IvLeague: Side Channel-resistant Secure Architectures Using Isolated Domains of Dynamic Integrity Trees

Md Hafizul Islam Chowdhuryy and Fan Yao
University of Central Florida
hafizul.islam@ucf.edu, fan.yao@ucf.edu

***Abstract*—Modern secure processors rely on hardware-assisted encryption and tree-based integrity verification to protect off-chip data. However, despite extensive research on performance optimization, there is a significant lack of emphasis on side channel vulnerabilities in secure architectures. Given the strong focus on data security, it is critical to ensure that the integration of new design elements into secure architectures does not inadvertently introduce additional vulnerabilities.**

Existing integrity verification mechanisms use a global integrity tree shared across security domains, which can introduce side channel leakage through integrity tree metadata sharing. In this work, we present IvLeague framework – a novel integrity verification mechanism for side channel-resistant isolated integrity trees among dynamic domains in secure processors. Specifically, IvLeague splits the global tree into multiple fixed-size subtrees, dynamically allocating these subtrees to domains during runtime. IvLeague enables efficient runtime scaling of memory coverage for individual domains. Additionally, we IvLeague-Invert, an optimization which shortens the integrity verification path by mapping data pages to high-level tree nodes. Finally, IvLeague-Pro further improves the integrity verification of hotpages by enabling efficient hotpage tracking and migrating hotpages closer to the root. We extensively evaluate all three IvLeague schemes using 16 real-world workloads with varying memory footprints. IvLeague scheme, along with its optimizations, demonstrates a 5%-19% speedup over the insecure *baseline*, while providing effective side channel protection for the integrity tree. Moreover, IvLeague ensures high utilization of TreeLings (over 99.5%) and supports workloads with highly skewed memory footprints.

***Index Terms*—Secure architectures, Metadata Side channels, Trusted execution environment, Isolated integrity trees.**

I. Introduction

Trusted computing has drawn considerable attention due to the growing concerns of trust in remote computing platforms (e.g., cloud services). State-of-the-art mechanisms employ secure architectures that take the processor as the root-of-trust and employ hardware-based encryption and integrity verification to offer strong data protection. Best-practice solutions such as Intel SGX [1] provide a trusted execution environment (TEE) that protects program execution in enclaves against adversaries that can compromise off-chip hardware [2] and privileged software [3], [4], [5], [6], [7]. While protecting off-chip data is essential, ensuring on-chip data security is of paramount importance. Recent advances in microarchitectural attacks (for example, timing channels) [8], [9], [10], [11], [12], [13], [14], [15], [16] highlight that program secrets can be severely exfiltrated by attackers by modulating microarchitectural states in various *on-chip* hardware resources, which leads to numerous proposals on protecting microarchitecture security [17], [18], [19], [20], [21], [22], [23], [24]. As industry and academia increasingly advocate secure-by-design architectures [1], [25], [26], [27], it is imperative to investigate the impact of security mechanisms as they are integrated and to enhance data security holistically in future systems.

Although prior works have demonstrated microarchitectural attacks in secure processors [12], [28], [29], [30], they generally exploit *known vulnerabilities* that are already manifested in classical settings (e.g., timing channels on caches [9], [10]). As such, they do not necessarily expand the current microarchitecture attack surface. In fact, commercial-off-the-shelf SGX hardware explicitly excludes side channels from its threat model [1], stating that microarchitecture security should be handled separately. Unfortunately, security of microarchitecture cannot be treated as a standalone problem in secure architectures [31], [32]. Specifically, the recent research [32] unveils that the integrity verification (IV) mechanism using tree-based metadata in secure processors introduces new side channel leakage *by design*. With a global IV tree, memory accesses in one domain (e.g., an enclave) unavoidably exercise integrity tree nodes at certain levels shared with other domains [32]. This implicit metadata sharing enables new *shared-memory* side channels even when regular data sharing is prohibited among domains (i.e., to defeat existing attacks such as Flush+Reload [13], [33], [34]). Such leakage in secure processors *exacerbates* microarchitecture security, and more concerningly, cannot be effectively mitigated by directly adopting existing defenses such as resource access randomization and partitioning [35], [36], [37]. In contrast to prior side channels that predominantly exploit the sharing of *hardware resources*, this vulnerability stems from a new source of sharing–*metadata*.

This paper aims to thwart side channels that exploit shared security metadata in secure processors [32]. Since the underlying vulnerability is the use of a global integrity tree, a plausible solution is to enforce IV metadata isolation among security domains. While statically partitioning the integrity tree can be a straightforward approach [31], such a mechanism does not allow runtime scaling of the number and size of secure domains. Ideally, an isolated integrity tree *per domain* should be maintained to prevent metadata sharing across domains. This tree should also be adjusted at runtime in order to cover the dynamic range of memory footprints for the domain. However, partitioning integrity trees and dynamically managing them at runtime can introduce several main challenges: i) non-trivial

performance overhead due to the potential memory indirections needed for traversing dynamically-constructed integrity trees, ii) the need for efficient hardware-based mechanisms to manage tree nodes for runtime workload memory usages, and iii) support for easy scaling of domains with low on-chip and off-chip metadata overhead.

We propose IvLeague, an architecture support for side channel-resistant isolated integrity trees among dynamic domains in secure processors. At a high level, IvLeague splits the global integrity tree into many small *statically-addressed* subtrees (called TreeLings). Metadata sharing is prevented among TreeLings by keeping their roots on-chip. IvLeague enables efficient runtime scaling of memory coverage (upto entire system memory) by *assigning* and *detaching* TreeLings to each individual domain. To allow flexible mapping of physical pages to TreeLings, IvLeague integrates a hardware mechanism that efficiently assigns and reclaims tree nodes for data pages according to memory allocations and deallocations. Additionally, IvLeague synergistically sets the number of TreeLings and performs limited TreeLing expansion to support a considerable number of IV domains (maximum 2^{12}) and mitigate TreeLing starvation with low metadata overhead.

We further propose several optimizations on the basic IvLeague framework (IvLeague-basic). Firstly, it has been observed that workloads with small memory footprints typically utilize a limited fraction of TreeLing leaf nodes. The first optimization, IvLeague-Invert, shortens the path of integrity verification from leaf to root by directly mapping data pages to high-level intermediate nodes and gradually introducing nodes from lower levels (i.e., intra-TreeLing extension) only when all nodes in certain top levels are occupied. Invert's top-down allocation policy can significantly reduce the effective TreeLing height and the corresponding integrity verification latency. Secondly, real world workloads often access a subset of pages with very high frequency (i.e., hotpages). Reducing the integrity verification overhead for data accesses in hotpages can significantly improve overall system performance. Accordingly, our second optimization, IvLeague-Pro, reserves a sub-region of each IvLeague that is dedicated for mapping hotpages. IvLeague-Pro integrates a lightweight hotpage tracker into the memory controller. When a page is designated as a hotpage, IvLeague-Pro performs *low-overhead* runtime relocation that maps the newly-identified page to a TreeLing breach closer to the root. Similarly, untracked pages are mitrated to the regular TreeLing nodes from the hot region of TreeLing. Notably, both IvLeague-Invert and IvLeague-Pro take advantage of the existing mechanism of dynamic physical page-to-leaf mapping in IvLeague-basic, requiring minimal additional hardware support.

We build a prototype of IvLeague in a cycle-level simulator and extensively evaluate its performance and runtime behavior across *16 multi-programmed* workloads built from SPEC2017, PARSEC and graph benchmarks [38], [39], [40]. Our evaluation indicates that IvLeague-basic provides strong side channel security for integrity tree designs with reasonable overheads ranging from 2.7% to 17.4% compared to the insecure baseline that uses a globally-shared IV metadata. More importantly, IvLeague-Invert enables a shorter effective tree height for programs with low memory footprints, leading to *5% speedup* (on average) over insecure scheme for small and medium workloads. Furthermore, IvLeague-Pro optimizes all workloads with faster integrity verification for frequently accessed pages, further offering up to 19% (14% on average) performance gain over the insecure baseline. Finally, we evaluate the scalability of IvLeague compared to static tree partitioning, and observe that IvLeague achieves near-optimal utilization of TreeLings (>99.5%) and scale well with workloads with various sizes of memory. IvLeague incurs modest on-chip hardware logic and storage cost. Overall, IvLeague shows the promise of designing performance-friendly secure processors with enhanced microarchitecure security in the future. In summary, the main contributions of this work are:

- We motivate the need to re-think the integrity verification mechanism in secure processors for leakage protection, and propose the first architectural support for isolated domains of dynamic integrity trees–IvLeague. IvLeague partitions the global integrity tree into small TreeLings and associates them with IV domains on-demand.
- We enhance IvLeague with IvLeague-Invert that maps data pages to TreeLing using a top-down extension mechanism to reduce the verification path length for among workloads with smaller memory footprints.
- We further propose IvLeague-Pro, which optimizes the integrity verification latency for frequently accessed pages by placing them in reserved nodes of TreeLing. IvLeague-Pro tracks hotpages at runtime and dynamically migrates them closer to the root in the reserved region of TreeLing to accelerate hotpage access with low overhead.
- We extensively investigate the performance of IvLeague schemes and find IvLeague can provide strong side channel protection for security metadata, with an overhead of 2.7%-17.4% for IvLeague-basic compared to the insecure scheme. Moreover, IvLeague-Invert and IvLeague-Pro optimize the performance significantly, resulting in 5%-19% speedup over the baseline.
- We perform scalability analysis of IvLeague against static partitioning schemes. IvLeague demonstrates significantly higher scalability compared to static partitioning for workloads with skewed memory footprints.

II. Background

A. Microarchitectural Attacks and Defenses

Microarchitectural attacks are a form of information leakage threat where illicit communication is established through the modulation of access timing to shared resources. These attacks can materialize as covert channels, facilitating unauthorized data transmission between isolated domains, or as side channels, where a malicious process illicitly extracts secrets from a victim process. Previous studies have shown timing channels exploiting various hardware resources in modern processors [14], [16], [41], [42], [43], [44], [45],

[46], [47], [48], [49], [50], [51], [52], [53], [54], [55], [56], [57], many of which are demonstrated on caches. Mainstream cache attacks such as Prime+Probe [14] observes victim activities through cache evictions. Shared-memory attacks (e.g., Flush+Reload [13], Evict+Reload [11]) monitor secret-dependent cache accesses via shared memory lines (e.g., shared libraries). Recent studies have unveiled various side channels in TEE environments such as Intel SGX [3], [6], [12], [28], [29], [58], [59], [60]. These attacks mostly utilize the same attack vectors (e.g., conflicts on caches [43], [61]), but with more sophisticated manipulations of victim's execution (e.g., replay and stepping [3]) enabled under the assumption of privileged adversaries [6]. Distinctively, recent works [31], [32] have identified that security metadata in secure processors (i.e., SGX) creates new source of leakage beyond the conventional sharing of hardware resources. In particular, MetaLeak [32] harnesses the multi-level sharing of integrity tree metadata, and formulates highly accurate side channel exploitation over metadata against real-world applications running in enclaves. Note that state-of-the-art microarchitectural defenses typically employ resource partitioning or obfuscation to either eliminate contention or disrupt the attacker's timing observations [21], [31], [35], [36], [37], [62], [63]. These schemes are not designed to mitigate information leakage due to the implicit sharing of metadata [32].

B. Secure Processors for Trusted Computing

State-of-the-art secure architectures minimize the trusted computing base (TCB) by treating the processor chip as the root of trust, and assuming off-chip components can be compromised via physical attacks (e.g., data stealing/spoofing and replay attacks [64]). Representative secure processors [31], [65], [66], [67], [68], [69] employ three key mechanisms: i) data confidentiality protection via *counter-mode encryption*, ii) data authentication with message authentication codes, and iii) data integrity (i.e., freshness) using integrity trees.

