MetaLeak: Uncovering Side Channels in Secure Processor Architectures Exploiting Metadata

Md Hafizul Islam Chowdhuryy, Hao Zheng and Fan Yao
Department of ECE, University of Central Florida
hafizul.islam@ucf.edu, hao.zheng@ucf.edu, fan.yao@ucf.edu

***Abstract*—Microarchitectural side channels raise severe security concerns. Recent studies indicate that microarchitecture security should be examined holistically (rather than separately) in systems. Although the effects of performance optimizations on side channels are widely studied, the impacts of integrating *security mechanisms* intended for other threats on microarchitecture security are not well explored.**

In this paper, we perform the first side channel exploration in secure processor architectures that offer data confidentiality and integrity protection through hardware. We investigate microarchitecture security in the *design space* of secure processors and identify unique properties in the underlying *metadata management* schemes, which can be leveraged for new information leakage attacks. We present MetaLeak, an end-to-end side channel attack framework that exploits timing variations due to metadata maintenance to exfiltrate program secrets in secure processors. Particularly, we present two variants of the attack: MetaLeak-T that exploits the sharing of integrity tree metadata, and MetaLeak-C that manipulates counter metadata states. Our evaluation first shows highly accurate covert communication using the security metadata that can operate across cores and sockets without explicit data sharing. We further perform extensive side channel case studies on state-of-the-art secure architecture designs as well as the SGX processors. Our results show that MetaLeak can successfully exfiltrate program secrets (up to 97% accuracy) from image-processing application and cryptographic software running in enclave. Our study indicates that the fundamental metadata mechanism is the root cause of the leakage, which necessitates the use of leakage-taming techniques in future secure processors. This work highlights the need to synergistically understand microarchitecture security, as new security mechanisms are integrated.

***Index Terms*—Microarchitecture security, Side channel, Secure architectures, Trusted execution environment, Metadata**

I. Introduction

Information leakage through microarchitectural attacks has raised critical security concerns in modern and emerging computing systems [1], [2], [3], [4], [5], [6], [7], [8]. These attacks can exfiltrate secretive data by modulating microarchitectural states (e.g., cache occupancy) during a victim program's execution. While a plethora of prior works have studied side channel vulnerabilities in existing processor components individually, recent advances have shown that microarchitecture security cannot be considered as a standalone problem [9], [10]. As the community increasingly advocates for secure-by-design architectures and proposes deploying more security countermeasures from *different aspects* into hardware, it is critical to understand the **composability of security**. Specifically, *how does the introduction of certain security mechanism for one threat reshape the hardware attack surface of another*?

Alongside the efforts to enhance microarchitecture security, trusted computing has rapidly gained attention in recent years due to the growing concerns of trust in remote computing platforms (e.g., cloud computing). Under such a threat model, adversaries can launch privileged software or physical attacks to compromise program data either *in transit* or *at rest* [11], [12], [13]. State-of-the-art trusted computing mechanisms employ *secure processor architectures* [12], [13], [14], [15], [16], [17] that take the CPU as the root of trust (e.g., trusted execution environment or TEE) and provide robust data protection through hardware. Best practices of secure processors perform encryption and integrity verification with the help of processor-maintained security metadata [12]. Variants of such approaches are adopted in commercialized solutions (e.g., Intel SGX [11]), where off-chip data security is safeguarded inside an enclave. Although there have been many studies on microarchitectural attacks and secure processor architectures *alone*, the security impact of supporting secure architectures on microarchitectural information leakage has not been well understood. Note that while prior works have investigated SGX side channels [18], [19], [20], [21], [22], [23], they primarily focus on exploring elevated side channels due to attackers' capability of privileged access, rather than identifying new leakage vulnerabilities within the *underlying architectural mechanisms* in secure processors. With the increasing demand for both *on-chip* and *off-chip* security, understanding the interplay between microarchitecture security and secure processor designs becomes imperative.

This paper explores microarchitecture security vulnerabilities in secure architectures. We systematically examine the design space of secure architectures in both state-of-the-art academic proposals [12], [14], [15], [16], [24] and the commercial-off-the-shelf SGX hardware [11], [25]. Our investigation reveals the key finding that the fundamental *metadata* management mechanism introduces new on-chip information leakage threats. *Firstly*, unlike in conventional architectures, memory accesses in secure processors exhibit abundant and highly distinguishable slow/fast paths due to the complicated interactions with metadata. *Secondly*, certain types of metadata (i.e., integrity tree counters) are incremented along with data block writes, which would introduce tremendous variations in memory accesses upon overflow. *Finally*, the use of metadata by design enables implicit sharing of memory (i.e., integrity

tree blocks) across domains, leading to side channels through metadata even when regular data sharing among untrusted domains is prohibited (e.g., to defeat shared memory attacks [3]).

