Performance-Enhanced Integrity Verification for Large Memories

Yanan Guo
University of Pittsburgh
Pittsburgh, PA, USA
yag45@pitt.edu

Andrew Zigerelli
University of Pittsburgh
Pittsburgh, PA, USA
anz37@pitt.edu

Yueqiang Cheng
NIO
San Jose, CA, USA
strongerwill@gmail.com

Youtao Zhang
University of Pittsburgh
Pittsburgh, PA, USA
zhangyt@cs.pitt.edu

Jun Yang
University of Pittsburgh
Pittsburgh, PA, USA
juy9@pitt.edu

***Abstract*—Modern memory systems on cloud servers are vulnerable to many security threats including memory integrity attacks. To protect user data, secure infrastructures like Intel SGX have deployed cryptographic memory protection mechanisms such as MAC+integrity tree. However, using an integrity tree can significantly increase the latency of memory data accesses and thus decrease performance. Although there are many tree optimizations that have obtained performance improvements, the overhead of using an integrity tree remains high. This overhead has become even worse with the recent increase of cloud memory size, making integrity protection impractical on cloud servers.**

We argue that most prior optimization works do not consider the architecture features of cloud server processors and thus miss the opportunity for further performance improvement. Based on this, we propose Parallelized-Compressed-Prefetched-Tree (PCPT), a tree optimization design tailored for cloud servers. PCPT consists of three methods including 1) parallelizing the memory accesses along a tree path to shorten the critical path, 2) compressing data cache lines and storing counters together with the data to reduce meta data accesses, and 3) prefetching in a tree-aware manner. We evaluate PCPT using 25 benchmarks drawn from 3 suites, and we show that PCPT improves the performance of the state-of-the-art by over 35%.

I. Introduction

Cloud computing has been one of the most promising Internet service hosting technologies. However, outsourcing the processing of data to a third-party platform also increases security concerns. The cloud is owned and thus fully controlled by the system provider, which includes the system software (e.g., the hypervisor) that enables the sharing of a hardware platform among users, and the hardware itself. If either the hypervisor or any hardware device is compromised due to hidden vulnerabilities, the provider will become malicious [10]. Additionally, cloud users may be able to exploit system flaws, utilizing them to obtain/modify other co-running users' data.

An essential part of protecting cloud users' data is to protect their data integrity, i.e. to guarantee that a user always sees the data it last wrote. This is important because the memory system on cloud servers is vulnerable to integrity attacks: the hypervisor can directly modify a user's memory data since it has privileged access permission, and a non-privileged user could launch Rowhammer attacks to indirectly modify other users' data [15], [25], [52], [53], [54]. In addition, an attacker with physical access to the memory system could manipulate the memory bus and modify the data [18] transferred between CPU and memory. Given these attack possibilities, it is important to verify the data integrity when it is loaded from memory devices, and processor vendors have put out many secure infrastructures to achieve this (e.g., Intel SGX [11], [21], [30]).

Memory integrity protection mechanisms typically consist of a *MAC+integrity tree* structure [13], [16], [37], [38], [43], [46]: each data block in memory is protected by a Message Authentication Code (MAC), which is essentially a keyed hash of the data block. When a data block is loaded from memory, the MAC is used to detect unauthorized data modifications. However, an attacker with precise control of the memory bus might be able to return an old version of data+MAC to the CPU, and cause a replay attack [18]. To prevent this, a hash tree (e.g., Bonsai Merkle Tree [37]) is built over the version id of each data block (which is also used as encryption counter), and the root node of the tree is stored on the CPU chip. To verify the freshness of a data block, we need to fetch all the nodes along a vertical tree path from the bottom up, until reaching the root node. Unfortunately, these extra memory accesses for MAC and tree nodes can introduce significant performance overhead since they ❶ lengthen the critical path of execution, and ❷ increase the memory bus traffic which slows down memory accesses on average.

Many prior works have been proposed to reduce the overhead of integrity trees. Synergy [40] stores MACs in the place usually reserved for ECCs in ECC memory, and stores ECCs elsewhere in memory. Later, a small dedicated ECC cache was proposed [47]. These two designs together can effectively reduce the overhead of fetching/updating MACs. In addition, to reduce the tree overhead, Vault [48] uses a fat and variable-arity tree which can decrease the tree depth; ITESP [47] builds one separate tree for each application to increase tree cache utilization; Morphable counters [39] dynamically adjusts counter size to reduce counter overflow frequency. Although these optimization works obtain performance improvements, the overhead caused by tree node accesses remains high, especially when the cloud server has a large memory size and thus requires a very deep tree.

Our goal is to further combat the performance barrier to the adoption of integrity tree in clouds. We argue that most previous tree optimization works assume a standalone system and are not tailored for cloud server architectures; thus, they miss important opportunities for further overhead containment. In this work, we propose a new tree optimization design named

Parallelized-Compressed-Prefetched-Tree (PCPT). PCPT consists of performance optimization methods that suit cloud server processors with large memories.

Specifically, we first notice that most recent cloud server processors have many memory channels and high-speed communication paths among memory controllers. However, currently, nodes in an integrity tree are not organized to leverage the memory access parallelism introduced by this architecture. Thus, we propose to reorganize the tree to map the data and nodes in each vertical tree path to different channels, to fully utilize the parallelism and reduce the total integrity verification latency for a memory data block. Second, we observe that most of the increased memory bus pressure comes from the accesses to the lowest-level counters (i.e. L0 counters), and that modern applications' memory data are highly compressible. Thus, we propose to lightly compress each memory data cache line (if possible), and store a copy of its L0 counter together with the data in the same cache line. Consequently, for a compressed data cache line, we do not need a separate memory transaction for fetching its L0 counter. Third, we discover that an integrity tree negatively impacts hardware prefetchers which are critical to the performance of cloud processors. We propose to dynamically adjust prefetchers based on memory bandwidth utilization and prefetch accuracy to keep their efficacy while the integrity tree is in operation. We evaluate PCPT using 25 benchmarks drawn from 3 suites, on 512GB and 1TB memory. The experimental results show that PCPT improves the performance of the baseline SGX integrity tree by about 55%, and improves the state-of-the-art tree designs [47], [48] by over 35%. Our contribution is listed as follows:

- We show that the performance overhead of integrity tree is still high even with prior optimization methods.
- We observe that some cloud server architecture features can be utilized to further optimize tree performance, and propose three optimization designs based on this observation.
- We implement our design in a hardware simulator, and evaluate its performance against the state-of-the-art.

II. Background

A. Threat Model

We assume a threat model and security guarantee similar with popular security infrastructures [13], [32], [48]: we protect the confidentiality and integrity of a user's sensitive data from ❶ an untrusted system software (e.g., hypervisor) and co-located users, and ❷ any attacker with physical access to off-chip memory devices and the memory bus. The former can be achieved by hardware-enforced access control policies, as done in [13], [23]. In this work, we mainly focus on the latter: we further assume that ❶ the data stored in memory or transferred on memory bus can be physically *obtained* by the attacker (e.g., via cold-boot attack), and ❷ the attacker can manipulate the memory bus to *modify* the data transferred from memory to CPU (e.g., by precisely injecting packets). In addition, the attacker is not able to physically read/modify the content of on-chip resources such as registers and caches. Side-channel attacks are excluded as they can be defended by orthogonal works [19], [20], [50]. Note that these assumptions are same with prior memory security techniques [16], [37], [39], [40].

B. Memory Encryption

The goal of memory encryption is to prevent an attacker from obtaining the memory data plaintexts, i.e. protect the confidentiality of memory data. The memory controller (MC) encrypts a data block when it is written to memory, and decrypts it when it is loaded onto CPU. To guarantee semantic security while maintaining acceptable performance, counter-mode encryption is commonly used in secure memory designs: each data block in memory is associated with a counter, which is part of the seed of the encryption/decryption cipher. All the counters are also stored in memory.

C. Memory Integrity Protection

Data integrity guarantees that the CPU always sees the data it last wrote to memory. One of the most efficient method to protect memory data integrity is *MAC+integrity tree*. In memory, every data block is associated with a MAC, which is a keyed hash of the data block, to detect data modifications (typically the data block size is 64B, and the MAC size is 8B). When a data block is fetched from memory, we recalculate its MAC using the loaded data; the integrity of this data block is verified by comparing the stored MAC with the re-calculated MAC: if an attacker modifies a data block in memory, the re-calculated MAC will unlikely match the stored MAC.

However, a powerful attacker with precise control of the memory bus could deploy replay attacks by returning an old version of data+MAC to CPU. To prevent this, an integrity tree (tree for short) is needed in which data blocks are the leaves and each parent node is the hash of the child nodes (e.g., Merkle Tree [16]). To verify the freshness of a data block, all the ancestor nodes of the data block in the tree are fetched from memory except the root node which is stored on-chip, and hashes are verified. The depth of the tree directly affects performance: deeper the tree is, more tree nodes to fetch and verify for each memory data read. To reduce the overhead of fetching tree nodes (nodes for short), typically a small separate cache for tree nodes is used: since nodes in cache are already integrity-verified, for each memory data read, we only need to fetch nodes in the tree path from the bottom up until a cache hit is encountered. Note that most of the high-level nodes are always cached due to their high locality.