Data Encryption and Authentication. In counter-mode encryption, per-block counters are used. A data block P (e.g., 64B) is broken into n chunks (p_i for $i \in [0, n-1]$, 16B each under 128-bit AES). When the processor writes P to memory, the processor encrypts each data chunk (p_i) with $c_i = p_i \oplus Enc(S, K)$ with key K. S is the *encryption seed* derived from the physical address of p_i and the counter associated with P. The encrypted block C consists of the encrypted chunks (c_i for $i \in [0, n-1]$). Counters are incremented after each data write to ensure *uniqueness* of encryption seeds. To offer data authenticity, the processor keeps an MAC (e.g., using keyed-hash) over the data block using its block address and encryption counter. The use of MAC can detect data spoofing and splicing attacks [70], [71], [72].

Integrity Verification. For more advanced attack scenario where attacker can arbitrarily replace certain data with an older version (i.e., replay attack), secure processors use an integrity tree to guarantee data freshness. A classical integrity tree is constructed with tree nodes consisting of hashes (i.e., hash

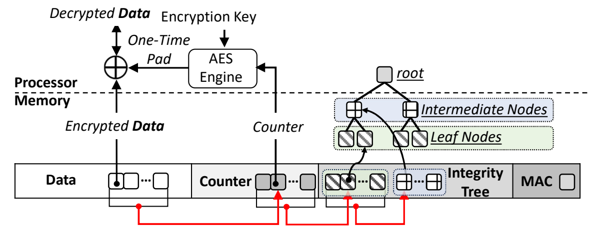


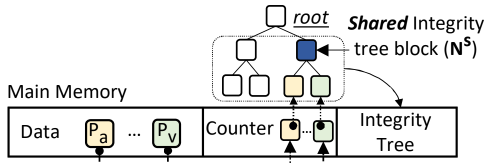
Fig. 1: Secure architecture mechanisms overview. Counters, MAC and integrity tree metadata are stored in memory. A *fixed* address mapping scheme (denoted as →) is used to locate the counter/MAC block for a data block, and the tree node block for a counter block (under a Bonsai Merkle tree).

tree) [31], [67], [68]. The hash is computed over a data block (e.g., 64B to 128-bit hash), and multiple hashes (e.g., 8) form a leaf node (tree memory block). The entire tree is built by further hashing the tree node to form the parent nodes level by level, eventually converging to the tree root. The number of hashes in one tree node determines the tree arity. When the processor reads from memory, the hashes are computed from the leaf to the root, which is then compared with the root on-chip. A mis-match indicates that the data in memory has been tampered. Typical secure processors integrate metadata caches that store partial integrity tree blocks on-chip. As such, the verification and update of tree nodes (i.e., for data read and write) only need to be performed up to the level *cached* on-chip since the processor is trusted.

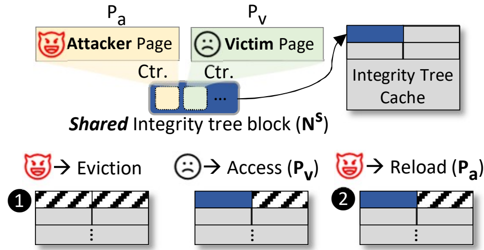
Integrity Tree Designs. The integrity verification procedure can incur considerable overhead as the tree size (e.g., height) grows. To reduce the verification overhead, state-of-the-art designs propose Bonsai Merkle Tree (BMT) built over *encryption counters only* (shown in Figure 1). As the MAC is computed over both the data block and its counters, a matching MAC with verified counter (using the tree) also proves the freshness of the data block. Since the size of counter metadata is significantly smaller than data, BMT can be considerably smaller than the one built over both data and counters [65], [66], [67], [73]. An alternative design to the hash tree is *tree of counters* [65], [74], [75] where a tree node contains a combination of *counters* (i.e., a large major counter shared among memory blocks in a page and small minor per-block counters) along with an embedded hash for the counters. When a data write occurs, the counter tree is updated by incrementing the minor counters. Intel SGX adopts a similar counter tree design but uses monolithic counters (56-bit) instead [75]. Note that regardless of the design choices, integrity tree is statically constructed, under which the memory controller uses a *fixed mapping function* to find the physical address of encryption counters and integrity tree nodes for a certain data block, as shown in Figure 1.

III. Threat Model

We assume that an adversary attempts to exfiltrate sensitive information from a victim process via microarchitectural



(a) Integrity tree block sharing across pages



(b) Attacker exploiting shared nature of integrity tree block to monitor victim accesses

Fig. 2: Exploitation of integrity tree block sharing.

attacks (e.g., timing side channels [9], [11], [13], [14]). The victim runs a process protected by TEE (e.g., an enclave in SGX). The adversary is a privileged attacker who can control the operating system (OS) and can also execute programs in enclaves. The running enclaves are considered mutually distrusting, and the TEE runtime ensures isolation between them [1]. Similar to existing TEE security studies [3], [6], [12], [28], [29], [58], [59], [60], we assume that the system software (e.g., OS/hypervisor) is untrusted and may be compromised by the adversary. The attacker may employ advanced noise filtering techniques, such as fine-grained program execution stepping and replay [3], [60].

We assume that the processor is equipped with state-of-the-art TEE support, including counter-mode encryption and BMT for integrity verification, to thwart various off-chip attacks (e.g., bus snooping and cold boot attacks [76]). To mitigate timing channel leakage between two security domains, data sharing between attacker and victim processes (e.g., through shared libraries) is either audited [51] or completely disabled [11], [13], [37]. Additionally, to defeat contention-based side channels (e.g., Prime+Probe [14] on caches), we assume that a state-of-the-art cache randomization scheme [19], [21], [35] is employed. Finally, non-timing-based leakage, such as attacks that exploit physical properties like electromagnetic emanations [52] and power consumption [53], [77], [78], [79], is considered out of scope.

IV. Motivation: Side Channel Attack on Shared Integrity Tree

Since the integrity tree is built over the entire memory as one unit, it creates shared integrity tree blocks among data pages. The integrity of counter blocks (and hence their corresponding data blocks) is verified through a certain path of nodes in the integrity tree. Shared integrity tree blocks are the ones with tree nodes common between two (or more) verification paths,

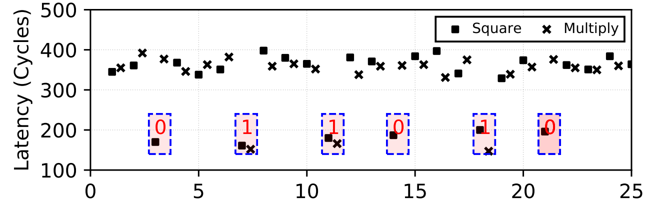


Fig. 3: Traces of attacker observed latencies to access P_a^1 and P_a^2. The inferred secret value (e_i) is highlighted in red.

as illustrated in Figure 2a. Consecutive data pages share the same lower-level tree blocks (i.e., leaf), and the higher-level blocks feature sharing across larger regions of the data pages (Figure 1). As shown in MetaLeak [32], an adversary can utilize its own data page to share an integrity tree node with the victim domain and launch shared-memory cache attacks [32] (i.e., similar to Evict+Reload [11]).

Figure 2b provides a high-level overview of this attack. Specifically, for a given victim page P_v in secure memory (e.g., the EPC in SGX), the attacker can allocate a physical page P_a such that P_a and P_v share a common tree node block (e.g., N^s in Figure 2a) used for integrity verification. The attacker can then *indirectly* infer the victim's access to P_v by observing the latency of their own access to P_a, which is carefully set to trigger the traversal of the integrity tree up to N^s. A shorter (or longer) access latency, caused by verification, indicates whether the victim has (or has not) accessed P_v due to a hit or miss of the shared N^s in the metadata cache. This exploitation shares some similarity with the Flush+Reload attack [13], but instead manipulates metadata. Figure 2b illustrates the attack steps. The attacker first performs metadata eviction of N^s (and its child nodes) ❶[1], and then infers the caching state of N^s based on the access latency to P_v ❷. If the victim's access to P_v depends on a secret, this secret can be exfiltrated by the attacker.

We demonstrate a real-world attack on systems equipped with Intel SGX processors. The attack targets the vulnerable modular exponentiation algorithm in the OpenSSL library [80], where a bit in the secret exponent e determines access patterns to the square (`sqr`) and multiplication (`mul`) functions. In this scenario, the victim is running a cryptographic application inside an enclave. Specifically, we launch the attack on an Intel i7-9700K processor, which uses tree-based integrity verification for enclave data [75]. Intel SGX employs a 4-level and 8-ary counter tree. A 64B tree node block includes eight 56-bit monolithic counters and a 64B hash. The leaf level of the tree covers eight encryption counter blocks (each containing eight 56-bit monolithic counters corresponding to data pages). Given the deterministic design of the integrity tree, the attacker can locate pages that share tree node at any specific integrity tree level with certain targeted EPC page (e). The attacker runs a malicious enclave and selects two pages, P_a^1 and P_a^2, which are empirically set to share

[1]Eviction is necessary as there is typically no ISA support for metadata flush, in contrast to `clflush` for data.

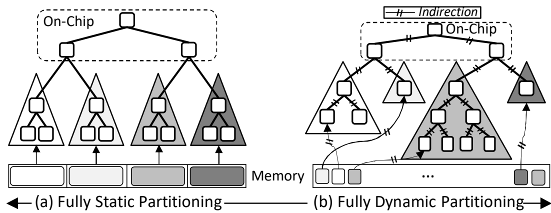


Fig. 4: Design space of integrity tree partitioning schemes.

tree node blocks at the second level (starting from the leaf) with victim pages $P_v^{\texttt{sqr}}$ and $P_v^{\texttt{mul}}$, respectively. We launch the metadata-based attack following the steps illustrated in Figure 2b. Figure 3 shows the attacker-observed access latency to both P_a^1 and P_a^2, which correlates with the victim's access to `sqr` and `mul`. Overall, we achieve 91.6% accuracy in recovering the 2048-bit RSA private exponent from a victim enclave running in SGX.

Uniqueness of the Metadata-based Side Channel. This attack reveals a new attack vector affecting secure processors. Specifically, the inherent *sharing of IV metadata* fundamentally breaks the isolation required for effective side channel protection among security domains [32]. More importantly, although the demonstrated attack exploits caches, existing secure cache defenses, such as partitioning [21], [35], [36], are ineffective. This is because mainstream cache defenses assume that the attacker and victim do not share *writable* data. Such sharing is feasible through implicit metadata sharing, which can lead to cache coherence issues with cache partitioning. Essentially, the root cause of this vulnerability is the IV metadata mechanism, which necessitates modifications to secure architectures to enhance microarchitecture security.

V. Design Objectives of IvLeague

In this section, we present the *design space* for architectural techniques to defeat integrity verification (IV) metadata-based leakage. We discuss the challenges and limitations ranging from the straightforward static partitioned metadata designs to fully dynamic integrity tree schemes (illustrated in Figure 4), and motivate the key design principles for IvLeague.

Completely Static Integrity Trees: A simple approach to enable isolation of IV metadata is to statically partition the global tree into a fixed number of subtrees. At runtime, each enclave domain is assigned to a subtree. Every subtree covers a pre-defined chunk of the physical memory. Since static addressing is used for locating tree nodes for data reads (as in the default global tree), such approach does not introduce additional per-domain tree construction and node management overhead. However, it has several key drawbacks: Firstly, it cannot easily scale with a varying number of security domains at runtime without limiting its utility (e.g., coverage of individual subtrees); Secondly, static partitions cannot accommodate workloads with larger memory footprints and, on the other hand, can lead to unused metadata (e.g., leaf nodes) for workloads with smaller memory requirements. Lastly, such scheme relies on the OS to strictly allocate physical pages for each domain from its designated memory chunk, which is problematic as the OS is typically not trusted with the TEE threat model.