Based on the above observations, we present **MetaLeak**, a novel side channel framework in secure processors that exploits the secret-dependent *metadata access*, in contrast to *data access* primarily targeted in prior microarchitectural attacks [2], [3], [26], [27]. We design two variants of this attack, namely *MetaLeak-T* and *MetaLeak-C*. MetaLeak-T infers program secrets (e.g., in enclave) by monitoring victim's accesses to the shared integrity tree nodes using a new exploitation technique–`mEvict+mReload`. Since security metadata cannot be directly accessed from software, `mEvict+mReload` generates carefully curated data accesses to exercise the desired integrity verification path indirectly. Differently, MetaLeak-C manifests as a write-observing leakage channel that identifies the timing differences in counter metadata maintenance through counter preset and overflow (i.e., `mPreset+mOverflow`). We first demonstrate covert channels capable of communicating across cores and sockets using both attack variants, which achieve above 99.3% bit accuracy. We then conduct side channel case studies in both academic designs [12], [14] and the commercially-available SGX processors, targeting real-world applications. Notably, under the state-of-the-art secure processor design [12], [14], MetaLeak manages to recover input images from a `libjpeg`-based image-processing program with high fidelity (up to 97% stealing accuracy). We further launch MetaLeak attacks against `libgcrypt` and `mbedTLS` based cryptographic software running in SGX enclaves. Our results show that MetaLeak can successfully exfiltrate cryptographic secrets with up to 91% accuracy. Finally, our investigation reveals that the identified vulnerabilities are uniquely tied to the underlying components in secure processors, which can render mainstream side channel mitigation ineffective (e.g., hardware obfuscation and partitioning [28], [29], [30], [31], [32], [33], [34]). Our work highlights the need to rethink secure architecture designs for microarchitecture security. In summary, the main contributions of this work are:

- We highlight the importance of understanding microarchitecture security as systems are integrated with trusted computing mechanisms for data confidentiality and integrity protection.
- We investigate and characterize the secure processor architecture design space, alongside on-chip side channel exploitation in state-of-the-art designs. We identify unique properties that expose new attack surfaces for information leakage in secure processors.
- We present MetaLeak, the first side channel attack framework that exfiltrates secrets by exploiting security metadata. We build two variants of attacks: MetaLeak-T that can manifest cross-core and cross-socket without the need for explicit data sharing; and MetaLeak-C, the fine-grained write-based channel through counter states. We implement two covert channels attacks, demonstrating high bit rate (MetaLeak-T) and accuracy (MetaLeak-C).
- We perform three case studies of MetaLeak side channels that can accurately infer secrets from real-world applications including image processing and cryptographic software. We demonstrate successful exploitation in both academic designs and real systems with Intel SGX.
- We discuss the unique challenges in architectural design that must be addressed to defend against MetaLeak. Our work highlights the need to rethink secure processor designs to sustain microarchitecture security.

II. Background and Related Works

A. Microarchitectural Attacks and Defenses

Microarchitectural side channel attacks are a form of information leakage attack where an adversary establishes illicit communication through modulating microarchitectural states that influence the timings of instruction executions (e.g., latency). These attacks typically consist of manipulating microarchitectural state changes dependent on secretive data, and inferring the secrets based on microarchitectural state observations. With extensive studies of microarchitectural side channels over the past decade, it is evident that hardware performance optimization techniques are one of the underlying causes of these vulnerabilities. Examples of such exploited performance-enhancing techniques include caching [2], [3], [27], [35], branch prediction [6], [7], [36], memory optimizations [37], [38], [39], [40], [41], and speculative execution [5], [36], [42]. These attacks significantly compromise on-chip data security. To mitigate microarchitectural timing channels, many defensive mechanisms have been proposed [43], [44], [45], [46], [47], [48], [49]. These works generally fall into the following categories: i) obfuscation-based approaches [28], [43], [46], which randomize the accesses to certain microarchitectural components (e.g., address-to-cache set mapping) to disrupt timing observations, and ii) isolation-based approaches [30], [31], [32], [33], [34], which employ spatial or temporal partitioning of shared hardware resources. In addition, timing channel detection mechanisms audit shared hardware resource usage for potentially malicious access patterns (e.g., periodic contention) [35], [44], [50], [51], [52]. Nonetheless, new attack variants are regularly published that can bypass current defenses [4], [46], [53], [54]. To ensure the security of future systems, it becomes crucial to understand microarchitectural security in a holistic manner by meticulously exploring the interplay among different microarchitectural mechanisms.

