(p_{dram}, p_{onchip}, swap)

```
if swap then
    tmp ← p_dram
end if
p_dram ← p_onchip
if swap then
    p_onchip ← tmp
end if
```

4.3 Memory Encryption

Confidentiality for memory leaving the scratchpad is accomplished by encrypting all page content with AES-CTR. CTR Mode (CM) requires some important security considerations [11]: (1) a specific key and counter combination may never be repeated with different plaintexts, and (2) the system is *malleable* — that is, an attacker may freely manipulate decrypted output with some knowledge of the plaintext.

To minimize the amount of memory required for bookkeeping, we randomly generate an enclave-specific key on initialization, which is used for every encrypted page and must be kept private inside the on-chip memory. The counter value for each page is randomly initialized before use, and incremented whenever a page is swapped. Importantly, a single 4096-byte page contains 256 AES blocks, so the AES-CTR operation itself increments by some value greater than 256 in between pageouts to prevent reuse of a keypair for two blocks. For read security without integrity protection, it is acceptable to store these counters in off-chip memory as an attacker reading the IV will still be unable to decrypt without the secret encryption key. See Algorithm 2 for the complete operation.

Algorithm 2 CryptoSwapPage(p_{dram}, p_{onchip}, swap)

```
if swap then
    tmp ← p_dram
end if
ctr_n ← 32 × Uniform
ctr_p ← CtrStoreSwap(ctr, key = &p_dram)
p_dram ← AesEncrypt(p_onchip, key, ctr_n)
if swap then
    p_onchip ← AesDecrypt(p_tmp, key, ctr_p)
end if
```

4.4 Page Integrity

To prevent replay attacks, an attacker must be prevented from reloading both a stale page's contents to the page store, and a stale counter to the counter store. By preventing the replay of any page's counter, the attacker will be unable to replay the page swap and obtain decipherable results (not accounting for the AES malleability property, which is addressed below).

The simplest solution to protecting these counters would be to place the counter store inside on-chip memory, where attackers are unable to access it. Unfortunately, this becomes space prohibitive; with an off-chip backing store of potentially tens of thousands of pages, this counter store would occupy hundreds of kilobytes of limited secure memory. Similarly, storing only a single hash of the entire counter store results in a prohibitive performance cost of $O(N)$ hashes per page swap where N is the number of pages swapped by the enclave.

We instead reduce the number of hashes per swap to $O(\log N)$, and additionally solve the AES malleability problem, by use of a *Hash Tree* data structure. We store, for each leaf, a hashed tuple of the page contents, the page's counter, and the page's backing address in DRAM.

4.4.1 Hash Trees.

We adopt a similar approach to the Merkle tree data structure[12], with a few critical differences. Merkle trees are designed for use in distributed cryptography,

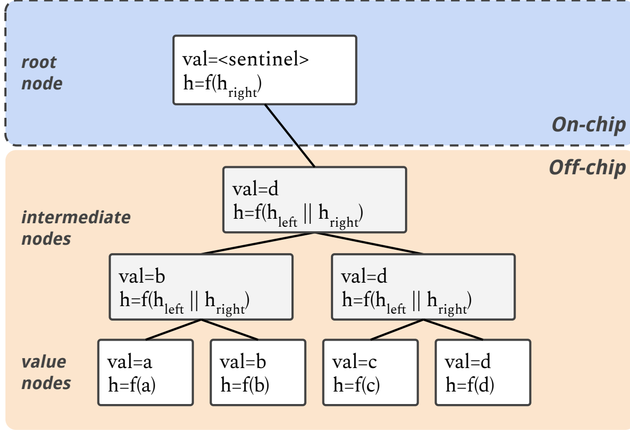

Figure 3. A 4-entry hash tree. For simplicity, values are stored only at the leaves of the tree. Under this structure, only the root node must be trusted for effective integrity protection.

where multiple clients need to validate the authenticity of some data with regards to a root of trust. Thus, a Merkle tree consists of authenticated nodes — certificates, signing other certificates, all the way up to a publicly known root of trust — such that users can walk up the tree to validate any given certificate.

Our problem is the inverse: given known values for our nodes, and a private root of trust, how can we ensure the validity of our nodes without privately storing all the known values? This question has been explored many times by authors discussing memory authentication schemes, usually with hardware support[7][6].

For our purposes, we wish to minimize the amount of secure memory needed to form a root of trust, with which we can validate page swaps to/from the enclave. Our implementation of a hash tree stores hashes at the leaves of the tree, and each non-leaf node stores a hash of its immediate children's hashes (see figure 3). So inserting to the tree implements a series of recursive hashes, ending at the root. Querying the tree recursively checks these hashes[1].