Per-domain Dynamically-constructed Integrity Trees: Secure processors can alternatively maintain fully dynamic integrity trees (Figure 4b). In this scheme, each domain hosts its own tree and is able to grow and shrink the tree size (i.e., coverage of verified memory) according to the dynamic memory usage of the workload. The addressing for *child nodes to their parent* and *memory blocks to their leaf node* is dynamically determined (e.g., via lookup tables) by the secure hardware without OS intervention. This design promises maximum flexibility in terms of the number of runtime domains and the size of each domain, but unavoidably brings substantial metadata maintenance overhead. For instance, verifying the integrity of a data block involves a memory lookup to find the corresponding leaf node, followed by multiple lookups to locate the tree node from leaf to root of the tree. Such memory indirection can impose significant slowdown on the already complex tree traversal for integrity verification. Moreover, management of tree nodes (e.g., allocation and reclamation) by hardware could introduce non-deterministic runtime complexity, impacting the critical path of program execution.

Based on the above discussion, a practical side channel-resistant IV mechanism should incorporate the following design considerations: i) support for a sufficient number of isolated IV domains, which can be dynamically constructed and destroyed as needed; ii) ability to efficiently adjust the integrity coverage area in each domain to match the runtime memory footprint of workloads; iii) low-overhead techniques to manage the tracking and mapping of tree nodes at runtime. Lastly, though less obvious, given a finite budget for IV metadata (e.g., storage), the secure IV mechanism should avoid tree node starvation during *multi-domain* execution, where no available tree nodes can be used for newly allocated data pages, even when system memory has not been exhausted. It can be seen that the aforementioned design choices are complementary in terms of fulfilling the properties, but neither of them meets all the requirements. With this observation, our work aims to design an isolated dynamic integrity verification framework that offers high efficiency in performance and effectiveness in leakage prevention by employing an advanced hybrid scheme.

VI. IvLeague Design

A. Design Overview and Challenges

The key principle of IvLeague is to maintain many tiny and *isolated* integrity trees that have no shared nodes, referred to as TreeLings. TreeLings serve as the basic allocation units for each IV domain, and TreeLing roots are kept on-chip. Moreover, the nodes *within* a TreeLing are *statically-addressed*, and hence the leaf to root node traversal does not require memory indirection. IvLeague tracks the mapping between the data page and the TreeLing leaf node in the *Leaf Mapping Metadata* (LMM), which is embedded in the page table. This allows the mapping of arbitrary physical pages

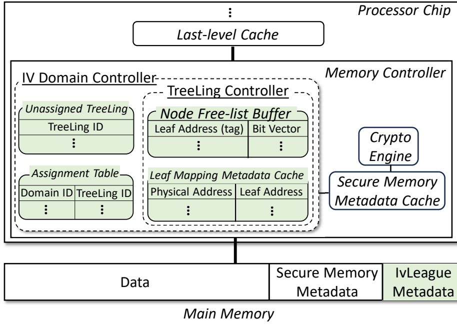


Fig. 5: Overall architecture of IvLeague. Shaded area represent added components.

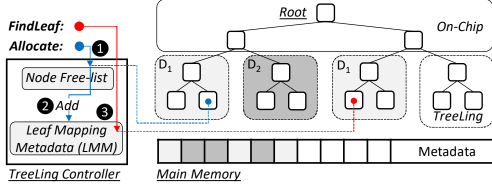


Fig. 6: High-level operation of IvLeague highlighting partitioning between two domains (D_1, D_2) via TreeLing isolation.

to TreeLings. Note that in BMT, a data page is mapped to a tree node when its counter block is assigned and directly verified with the tree node. A per-TreeLing *Node Free-list* (NFL) maintains the *available leaf nodes* that can be assigned to new page allocations. Figure 5 shows an overview of the IvLeague architecture. The high-level operation of IvLeague is illustrated in Figure 6. When a new page is allocated to an IV domain (❶), an available leaf node is assigned to the page from the NFL. The LMM is updated to associate the leaf node with the corresponding page frame (❷). During integrity verification, the leaf node for the physical data page is retrieved from LMM (❸), without requiring additional indirection. To enable runtime resizing of IV domain coverage, IvLeague incorporates an *IV Domain Controller* to dynamically map/unmap TreeLings within IV domains.

There are several design challenges that must be addressed to provide efficient and effective side channel protection for integrity verification metadata. *Firstly*, since the number of TreeLings is limited and the availability of TreeLing nodes is crucial for program data page allocation, the leaf nodes in each TreeLing must be utilized efficiently to avoid under-utilization within the TreeLing (Section VI-C1). Effective *intra-TreeLing* management is required to avoid premature TreeLing exhaustion caused by poor utilization. This is challenging because IvLeague performs dynamic mapping between pages and TreeLing nodes, requiring an additional hardware-controlled allocation policy for TreeLing nodes (Section VI-C). *Secondly*, the configuration of TreeLings (i.e., the size and number of TreeLings in the system) must be chosen carefully to prevent TreeLing starvation, where small-footprint domains occupy the majority of TreeLings, leading to a TreeLings shortage while system memory is still available. *Inter-TreeLing* management must consider these factors to select appropriate TreeLing

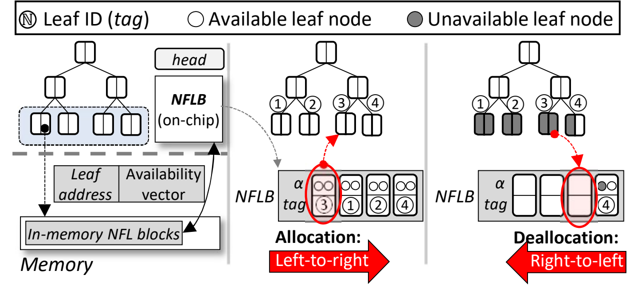


Fig. 7: High-level overview of NFL design.

configurations and allocate TreeLings to domains dynamically during runtime (Section VI-D). In the following sections, we present how each of these challenges can be addressed.

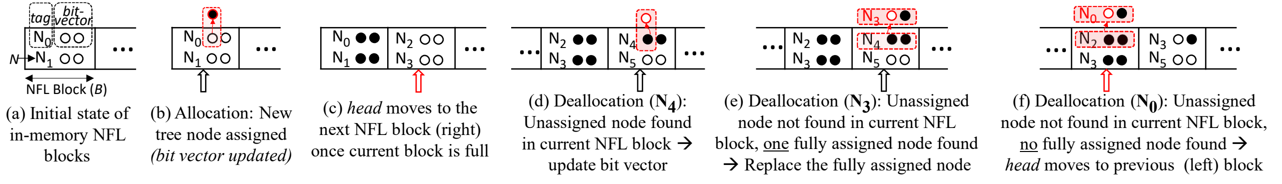
B. Definition of TreeLing

TreeLings originate from the split parts of the global integrity tree, each having a very small memory coverage (e.g., a few to tens of MBs). In the simplest form, a TreeLing is a subtree with a node at the lth level of the global tree as its root. Apparently, assuming a total number of m tree nodes in level l, there will be a set of m TreeLings, denoted as $\{\tau_i | i \in [1, 2, \ldots, m]\}$ (See the illustration in Figure 6). To offer the isolation guarantee, the roots of TreeLings and any level of nodes above are kept on-chip, which prevents sharing of the nodes in memory between any pair of TreeLings (e.g., τ_i and τ_j). Such an isolation can be achieved using various ways, including locking nodes on cache or hosting them on dedicated on-chip buffers. Particularly, IvLeague reserves a dedicated space in the IV metadata cache via way partitioning to hold all TreeLing roots.

C. Intra-TreeLing Management

1) Dynamic Mapping of TreeLing Nodes: When a new page is allocated to a domain, an available TreeLing node must be associated with the page. Traditional secure architectures rely on static mappings between a page and integrity tree leaf nodes. In a scheme with dynamic mapping between the data page and the integrity tree leaf, leaf node allocation requires scanning through all the leaf nodes in a TreeLing to find an available one (i.e., TreeLing leaf node which is not yet assigned to any data pages). This introduces a considerable runtime overhead of O(N), given the total number of leaf nodes N, which is in the critical path of data page allocation. On the other hand, simple approaches, such as consecutively assigning tree leaves in a predetermined direction, are incompatible with runtime page deallocation activities.

IvLeague integrates a hardware-based mechanism to perform TreeLing node mapping at runtime with the additional in-memory metadata storage, *NFL*. Specifically, NFL is a per-TreeLing structure that tracks TreeLing node availability during runtime. Each NFL entry includes two fields: a *tag* for the address of a TreeLing node block, and an *availability vector* tracking the available slots in this node where counter blocks can be attached (e.g., slots for hashes). NFL maintains



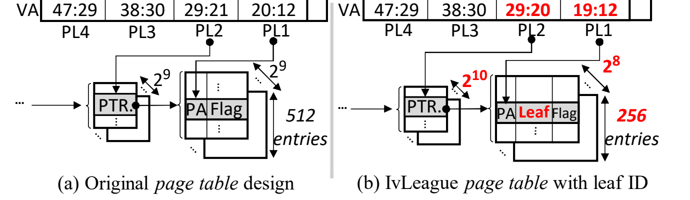
(a) Initial state of in-memory NFL blocks (b) Allocation: New tree node assigned *(bit vector updated)* (c) *head* moves to the next NFL block (right) once current block is full (d) Deallocation ($\mathbf{N_4}$): Unassigned node found in current NFL block → update bit vector (e) Deallocation ($\mathbf{N_3}$): Unassigned node not found in current NFL block, one fully assigned node found → Replace the fully assigned node (f) Deallocation ($\mathbf{N_0}$): Unassigned node not found in current NFL block, no fully assigned node found → *head* moves to previous (left) block

Fig. 8: Operations of NFL (❍: corresponding slot is available, ●: unavailable). The entries in an NFL block (shown as individual rows) denote the available slots in a TreeLing node block that could be mapped (attachable). The *arrow* indicates the position of *head* register.

one entry for each TreeLing leaf node, and multiple entries are stored in one memory line. A head register is kept to point to the current NFL block under operation. Figure 7 illustrates the design and high-level operations of node mapping using NFL. When a new TreeLing is assigned to an IV domain, all the tags in its NFL are initialized with the corresponding node block addresses, the α vectors are reset, and the head is set to the first NFL block. Upon requests for node mapping and unmapping, IvLeague accesses the current NFL block to either find an available attaching slot (for page allocation) or add tracking of reclaimed slots (for page deallocation). To reduce the overhead due to NFL memory reads, an on-chip CAM buffer (**NFLB**) is maintained that caches the most recently accessed NFL blocks. Figure 8a illustrates the organization of the NFL in memory. Figure 8b-8f show the *logical* view of the NFL reflecting the changes made through NFLB due to the mapping and unmapping of data pages.

NFL Operations for Page Allocation. Upon a new memory page allocation, if the currently active NFL block pointed by the head register (B_i) has an available slot, the new page is mapped to that slot (Figure 8b). Here, i represents the currently active NFL block in the NFLB. The address of B_i is stored in the *head* register. Once B_i is fully mapped, the *head* register is moved to the next NFL block, B_{i+1} (Figure 8c). Since the *head* register is advanced only when all previous NFL blocks are fully utilized, this design ensures that IvLeague can always locate an available slot for new page allocation with at most one NFL read (i.e., the *next* NFL block). As a result, IvLeague with NFL offers a maximum of O(1) overhead for locating an available slot during page allocation. Note that if B_i already contains an available slot, which would be the common case for most of the page allocations, the page to TreeLing node mapping via NFLB incurs no additional overhead.

NFL Operations for Page Deallocation. During memory page deallocation, the α vector of the corresponding TreeLing node block (N) is updated to reclaim the unmapped node slots. Note that IvLeague performs an *in-place* update of NFL entries for TreeLing nodes with updated availability. Particularly, when node block N has an unmapped slot, IvLeague first checks the entries in the *current* NFL block. If the entry for node block N exists (i.e., a tag match), this entry is directly updated to track the newly available slot (Figure 8d). Otherwise, instead of *scanning* sequentially through NFL to find the NFL block that potentially tracks N's availability, IvLeague attempts to *re-use* one entry into the current NFL block with full slot occupation (e.g., N_4 in Figure 8e). If



(a) Original *page table* design (b) IvLeague *page table* with leaf ID

Fig. 9: Modified page table design in IvLeague.

no such entry is found, IvLeague will move *head* to the previous NFL block (B_{i-1}), and reuse an entry to track N there (Figure 8f). Note that the management of nodes in NFL ensures that all previous NFL blocks (before the location at *head*) are fully mapped. As a result, IvLeague only needs to move backwards one NFL block to find an entry for the tracking of a TreeLing node block with page deallocations. In cases where the *head* points at the very first NFL block in the current TreeLing, IvLeague can utilize the NFL from the previous TreeLing assigned to the same IV domain. This cross-TreeLing maintenance of available node slots allows IvLeague to efficiently track attachable IvLeague nodes with high utilization as data pages are allocated and freed.