B. Secure Processor Architectures

Hardware-based threats on off-chip data span two categories that can severely compromise security: i) data stealing (e.g., bus snooping [55] and cold boot attacks [56]), and ii) data tampering (e.g., transient faults [57], [58] and physical memory manipulation [59]). These attacks motivate the hardware and architecture community to design secure processor architectures. In this pursuit, treating the processor as the root of trust has gained significant attention. Typically, secure processors

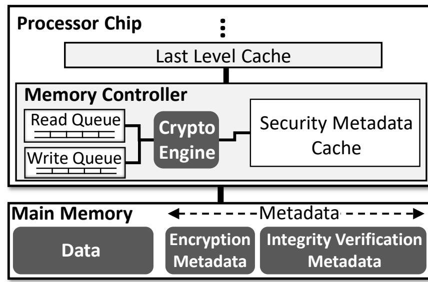


Fig. 1: Overview of secure processor architecture.

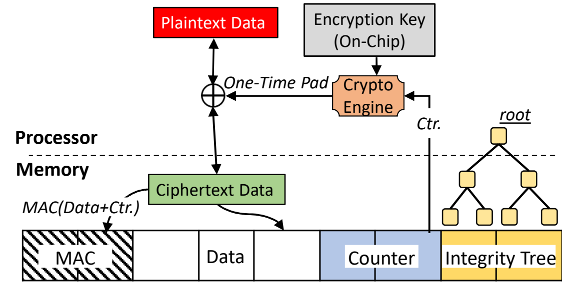


Fig. 2: Metadata structure in representative secure processors.

protect data confidentiality and integrity by incorporating a security engine on-chip [12], [13], [15], [60]. The processor encrypts plaintext data blocks and stores them in memory as ciphertext, together with encryption and integrity verification metadata. When data is read from memory, the ciphertext blocks are decrypted and checked for potential tampering. These hardware security primitives in secure architectures serve as the foundation for commercial processors featuring TEE, such as Intel SGX and AMD SEV [11], [61], [62]. Figure 1 shows an overview of state-of-the-art secure processor design. The key components are:

Data Confidentiality Protection. Data encryption is typically performed using state-of-the-art block ciphers such as AES [63]. To offer a strong confidentiality guarantee, hardware-based encryption mechanisms must ensure that the same data content written to the memory maps to different ciphertexts (e.g., to mitigate known-ciphertext attacks [64]). While there exist various encryption modes (e.g., AES electronic codebook [65]), *counter-mode* encryption is particularly appealing in secure processors. With such techniques, a seed is applied in the block cipher to generate a unique one-time pad (OTP). Data in a block is then XORed with the OTP for encryption and decryption. To ensure the temporal uniqueness of the OTP (i.e., for writes to a block over time), a counter (e.g., for each block) is used as part of the seed, which is incremented each time for a new encryption. Meanwhile, to ensure OTPs are unique among different blocks, the block address is further added as a seed component. Counter-mode encryption improves security and offers high performance by effectively hiding the cryptographic operation (OTP generation) from the critical path of memory reads. Figure 2 illustrates the counter-mode encryption in secure architectures where encryption counters are maintained in memory [12], [14], [66], [67].

Data Integrity Verification. Memory data is vulnerable to various forms of tampering, including: i) data spoofing that directly alters data, ii) data splicing where values from two memory locations are swapped, and iii) data replay in which older data of a block is used to replace its current value. Conventionally, data integrity is protected through an authentication mechanism using message authentication codes (MAC) based on keyed hash [68]. The MAC is loaded along with its data block from memory and checked with the re-computed MAC from the loaded block. A detected mismatch indicates the presence of data spoofing. This can be further augmented by adding the block address into the MAC generation to detect data splicing attacks [66]. However, MAC-based authentication cannot defend against *data replay attacks* that replace paired data and MAC captured at a previous time. To further detect data replay, the processor essentially has to keep track of an untampered digest of *the whole memory*. This is achieved by additionally maintaining an integrity tree (e.g., tree of hashes) constructed over memory data, with the root of the tree always kept on-chip [69], [70]. Using the integrity tree, any off-chip data manipulation (including data replay) will result in a mismatch eventually with the tree root. Note that secure processors can also integrate ECC [15] in memory, which specializes in error correction rather than tamper detection. Recent works show the possibility of deriving version counters on-chip and eliminating the need for tree-based metadata in domain-specific accelerators [24], [71], [72], [73]. However, these techniques are mainly designed to protect workloads that have *deterministic and uniform* memory access patterns (e.g., machine learning). The integrity tree is still the most effective way to ensure data freshness in general-purpose computing.