Bonsai Merkle Tree (BMT) [37] and SGX integrity tree (SIT) [13] are optimizations for Merkle Tree. Instead of building a tree over the data blocks, BMT and SIT use the memory encryption counter also as a version id for calculating the MAC, and thus only build a tree over all the counters, resulting in a smaller tree. Additionally, SIT uses a different tree organization to achieve parallel update of nodes in a vertical tree path, enabling better performance. In SIT (Figure 1), each node (in cache line level) consists of eight 56-bit counters and one 64-bit hash; the link between a parent and a child node is

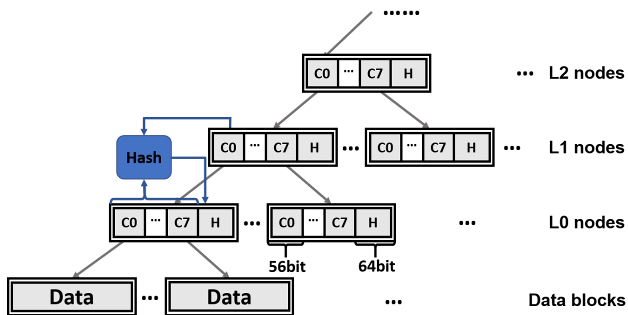


Fig. 1: SGX integrity tree (SIT).

formed by hashing all the counters in the child node and one in the parent node, and placing the hash in the child node.

D. Prior Performance Optimization Works

Although the MAC+tree mechanism can protect memory data, it also introduces significant performance overhead, because of the extra memory accesses for fetching MACs and tree nodes. There have been many previous works focusing on reducing these two overheads. Synergy [40] was proposed to reduce the overhead caused by accessing MACs for ECC memory. Synergy stores MACs in the ECC field and stores ECCs elsewhere in memory. Therefore, the MAC can be fetched along with the corresponding data block instead of generating an extra memory transaction. When the MAC verification succeeds, there is no need to fetch the ECC. In the very rare cases where MAC verification fails, the ECC will be loaded for auto-correction. However, using Synergy only benefits memory data reads; for data writes, a separate memory write transaction is still needed for updating the ECC. To further improve the write performance, ITESP [47] was later proposed which uses a small dedicated ECC cache and only updates the ECC in memory when there is ECC cache eviction. These two designs together can effectively reduce the overhead of using MACs.

To reduce the tree overhead, the authors of [47] also proposed to build a separate tree for each co-running application, instead of using one large tree for the entire memory. This design can significantly increase the tree access locality and thus increase the tree cache utilization. In addition, Vault [48] stores a shared large global counter and several small independent local counters in each tree node, instead of all independent large counters; Vault can fit more counters in a tree node and thus increase the tree arity, resulting in a much shallower tree and less tree node fetches for each memory data access. At the same time, Vault uses variable arities to reduce counter overflows. Further, Morphable counters [39] observes the temporal locality in counter values and dynamically adjusts the global counter value. Morphable counters can also reduce counter overflows.

E. Motivation

Although prior works have improved tree performance, the overhead of using an integrity tree remains high and cannot scale with the ever increasing memory capacity, especially for virtual machine (VMs) on a cloud server. The large memory size (e.g., 1TB) of cloud systems requires a very deep tree and excessive memory accesses for nodes. These additional memory accesses can impair user performance from two perspectives: 1) *sequentially* fetching more nodes for each data block makes integrity verification slower; 2) more memory accesses for nodes render higher memory bus pressure, making memory accesses slower on average. Therefore, we aim at further reducing the tree overhead from these two perspectives, based on new observations that were neglected in prior optimization works. *One difference between our work and prior works is that we take into account the architecture features of real cloud computing processors, which gives us more opportunities for optimization.*

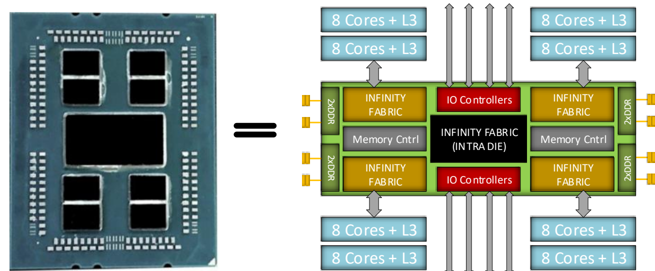


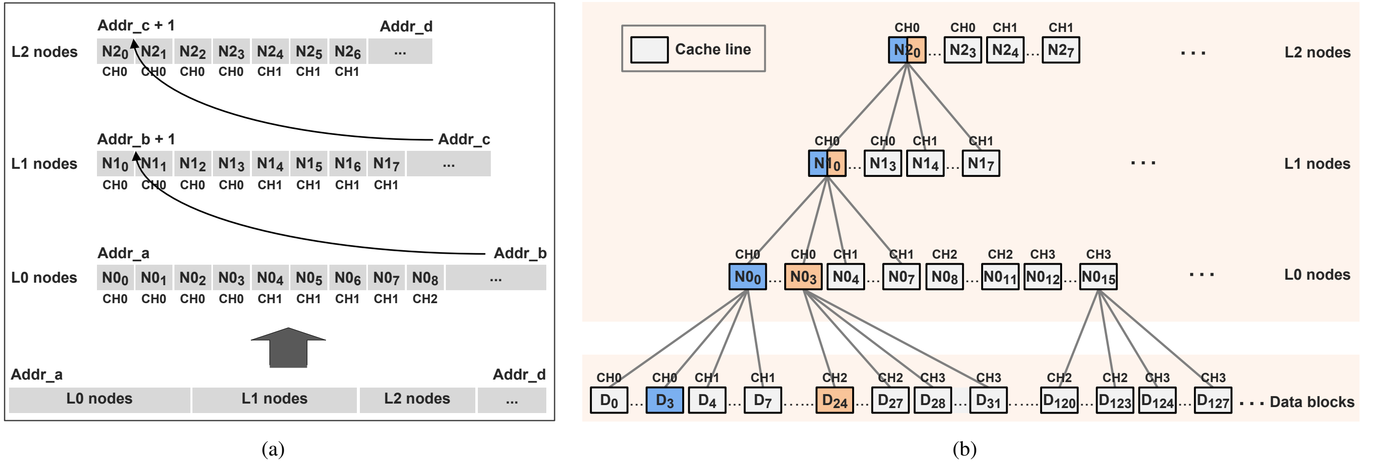
Fig. 2: The architecture of AMD EPYC 7002 processors.

III. PCPT Design

In this section, we introduce PCPT, which consists of three optimization methods to improve the tree performance. We use SIT, the SGX integrity tree, as the baseline to describe our optimizations in this section, and will discuss how to apply these optimizations to other optimized tree architectures (e.g., Vault) in Section V. To make it clear, "data block" and "tree node" ("node" for short) referred to in this section are both in cache line size.

A. Parallelizing Tree Path Fetching

With the integrity tree, each memory data read requires multiple reads of the tree nodes. Hence, the latency of fetching all the nodes for a data block lengthens the critical path of execution. If a data block and the corresponding nodes can be fetched in parallel, the latency would be much shorter than fetching them sequentially. Fortunately, the technology advancement and new architecture in modern cloud processors make this possible: most server processors used by cloud providers (e.g., AWS, Azure) now support at least 4 independent memory channels, and sometimes 8 [3]. More importantly, recent cloud server processors tend to place MCs closely to mitigate the effect of non-uniform memory access (NUMA) and improve performance [1], [2]. For example, in most AMD server processors (e.g., EYPC 7002), all the MCs are integrated in one central die, rather than distributed into each core cluster near edges, as shown in Figure 2. With this kind of architectures, MCs can communicate via the intra-die infinity fabric at high speed. Thus, if a data block and the associated nodes are stored in different memory channels, they can be fetched in parallel and verified with efficient communication between MCs. From our experiments, assuming a system with 32 cores, 512GB



(a) (b)

Fig. 3: Tree-to-memory mapping; (a) how nodes are stored in memory, and (b) a vertical view of how nodes and the corresponding data are mapped into each memory channel, for the bottom three levels of the tree and the data blocks; $\mathbf{NA_B}$ indicates the Bth node in level A.

memory, and a 12-level SIT with a 512KB cache for the tree, on average over 70% memory data reads only require no more than 2 additional memory fetches for nodes, and over 80% require no more than 3. As mentioned in Section II-C, this is because high-level nodes are mainly cached due to their high locality. Thus, when there are at least 4 channels, each data block and the associated lowest 3 nodes in the tree path can be stored in 4 different channels, and fetched in parallel.