2) Management of Leaf Mapping Metadata (LMM): IvLeague stores the mapping between PFNs to TreeLing leaf nodes in the page table. As illustrated in Figure 9a, the page table is a multi-level radix-tree structure, where each level is indexed by a portion of the virtual address bits and contains pointers to the data storage of the subsequent level. The last level contains the PFN corresponding to the virtual address (i.e., page table entry or PTE). IvLeague extends the PTE with an additional field to store the address of the leaf node mapped to the current page. Specifically, as shown in Figure 9b, each extended PTE reserves an additional 64 bits for the leaf ID. Due to such extension, each PTE page in IvLeague contains a reduced number of 256 PTE entries. The collection of the additional leaf IDs embedded in the page table is called Leaf Mapping Metadata, or LMM. When a page table walk is triggered, the LMM is separated from the page table data and stored in the LMM cache within the memory controller (Figure 5). When a TLB entry is evicted, the LMM cache entry is also evicted to maintain consistency.

D. Inter-TreeLing Management

1) Allocation of TreeLings to IV Domain: IvLeague manages the runtime allocation and deallocation of TreeLings to IV domains on-demand. When all leaf nodes within a TreeLing are fully occupied (i.e., no available α entries in

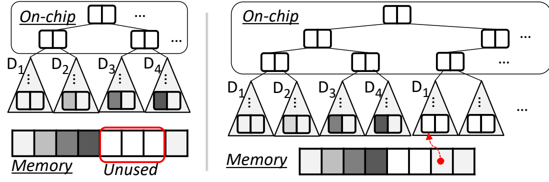


Fig. 10: Memory utilization in skewed allocation corresponding to number of TreeLings. Left: New allocation request from D_1 is failed due to TreeLing starvation, although memory is available; Right: New allocation request from D_1 is successful for the same memory distribution.

NFL), a new TreeLing is assigned to the domain. Specifically, IvLeague integrates an on-chip FIFO, called *Unassigned TreeLing* (Figure 5), to keep track of the currently unallocated TreeLings. Additionally, an *Assignment Table* (Figure 5) is used to track domains and their associated TreeLings. Note that these structures are only accessed during the TreeLing assignment process. We set the maximum number of IV domains supported to 2^{12}, which aligns with the limit on the number of contexts supported by hardware (i.e., Intel processors use 12-bit process-context IDs [81]).

2) Addressing TreeLing Starvation: Since IvLeague allocates IV tree nodes to security domains in the unit of TreeLing, there are chances when no TreeLing is available to attach newly allocated data pages (i.e., exhaustion of IvLeague before depletion of main memory). Figure 10a illustrates one such scenario with skewed memory footprints among domains. Provisioning additional TreeLings could mitigate this issue, as shown in Figure 10b. Particularly, the number of TreeLings needed to ensure full system memory coverage under the worst-case memory usage patterns across domains[2] can be modeled as: $\#\tau = (D-1) + \frac{M-(D-1)\times 4KB}{S}$, where D is the maximum number of IV domains to support, M is the total system memory, and S is the TreeLing size. Apparently, S and $\#\tau$ are inversely related given fixed D and M. When S is set to the smallest (e.g., covering one 4KB page), the combined coverage of all TreeLings is the same as M, indicating that no wasteful metadata is provisioned in memory. However, such minimal TreeLing size means that all leaf nodes have to be kept on-chip, which is impractical. Alternatively, when tuning $\#\tau$ to the theoretically minimal (i.e., D), the on-chip storage for TreeLing roots is minimized. Unfortunately, each TreeLing has to be large enough to cover the whole memory, bringing prohibitively high memory metadata storage overhead. Note that real-world workloads typically do not exhibit the worst-case behavior. As a result, TreeLing starvation can be empirically avoided by using an efficient tradeoff between on-chip and memory metadata overhead. In practice, we first determine $\#\tau$ in the system by identifying the level of the global tree nodes kept on chip, with reasonable storage overhead (e.g., according to IV metadata cache sizes). We then analyze the size of each TreeLing S, for this specific $\#\tau$ using the previous equation. Particularly, under IvLeague, to minimize the chance of starvation, the subtree (i.e., each of the $\#\tau$ equal splits from the global tree) is expanded with a limited number of levels (e.g., one to two) to form a TreeLing. We provide a detailed analysis of the selection of $\#\tau$ and S and their impact on overall system performance in Section X-C.

[2]The worst-case memory distribution occurs when all but one domain occupy only one data page, and the remaining domain occupies the rest of the available system memory.

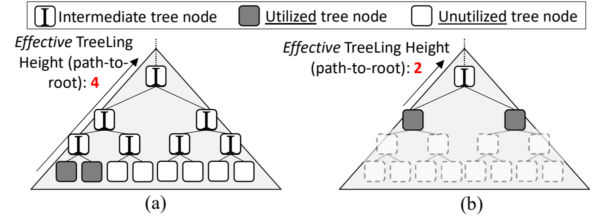


Fig. 11: Overview of IvLeague-Invert scheme.

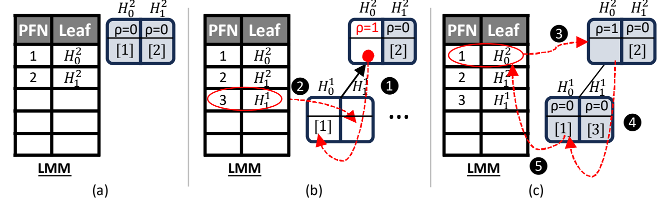


Fig. 12: IvLeague-Invert: a) allocation of page without introducing next level; b) operation to convert a slot to a parent slot; c) LMM update procedure after the conversion. Here H_i^l represents the address of the ith hash slot of the lth level of the tree. ρ is the `is_parent` flag and `[i]` denotes the hash corresponding to the attached page (PFN i).

VII. IVLEAGUE OPTIMIZATIONS

A. IvLeague-Invert: On-demand Extension of TreeLing

IvLeague-basic allocates pages at the leaf nodes of the tree only (Figure 11a). This can result in unnecessarily long integrity verification paths for programs with small memory footprints (i.e., the TreeLing is underutilized). Note that the IV operation overhead can be substantially reduced if tree nodes mapping to data pages can be collapsed towards the root compactly and expanded later within the TreeLing as more nodes are needed. Traditional statically-mapped global integrity tree does not allow tree node consolidation and extension, as only leaf nodes are mapped to data pages. Prior works [68] propose a pruning mechanism that restructures portions of the tree to enable such optimization in classical integrity trees. However, this approach incurs additional overhead for dynamic tree reconstruction. Interestingly, IvLeague's LMM structure in IvLeague naturally supports the use of intermediate nodes as leafs without requiring additional hardware support. Based on this observation, we propose an optimization over IvLeague (IvLeague-Invert) that enables data page to intermediate node mapping, shortening the path from leaf to root. As shown in Figure 11b, when a page is accessed, the LMM bookkeeps the address of the corresponding TreeLing node, which can be either a leaf or an intermediate node. Since TreeLing itself

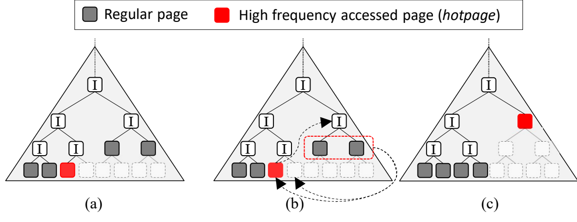
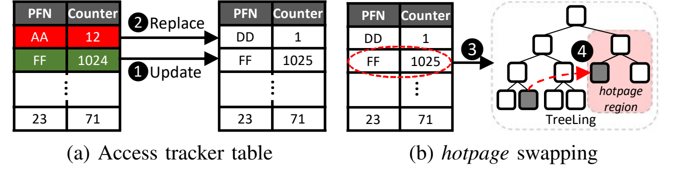


Fig. 13: High-level overview of IvLeague-Pro scheme.



(a) Access tracker table (b) *hotpage* swapping

Fig. 14: IvLeague integration with hotpage detection module.

is statically addressed, determining the path to the root for any node in the tree does not require indirection.

To support the assignment of intermediate nodes as leaf nodes, NFL is modified to track all nodes of TreeLing by placing the availability entries in NFL for TreeLing nodes *level by level* in a top-down manner. Upon a new mapping request, NFL first assigns *available* slots from the higher levels of TreeLing (ie H^2 in Figure 12a). When the NFL head reaches the *last* node block at the current level, IvLeague-Invert extends the effective tree by using nodes in the next level. This is achieved by first converting a tree node slot from the current level to a parent slot, followed by mapping pages to the corresponding child slots in the lower level. As shown in Figure 12b: ❶ IvLeague-Invert first copies the hash content of the selected H_0^2 slot into the *first available slot* of its child TreeLing block as determined by NFL (i.e., H_0^1). This ensures that the integrity tree metadata (i.e., hash) for H_0^2 is preserved when it becomes a parent slot. ❷ Afterward, new mapping requests are served starting from the subsequent available slots at this level (i.e., the H_1^1 slot in the H^1 level). To track whether a slot represents a parent or a leaf, a 1-bit `is_parent` flag (ρ) is reserved from each 64-bit hash slot in the TreeLing node. This conversion does not incur additional overhead, as H_1^1 would normally require its parent (H_0^2) for verification. Furthermore, LMM does not need to be updated immediately. As shown in Figure 12c, when the LMM is accessed after conversion, IvLeague-Invert first locates the old *leaf* H_0^2 using the LMM (❸) and detects that it has been converted into a parent. The *new leaf* for the page is now the first slot of H_0^2's child, H_0^1 (❹). Once the non-parent slot for the page is found, the LMM is updated to reflect the new leaf (❺). IvLeague-Invert enables dynamic extension within a TreeLing as the memory footprint of a domain grows, which can significantly reduce integrity verification overhead by decreasing the number of levels to be traversed.

B. IvLeague-Pro: Optimization for Frequently Accessed Pages

IvLeague-Basic maintains the same integrity verification length for all memory pages, regardless of their access frequency (similar to traditional secure architecture). However, in real-world workloads, a small portion of memory is accessed very frequently (i.e., hotpages) [68], [82], [83]. IvLeague-Pro further optimizes system performance by prioritizing integrity verification for these *hotpages*, placing them closer to the TreeLing root (Figure 13). A hotpage is determined by its access count (AC_i) divided by the accumulated access counts of all pages ($\sum_{j=1}^{n} AC_j$). Particularly, $Hotness(i) = \frac{AC_i}{\sum_{j=1}^{n} AC_j}$ quantifies the relative frequency of access to page i.

Figure 14 illustrates the IvLeague-Pro mechanism. Specifically, a small sub-region of the TreeLing (denoted as τ_{hot}) is reserved from the rest of the tree (τ_{reg}). τ_{hot} is only used for mapping hotpages. IvLeague-Pro discards the last level of nodes in τ_{hot}, thereby shortening the longest path compared to the original τ_{reg}. To support such scheme, two NFLs are maintained: a smaller one for τ_{hot} (NFL_{hot}) and a larger one for τ_{reg} (NFL_{reg}). Figure 14b demonstrates the use of an n-entry access frequency tracking table integrated into the memory controller, serving as a low-overhead hotpage detector. When a page is accessed, the corresponding entry in the tracker is updated. In case a new entry is needed and there is no free table entry, the entry with the smallest counter value is replaced. When a counter reaches a predefined frequency threshold, the associated page is migrated to the hotpage region (Figure 14a). This is done by first finding an available TreeLing node/slot from NFL_{reg}. The hash is then copied to the new TreeLing slot in (τ_{hot}), and the page to leaf mapping is updated in LMM. All counters within the table are cleared after a predefined fixed interval. When a hotpage becomes inactive (i.e., cleared from the tracker), the corresponding node in τ_{hot} is migrated back to τ_{reg}. This procedure is similar to the hotpage migration, with the distinction that NFL_{reg} is used to identify the available node. Note that the efficacy of the n-entry counter in IvLeague-Pro depends on the access striping of hotpages (i.e., the number of unique regular pages accessed between the accesses to the same hotpages) to be less than n. For more intricate access patterns, IvLeague-Pro can seamlessly integrate more advanced hotpage detection mechanisms [82], [83], [84], [85].