Crucially, for our purposes, the root of the tree must remain in secure memory. The hash stored here may only be written by the enclave, and so if some node P of the tree has been tampered with by an outside actor, the enclave's attempt to query the tree at, below, or adjacent to P will cause a hash mismatch.

Insertion takes a two-step approach, where the algorithm must first recurse[1] down to an appropriate intermediate node, insert a new leaf below it, and propagate newly-computed hashes back up the tree. As we walk down the tree, we validate intermediate nodes. This is to ensure we don't implicitly

[1]In C, this algorithm is implemented iteratively for predictable stack usage analysis.

Algorithm 3 Query(root, x, h_x)

$h_l \leftarrow$ root.left.hash
$h_r \leftarrow$ root.right.hash
$h_n \leftarrow$ root.hash
if x = root.val **then**
  return $h_n = h_x$
end if
if $h_n \neq$ Sha256(h_l, h_r) **then**
  return False
end if
next $\leftarrow$ Traverse(root, x)
return Query(next, x, h_x)

trust nodes adjacent to the insertion path when propagating hashes upward.

Algorithm 4 Insert(root, x, h_x)

if root.leaf **then**
  if root.val = x **then**
    root.hash $\leftarrow h_x$
  else
    oldroot $\leftarrow$ *root
    next, sibling $\leftarrow$ Traverse(root, x)
    *next $\leftarrow$ MakeNode(x, h_x)
    *sibling $\leftarrow$ oldroot
    root.hash $\leftarrow$ Sha256(left.hash, right.hash)
  end if
else
  $h_l \leftarrow$ root.left.hash
  $h_r \leftarrow$ root.right.hash
  if root.hash $\neq$ Sha256(h_l, h_r) **then**
    return Error
  end if
  if root.val < x **then**
    $h_l \leftarrow$ Insert(root.left, x, h_x)
  else
    $h_r \leftarrow$ Insert(root.right, x, h_x)
  end if
  root.hash $\leftarrow$ Sha256(h_l, h_r)
end if
return root.hash

Finally, some special timing considerations must be made so that no race conditions appear between stages of either the Insert or Query operations. With a literal implementation of the algorithms as described here in pseudocode, an attacker may be able to exploit the race between writing a hash at depth N of the tree, and then reading it at depth $N-1$ for the intermediate node update. With our implementations, which are completely iterative, two layers of the tree are resident in secure stack memory at any given time. For Query, this allows comparing hashes and iterating down the tree to be a

single atomic operation with respect to DRAM. For Insert, comparing values and iterating is similarly atomic, and then iterating back up the tree is atomic with respect to each modified node.

This access pattern does not prevent multiple threads from racing. Rather, it prevents an adversary from injecting data into the tree which will cause false positive integrity checks. Should an attacker modify data in a tree's node at any point before, during, or after a write, we expect some later query involving that node to fail its integrity check.

4.5 Comparison to SGX

Intel SGX[8] is a TEE system, available for use in many commercial Intel processors today. As part of its protections, SGX provides a similar set of off-chip memory defenses as in this paper. It protects memory content against a passive adversary snooping the memory bus, as well as an active adversary injecting changes into DRAM. To do so, Intel SGX includes both a memory engine for confidentiality, and an integrity tree with hardware support.

For encryption, Intel employs memory "version counters" that fulfill the same role as our AES-CTR counters, namely to track changes to memory over time in order to prevent replay attacks. SGX operates on 512-bit chunks of memory at a time, using a modified AES-CTR 128 with four counters per chunk. Our scheme is more conservative, using AES-256, and we are limited by the 4K page boundary as our minimum chunk size.

For integrity protection, Intel too uses a tree with its root isolated in on-chip SRAM, though SGX uses 56-bit Carter-Wegman MACs instead of hashes. Again, this paper uses a more conservative scheme, coming at a performance penalty; we opt to use a SHA256 Merkle tree.

The SGX white paper provides practical justifications for the security of these less conservative choices. Future versions of our implementation can similarly evaluate loosening some of the security guarantees for the sake of performance.

5 Implementation

We implemented the complete page encryption and hash-tree-based page integrity protection system described in Section 4 on top of the paging module for Keystone's Eyrie runtime. This implementation is publicly available on the Keystone github[1].

To render this paper's hash tree Query/Insert functions iterative, a few transformations were done. For Query, we note that the function ends with a tail call, and use tail call elimination to transform it into a loop with space complexity $O(1)$. For Insert, the recursive call is not a tail call, so we must use a stack on-chip to store intermediate data. On average, we need to store $O(\log(P \text{ nodes}))$. We specify the maximum tolerable tree depth as a build-type parameter, and store this insertion stack in the runtime's `.data` section.