However, current tree organization is unaware of the opportunities for parallel accesses. Figure 3 (b) shows an example of a subtree at the tree bottom and the attached data blocks. Typically, a contiguous memory address space is reserved for storing the tree [13], as shown in Figure 3 (a). Nodes are stored left-to-right, level-by-level, starting at the bottom, i.e. level-order traversal from the bottom up. Which channel a node (at cache line granularity, as defined before) belongs to depends on its physical address. Previous works [35], [49] have reverse-engineered modern MCs' address mapping algorithms for DDR4 products: generally, the 9th and some higher least significant address bits (LSBs) are used to index memory channels, i.e. every four consecutive cache lines (64B each) are stored in one memory channel, as shown in Figure 3 (a). Figure 3 (b) gives a vertical view of the channel mapping. As we can see, for each level of the tree, from the left most node, the memory channel for each node is CH0, CH0, CH0, CH0, CH1, CH1, CH1, CH1, CH2...[1]. Same organization goes to data blocks as well (also at cache line granularity). If we randomly pick a tree path (e.g., the blue or pink path in Figure 3 (b)), the data block will have a probability of 1/4 to be in the same channel with the level-0 node (L0 node for short), when there are 4 channels (1/8 for the 8-channel case). Similarly, each child node has a probability of 1/4 to be in the same channel with its parent node.

The reason behind this phenomenon is that *according to how nodes and data are organized in memory, there is a uniform distribution on channel assignments for nodes and data along a tree path (except for the top nodes), i.e. a child node or a data block always has a probability of* $1/Num_{channel}$ *to be in the same channel with its parent node.* Using the birthday paradox, there is a probability of ∼63% and 91% for channel collisions when accessing a data block with fetching 2 and 3 nodes, respectively, significantly limiting parallelism opportunities.

Proposed solution. To solve this problem, we propose to ❶ reorganize the tree in memory such that for each tree path, the nodes in the lowest levels are all mapped to different memory channels; ❷ rearrange how data blocks are connected to the tree, so that for each tree path the lowest nodes are all mapped to different memory channels than the data block.

When there are 4 memory channels, if we want to map the lowest 3 nodes and the corresponding data block in a tree path to 4 different channels, then all the 8 data blocks that are attached to one L0 node have to be in the same channel, since the corresponding L0/L1/L2 nodes are fixed and occupy the other three channels. However, as we mentioned, 8 data blocks that are currently connected to one L0 node are consecutive and distributed into 2 channels, according to modern mapping policies. Therefore, we need to choose 8 data blocks that are in the same channel, but not consecutive, to be attached to one L0 node. This, however, potentially reduces the cache hit rate of L0 nodes, due to the impaired locality. Thus, we change to only guarantee that the lowest 2 nodes and the data block are in different channels, so there will be 2 available channels for the 8 data blocks. Hence, 8 consecutive data blocks can still share one L0 node. For the case where the 9th and 10th LSBs are used for indexing channels, our reorganizing and rearranging algorithms are as follows (slight modifications are needed with different channel indexing bits):

1) Consider the subtree with 1 L1 node, 8 L0 nodes, and 64 data blocks, as shown in the left part of Figure 4. To guarantee that L0 and L1 nodes are in different channels, we place all those 8 L0 nodes in one certain channel, different from the channel for the L1 node, i.e. these L0

[1]Here we assume the 9th LSB and 10th LSB are used for indexing channels (when there are 4 channels). This result may vary with different mapping algorithms. For example, if the 9th LSB and 9th LSB ⊕ 10th LSB are used, then the result will be CH0..., CH3..., CH1..., CH2...

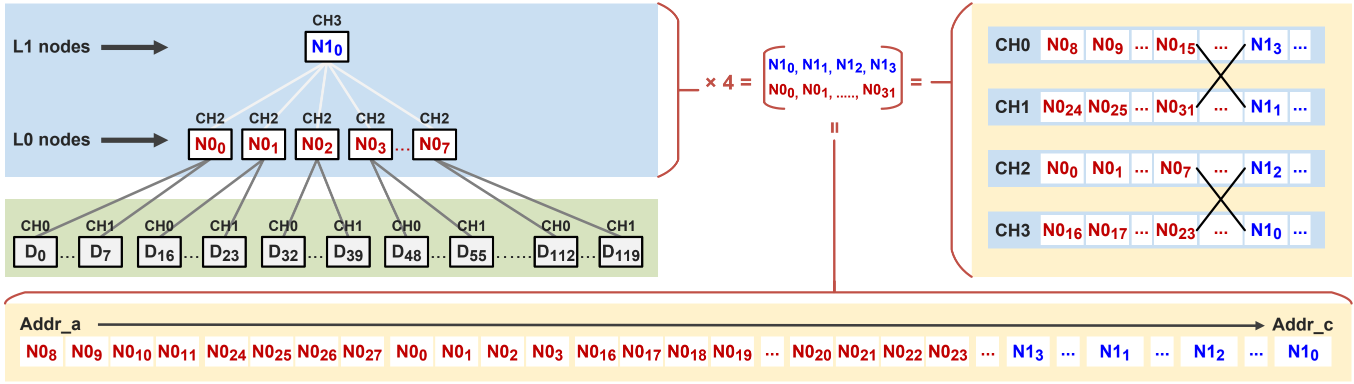


Fig. 4: The modification summary, with the left part showing a subtree with 1 L1 node, 8 L0 nodes, and 64 data blocks, the bottom part showing how to store 32 L0 nodes and the corresponding 4 L1 nodes in memory, and the right part showing the vertical view of the bottom part. Addr_a and Addr_c are same with the ones in Figure 3.

nodes can no longer be stored in consecutive addresses. At the mean time, we place adjacent 4 L1 nodes in 4 different channels to ensure the symmetry of the mapping. These can be realized by reordering the nodes in the same level in memory: the bottom part of Figure 4 shows how to reorder and store 32 L0 and 4 L1 nodes in memory; the right part shows the content of each channel after reordering: each L1 node is now in a different channel with the attached 8 L0 nodes.

2) We want each 64 data blocks to be in two channels, distinct from the channels of the 8 parent L0 nodes and 1 grand-parent L1 node, to eliminate channel collisions. As an example, in Figure 4 (left), the L0 nodes are in CH2, and the L1 node is in CH3, so we want the 64 data blocks to be in CH0 and CH1. Previously, these 64 data blocks are consecutive and thus are distributed evenly among 4 channels, i.e. 16 in CH0 and 16 in CH1. We switch the remaining 16 in CH2 and 16 in CH3 with the next subtree's 16 in CH0 and 16 in CH1. As in Figure 4 (left), then we have 64 data blocks in CH0 and CH1 for this subtree and 64 in CH2 and CH3 for the next subtree. From Figure 4 (right), these 64 in CH2 and CH3 are actually the ones the next subtree needs to eliminate channel collisions.

In the situation of 8 memory channels, for each tree path, the bottom 3 nodes and the data block can be placed in different channels simply, without affecting locality. In fact, this tree reorganization method is not unique. For an arbitrary number of channels, an example generalization of this method is shown in Algorithm 1, with CH(X) denoting the channel that X is mapped into.

B. Curtailing Tree Node Fetches

The design introduced in Section III-A does not decrease the number of extra memory accesses caused by fetching tree nodes: every memory data read still expands into multiple memory reads which significantly increase the pressure on memory bus. From our experiments with the same setup in Section III-A, over ***56%*** of these extra memory accesses come from fetching L0 nodes (as shown in Figure 7). As mentioned earlier, this is because high-level nodes are mainly cached. Thus, for verifying the integrity of a memory data block, if we could obtain the corresponding L0 counter without introducing an extra memory access, we would significantly reduce the memory bus pressure and improve performance.

Algorithm 1: Tree Reorganization Method

Input: c (# of channels), a (arity of the tree)

// Step1: Remap L2 nodes (**N2**$_*$)
for $i=0$; $i<$ #**N2**; $i+=c$ **do**
  for $j=0$; $j<c$; $j++$ **do**
    $\mathrm{CH}(\mathrm{N2}_{i+j}) \leftarrow j$

// Step2: Remap L1 nodes (**N1**$_*$)
for $i=0$; $i<$ #**N1**; $i+=c\cdot a$ **do**
  for $j=0$; $j<c\cdot a$; $j+=a$ **do**
    $\{\mathrm{CH}(\mathrm{N1}_{i+j+m})\}_{m=0}^{a-1} \leftarrow (j\%3+1)\%3$

// Step3: Remap L0 nodes (**N0**$_*$)
for $i=0$; $i<$ #**N0**; $i+=c\cdot a^2$ **do**
  for $j=0$; $j<c\cdot a^2$; $j+=a^2$ **do**
    $\{\mathrm{CH}(\mathrm{N0}_{i+j+m})\}_{m=0}^{a^2-1} \leftarrow (j\%3+2)\%3$

// Step4: Connect L0 nodes to data blocks (**D**$_*$)
for $i=0$; $i<$ #**D**; $i+=c\cdot a^3$ **do**
  for $j=0$; $j<c\cdot a^3$; $j+=a^3$ **do**
    Pick a^3 D_* to connect to $\{\mathrm{N0}_{i+j+m}\}_{m=0}^{a^2-1}$ s.t. $\mathrm{CH}(\mathrm{N2}_{i+j}), \mathrm{CH}(\mathrm{N1}_{i+j}), \mathrm{CH}(\mathrm{N0}_{i+j}) \notin \{\mathrm{CH}(\mathrm{D}_*)\}$