VIII. Security Analysis of IvLeague

Protection against Side Channels Exploiting Metadata Sharing. The root cause of the metadata-based information leakage [31], [32] is *sharing* the IV metadata *in memory* across domains. IvLeague fundamentally eliminates IV metadata sharing because: ❶ IvLeague assigns each TreeLing to a unique domain; ❷ no nodes are shared among TreeLings, and ❸ the roots of active TreeLings are kept on-chip. Therefore, IV operations in one domain cannot influence the timing of another, which indicates strong protection against the shared metadata-based leakage [32]. By default, IvLeague locks all TreeLing roots (i.e., one specific level of the global integrity tree) to cache during system power cycle. This ensures no leakage exists based on runtime TreeLing allocation activities.

Hardware	Configurations
Processor	8 OoO x86 Cores
L1 / L2 Cache	Private, 32KB, 8-way / Private, 1MB, 4-way
L3 Cache	Shared, 8MB, 16-way, 40-cycle hit
Crypto engine	20-cycle AES latency
Mem. Ctrl.	64 RD & WT queue, FR-FCFS, open-row 8-way 256KB counter & Tree cache
Main Memory	32GB, dual channel, 2 ranks/channel
IvLeague Params.	LMM cache: 16-way 204KB NFLB: 2 entries per-domain TreeLing size: 64MB; # of TreeLing: $4K$
Secure Architecture Configuration	
Enc. Counter	64-bit major, 7-bit minor counter
MAC	8 byte per block
Integrity Tree	8-ary Bonsai Merkle Tree
Metadata Cache	8-way 256KB counter and integrity tree caches

TABLE I: Architecture configurations.

Small (SPEC2017)	**S-1:** gcc-cactuBSSN-perlbench-deepsjeng **S-2:** mcf-omnetpp-lbm-xalancbmk **S-3:** bwaves-lbm-x264-cactuBSSN **S-4:** perlbench-xalancbmk-gcc-omnetpp **S-5:** mcf-bwaves-deepsjeng-x264 **S-6:** omnetpp-gcc-mcf-perlbench
Medium (PARSEC)	**M-1:** dedup-ferret-blackscholes-bodytrack **M-2:** canneal-swaptions-vips-ferret **M-3:** freqmine-fluidanimate-canneal-facesim **M-4:** vips-swaptions-dedup-ferret **M-5:** blackscholes-bodytrack-freqmine-fluidanimate **M-6:** dedup-facesim-bodytrack-swaptions
Large (Graph)	**L-1:** bfs-pr-bc-sssp **L-2:** bfs-pr-cc-tc **L-3:** bc-sssp-cc-tc **L-4:** sssp-pr-bc-tc

TABLE II: List of multi-programmed workloads.

Alternatively, a dynamic locking mechanism can be used where only the roots of *allocated* TreeLings are kept on-chip, which can reduce the runtime cache pressure. Prior studies have shown that dynamic resource allocation can potentially exfiltrate *coarse-grained* information across domains by observing the victim domain's resource needs [62], [86]. Note that recent works show that information leakage can be bounded to a low level with principled partitioning schemes (e.g., [62]). Overall, IvLeague can effectively eliminate the metadata sharing side channels by design.

Security Implication of Architecture Support for IvLeague. Microarchitectural components shared across domains can be exploited to carry out side channels without memory sharing (i.e., conflict-based cache attacks). Though not demonstrated in prior studies, such attacks (e.g., Prime+Probe) can be potentially applicable on the IvLeague secure processor caches (e.g., metadata and LMM cache). Note that IvLeague is specifically designed to thwart information leakage via *shared metadata* across security domains [32], which represents a fundamentally new attack vector that cannot be addressed with existing defense techniques (see Section IV). Moreover, secure cache techniques [35], [36], [63], [87], [88], [89], [90] are orthogonal and can be integrated into secure processors to form a baseline with classical side channel protection, upon which IvLeague is built to offer comprehensive side channel security. IvLeague utilizes dynamic mapping of the data pages to TreeLing leaf nodes through indirection from LMM. This mapping process is managed entirely by the hardware in the memory controller, preventing manipulation by off-chip attackers or malicious privileged software, similar to how virtual address to EPC frame mapping is protected against page table manipulation by a malicious OS in SGX. Finally, the in-memory NFL and NFLB are per-domain structures, which are not exploitable for cross-domain leakage.

Cryptographic Security of IvLeague. IvLeague maintains the same-level of cryptographic security assurance as in conventional secure architectures [65], [67], [68], [74], [75]. Specifically, IvLeague preserves the original physical structure of the integrity tree but dynamically splits the tree using TreeLings by sustaining an intermediate level of the tree in the cache. Integrity verification is performed from the direction of leaf to root, ending at the first node cached on-chip. IvLeague utilizes the same arity and hash size configuration as in the global integrity tree. Hence, it offers the same level of cryptographic security as the baselines.

IX. Experimental Setup

IvLeague Architecture Configurations. We implement and evaluate IvLeague in the gem5 simulator [91] under full-system simulation with Linux kernel 4.9.92. We first implement the state-of-the-art secure processor architecture [67] (***baseline***), which adopts an 8-ary Bonsai Merkle Tree. We simulate an eight-core out-of-order processor with dual-channel 32GB main memory. To enable protection against conflict-based attacks, the baseline is integrated MIRAGE [35], a representative randomized cache technique in the shared data cache (i.e., LLC) and metadata caches (i.e., counters and IV metadata). We implement the three variants of IvLeague on top of **baseline**, including IvLeague-Basic, IvLeague-Invert and IvLeague-Pro. Note that for IvLeague schemes, the MIRAGE defense is additionally applied on the LMM cache. IvLeague schemes use a $8K$-entry LMM cache and 2-entry NFLB per-domain. Each TreeLing is a 4-level subtree that covers 64MB of data. Also, under the evaluated system memory and TreeLing configuration, the global integrity tree height is 6-levels in *baseline* and 7-levels for all IvLeague schemes. To keep TreeLing roots on-chip, the processor performs locking of the first three levels (excluding the global root) in the IV metadata cache. Notably, this effectively makes Level 3 nodes the roots for TreeLings. IvLeague-Pro uses a per-domain 128-entry access frequency tracker with 8-bit counter for hotpage prediction. Table I illustrates the key architecture configurations for IvLeague.

Workload Configurations. We configure 16 multi-program workloads assembled using reasonably high memory intensive benchmarks from SPEC2017 [38], [92], [93], PARSEC3 [39], and the GAP Benchmark Suite [40]. Based on the combined memory footprint of each workload, we classify them based on their memory footprints as *small* ($<$ 5GB), *medium* ($5-10$GB), and *large* ($>$ 10GB). For the SPEC2017 and PARSEC3 workloads, we perform simulations using reference and native input sizes, respectively. For GAP benchmarks, we

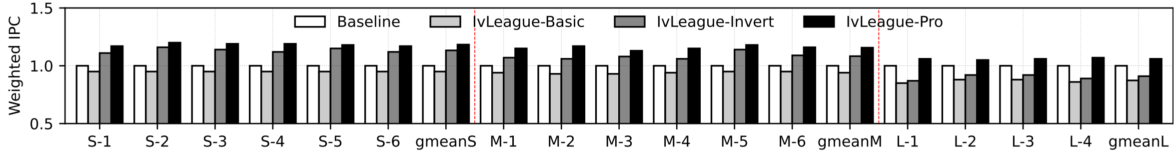


Fig. 15: Comparison of performance (i.e., Weighted IPC normalized to *Baseline*) under different schemes.

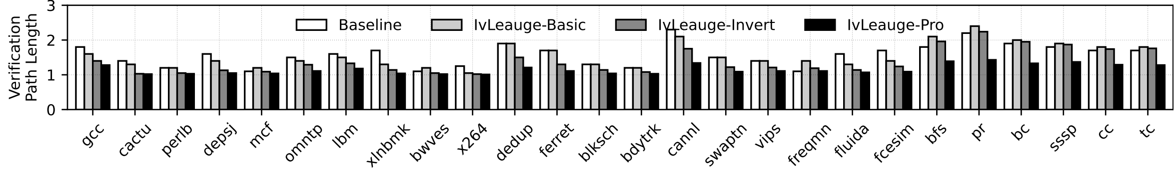


Fig. 16: Average integrity verification path length in IvLeague.

use twitter-large (5GB) graphs [94]. Each workload consists of four individual benchmark processes (single-threaded in small, multithreaded with two worker threads in medium and large), each process operating as separate IV domains. Multiple threads within the same process are grouped in the same IV domain. In the simulation setup, we configure gem5 to skip the initial *2B* and *5B* instructions for small/medium and large workloads, respectively, and collect statistics from detailed simulations over *1B* instructions for each core. Table II provides detailed configurations of all workloads.

X. Evaluation

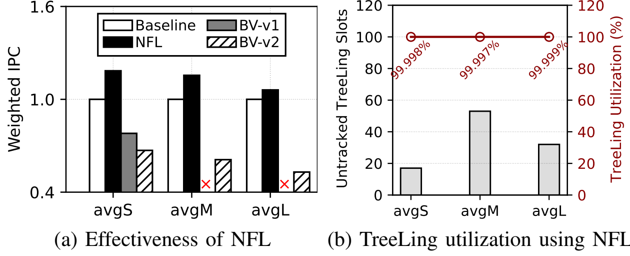
A. Performance Evaluations

1) System Performance Analysis of IvLeague Schemes: To comprehensively evaluate IvLeague, we analyze the overall system performance of various IvLeague schemes. Figure 15 illustrates the weighted IPC [95] for each IvLeague scheme, normalized to the default secure architecture with global integrity tree (i.e., *Baseline*). We observe that the basic IvLeague scheme (IvLeague-Basic), which features dynamic allocation of TreeLings, demonstrates a range of performance degradation compared to *Baseline*, including a modest 2.7% to 5.5% for *Small* and *Medium* workloads and 17.4% for *Large* on average. The performance impact with IvLeague-Basic comes mainly from additional memory accesses needed for leaf node management (i.e., using NFL and LMM), and additional accesses to tree nodes due to the expansion of the global tree (i.e., one additional level). These additional memory accesses can prolong the integrity verification latency. Note that such overhead is relatively more pronounced in the *Large* workloads with larger memory footprints, which can lead to higher misses in the IV metadata cache. The performance overhead of the basic scheme is mitigated with IvLeague-Invert, which reduces the length of the IV path by enabling the mapping of data pages to intermediate IvLeague nodes. Compared to IvLeague-Basic, IvLeague-Invert achieves an average of 10.9%, 8.8% and 3.3% IPC improvement over *Small*, *Medium* and *Large* workloads, respectively. Importantly, these reflect *performance gains* of 8.2% for *Small*, 3.4% for *Medium*, and a reduced overhead of 13.2% for *Large* over the insecure baseline. Furthermore, by applying the dynamic positioning of the nodes of hotpages closer to TreeLing roots, IvLeague-Pro *consistently* demonstrates *performance speedup* up to 19% (14% on average) across all workloads over *Baseline*, while preventing the metadata-based leakage in **Baseline**. Overall, IvLeague shows the promise of architecting secure processors resistant to side channel leakage without adversely impacting performance.