While secure processors provide off-chip data security, their designs have not considered on-chip security, particularly with respect to microarchitecture security. To holistically understand the overall security landscape of modern processors, it is critical to explore how secure processor architecture designs impact microarchitectural information leakage.

III. Threat Model

We assume that an adversary attempts to exfiltrate sensitive information from a victim process via microarchitectural attacks. To protect against adversaries with the capability of accessing/probing machines physically (e.g., bus snooping and cold boot attacks [55], [56]), state-of-the-art secure processors are deployed in the systems [12], [14]. We use a general threat model considering various use cases of secure processors. Specifically, conventional secure architectures assume that the operating system (OS) is trusted and the attacker only manifests as unprivileged userspace processes. Under a trusted execution environment (e.g., SGX), the OS can be maliciously controlled. In such a scenario, the attacker may have privileged control of victim's process such as fine-grained stepping [25] and execution replay [22]. We further assume read-only data sharing between the attacker and victim processes (e.g., shared

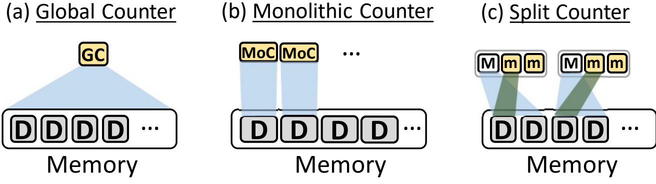


Fig. 3: Counter-sharing groups during encryption in different counter schemes. GC = *Global counter*, MoC = *Monolithic counter*, M = *Major counter* and m = *Minor counter*.

libraries) is either disabled [30], [31] or audited [52], which can thwart previous side channels exploiting shared memory (e.g., Flush+Reload [3]). Furthermore, similar to prior side channel attacks leveraging uncore hardware resources [74], [75], we assume that the victim's memory accesses of interest reach LLC/memory controller due to either system security policies (e.g., cache cleansing [6], [74], [75]) or programming models (e.g., persistent applications [76]). The adversary aims to launch the attacks cross-core or cross-socket, which is plausible as metadata are shared across all cores in the system. Finally, we exclude attacks that harness physical properties of the machines as sources of side channels, such as power consumption and electromagnetic emanations [77], [78], [79].

IV. Microarchitecture (In)security in Secure Processor Design Space

In this section, we perform a systematic investigation of the architectural design and optimization in secure processors. The key objective is to examine the fundamental design components of secure architectures and understand the potential vulnerabilities that could lead to timing-based leakage.

Definition of Core Elements. Secure processor architecture is built on three cryptographic primitives:

- Counter-mode encryption: $c = p \oplus Enc_k(s)$, where p is a plaintext data chunk within a memory block (P), c is the encrypted chunk and s is the seed. The encrypted chunks (c) in a block forms an encrypted block C.
- Data authentication: $\mathcal{M} = MAC_k(C, addr_b)$, where MAC is typically a keyed hash operation (e.g., GHASH [68]) performed over the ciphertext memory block C, and its block address $addr_b$.
- Integrity tree: $h_i^l = Hash(h_{i_1}^{l-1} \parallel h_{i_2}^{l-1} \parallel ... \parallel h_{i_k}^{l-1})$ for hash-based integrity tree or $ctr_i^l = \sum_{j=1}^{k} ctr_{i_j}^{l-1}$ for tree of counters. l is the level of a tree node, h_i^l (or ctr_i^l) is the parent hash (or counter) node at the lth level, and $h_{i_{1...k}}^{l-1}$ (or $ctr_{i_{1...k}}^{l-1}$) are the corresponding child hash (or counter) nodes. $\parallel$ is the concatenation operator. The tree is constructed recursively where each intermediate node is a hash (or counter) derived from its child nodes. The root of the tree (h^{root} or ctr^{root}) is stored on-chip.