Proposed solution. To achieve this, we propose to compress each data block in memory (if possible), and place a copy of the corresponding L0 counter together with the data block in one cache line. Compression techniques have been well devised for cache/memory data and proven to be effective, as the data of many modern applications such as machine learning are highly compressible [22], [55]. For example, cache line level data compression algorithms, such as Base-Delta Compression [34], are well developed and used in modern computer systems. Given a 64-byte data cache line, if we can compress the data to below 57 bytes (compression ratio $\approx$ 8/7), then we are able to place at least a copy of the corresponding L0 counter

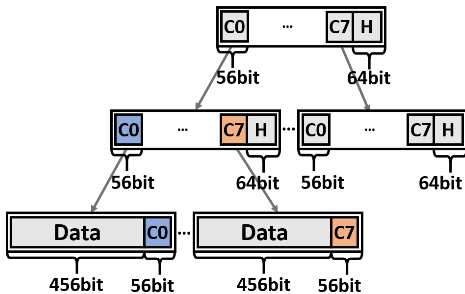


Fig. 5: The vertical view of the tree with storing a copy of the L0 counter in the data cache line.

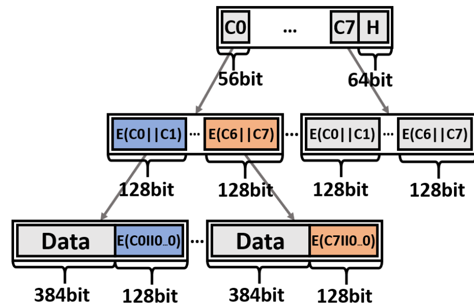


Fig. 6: The tree with encrypted L0 nodes; each L0 counter is now extended to 64 bits since the hash is eliminated.

inside this cache line[2], as shown in Figure 5. Note that this compression should happen before data encryption, because after encryption the compressibility is very small.

Once a memory data block is compressed and accompanied with its L0 counter, we do not need a separate memory transaction to fetch the tree node that contains this counter: when the data block is fetched from memory, the copy of the L0 counter stored in the same cache line will be fetched together with the data. However, there is one problem of this approach. As explained in Figure 1, to verify the integrity of an L0 counter, we need to know the value of its seven sibling counters and their corresponding hash (in the same node). Since for a compressed data block, we are now obtaining the L0 counter from the data cache line and not fetching the tree node, we will not be able to get the sibling counters and verify the integrity of the L0 counter. To address this problem, we utilize the design proposed in [48]: as shown in Figure 6, we eliminate the hash in each L0 node, and protect L0 counters by encrypting them using the parent L1 counter as part of the key, explained below. If an attacker tries to fabricate either the L1 or the encrypted L0 counter, it will very likely yield an incorrect decrypted L0 counter, which will further yield an incorrect data block that fails the MAC verification. The detailed security analysis can be found in [48]. Note that although encrypting L0 counters adds latency on the critical path, it can significantly improve the throughput of memory accesses.

This encryption in [48] is carried by first slicing an L0 node into four 128-bit chunks, and then encrypting each chunk individually using AES-128: $C_{L0} = E(P_{L0}, (CTR_{L1}||Addr) \oplus K)$, where C_{L0} and P_{L0} are the ciphertext and plaintext of the L0 chunk, respectively, CTR_{L1} is the parent L1 counter, Addr is the address of CTR_{L1}, and K is the on-chip secure key. We adapt this design in our approach as follows. We use two types of encryption: 1) for each L0 node in the tree, we still encrypt it in the same way as in [48]; 2) for a compressed data block, we first pad its L0 counter with "0"s to achieve a length of 128 bits, and then encrypt it (in the same way as the chunk mentioned above) and store the 128-bit ciphertext together with the data in one cache line. After doing so, there will be two cases for fetching and decrypting an L0 counter:

[2]Here we assume a SGX-type counter which is 7 bytes; other counter types (e.g., split counters) require different compression ratios and will be discussed in Section V.

Case 1: if the corresponding data block is compressed and the encrypted L0 counter is fetched together with it, we first decrypt this counter ciphertext using $(CTR_{L1}||Addr) \oplus K$ as the key; then we ignore the last 64 bits of the plaintext (which are supposed to be zeros), and use the first 64 bits as the L0 counter;

Case 2: if the corresponding data block is not compressed and the L0 node in the tree is thus fetched, we decrypt each slice of this node individually and obtain the target L0 counter from the corresponding slice.

Storing a 128-bit counter ciphertext with data requires a higher data compression ratio (4/3); however, we will show in Section V that it still leads to good performance as this ratio is not difficult to achieve for many applications. Note that since AES does not suffer from known-plaintext attack, this padding is secure: for manipulating the L0 counter and performing a replay attack, the attacker needs to first brute force K which requires 2^{128} searches and is not realizable. We do not encrypt an L0 counter without padding since the input size of AES block cipher is at least 128 bits.

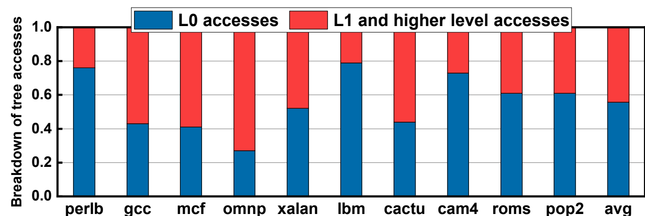


Fig. 7: The breakdown of tree node fetches.

Caching policy. When a loaded memory data block is compressed, after the loaded data is integrity-verified and decrypted, the data will be decompressed (by the MC) and stored in the data cache, and the loaded L0 counter will be discarded: we can choose to store this counter in the tree cache; however, the tree cache is accessed/modified with cache line size (tree node size) granularity. To store one counter (which is 1/8 of a cache line) in the tree cache, we will need additional metadata to track the status of each counter in a cache line, making the tree cache structure very complicated. Discarding the L0 counter will not degrade performance: next time when the same data block is loaded from memory, it is very likely that this data block is still compressed and no extra memory transaction is needed for obtaining the L0 counter. Although we will need

to decrypt this L0 counter using higher-level counters, those counters are usually cached, as explained earlier.

Counter consistency. The main challenge of storing a copy of the L0 counter with the data block is to maintain the data consistency between this copy and the one stored in the tree node. Our solution is to always update these two copies of counter together. An L0 counter is only updated when the corresponding data block is being written back to memory. Thus, for a memory data write, we first fetch the tree node that contains the associated L0 counter if it is not cached. Then we update and re-encrypt the tree node, as while as updating, padding, and re-encrypting this L0 counter individually. We store the L0 counter ciphertext together with the data block if the data block is compressible.

There is one natural question about this design: since we do not cache the L0 counter if the data is compressed, then it is very likely that when a data block (that was compressed) is being written to memory, its L0 counter is not in the tree cache and needs an extra memory transaction for fetching it. We argue that this will not significantly degrade performance for two reasons: ❶ the data write operation is not on the critical path; ❷ the probability that the L0 counter is not cached when a data block is being written back is high no matter we cache the L0 counter (when it is loaded) or not: tree cache is usually very small. Thus, when a data block is relatively "old" in the data cache, and being evicted, its L0 counter has a very high probability to be already evicted from the tree cache. We will quantitatively prove this argument in Figure 11.

When loading a data block from memory, the MC needs to know whether its L0 counter is stored with the data block or not, to generate tree node accesses. To achieve this, we use one bit to record the compression status of each data block in memory. This metadata is stored together with the secure metadata for enclaves/secure VMs: to protect a process/VM from the untrusted OS/hypervisor, modern secure infrastructures usually maintain an access control table (e.g., RMP in AMD SEV-SNP [23]). This table records the access permission and other related metadata of each secure page, and cannot be corrupted by the OS/hypervisor. Thus, we store the compression status of each data block in a secure page also in this table, resulting in 64-bit metadata for a 4KB page. However, we may need to fetch/update the metadata every time when a data block is loaded from/written to memory. This can introduce overhead because 1) fetching metadata will increase the memory bus pressure, and 2) the memory accesses for fetching data and counters will be sequentialized, since counters are not fetched until the compression status is known. In fact, this is not a severe problem because recent secure infrastructures start to use a small dedicated cache for the access control table [23]. As the metadata size is very small (1/256 of the data size), this cache hit rate is very high. Thus, as later shown in the experiments, our design can still significantly improve performance.