Our implementation is written entirely in C, and uses externally sourced implementations of SHA256 and AES (1200 lines-of-code). Our hash-tree implementation is 300LoC, and the page encryption integration adds 100LoC to the existing Eyrie paging module.

5.1 Memory and Performance

The amount of on-chip and off-chip memory used for encryption and integrity protection are both functions of the size of pageable memory. For on-chip overhead, we pay a constant space penalty of one page for the temporary storage during a swap. On top, insertion stores $(H+W)\log(P)$ bytes, where P is the total number of swapped pages, $H = 32$ is the hash size for SHA256, and $W = 8$ is for a traversed parent pointer (for walking back up the tree). For the off-chip overhead of our hash tree, we store approximately $2NP$ bytes, where $N = 64$ is the size of a hash tree node (including the hash, node value, and pointers to left/right children). Our pageout counter store, also off-chip, comprises P 64-bit counters, or $8P$ bytes. The practical limitation for how many pages can be safely stored off-chip is thus not limited by the storage space on-chip but by the performance impact of tree depth and available off-chip storage.

6 Summary of Security Analysis

We protect against (1) an attacker both attempting to snoop the off-chip memory, as well as (2) an attacker modifying data stored off-chip, including by replaying old pages.

The read-only or cold-boot adversary is thwarted by our use of AES-CTR 256 to encrypt any memory leaving the chip. CM provides adequate protection if we never reuse a (key, counter) tuple, and this is ensured by a randomly generated key on boot, coupled with a unique counter per page which increments on each page swap. Counter overflow should be handled by terminating the enclave safely.

The active adversary will result in eventual failure once the enclave attempts to swap in the modified page. A hash over (page contents, pageout counter, page address) is checked against the hash tree. The tree's integrity is protected by recursive layers of hash checking, and its final layer is checked against a hash stored on-chip, out of attacker reach.

The failure mode is the immediate safe self-shutdown of the enclave. Thus an attacker may learn that a specific page is being queried by the enclave — though as we presume they have access to the DRAM bus, they may simply read the bus to see when that page is swapped in or out. But the attacker cannot use such a failure to learn information, as the enclave does not proceed to execute after a page integrity violation.

[1] https://github.com/keystone-enclave/keystone-runtime/pull/27

benchmark	# Page Faults	Paging Overhead
qsort	285147	128.6×
aes	59716	16.4×
norx	58834	28.6×
miniz	18341	1.8×
bigint	168	1.0×
sha512	0	1.0×
dhrystone	0	1.0×

Table 1. Number of page faults reported by Lee *et al.*[10] and paging overhead over baseline for a 1MB on-chip scratchpad

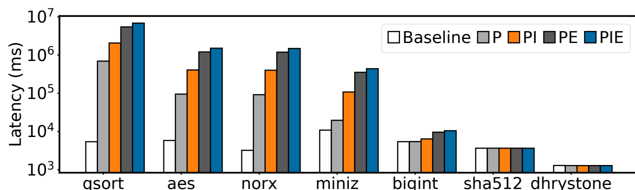

Figure 4. Average execution time for rv8 benchmark tests under various configurations. Note log scale.
Baseline: Not in enclave, **P**: In enclave scratchpad + Paging support to DRAM, **PI**: P + Hash tree Integrity protection, **PE**: P + Encryption of pages, **PIE**: P + Integrity and Encryption

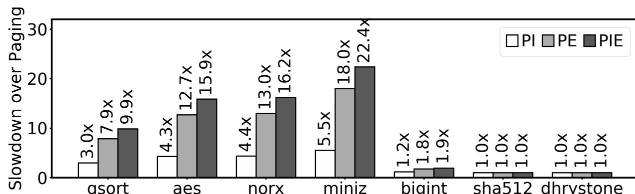

Figure 5. Multiplicative slowdown of PI, PE, and PIE configurations over plain Paging configuration. Linear scale.

7 Evaluation

For our experiments, we used a SiFive HiFive Unleashed development board, with an FU540 SoC. On this, we run a Keystone-derived TEE with support for managing the FU540's LIM enabled. We allocate a default of 1MB of an on-chip scratchpad to the enclave during test and 260MB of swap space in DRAM. Test results are averaged across 10 runs for each configuration.

We ran eight tests from the rv8 benchmark suite for RISC-V[2], the results of seven of which are shown in figures 4 and 5. *primes*, is excluded as the total execution time under our test configurations was in excess of 10 hours per-test.

We expected many tests to perform orders of magnitude slower when any form of paging was necessary, as established in the original Keystone paper[10], and our findings were consistent with this expectation. More interesting was the additional cost of enabling extra security features: Page Encryption (PE), Page Integrity (PI), and Page Integrity with Encryption (PIE). By running each of these configurations independently impact of each feature is clear.