2) Integrity Verification Path Length with IvLeague: IvLeague ensures that the TreeLing roots are kept on-chip to enable metadata isolation. Since memory data on-chip are trusted, the height of TreeLing significantly influences the integrity verification path for a data block read. We profile the runtime integrity verification transactions for data reads (i.e., the TreeLing blocks read and verified up to a cached node). Figure 16 shows the average path length for each benchmark, computed across all workloads that contain it. We observe benchmarks with larger working sets (e.g., graph) show longer verification paths, due to higher metadata cache pressure. Under IvLeague-Basic, benchmarks within *Small* and *Medium* workloads have average path lengths of 1.31 and 1.52, respectively, which are shorter than those of *Baseline* (1.42 and 1.57). This reduction occurs because locking tree nodes from the TreeLing root level improves the locality of top-level nodes for data reads that require long tree traversals. With the use of intermediate nodes, IvLeague-Invert reduces the average length to 1.15 and 1.27 for *Small* and *Medium*. For *Large* workloads, IvLeague-Basic and IvLeague-Invert incur slightly longer verification path (2 and 1.92) over *Baseline* (1.85), which are also evidenced from the negative performance impact shown in Figure 15. Note that workloads with large memory introduce higher metadata cache access conflicts, under which the overhead due to the static TreeLing extension (see Section VII-A) becomes more influential. Finally, with the integration of IV optimization for hotpages, IvLeague-Pro decreases the average path to 1.08, 1.10, and 1.22 for *Small*, *Medium*, and *Large* workloads, leading to *performance improvement* across all workloads.

3) Runtime Efficiency Analysis of IvLeague Components: We investigate the effectiveness of IvLeague design and the overheads corresponding to various IvLeague operations:

Effectiveness of NFL Design in IvLeague. To understand the



(a) Effectiveness of NFL (b) TreeLing utilization using NFL

Fig. 17: Effectiveness of NFL. Left plot shows the performance under NFL and the naive bit vector implementations for *Small* (`avgS`), *Medium* (`avgM`) and *Large* (`avgL`) over baseline. × means runs unsuccessful; Right plot illustrates node utilization.

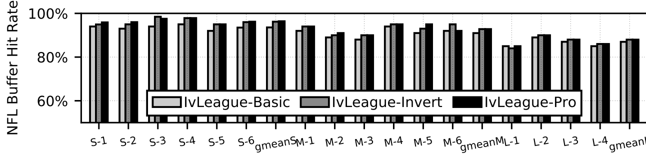


Fig. 18: NFLB hit rate for all workloads.

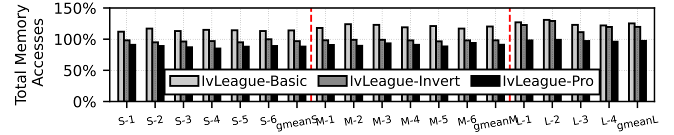
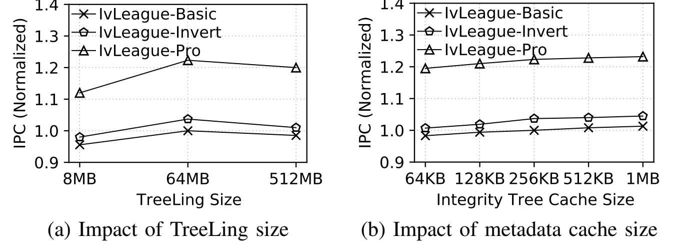


Fig. 19: Additional memory accesses due to IvLeague operations (normalized to *baseline* scheme).



(a) Impact of TreeLing size (b) Impact of metadata cache size

Fig. 20: Sensitivity analysis (normalized to IvLeague-Basic with 64MB TreeLing and 256KB metadata cache).

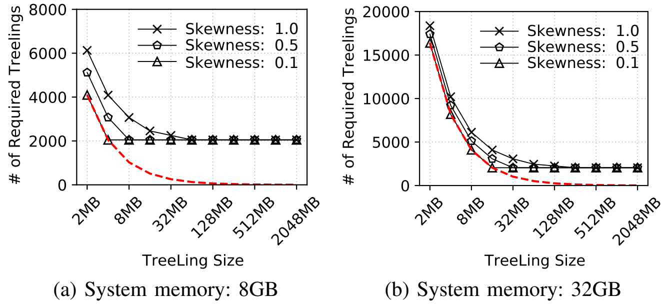
effectiveness of NFL, we replace it in IvLeague with a naive design that uses per TreeLing bit vectors. Specifically, each bit is statically mapped to a leaf node in TreeLing to indicate availability (e.g., '1' for occupied and '0' for free). Once *head* register points to the last active position among all bit vectors, and TreeLing node assignment is made to the first available node from the head. In case of deallocation, the head moves back to the freed node. We implement two different variants of such scheme: *BV-v1* and *BV-v2*. BV-v1 reacts to deallocations of nodes mapped to the currently active TreeLing (i.e., head not moving across TreeLings), and performs search (sequential scan) for free nodes in the current IvLeague only. In contrast, BV-v2 tracks node reclamation across TreeLings, and hence a cross-TreeLings search of free nodes is potentially needed for the allocation request. We run the same set of workloads in IvLeague (i.e., IvLeague-Pro version) with BV-v1 and BV-v2 and compare their performance with the NFL mechanism. As shown in Figure 17a, both schemes incur substantial performance overheads due to the expensive free node search operation, which delays normal data reads. Overall, there is a 33% to 47% performance loss in BV-v2 over *baseline*, compared to 6% to 18% performance gain in IvLeague (with NFL). Moreover, while BV-v1 performs slightly better than BV-v2 (i.e., 22% degradation over *Baseline*), it fails to accommodate leaf node mapping (for page allocations) in all *Medium* and *Large* workloads, as TreeLings quickly becomes exhausted when deallocation occurs across TreeLing. We further analyze the utilization of TreeLing nodes at runtime with NFL. Utilization is computed as the ratio of practically-used nodes over available nodes among all allocated TreeLings. Figure 17b reports the utilization ratio and the total number of untracked TreeLing nodes in the NFL. It is observed that only a small number of nodes (17-52) are untracked, which is negligible with respect to the overall number of TreeLing nodes actively mapped (a total of 2^{26} available TreeLing nodes). Overall, we observe that with NFL, IvLeague can achieve near-optimal tree node utilization ($> 99.99\%$) while maintaining low leaf node mapping overhead.

NFLB Hit Rate. The on-chip NFLB is accessed when there are page allocations and deallocations. Upon NFLB miss, the in-memory NFL is loaded (e.g., identifying free leaf nodes), which introduces additional metadata accesses overhead. As such, NFLB hit rate can have non-trivial impact on system performance. Figure 18 illustrates the NFLB hit rate for all workloads. Specifically, for *Small* and *Medium* workloads, the NFLB exhibits extremely high hit rates, with the average range between 91% and 96.5%. Furthermore, there is certain drop in the hit rate for the *Large* workloads due to a substantially higher activity of page deallocations originating from more diverse memory ranges. However, the NFLB still maintains a high hit rate of at least 86.9% across all IvLeague schemes.

Additional Memory Accesses with IvLeague. The introduction of NFL, integration of LMM in the page table, and expansion of TreeLing could incur additional memory accesses. We further perform studies to investigate the memory access overhead with the IvLeague mechanisms. Figure 19 presents the total memory accesses in IvLeague for the workloads normalized to *Baseline*. Specifically, we observe additional 14%-25% and 0%-15% memory accesses across all workloads for IvLeague-Basic and IvLeague-Invert over *Baseline*. Differently, IvLeague-Pro demonstrates a *reduction* of 3% to 9% total memory accesses. This improvement mainly stems from the reduced number of integrity tree node traversal from memory, particularly for high-frequency accessed pages.

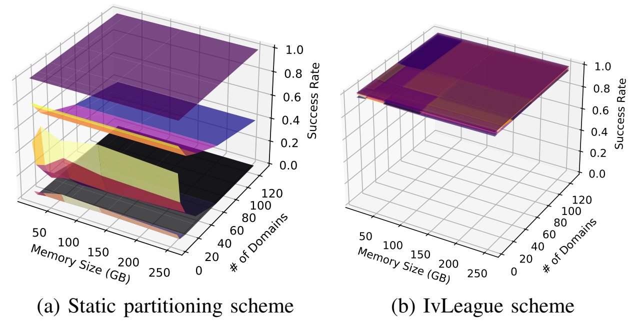
B. Sensitivity Analysis of IvLeague's Configurations

We perform several sensitivity studies for IvLeague configurations as follows:



(a) System memory: 8GB (b) System memory: 32GB

Fig. 21: TreeLings required under different memory allocation distributions across programs (number of IV Domains: 2^{12}). Red dashed line represents *minimum* number of TreeLing that covers the entire available memory (i.e., assuming all TreeLings are fully utilized).



(a) Static partitioning scheme (b) IvLeague scheme

Fig. 22: Comparison of different number of domain support without requiring memory swapping. From top to bottom: $\sum_{i=1}^{n} M_i = \langle 20\%, 40\%, 60\%, 80\% \rangle$.

Performance Impact on Sizes of TreeLing and Metadata Cache. First, we evaluate the performance impact of TreeLing when varying its size from 8, 64 to 512MB, corresponding to three, four and five levels inside TreeLings. Meanwhile, these configurations lead to locking of the top 5, 4 and 3 IV tree levels on-chip, respectively. Note that while keeping more levels on-chip enhances for top level nodes, this limits the usable cache space for intra-TreeLing node blocks, leading to potentially higher cache thrashing. As we observe from Figure 20a, the 64MB TreeLing has the highest performance across all configurations (up to 12% and 3% higher performance compared to 8 and 512MB TreeLing configurations respectively). While the performance gain from 8 to 64MB configuration is due to less cache thrashing, the performance degradation from 64 to 512MB configuration is primarily because of a higher amount of integrity tree cache misses in 512MB for tree nodes which are locked in cache in the 64MB configuration. This result shows the 64MB TreeLing configuration offers the best balance between levels inside TreeLing and on-chip locked blocks. We further explore the impact of different IV metadata cache sizes (from 64KB to 1MB) in IvLeague, normalized to the IvLeague-Basic with a 256KB cache. Figure 20b demonstrates average IPC (gmean) across all workloads for various IV metadata cache sizes between 64KB and 1MB. We observe that while in general larger cache shows higher system performance, the additional performance gains the IvLeague schemes are diminishing beyond the size of 256KB among all workloads.

TreeLing Size vs Number of TreeLings. We additionally perform empirical analysis on number of TreeLings required for different TreeLing sizes to sustain domains with different memory distributions (i.e., skewness). We define the *skewness factor* (S) as $S = \frac{M_{max}}{M_{total}}$; where M_{max} is the memory footprint for the domain with the largest memory usage, and M_{total} denotes the total memory footprint among all domains. A higher value of S (i.e., closer to 1) represents a higher variance in memory footprint across domains. Figure 21 highlights the trend of minimum number of TreeLings required to support different skewness and different TreeLing size (for two system memory configurations: 8GB and 32GB). One clear trend we observe for both is the number of required TreeLings significantly reduces as the TreeLing size increases (up to a certain TreeLing size). This is because larger TreeLing can cover a larger memory area, requiring a smaller number of TreeLings. However, beyond a specific TreeLing size, the number of required TreeLing observes minimal change (i.e., beyond 64MB TreeLing across both memory configurations). This is because a certain number of TreeLing is required to provide isolation across the supported number of domains, regardless of the coverage area of individual TreeLing. We observe this trend for all configured skewness and system memory configurations. Overall, this result highlights that 64MB TreeLing provides a good balance between the required number of TreeLings and the length of individual TreeLing.

C. Scalability Analysis of IvLeague

One of the key advantages of IvLeague is that it can support a dynamic range of IV domains with varying runtime memory footprints. In contrast, static partitioning schemes (e.g., [31]) can only support up to a fixed amount of memory per domain, which depends on the total number of partitions in the system.