C. Adaptive Tree-aware Prefetching

The increase in memory bus traffic caused by nodes fetching would also impose a negative impact on hardware prefetchers

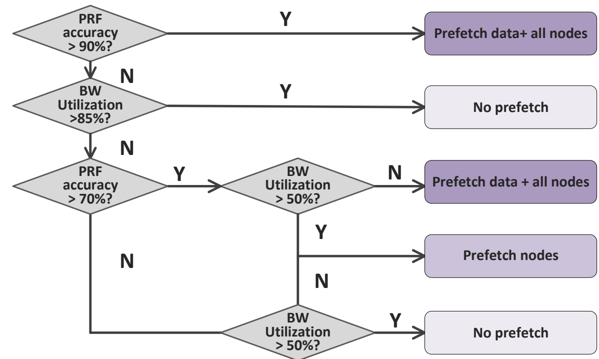


Fig. 8: Adaptive prefetching algorithm based on the memory bandwidth utilization (normalized to the peak utilization), and the prefetch accuracy.

that are widely used in cloud server processors to increase data cache hit rate [6]. However, previous works on tree optimization ignore the importance of prefetchers and assume a baseline without any prefetch.

Actually, researchers have proposed hardware prefetcher designs with high accuracy and at low cost [26], [31], [42]. Those prefetchers work efficiently in insecure memory where there is no protection on data privacy and integrity. However, for secure memory, those designs may have new problems. In secure memory where there is a tree involved, when a data block is being prefetched, its associated nodes should also be prefetched to verify the integrity of the data, otherwise, the prefetched data cannot be accessed securely. Thus, one problem will occur when using previous hardware prefetchers in secure memory: *in secure memory, each prefetch likely triggers a sequence of memory reads for both data and its associated nodes, and thus generates heavier memory bus traffic than in insecure memory, coupled with the fact that the bus traffic is already heavier in secure memory due to node accesses.*

In previous works [7], [9], [45], it has been proposed to use memory bandwidth utilization as a feedback signal to adjust the prefetcher's behavior. The overall ideas of these designs are similar: when the bandwidth utilization is low, increase the prefetch coverage; when it's high, increase the prefetch accuracy. These designs ensure that the prefetcher will not overwhelm the memory bus and hurt the performance, by adjusting "what to prefetch" according to the bandwidth utilization. However, they do not consider the case of secure memory. In secure memory, "what to prefetch" is not only about deciding which data block to prefetch, but also which node to prefetch. *Our key insight is that nodes and data should have different priorities during the prefetch decision:* after a prefetch request is generated (based on the data access pattern), instead of prefetching the data and all its associated nodes, prefetching nodes only is more beneficial when constrained, as nodes (especially higher-level nodes) are shared by many data blocks and are thus more likely to be useful.

Proposed solution. We therefore propose a new algorithm to adjust general hardware prefetchers' behavior in secure memory, based on the memory bandwidth utilization and

TABLE I: Evaluated secure/insecure memory designs.

Design	Tree structure	Tree Optimiz.			MAC	MAC Optimiz.	
		Tree path parallelization	Curtailed tree fetch	Adaptive prefetch		Synergy [40]	ECC cache [47]
Insecure baseline	✗	✗	✗	✗	✗	✗	✗
Secure baseline	SIT [13]	✗	✗	✗	64bit	✓	✓
Vault_O	Vault [48]	✗	✗	✗	64bit	✓	✓
ST_O	Separated SIT [47]	✗	✗	✗	64bit	✓	✓
PT	SIT	✓	✗	✗	64bit	✓	✓
CPT	SIT	✓	✓	✗	64bit	✓	✓
PCPT	SIT	✓	✓	✓	64bit	✓	✓
Vault_O+PCPT	Vault	✓	✓	✓	64bit	✓	✓
ST_O+PCPT	Separated SIT	✓	✓	✓	64bit	✓	✓

the prefetch accuracy: our algorithm is decoupled from the internal prefetcher design which decides how to predict the data address for the next prefetch, it's only about after the prefetcher generates the prefetch request, whether and how should this be served by the MCs. We assume that the bandwidth utilization can be tracked by simple counters near the MCs as in [9], and the prefetch accuracy in each epoch is recorded as in [27].

Our algorithm is sketched in Figure 8. There are two scenarios where we always fetch the target data block and all the associated nodes for each prefetch request. The first scenario is when the prefetch accuracy is very high (e.g., $> 90\%$) because correct prefetch of data and its nodes would very likely benefit the performance. The second scenario is when the bandwidth utilization is relatively low (e.g., $< 50\%$ of peak utilization) and the prefetch accuracy is still good (e.g., $> 70\%$) because the pressure on the memory bus is low. However, if the bus is quite busy (e.g., $> 85\%$ of peak utilization) but prefetch accuracy is not very high, the prefetch requests are rejected by the MCs.

There is one situation where we only prefetch nodes. From the experiments, when using Best Offset Prefetcher [31], there are over ***50%*** memory data prefetches that require to fetch at least one node for integrity verification. In these cases, when the prediction is wrong and the prefetched data block is useless, the prefetched nodes might still be useful since they are shared by multiple data blocks. Thus, when we have to partially give up some prefetch accesses due to the limited memory bandwidth, we still want to prefetch these nodes to shorten the verification path for future data blocks.

The thresholds in Figure 8 are empirical results. When implementing this design into processors, these numbers can be left configurable for the system software.

IV. Methodology

We evaluate our techniques on 25 benchmarks including 15 from SPEC 2017 [5], 6 from GAP [41], and 4 machine learning workloads (ML) [36]. The chosen 15 SPEC workloads cover the ones that have high/medium/low percentage of memory operations to give a comprehensive evaluation. We implement our design in USIMM [12], a trace-driven cycle accurate memory simulator. To simulate the cloud computing system which is typically virtualized, we collect the full-system trace of each VM: we modify QEMU [8], an open-source emulator and virtualizer, to collect the entire trace of a VM during binary translation. For each simulation, we first run 1 billion instructions to warm up the cache, and then run 10 billion instructions for collecting the statistic data. For energy consumption evaluation, we use Micron power calculator [4] to estimate the power of each memory chip.

For the baseline, we implement a small cache dedicated for the tree, as done in most prior optimization works [40], [48]. The size of the tree cache is set to be `16KB×Num_core`. For memory prefetch, we use Best Offset Prefetcher (BOP) [31], which is one of the most effective hardware prefetchers. In addition, we use a variant of Base-Delta-Immediate Compression (BDI) [34] in PCPT: for a compressed data block, we encode the compressed data length into the first four bits of this cache line. Most of our evaluations are done in systems with 32 cores, 4 channels, and 512GB memory; however, we do test PCPT with different configurations in Section V-B. For the evaluation, we first test the performance of the systems in which n VMs run concurrently, where n is the number of CPU cores. In this case, we reserve one core for each VM; each runs the same single-threaded benchmark (from SPEC) at a time. Then we test the systems in which multiple cores are assigned to each VM, and each runs the same multi-threaded benchmark (from GAP or ML). Detailed simulation parameters are shown in Table II.

V. Results

In Section V-A and Section V-B, we use two baselines to evaluate our design. As shown in Table I, the first baseline is the

TABLE II: Simulation Parameters

ROB/width	128-entry/8-wide
Number of cores (Num_core)	32/64
L1 cache, private	32KB, 64B line, 8-way associative
LLC, shared	4MB per core [3], 64B line 16-way associative
Hardware prefetcher	BOP
Compression algorithm/latency	Optimized BDI/2 cycles
Counter encryption/decryption latency	50 cycles
Tree cache	16KB × Num_core
RMP cache	8KB × Num_core
DDR4 memory channels	4/8
Memory size	512GB/1TB
Bus speed	3200MHZ
DIMMs per channel	2, according to [3]