As figure 5 clearly demonstrates, encryption/decryption accounted for the bulk of the additional cost among these features. Most page swaps involved two passes of AES-CTR 256, one to encrypt the outgoing page, and one to decrypt the incoming page. The software implementation we used was not particularly optimized for speed[3], nor did the hardware feature any crypto acceleration or even instruction-level parallelism (the FU540 SoC used to measure uses a single-issue, in-order pipeline[14]).

8 Future Work

For simplicity of implementation, the hash tree presented in this paper is a classical binary search tree. We allocate backing pages sequentially, so to ensure the BST is balanced, we use an injective key scrambling function. We would like to explore alternative tree structures that self-balance (red-black or B-trees, for simple examples), and how performance characteristics change.

Also discussing memory authentication using a hash tree, Gassend, Suh, et. al. described a tree node caching scheme that reduced their runtime overhead from near a factor of 10, down to about 25%[7]. Under their system, the tree's most recently used intermediate node hashes are cached to allow the tree query mechanism to short circuit in the average case. Implementing caching this way in Keystone is likely to spare significant processing time during a page swap.

Additionally, we would like to explore integrating hardware cryptography blocks or ISA extensions as they become available for commercial RISC-V boards. Specifically, whether hardware implementations of AES or SHA can match the total latency of a software implementation, given burst sizes of 4096 bytes (1 page). With hardware blocks, we may also be able to encrypt and hash a page in parallel, instead of sequentially. Starting with a complete software implementation allows for finding a balance between the speed of hardware acceleration and the flexibility of software.

9 Conclusion

We designed, implemented, and evaluated a software off-chip memory encryption and integrity protection system for Keystone-based RISC-V Trusted Execution Environments. Our novel design uses a simple modular approach allowing for each component to be enabled separately and evaluated independently. This approach allows a Keystone TEE system to choose an appropriate set of defenses for a given application and adversary, without any modifications to hardware or core TEE systems. For applications with a limited memory footprint, software-based encryption and protection offers a compelling alternative to expensive hardware extensions.

References

[1] Xi Chen, Robert P Dick, and Alok Choudhary. 2008. Operating system controlled processor-memory bus encryption. In *DATE*.

[2] Michael Clark. 2019. rv8-bench. https://github.com/rv8-io/rv8-bench/

[3] Brad Conte. 2012. crypto-algorithms. https://github.com/B-Con/crypto-algorithms

[4] DanglingPointr. 2019. Hacking The 3ds IV: Hardware attacks. https://pedro-javierf.github.io/devblog/hacking3ds4/

[5] Advanced Micro Devices. 2020. *AMD SEV-SNP: Strengthening VM Isolation with Integrity Protection and More*. Technical Report.

[6] Reouven Elbaz, David Champagne, Catherine Gebotys, Ruby Lee, Nachiketh Potlapally, and Lionel Torres. 2009. Hardware Mechanisms for Memory Authentication: A Survey of Existing Techniques and Engines. *Transactions on Computational Science* 4 (01 2009), 1–22. https://doi.org/10.1007/978-3-642-01004-0_1

[7] B. Gassend, G. E. Suh, D. Clarke, M. van Dijk, and S. Devadas. 2003. Caches and hash trees for efficient memory integrity verification. In *The Ninth International Symposium on High-Performance Computer Architecture, 2003. HPCA-9 2003. Proceedings.* 295–306. https://doi.org/10.1109/HPCA.2003.1183547

[8] Shay Gueron. 2016. *A Memory Encryption Engine Suitable for General Purpose Processors*. Technical Report. Intel Corporation, Intel Development Center, Israel.

[9] J Alex Halderman, Seth D Schoen, Nadia Heninger, William Clarkson, William Paul, Joseph A Calandrino, Ariel J Feldman, Jacob Appelbaum, and Edward W Felten. 2009. Lest we remember: cold-boot attacks on encryption keys. *Commun. ACM* (2009).

[10] Dayeol Lee, David Kohlbrenner, Shweta Shinde, Krste Asanović, and Dawn Song. 2020. Keystone: An Open Framework for Architecting TEEs. In *Proceedings of the Fifteenth European Conference on Computer Systems (EuroSys'20)*.

[11] John McGrew. 2002. Counter Mode Security: Analysis and Recommendations. (November 2002).

[12] Ralph C. Merkle. 1990. A Certified Digital Signature. In *Advances in Cryptology — CRYPTO' 89 Proceedings*, Gilles Brassard (Ed.). Springer New York, New York, NY, 218–238.

[13] Sen Nie, Ling Liu, and Yuefeng Du. 2017. Free-fall: Hacking Tesla from Wireless to CAN Bus *(Black-Hat)*.

[14] SiFive. 2018. SiFive FU540-C000 Manual.