To compare IvLeague's support for domains with highly dynamic range of memory requirements, we analytically model partition management and memory allocation in static partitioning. For a fixed number of partitions P and memory occupied by each domain M_i, the success/failure of static partitioning is defined as whether the domains can be scheduled *without requiring memory swapping* (i.e., when $\forall i,\ M_i \leq S$; where S is the size of a partition). We empirically calculate the success rate for a specific configuration by: i) generating random memory footprint per domain such that $\sum_{i=1}^{n} M_i = Fixed\ Percentage\ of\ T$ (T is the total system memory); ii) for each generated memory usages among domains, check if the condition $\forall i,\ M_i \leq S$ is satisfied. If the condition is met, this case is successful. We perform similar TreeLing exhaustion analysis on IvLeague. The scalability experiment is done by configuring multiple different levels of system memory utilization (20% to 80%). For each level, we vary

Component	Storage	Area
NFL Logic and Buffer	528-byte	$0.0071mm^2$
LMM Cache	204KB	$0.33mm^2$
Hotpage Predictor (IvLeague-Pro)	848-byte	$0.018mm^2$

TABLE III: On-chip hardware cost for IvLeague components.

the number of active domains from 8 to 128, and the total system memory between 8 to 256GB. Figure 22 shows the success rates of running these configurations (with a fixed 4096 TreeLings). We observe that static partitioning only has a high success rate when system memory utilization is low (i.e., <20%). Its success rate falls significantly with higher memory utilization (i.e., >40%), and with larger number of runtime domains (i.e., >32). In contrast, IvLeague consistently maintains a success rate of $> 98\%$ across all configurations.

D. Hardware Overhead Analysis

IvLeague requires on-chip hardware support and off-chip storage for the proposed mechanisms. Table III provides an overview of the main hardware overheads (storage and area) based on the default setup (Section IX). We evaluate the area overheads using CACTI [96] with 45nm process node. Specifically, IvLeague utilizes a 204KB on-chip LMM cache. For each core, IvLeague maintains a 128-byte NFL on-chip buffer and a 4-bit NFL head register in the memory controller. In addition, IvLeague-Pro integrates a 128-entry hotpage predictor (848 bytes of on-chip storage) per-core. The overall on-chip area overhead is 0.3551mm^2 for the evaluated configuration, which is negligible with respect to the typical chip area of modern processors [97]. Moreover, our IvLeague implementation does not require additional on-chip storage for metadata isolation. Instead, a portion of the IV metadata cache (32 KB out of 256 KB) is reserved to lock the TreeLing roots. In terms of off-chip storage, each TreeLing node requires 64 bits of NFL metadata (i.e., 56-bit tag and 8-bit availability bit vector), leading to a total of 16MB of system memory storage for all TreeLings (0.05%). Additionally, due to the integration of LMM, each page table entry (PL1) is additionally associated with 64-bit leaf ID. Finally, the globally-formed integrity tree from TreeLings is one-level taller than the static tree in *Baseline*, resulting in use of additional 0.7% usage of system memory for IV metadata (0.89% in *Baseline*).

XI. Related Works

As TEE designs mature and are increasingly adopted in real-world applications, the focus on preventing side channel vulnerabilities in TEE architectures has gained significant attention. Consequently, TEE-centric defenses are necessary to mitigate side channel leakage in these scenarios. Unlike protection mechanisms in classical side channels that can rely on support from system software, leakage prevention mechanisms in TEE environments have to address the potential threats from privileged attacks that possess powerful system-level control (e.g., instruction stepping and replay [3], [79]). Therefore, hardware-based side channel protections are necessary against TEE-enabled attacks. Prior works([18], [21], [31])) propose to mitigate side channels in TEEs by partitioning or isolating shared microarchitectural components for enclaves. For instance, MI6 [18] provisions major shared hardware components (e.g., caches and on-chip networks) for enclaves to defeat cross-core contention-based attacks, and performs cache flushing upon context switches to disrupt same-core leakage [18]). Cachelets [21] enables dynamic on-demand allocation of fine-grained cache regions to enclaves through set and way-based cache partitioning. Bespoke [19] leverages set-wise cache partitioning through a flexible cache set indexing mechanism, isolating cache accesses from specific domains into dedicated sets. Additionally, recent secure cache designs [35], [36], [63], [89] employ cache access randomization (e.g., address to set mapping) to thwart conflict-based attacks on caches with various security guarantees. These techniques effectively protect against contention-based information leakage on shared hardware and can be integrated with IvLeague for holistic leakage protection in secure processors.

Alongside efforts in microarchitecture security, ongoing research aims to enhance the performance of secure architecture designs. Despite the commercial adoption of secure processor architectures (TEE), their performance overhead compared to non-TEE execution remains substantial. Recent architecture-level optimizations for secure architectures include innovations such as using variable-arity integrity trees (e.g., VAULT [65]) to minimize metadata maintenance overhead. Synergy [66] re-purposes the ECC chip for MAC storage, eliminating memory traffic overhead for MAC accesses. Morphable counter [73] employs a compact counter structure with overflow prevention to reduce page re-encryptions. Each of these techniques targets different components of secure architectures. IvLeague operates independently of these optimizations and has the potential to integrate them, further enhancing performance without compromising security guarantees.

XII. Conclusion

Secure processors utilize tree-based integrity verification to protect off-chip data. However, this integrity tree is a global structure shared across the security domain. This can introduce severe side channel leakage through shared integrity tree metadata. In this work, we present IvLeague, an architecture framework technique that provides cross-domain isolation of integrity tree. IvLeague breaks the integrity tree into many small isolated TreeLings, which are allocated to domains to provide isolation. We further propose two optimizations: i) IvLeague-Invert, which reduces the integrity verification path by utilizing intermediate tree nodes as leafs; and ii) IvLeague-Pro, which tracks and places hotpages closer to root. Overall, IvLeague scheme offers upto 19% speedup over insecure baseline, while providing proper side channel protection for shared integrity tree metadata.

XIII. Acknowledgements

This work is supported in part by U.S. National Science Foundation under CAREER Award CNS-2340777 and CNS-2008339.

References

[1] V. Costan and S. Devadas, "Intel SGX Explained." *IACR Cryptol. ePrint Arch.*, 2016.

[2] D. Lee, D. Jung, I. T. Fang, C.-C. Tsai, and R. A. Popa, "An off-chip attack on hardware enclaves via the memory bus," in *USENIX Security*, 2020.

[3] J. Van Bulck, F. Piessens, and R. Strackx, "SGX-Step: A practical attack framework for precise enclave execution control," in *ACM SsyTEX*, 2017.

[4] S. Johnson, R. Makaram, A. Santoni, and V. Scarlata, "Supporting intel sgx on multi-socket platforms," *Intel Corporation*, 2021.

[5] A. Ahmad, K. Kim, M. I. Sarfaraz, and B. Lee, "OBLIVIATE: A Data Oblivious Filesystem for Intel SGX." in *IEEE NDSS*, 2018.

[6] V. Karande, E. Bauman, Z. Lin, and L. Khan, "Sgx-log: Securing system logs with sgx," in *ACM AsiaCCS*, 2017.

[7] D. Harnik, E. Tsfadia, D. Chen, and R. Kat, "Securing the storage data path with SGX enclaves," *arXiv preprint arXiv:1806.10883*, 2018.

[8] M. Yan, J.-Y. Wen, C. W. Fletcher, and J. Torrellas, "Secdir: a secure directory to defeat directory side-channel attacks," in *IEEE ISCA*, 2019.

[9] W. Xiong and J. Szefer, "Survey of transient execution attacks," *ACM Computing Surveys*, 2021.

[10] C. Canella, J. Van Bulck, M. Schwarz, M. Lipp, B. Von Berg, P. Ortner, F. Piessens, D. Evtyushkin, and D. Gruss, "A systematic evaluation of transient execution attacks and defenses," in *USENIX Security*, 2019.

[11] D. Gruss, R. Spreitzer, and S. Mangard, "Cache template attacks: Automating attacks on inclusive {Last-Level} caches," in *USENIX Security*, 2015.

[12] A. Moghimi, G. Irazoqui, and T. Eisenbarth, "Cachezoom: How sgx amplifies the power of cache attacks," in *Springer CHES*, 2017.

[13] Y. Yarom and K. Falkner, "Flush+reload: A high resolution, low noise, l3 cache side-channel attack," in *USENIX Security*, 2014.

[14] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee, "Last-level cache side-channel attacks are practical," in *IEEE S&P*, 2015.

[15] F. Yao, H. Fang, M. Doroslovački, and G. Venkataramani, "COTSknight: Practical defense against cache timing channel attacks using cache monitoring and partitioning technologies," in *IEEE HOST*, 2019.

[16] M. H. I. Chowdhuryy and F. Yao, "Leaking secrets through modern branch predictor in the speculative world," *IEEE TC*, 2021.

[17] S. Constable, J. Van Bulck, X. Cheng, Y. Xiao, C. Xing, I. Alexandrovich, T. Kim, F. Piessens, M. Vij, and M. Silberstein, "Aex-notify: Thwarting precise single-stepping attacks through interrupt awareness for intel sgx enclaves," in *USENIX Security*, 2023.

[18] T. Bourgeat, I. Lebedev, A. Wright, S. Zhang, Arvind, and S. Devadas, "Mi6: Secure enclaves in a speculative out-of-order processor," in *IEEE MICRO*, 2019.

[19] G. Saileshwar, S. Kariyappa, and M. Qureshi, "Bespoke cache enclaves: Fine-grained and scalable isolation from cache side-channels via flexible set-partitioning," in *IEEE SEED*, 2021.

[20] D. Lee, D. Kohlbrenner, S. Shinde, K. Asanović, and D. Song, "Keystone: An open framework for architecting trusted execution environments," in *ACM EuroSys*, 2020.

[21] D. Townley, K. Arıkan, Y. D. Liu, D. Ponomarev, and O. Ergin, "Composable cachelets: Protecting enclaves from cache side-channel attacks," in *USENIX Security*, 2022.

[22] H. Omar and O. Khan, "Ironhide: A secure multicore that efficiently mitigates microarchitecture state attacks for interactive applications," in *IEEE HPCA*, 2020.

[23] T. Yavuz, F. Fowze, G. Hernandez, K. Y. Bai, K. R. Butler, and D. J. Tian, "Encider: detecting timing and cache side channels in sgx enclaves and cryptographic apis," *IEEE TDSC*, 2022.

[24] F. Lang, W. Wang, L. Meng, J. Lin, Q. Wang, and L. Lu, "Mole: Mitigation of side-channel attacks against sgx via dynamic data location escape," in *ACM ACSAC*, 2022.

[25] D. Kaplan, J. Powell, and T. Woller, "AMD memory encryption," *AMD*, 2016.

[26] S. Pinto and N. Santos, "Demystifying arm trustzone: A comprehensive survey," *ACM CSUR*, 2019.

[27] G. Dhanuskodi, S. Guha, V. Krishnan, A. Manjunatha, M. O'Connor, R. Nertney, and P. Rogers, "Creating the first confidential gpus: The team at nvidia brings confidentiality and integrity to user code and data for accelerated computing." *ACM Queue*, 2023.

[28] M. Schwarz, M. Lipp, D. Moghimi, J. Van Bulck, J. Stecklina, T. Prescher, and D. Gruss, "Zombieload: Cross-privilege-boundary data sampling," in *ACM CCS*, 2019.

[29] S. Lee, M.-W. Shih, P. Gera, T. Kim, H. Kim, and M. Peinado, "Inferring fine-grained control flow inside sgx enclaves with branch shadowing," in *USENIX Security*, 2017.

[30] M. Dai, R. Paccagnella, M. Gomez-Garcia, J. McCalpin, and M. Yan, "Don't mesh around:side-channel attacks and mitigations on mesh interconnects," in *USENIX Security*, 2022.

[31] 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 *IEEE ISCA*, 2020.

[32] M. H. I. Chowdhuryy, H. Zheng, and F. Yao, "Metaleak: Uncovering side channels in secure memory architectures exploiting metadata," in *IEEE ISCA*, 2024.