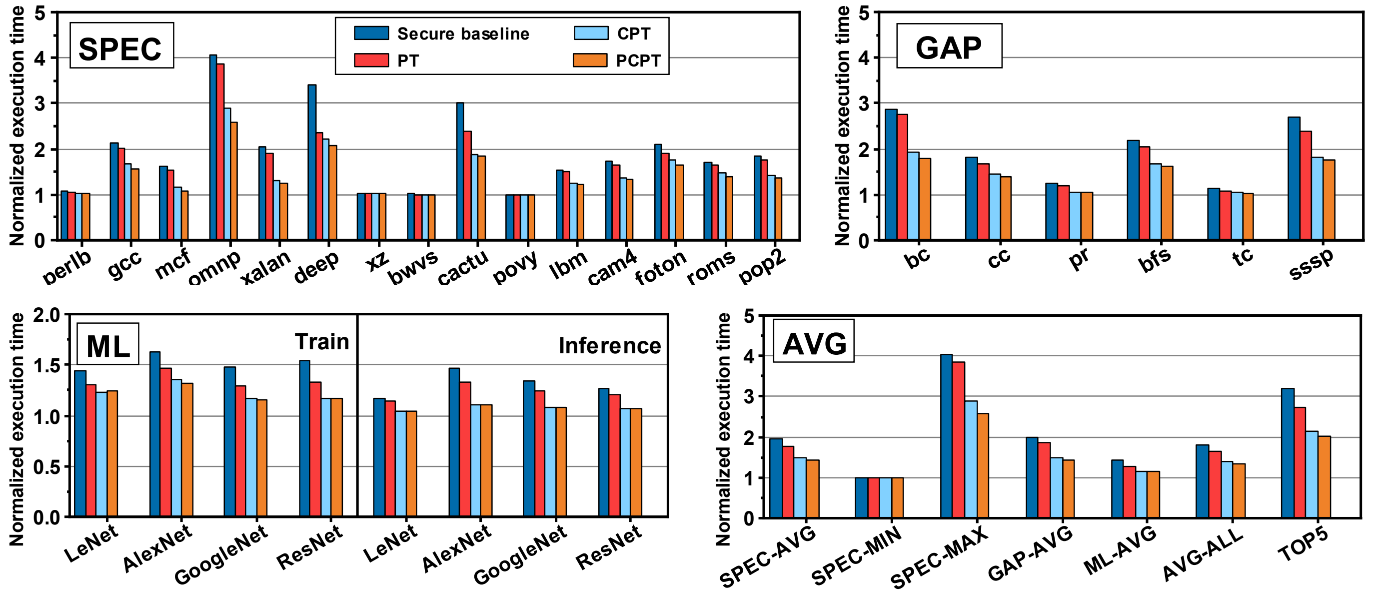


Fig. 9: Execution time for the secure baseline (SB), secure baseline with parallelizing tree path accesses (PT), PT with curtailed tree node fetches (CPT), and CPT with the adaptive prefetching algorithm (PCPT), all normalized to the insecure baseline, assuming 32 cores, 4 memory channels, and 512GB memory.

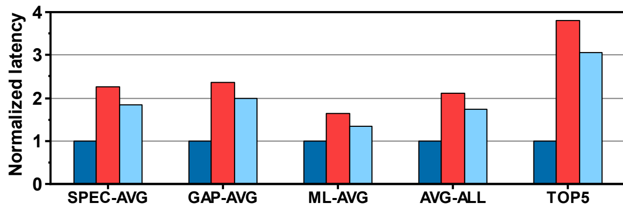


Fig. 10: The load instruction latency, with each three bars from the left to right showing the latency for the insecure baseline, secure baseline, and PT, all normalized to the insecure baseline.

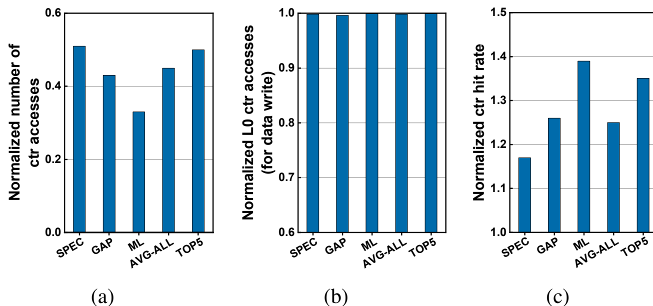


Fig. 11: (a) The total number of node fetches, (b) the probability that the corresponding L0 counter is not cached when writing data to memory, and (c) the cache hit rate of L1 and higher-level nodes, in CPT; all the results are normalized to PT.

insecure baseline, in which there is no protection on memory confidentiality and integrity. The second baseline is the *secure baseline* in which we implement Counter-Mode AES (AES-CTR) for memory encryption, and the MAC+tree structure for memory integrity. We use a simple 12-level SIT (SGX Integrity Tree) which covers up to 1TB memory. We also implement Synergy+ECC cache [40], [47] in the secure baseline which can together mitigate the overhead caused by MAC accesses, as introduced in Section II-C. In Section V-C, we evaluate PCPT with optimized tree structures instead of SIT to show how using PCPT can further improve the performance of prior optimization designs. Table I provides the details regarding the configurations we evaluate.

A. Performance and Energy Evaluation

Figure 9 shows the execution time for different benchmark suites when our proposed optimizations are applied step by step. All the results are normalized to the insecure baseline. As we can see, the secure baseline, which only optimizes the performance of MAC accesses, still suffers very high overhead on certain benchmarks (up to 4 times). In comparison, PCPT adds three optimizations on the tree performance, and thus achieves much smaller overhead.

First, as shown in the first and second bars from the left in Figure 9, compared to the secure baseline, the execution time for *secure baseline + parallelizing tree path accesses* (PT) is reduced from 1.95 to 1.77 for SPEC, 1.99 to 1.85 for GAP, 1.41 to 1.29 for ML, and 3.21 to 2.74 for the 5 most memory-intensive benchmarks (among all 25 benchmarks, referred by Top5), on average. The primary reason of this improvement is the shortened latency of load instructions. Figure 10 shows the execution latency of load instructions that miss in the data cache. For the insecure baseline (shown by the left bars), as long as the target data block is fetched and returned from memory to CPU, the instruction is ready to retire. In contrast, with integrity verification, since we also need to fetch all the associated tree nodes, this latency becomes much longer. As we can see, compared to the secure baseline (shown by the middle bars), where some of the node fetches happen sequentially, parallelizing tree path accesses (shown by the right bars) can effectively reduce the latency of load instructions. As shown

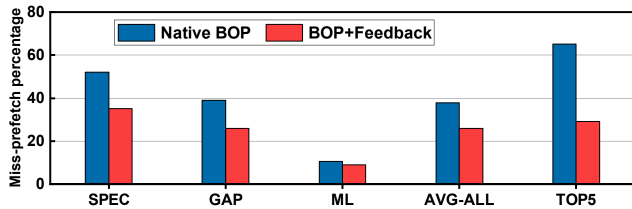


Fig. 12: The percentage of useless prefetch accesses, in PCPT with/without the proposed adaptive prefetching algorithm, counting both the node accesses and the data accesses.

in Figure 9, this reduction is important especially for memory intensive applications (e.g., **cactu**), as with the tree, the much longer latency for load instructions can cause the CPU pipeline to be stalled more easily, which is significant overhead. Note that this design does not benefit **omnetpp** much, because the tree cache utilization of this benchmark is much lower than other benchmarks: in **omnetpp**, about 40% memory data reads need the access to L4 and higher nodes. Thus, only parallelizing the lowest 2 or 3 nodes does not help very much.

Second, curtailing tree node fetches can further reduce the execution time from 1.77 to 1.50 for SPEC, 1.85 to 1.51 for GAP, 1.29 to 1.15 for ML, and 2.74 to 2.15 for Top5, as shown in the second and third bars from the left in Figure 9. Figure 11 (a) shows the total number of node fetches in *PT + curtailed tree fetches* (CPT), normalized to the results in PT. We can see that on average, CPT reduces about 50% of the node fetches in PT. Most of this reduction comes from the reduction of L0 node fetches. However, since in CPT, we do not cache the L0 counter for a compressed data block, the cache hit rate of the higher-level nodes (L1 to root) in CPT becomes much higher than the hit rate in PT, as shown in Figure 11 (c). Thus, CPT also has less L1 and higher-level node fetches than PT does, which can further benefit the performance. *This is why CPT achieves better performance improvement compared to prior works that also improve performance by reducing metadata accesses, such as Synergy.* In addition, as shown in Figure 11 (b), not caching L0 counter for compressed data blocks will not increase the probability that the L0 counter needs to be fetched when the corresponding data block is written to memory; the difference between PT and CPT is less than 1%.

The third and fourth bars from the left in Figure 9 show the performance difference of using BOP with and without the proposed adaptive algorithm: comparing with using the native BOP, after adding the proposed algorithm, the execution time can be further reduced to 1.42 for SPEC, 1.44 for GAP, 1.15

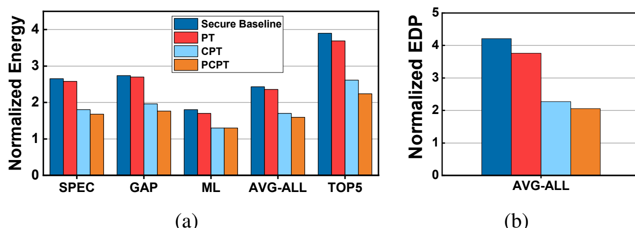


Fig. 13: (a) The normalized memory energy and (b) the average system energy delay product, with the same setup of Figure 9.

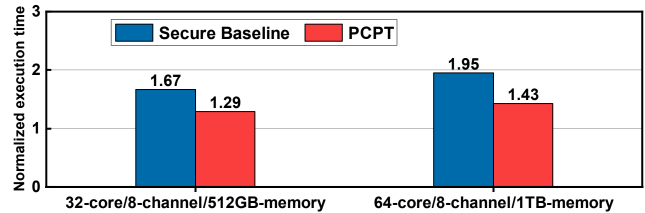


Fig. 14: The average execution time of the secure baseline and PCPT for various number of cores and memory channels, all normalized to the insecure baseline.

for ML, and 2.01 for Top5. Figure 12 shows the percentage of prefetch accesses that are not useful. As we can see, native BOP is very aggressive and can generate about 40% useless prefetch accesses which can significantly increase the memory bus pressure. In contrast, with the adaptive prefetching algorithm, the useless portion can be reduced to about 25% on average.