A. Counter-mode Encryption Design

To maintain data confidentiality, secure processor architectures predominantly use AES counter-mode encryption or its variants. Figure 3 illustrates the main design for encryption counters. The key differences among these schemes are the *group of memory blocks* ($\mathcal{G}$) sharing the same counter. Specifically, Global Counter (GC) scheme uses one counter shared among all memory blocks *for encryption* [80], [81]. The snapshot value of the counter encrypting each block is stored as metadata for subsequent decryption. When the global counter overflows, the encryption key must be changed and whole-memory re-encryption is performed with the new key (i.e., $\mathcal{G}_{GC}$ = all blocks in memory). Monolithic Counter (MoC) maintains one counter per memory block for both encryption and decryption, which considerably reduces frequency of overflow [82]. Note that overflow of one counter still needs re-encryption of the entire memory (i.e., $\mathcal{G}_{MoC}$ = all blocks in memory). Finally, the Split Counter (SC) scheme organizes memory blocks into groups (typically covering a physical page each [13]), each group is assigned a large counter (i.e., major counter) and a set of small-size per-block counters (i.e., minor counter). To encrypt/decrypt each block, a *fused counter* is formed by combining the shared major counter and its own minor counter. Writes to a data block increase its minor counter first. When a minor counter overflows, the shared major counter is incremented and only the blocks sharing the major counter are re-encrypted (i.e., $\mathcal{G}_{SC}$ = blocks in a major counter-sharing group). Typical SC schemes keep a 64-bit major counter and a group of 64 7-bit minor counters, forming exactly one counter block corresponding to one data page [12], [14], [15], [83], [84]. Such design keeps counter storage small and meanwhile reduces overhead of counter overflow compared to the MoC scheme. Note that encryption is done chunk by chunk (e.g., 16B for AES-128). Consequently, seed uniqueness needs to be maintained at chunk level. To achieve this, the seed is typically generated by combining the chunk address (i.e., $addr_b$ and chunk offset) with the block's counter: $addr_{ck} \parallel ctr$ (e.g., fused counter in SC schemes).

Algorithm 1: Counter-mode Encryption Mechanism

```
  Input: P_t: current block to encrypted
1 Function Encrypt(P_t):
2   ctr_old = ctr
3   Increment (ctr)      // Increment the counter
4   if ctr_old = ctr^Max then
      // Overflow detected. Change
         encryption key (GC/MoC) or
         increment major counter (SC)
      // Re-encrypt memory blocks in G
5     Decrypt(P_i) with old counters, Encrypt(P_i) with
       new counters for P_i in {G−P_t}
6     Encrypt(P_t) using ctr
7   else
8     Encrypt(P_t) using ctr // Only encrypt P_t
```

Maintaining encryption counters can bring non-trivial performance overhead. Accordingly, several architectural optimizations explore more compact and efficient counter designs [14], [85], [86]. Note that in all encryption counter

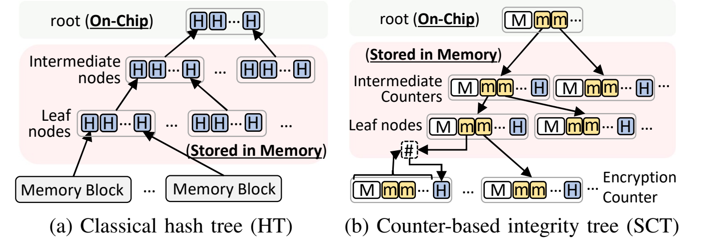
designs, counters must be updated following every write to ensure seed uniqueness. Since the counter size is finite, these counters will eventually overflow. As they are shared across multiple data blocks/pages, overflow in these requires re-encryption of the entire counter-sharing group along with the target block write. As highlighted in Algorithm 1, this creates a *metadata state dependent* execution paths: 1) the longer path in case of overflow, a set of data blocks (together with counters) are read, re-encrypted, and written back; or 2) the shorter path, only one data block is encrypted and written back (VUL-1).

B. Data Authentication

MAC is typically computed over ciphertext data and stored along with it for authentication. For secure processors which also keep encryption counters in memory, MAC could be additionally extended to counter blocks [81]. Alternatively, MAC can be maintained *over* each data block and counters by pairing the data block and its corresponding counters (i.e., $MAC_k(C, ctr, addr_b)$ [12]). Such scheme allows integrity verification with tree over encryption counters only. Classical designs typically do not cache MAC [13]. Instead MAC is read from memory for every data read and write. Alternatively, in-memory ECC bits can be repurposed to store the MAC [15], which allows fetching both data and MAC in a single memory read. During write, MAC is recalculated and stored in memory. Memory authentication using MAC does not result in variable execution latency differences (i.e., always requires fixed MAC calculation latency and one MAC read latency in classical design), hence it is agnostic of program or secure memory metadata access patterns.