[33] V. Docs, "Runtime Encryption of Memory with Intel® Total Memory Encryption–Multi-Key (Intel® TME-MK)." [Online]. Available: https://docs.vmware.com/en/VMware-vSphere/7.0/com.vmware.vsphere.resmgmt.doc/GUID-F9111E35-E197-46EC-8350-77827A5A2DEC.html

[34] M. Yan, R. Sprabery, B. Gopireddy, C. Fletcher, R. Campbell, and J. Torrellas, "Attack directories, not caches: Side channel attacks in a non-inclusive world," in *IEEE S&P*, 2019.

[35] G. Saileshwar and M. Qureshi, "Mirage: Mitigating conflict-based cache attacks with a practical fully-associative design," in *USENIX Security*, 2021.

[36] M. K. Qureshi, "Ceaser: Mitigating conflict-based cache attacks via encrypted-address and remapping," in *IEEE MICRO*, 2018.

[37] F. Liu, H. Wu, K. Mai, and R. B. Lee, "Newcache: Secure cache architecture thwarting cache side-channel attacks," *IEEE Micro*, 2016.

[38] S. P. E. Corporation, "SPEC CPU2017 ," 2017. [Online]. Available: www.spec.org/cpu2017

[39] X. Zhan, Y. Bao, C. Bienia, and K. Li, "Parsec3.0: A multicore benchmark suite with network stacks and splash-2x," *ACM CAN*, 2017.

[40] S. Beamer, K. Asanović, and D. Patterson, "The GAP Benchmark Suite," *arXiv preprint arXiv:1508.03619*, 2015.

[41] Z. Wang and R. B. Lee, "Covert and side channels due to processor architecture," in *IEEE ACSAC*, 2006.

[42] X. Ren, L. Moody, M. Taram, M. Jordan, D. M. Tullsen, and A. Venkat, "I see dead μops: Leaking secrets via intel/amd micro-op caches," in *IEEE ISCA*, 2021.

[43] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, "Spectre attacks: Exploiting speculative execution," in *IEEE S&P*, 2019.

[44] D. A. Osvik, A. Shamir, and E. Tromer, "Cache attacks and countermeasures: the case of aes," in *Springer CT-RSA*, 2006.

[45] F. Liu and R. B. Lee, "Random fill cache architecture," in *IEEE MICRO*, 2014.

[46] F. Yao, M. Doroslovacki, and G. Venkataramani, "Are coherence protocol states vulnerable to information leakage?" in *IEEE HPCA*, 2018.

[47] M. H. I. Chowdhuryy, R. Ewetz, A. Awad, and F. Yao, "R-saw: New side channels exploiting read asymmetry in mlc phase change memories," in *IEEE SEED*, 2021.

[48] M. H. I. Chowdhuryy, H. Liu, and F. Yao, "BranchSpec: Information Leakage Attacks Exploiting Speculative Branch Instruction Executions," in *IEEE ICCD*, 2020.

[49] M. H. I. Chowdhuryy, R. Ewetz, A. Awad, and F. Yao, "Understanding and characterizing side channels exploiting phase-change memories," *IEEE Micro*, 2023.

[50] F. Yao, G. Venkataramani, and M. Doroslovački, "Covert timing channels exploiting non-uniform memory access based architectures," in *ACM GLSVLSI*, 2017.

[51] F. Yao, M. Doroslovački, and G. Venkataramani, "Covert timing channels exploiting cache coherence hardware: Characterization and defense," *Springer IJPP*, 2019.

[52] Z. Zhan, Z. Zhang, S. Liang, F. Yao, and X. Koutsoukos, "Graphics peeping unit: Exploiting em side-channel information of gpus to eavesdrop on your neighbors," in *IEEE S&P*, 2022.

[53] Z. Zhang, S. Liang, F. Yao, and X. Gao, "Red alert for power leakage: Exploiting intel rapl-induced side channels," in *ACM CCS*, 2021.

[54] Z. Zhang, K. Cai, Y. Guo, F. Yao, and X. Gao, "Invalidate+compare: A timer-free gpu cache attack primitive," in *USENIX Security*, 2024.

[55] A. S. Rakin, M. H. I. Chowdhuryy, F. Yao, and D. Fan, "Deepsteal: Advanced model extractions leveraging efficient weight stealing in memories," 2022.

[56] K. Cai, M. H. I. Chowdhuryy, Z. Zhang, and F. Yao, "Deepvenom: Persistent dnn backdoors exploiting transient weight perturbations," in *IEEE S&P*, 2024.

[57] F. Yao, A. S. Rakin, and D. Fan, "Deephammer: Depleting the intelligence of deep neural networks through targeted chain of bit flips," in *USENIX Security*, 2020.

[58] S. van Schaik, M. Minkin, A. Kwong, D. Genkin, and Y. Yarom, "Cacheout: Leaking data on intel cpus via cache evictions," in *IEEE S&P*, 2021.

[59] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin *et al.*, "Meltdown: Reading kernel memory from user space," in *USENIX Security*, 2018.

[60] D. Skarlatos, M. Yan, B. Gopireddy, R. Sprabery, J. Torrellas, and C. W. Fletcher, "Microscope: Enabling microarchitectural replay attacks," in *IEEE ISCA*, 2019.

[61] A. Bhattacharyya, A. Sandulescu, M. Neugschwandtner, A. Sorniotti, B. Falsafi, M. Payer, and A. Kurmus, "SMoTherSpectre: Exploiting Speculative Execution through Port Contention," in *ACM CCS*, 2019.

[62] Z. N. Zhao, A. Morrison, C. W. Fletcher, and J. Torrellas, "Untangle: A principled framework to design low-leakage, high-performance dynamic partitioning schemes," in *ACM ASPLOS*, 2023.

[63] W. Song, B. Li, Z. Xue, Z. Li, W. Wang, and P. Liu, "Randomized last-level caches are still vulnerable to cache side-channel attacks! but we can fix it," in *IEEE S&P*, 2021.

[64] Y. Mo and B. Sinopoli, "Secure control against replay attacks," in *IEEE Allerton*, 2009.

[65] M. Taassori, A. Shafiee, and R. Balasubramonian, "Vault: Reducing paging overheads in sgx with efficient integrity verification structures," in *ACM ASPLOS*, 2018.

[66] G. Saileshwar, P. J. Nair, P. Ramrakhyani, W. Elsasser, and M. K. Qureshi, "Synergy: Rethinking secure-memory design for error-correcting memories," in *IEEE HPCA*, 2018.

[67] 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 *IEEE MICRO*, 2007.

[68] A. Freij, H. Zhou, and Y. Solihin, "Bonsai merkle forests: Efficiently achieving crash consistency in secure persistent memory," in *IEEE MICRO*, 2021.

[69] M. H. I. Chowdhuryy, M. Jung, F. Yao, and A. Awad, "D-shield: Enabling processor-side encryption and integrity verification for secure nvme drives," in *IEEE HPCA*, 2023.

[70] B. Gassend, G. E. Suh, D. Clarke, M. Van Dijk, and S. Devadas, "Caches and hash trees for efficient memory integrity verification," in *IEEE HPCA*, 2003.

[71] G. E. Suh, D. Clarke, B. Gasend, M. Van Dijk, and S. Devadas, "Efficient memory integrity verification and encryption for secure processors," in *IEEE MICRO*, 2003.

[72] C. Yan, D. Englender, M. Prvulovic, B. Rogers, and Y. Solihin, "Improving cost, performance, and security of memory encryption and authentication," *ACM CAN*, 2006.

[73] 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 *IEEE MICRO*, 2018.

[74] M. K. Qureshi, M. M. Franceschini, L. A. Lastras-Montaño, and J. P. Karidis, "Morphable memory system: A robust architecture for exploiting multi-level phase change memories," in *IEEE ISCA*, 2010.

[75] S. Gueron, "Memory encryption for general-purpose processors," *IEEE Security & Privacy*, 2016.

[76] J. A. Halderman, S. D. Schoen, N. Heninger, W. Clarkson, W. Paul, J. A. Calandrino, A. J. Feldman, J. Appelbaum, and E. W. Felten, "Lest we remember: cold-boot attacks on encryption keys," *USENIX Security*, 2009.

[77] M. Lipp, A. Kogler, D. Oswald, M. Schwarz, C. Easdon, C. Canella, and D. Gruss, "Platypus: Software-based power side-channel attacks on x86," in *IEEE S&P*, 2021.

[78] Z. Kenjar, T. Frassetto, D. Gens, M. Franz, and A.-R. Sadeghi, "V0ltpwn: Attacking x86 processor integrity from software," in *USENIX Security*, 2020.

[79] M. H. I. Chowdhuryy, Z. Zhang, and F. Yao, "Powspectre: Powering up speculation attacks with tsx-based replay," in *ACM ASIACCS*, 2024.

[80] The OpenSSL Project, "OpenSSL: The open source toolkit for SSL/TLS (v. 0.9.7)," 2006. [Online]. Available: https://www.openssl.org

[81] Linux, "Process Context IDentifiers." [Online]. Available: https://www.kernel.org/doc/Documentation/x86/pti.

[82] D. Xu, J. Ryu, K. Shin, P. Su, and D. Li, "{FlexMem}: Adaptive page profiling and migration for tiered memory," in *USENIX ATC*, 2024.

[83] M. Alwadi, A. Mohaisen, and A. Awad, "Promt: optimizing integrity tree updates for write-intensive pages in secure nvms," in *ACM ICS*, 2021.

[84] T. Lee, S. K. Monga, C. Min, and Y. I. Eom, "Memtis: Efficient memory tiering with dynamic page classification and page size determination," in *ACM SOSP*, 2023.

[85] R. Wang, S. Mittal, Y. Zhang, and J. Yang, "Decongest: Accelerating super-dense pcm under write disturbance by hot page remapping," *IEEE CAL*, 2017.

[86] G. E. Suh, L. Rudolph, and S. Devadas, "Dynamic partitioning of shared cache memory," *The Journal of Supercomputing*, 2004.

[87] T. Bourgeat, J. Drean, Y. Yang, L. Tsai, J. Emer, and M. Yan, "Casa: End-to-end quantitative security analysis of randomly mapped caches," in *IEEE MICRO*, 2020.

[88] V. Kiriansky, I. Lebedev, S. Amarasinghe, S. Devadas, and J. Emer, "DAWG: A defense against cache timing attacks in speculative execution processors," in *IEEE MICRO*, 2018.

[89] L. Giner, S. Steinegger, A. Purnal, M. Eichlseder, T. Unterluggauer, S. Mangard, and D. Gruss, "Scatter and split securely: Defeating cache contention and occupancy attacks," in *IEEE S&P*, 2023.

[90] M. Werner, T. Unterluggauer, L. Giner, M. Schwarz, D. Gruss, and S. Mangard, "Scattercache: thwarting cache attacks via cache set randomization," in *USENIX Security*, 2019.

[91] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti *et al.*, "The gem5 simulator," *ACM CAN*, 2011.

[92] S. Singh and M. Awasthi, "Memory centric characterization and analysis of spec cpu2017 suite," in *ACM ICPE*, 2019.

[93] A. Limaye and T. Adegbija, "A workload characterization of the spec cpu2017 benchmark suite," in *IEEE ISPASS*, 2018.

[94] H. Kwak, C. Lee, H. Park, and S. Moon, "What is twitter, a social network or a news media?" in *IEEE IW3C2*, 2010.

[95] V. Young, P. J. Nair, and M. K. Qureshi, "DEUCE: Write-efficient encryption for non-volatile memories," *ACM ASPLOS*, 2015.

[96] R. Balasubramonian, A. B. Kahng, N. Muralimanohar, A. Shafiee, and V. Srinivas, "Cacti 7: New tools for interconnect exploration in innovative off-chip memories," *ACM TACO*, 2017.

[97] "Intel® core™ i7-930 processor (8m cache, 2.80 ghz, 4.80 gt/s intel® qpi) product specifications," 2010. [Online]. Available: https://ark.intel.com/content/www/us/en/ark/products/41447/intel-core-i7-930-processor-8m-cache-2-80-ghz-4-80-gt-s-intel-qpi.html