Figure 13 shows memory energy results and system energy delay product (EDP). Compared to the secure baseline, PCPT reduces the energy consumption from 2.43 to 1.59, and system EDP from 4.21 to 2.04, on average. This improvement mainly comes from the reduction on the number of tree node fetches and useless prefetch accesses.

B. Performance Evaluation with Other Setups

In this section, we evaluate PCPT in systems with 32-core/8-channel/512GB-memory and 64-core/8-channel/1TB-memory, as shown in Figure 14. Compared to the 32-core/4-channel/512GB-memory system, in the 32-core/8-channel system, PCPT has less performance improvement over the secure baseline. This is because with 8 channels, there is a lower probability for channel collisions in a tree path, and useless prefetches have less effect on the performance with a higher memory bandwidth. In addition, both the secure baseline and PCPT have higher overhead in the 64-core system, compared to the 32-core/4-channel system. We believe this is due to a lower tree cache utilization, as explained in [47]. However, PCPT remains good improvement over the secure baseline.

C. PCPT with Prior Tree Optimization Designs

In previous sections, we only evaluate PCPT with assuming the baseline tree to be SIT. In fact, recently researchers have proposed many optimized tree designs based on SIT. Vault proposed in 2018 [48] is a tree architecture that has much better performance compared to SIT. We slightly modify and implement PCPT over a 8-level Vault to evaluate PCPT with Vault: ❶ Vault has a much higher arity (e.g., 64 for L0), and thus we need to correspondingly modify the reorganizing details in Figure 4; however, we cannot avoid hurting the locality of L0 nodes due to the higher arity); ❷ each L0 counter in Vault have 71 bits (64-bit global counter and 7-bit local counter). Hence, we pad an L0 counter with 57 "0"s and still use AES-128 to encrypt it before placing it in the data cache line. As shown in Figure 15, using PCPT can improve the performance of optimized Vault (Vault_O) from 1.66 to 1.41 on average. The configuration details of Vault_O are shown in Table I.

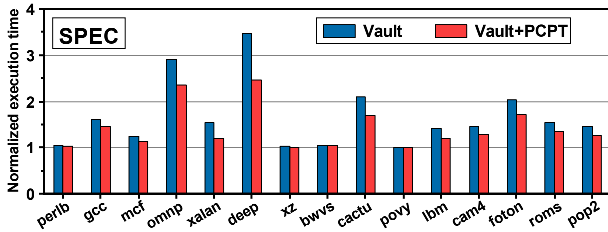


Fig. 15: The normalized execution time of optimized Vault (Vault_O) and Vault_O+PCPT.

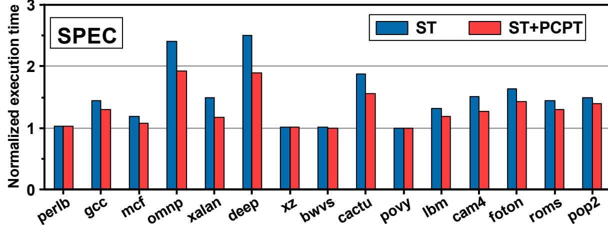


Fig. 16: The normalized execution time of optimized Separated Tree (ST_O) and ST_O+PCPT.

In [47], the authors proposed to build one separate tree for each process and isolate trees in cache. This design has been proved to significantly increase the tree cache utilization and thus reduce the overhead. PCPT can be directly applied on top of this design to further improve the performance: this separated tree design (ST) mainly focuses on reducing the overhead of high-level node accesses, and PCPT mainly focuses on reducing low-level node access overhead. From Figure 16, with optimized ST (ST_O) and PCPT together, the average overhead of using a MAC+tree structure is only 1.30. The configuration details of ST_O are also in Table I.

VI. Discussion

A. Side Channel Attacks

Side channel attacks are very potent. Although PCPT mainly focuses on memory integrity and excludes side channels from our threat model, we do not want to introduce additional side channels by this design. In this section, we discuss side channel attacks related to PCPT and explain why they will not cause severe security problems.

First, one may argue that data compression in general can introduce side channels, as shown in [24]. However, this compression-based side-channel attack has a prerequisite that the attacker is able to obtain the *exact compression ratio* of a cache line. This can be easily achieved by using a variant of Prime+Probe [29]. However, in PCPT this prerequisite is very difficult (if not impossible) to achieve for two reasons. First, we do not use variable-size cache lines in PCPT: even after applying data compression, a cache line in PCPT is still always the same length; the compression is only done to make space for a counter. Thus, the attacker cannot figure out the compression ratio of a data block by directly observing the cache line size anymore. Second, an attacker with precise control of the memory bus might be able learn if an L0 counter fetch is created after a data fetch, and thus learn if this data block is compressed. However, this only tells the attacker whether the compression ratio of this data block is larger than 4/3 or not, which is much less information than the attacker needs in [24]. There has not been any work showing that this information can be utilized to build a practical side channel attack. If an attack is discovered in the future, the compression can be performed with an element of randomness.

Second, prior research [44] has found that general hardware prefetchers can introduce powerful side channels. This is because the prediction result of the prefetcher is related to users' memory access patterns (which might be related to users' secrets). PCPT does not modify how prefetch addresses are generated. We only limit (but not increase) the prefetches based on the memory bus utilization and prefetch accuracy. Hence, we do not introduce additional side channel leakages, but rather, reduce potential leakages. The thresholds on bus utilization and prefetch accuracy in PCPT are only observable to the hardware logic in memory controller, and thus cannot be used by the untrusted OS/hypervisor or other users to leak information. However, since the prefetch decision affects memory bus utilization, a "smart" attacker could carefully design some algorithms to learn the memory bus utilization and further obtain the prefetch accuracy, which might be related to the victim's secrets. In fact, side channel leakages caused by the sharing of memory bus do not only exist in our design, modern secure infrastructures (e.g., SGX) all suffer from this problem. This is a significant and general challenge on designing secure infrastructures, and we have to rely on orthogonal defenses and more future works to defend them.

B. Related Works

We have already discussed most of the integrity protection techniques including MT, BMT, Vault, Synergy, Morphable counters, and ITESP. In addition, the work from Lehman et al. [28] analyzes the integrity metadata access patterns. Graphene [33] and BlockHammer [17] were proposed for defending Rowhammer attacks. Osiris [51], Anubis [56], and PLP [14] focus on integrity trees for persistent memory systems.

VII. Acknowledgments

This work is supported in part by US National Science Foundation #1422331, #1535755, #1617071, #1718080, #1725657, #1910413, and #2011146. The authors thank the anonymous reviewers for their constructive comments.

VIII. Conclusion

In this work, we first give a comprehensive summary of the state-of-the-art integrity tree designs, and identify critical performance problems of directly using them in cloud systems with large memories. Then, we give an important insight that some cloud server architecture features can be considered in the tree design for further performance improvement. Based on this insight, we propose a new design, PCPT, which consists of three tree optimizations tailored for cloud server architectures and critical hardware configurations. We implement and evaluate PCPT in USIMM, and the results show that PCPT can improve the performance of the state-of-the-art by over 35%.

References

[1] [Online]. Available: https://www.amd.com/en/processors/epyc-7002-series

[2] [Online]. Available: https://www.amd.com/en/processors/epyc-7003-series

[3] "The 2nd generation AMD EPYC processor redefines data center economics." [Online]. Available: https://www.amd.com/system/files/documents/TIRIAS-White-Paper-AMD-Infinity-Architecture.pdf

[4] "Micron system power calculator." [Online]. Available: http://www.micron.com/products/support/power-calc

[5] "SPEC CPU 2017." [Online]. Available: https://www.spec.org/cpu2017

[6] "Workload tuning guide for AMD EPYC™ 7002 series processor based servers." [Online]. Available: https://developer.amd.com/wp-content/resources/56745_0.80.pdf

[7] A. R. Alameldeen and D. A. Wood, "Interactions between compression and prefetching in chip multiprocessors," in *2007 IEEE 13th International Symposium on High Performance Computer Architecture (HPCA)*, 2007.

[8] F. Bellard, "Qemu, a fast and portable dynamic translator," in *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, 2005, p. 41.

[9] R. Bera, A. V. Nori, O. Mutlu, and S. Subramoney, "Dspatch: Dual spatial pattern prefetcher," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2019, pp. 531–544.

[10] R. Buhren, C. Werling, and J.-P. Seifert, "Insecure until proven updated: Analyzing AMD SEV's remote attestation," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2019, pp. 1087–1099.

[11] S. Chakrabarti, M. Hoekstra, D. Kuvaiskii, and M. Vij, "Scaling Intel® software guard extensions applications with Intel® SGX card," in *Proceedings of the 8th International Workshop on Hardware and Architectural Support for Security and Privacy (HASP)*, 2019.

[12] N. Chatterjee, R. Balasubramonian, M. Shevgoor, S. Pugsley, A. Udipi, A. Shafiee, K. Sudan, M. Awasthi, and Z. Chishti, "Usimm: the utah simulated memory module," *University of Utah, Tech. Rep*, pp. 1–24, 2012.

[13] V. Costan and S. Devadas, "Intel sgx explained," Cryptology ePrint Archive, Report 2016/086, 2016.

[14] A. Freij, S. Yuan, H. Zhou, and Y. Solihin, "Persist level parallelism: Streamlining integrity tree updates for secure persistent memory," in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2020, pp. 14–27.

[15] P. Frigo, E. Vannacc, H. Hassan, V. v. der Veen, O. Mutlu, C. Giuffrida, H. Bos, and K. Razavi, "Trrespass: Exploiting the many sides of target row refresh," in *2020 IEEE Symposium on Security and Privacy (SP)*, 2020, pp. 747–762.

[16] B. Gassend, G. E. Suh, D. Clarke, M. van Dijk, and S. Devadas, "Caches and hash trees for efficient memory integrity verification," in *Proceedings of the 9th International Symposium on High-Performance Computer Architecture (HPCA)*, 2003, p. 295.

[17] A. Giray Yağlıkçı, M. Patel, J. S. Kim, R. Azizi, A. Olgun, L. Orosa, H. Hassan, J. Park, K. Kanellopoulos, T. Shahroodi *et al.*, "Blockhammer: Preventing rowhammer at low cost by blacklisting rapidly-accessed dram rows," *arXiv e-prints*, 2021.

[18] S. Gueron, "A memory encryption engine suitable for general purpose processors," *IACR Cryptol. ePrint Arch.*, vol. 2016, p. 204, 2016.

[19] Y. Guo, A. Zigerelli, Y. Zhang, and J. Yang, "Ivcache: Defending cache side channel attacks via invisible accesses," in *Proceedings of the 2021 on Great Lakes Symposium on VLSI*, 2021, pp. 403–408.

[20] A. Harris, S. Wei, P. Sahu, P. Kumar, T. Austin, and M. Tiwari, "Cyclone: Detecting contention-based cache information leaks through cyclic interference," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2019, pp. 57–72.

[21] M. Hoekstra, R. Lal, P. Pappachan, V. Phegade, and J. Del Cuvillo, "Using innovative instructions to create trustworthy software solutions," in *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy (HASP)*, 2013.

[22] D. A. Huffman, "A method for the construction of minimum-redundancy codes," *Proceedings of the IRE*, vol. 40, no. 9, pp. 1098–1101, 1952.

[23] D. Kaplan, "AMD SEV-SNP: Strengthening VM isolation with integrity protection and more," *White paper*, 2020.

[24] J. Kelsey, "Compression and information leakage of plaintext," in *International Workshop on Fast Software Encryption*, 2002, pp. 263–276.

[25] J. S. Kim, M. Patel, A. G. Yağlıkçı, H. Hassan, R. Azizi, L. Orosa, and O. Mutlu, "Revisiting rowhammer: An experimental analysis of modern dram devices and mitigation techniques," in *Proceedings of the 47th Annual International Symposium on Computer Architecture*, 2020, pp. 638–651.

[26] J. Kim, S. H. Pugsley, P. V. Gratz, A. L. N. Reddy, C. Wilkerson, and Z. Chishti, "Path confidence based lookahead prefetching," in *Proceedings of the 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2016.

[27] C. J. Lee, O. Mutlu, V. Narasiman, and Y. N. Patt, "Prefetch-aware dram controllers," in *Proceedings of the 41st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2008, p. 200–209.

[28] T. S. Lehman, A. D. Hilton, and B. C. Lee, "MAPS: understanding metadata access patterns in secure memory," in *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2018, pp. 33–43.

[29] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee, "Last-level cache side-channel attacks are practical," in *2015 IEEE Symposium on Security and Privacy (SP)*, 2015, pp. 605–622.

[30] F. McKeen, I. Alexandrovich, A. Berenzon, C. V. Rozas, H. Shafi, V. Shanbhogue, and U. R. Savagaonkar, "Innovative instructions and software model for isolated execution," in *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy (HASP)*, 2013.

[31] P. Michaud, "Best-offset hardware prefetching," in *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2016, pp. 469–480.

[32] M. Orenbach, P. Lifshits, M. Minkin, and M. Silberstein, "Eleos: Exitless os services for sgx enclaves," in *Proceedings of the 12th European Conference on Computer Systems*, 2017, pp. 238–253.

[33] Y. Park, W. Kwon, E. Lee, T. J. Ham, J. H. Ahn, and J. W. Lee, "Graphene: Strong yet lightweight row hammer protection," in *Proceedings of the 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2020, pp. 1–13.

[34] G. Pekhimenko, V. Seshadri, O. Mutlu, M. A. Kozuch, P. B. Gibbons, and T. C. Mowry, "Base-delta-immediate compression: Practical data compression for on-chip caches," in *2012 21st international conference on parallel architectures and compilation techniques (PACT)*, 2012, pp. 377–388.

[35] P. Pessl, D. Gruss, C. Maurice, M. Schwarz, and S. Mangard, "DRAMA: Exploiting DRAM addressing for cross-cpu attacks," in *25th USENIX Security Symposium (USENIX Security)*, 2016.

[36] J. Redmon, "Darknet: Open source neural networks in c," 2013-2016. [Online]. Available: http://pjreddie.com/darknet/

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

[38] B. Rogers, M. Prvulovic, and Y. Solihin, "Efficient data protection for distributed shared memory multiprocessors," in *Proceedings of the 15th International Conference on Parallel Architectures and Compilation Techniques*, 2006, pp. 84–94.

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

[40] G. Saileshwar, P. J. Nair, P. Ramrakhyani, W. Elsasser, and M. K. Qureshi, "Synergy: Rethinking secure-memory design for error-correcting memories," in *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2018, pp. 454–465.

[41] D. P. Scott Beamer, Krste Asanović, "The gap benchmark suite," 2015.

[42] M. Shevgoor, S. Koladiya, R. Balasubramonian, C. Wilkerson, S. H. Pugsley, and Z. Chishti, "Efficiently prefetching complex address patterns," in *Proceedings of the 48th International Symposium on Microarchitecture (MICRO)*, 2015, pp. 141–152.

[43] W. Shi, H.-H. S. Lee, M. Ghosh, C. Lu, and A. Boldyreva, "High efficiency counter mode security architecture via prediction and precomputation," in *Proceedings of the 32nd Annual International Symposium on Computer Architecture (ISCA)*, 2005, pp. 14–24.

[44] Y. Shin, H. C. Kim, D. Kwon, J. H. Jeong, and J. Hur, "Unveiling hardware-based data prefetcher, a hidden source of information leakage," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2018, pp. 131–145.

[45] S. Srinath, O. Mutlu, H. Kim, and Y. N. Patt, "Feedback directed prefetching: Improving the performance and bandwidth-efficiency of hardware prefetchers," in *2007 IEEE 13th International Symposium on High Performance Computer Architecture (HPCA)*, 2007, pp. 63–74.

[46] G. E. Suh, D. Clarke, B. Gassend, M. van Dijk, and S. Devadas, "Aegis: Architecture for tamper-evident and tamper-resistant processing," in *Proceedings of the 17th Annual International Conference on Supercomputing (ICS)*, 2003, p. 160–171.

[47] M. Taassori, R. Balasubramonian, S. Chhabra, A. R. Alameldeen, M. Peddireddy, R. Agarwal, and R. Stutsman, "Compact leakage-free support for integrity and reliability," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, 2020, pp. 735–748.

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

[49] M. Wang, Z. Zhang, Y. Cheng, and S. Nepal, "Dramdig: A knowledge-assisted tool to uncover dram address mapping," in *Proceedings of the 57th Annual Design Automation Conference (DAC)*, 2020.

[50] M. Yan, J. Choi, D. Skarlatos, A. Morrison, C. W. Fletcher, and J. Torrellas, "Invisispec: Making speculative execution invisible in the cache hierarchy," in *Proceedings of the 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2018, p. 428–441.

[51] M. Ye, C. Hughes, and A. Awad, "Osiris: A low-cost mechanism to enable restoration of secure non-volatile memories," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2018, pp. 403–415.

[52] J. M. You and J.-S. Yang, "Mrloc: Mitigating row-hammering based on memory locality," in *Proceedings of the 56th Annual Design Automation Conference (DAC)*, 2019.

[53] Z. Zhang, Y. Cheng, D. Liu, S. Nepal, and Z. Wang, "Telehammer: Cross-privilege-boundary rowhammer through implicit accesses," *arXiv*, 2019.

[54] Z. Zhang, Y. Cheng, D. Liu, S. Nepal, Z. Wang, and Y. Yarom, "Pthammer: Cross-user-kernel-boundary rowhammer through implicit accesses," in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2020, pp. 28–41.

[55] J. Ziv and A. Lempel, "A universal algorithm for sequential data compression," *IEEE Transactions on information theory*, vol. 23, no. 3, pp. 337–343, 1977.

[56] K. A. Zubair and A. Awad, "Anubis: Ultra-low overhead and recovery time for secure non-volatile memories," in *2019 ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA)*, 2019, pp. 157–168.