C. Integrity Verification Tree Design

To detect data replay attacks, a hierarchical tree structure (i.e., integrity tree) has to be used in secure processors (See Section II-B). State-of-the-art secure processors typically adopt one of the two integrity tree designs (shown in Figure 4): i) *counter tree* (CT) where the tree is built primarily with a combination of counters [14], [15], [16]; ii) *hash tree* (HT) where tree node contains hash values [12], [13]. Multiple logical tree nodes form a tree node block. Each tree leaf node has a memory block *attached* to it (i.e., covered by the tree). A standard integrity tree has the leaf nodes cover all non-integrity tree data [13], [81]. However, when the MAC is computed over coupled data and encryption counters (Section IV-B), the tree can be built to cover encryption counter blocks only, resulting in a shorter and more efficient Bonsai Merkle Tree [12]. *Reads of one tree-attached block* from the main memory trigger integrity verification through a tree traversal. This involves loading the node blocks from the leaf to the root node (i.e., bottom-up), and checking the tree nodes along the path for tamper detection. *Writes to a covered memory block* first activate the same verification process as in read, followed by an update to the tree nodes along the path. Since a complete leaf to root traversal introduces considerable metadata accesses, secure processors typically cache tree nodes in the on-chip metadata cache. As the processor defines the security boundary, cached node blocks are trusted. Thus, the aforementioned tree traversal path only needs to proceed to the *first node block* cached on-chip (essentially serving as a temporary root) without compromising security [81].



(a) Classical hash tree (HT) (b) Counter-based integrity tree (SCT)

Fig. 4: Different integrity tree schemes (H = *Hash*, # = *Hash operation*). Multiple logical tree nodes form a metadata memory block (i.e., node block).

Algorithm 2: Data Integrity Verification

Input: $\mathcal{B}$ `// Memory block to verify`
1 **Leaf node:** N_A^i `// The` i^{th}`-level ancestor tree node for an attached memory block` $\mathcal{B}$
2 **Function** `Verify(`$\mathcal{B}$`)`**:**
  `// Assume Block(`N_A^L`) is cached`
3 **for** i *from* 1 *to* L **do**
4   **Load** Block(N_A^i)
5   **Verify** (Block(N_A^{i-1})) with N_A^i
6 **Verify**($\mathcal{B}$) with N_A^0

For HT, each node is a *hash* of its child nodes grouped as one node block (Figure 4a). The number of hashes in a node block defines the *tree arity*. Verification of one attached memory block involves matching its hash with its leaf node hash (e.g., h_i^0), and further verifying this tree node using the hash in its parent node (i.e., $h_i^l \stackrel{?}{=} Hash(h_{i_1}^{l-1} \| h_{i_2}^{l-1} \| ... \| h_{i_k}^{l-1})$). For CT designs, a node block contains a combination of *counters* and an embedded per-block hash. State-of-the-art academic designs [14], [15], [16] use the split-counter tree (SCT), where one node block maintains a tree major counter ($\overline{\mathbf{ctr}}$) and a set of tree minor counters (ctr), similar to the encryption counter design in SC mode (Section IV-A). Figure 4b shows the split-counter tree over encryption counters. Each tree minor counter essentially serves as the parent node for its child nodes. The tree is constructed such that the value of a parent minor counter is the sum of all its child minor counters (i.e., $ctr_i^l \leftarrow \sum_{j=1}^{k} ctr_{i_j}^{l-1}$). Additionally, the embedded per-block hash is computed over its own major/minor counters and its parent minor counter as illustrated in Figure 3, defined as $h_i^l \leftarrow Hash(ctr_i^{l+1} \| \overline{\mathbf{ctr}}^l \| ctr_{i_1}^l \| ctr_{i_2}^l \| ... \| ctr_{i_k}^l)$. In SCT, a memory block verification (e.g., encryption counter block) involves comparing the parent minor counter with the sum of the current node block's minor counters (together with verifying the embedded hash if necessary) all the way to a cached node. Note that SCT counters are also subject to overflow. When a tree minor counter overflows, the current

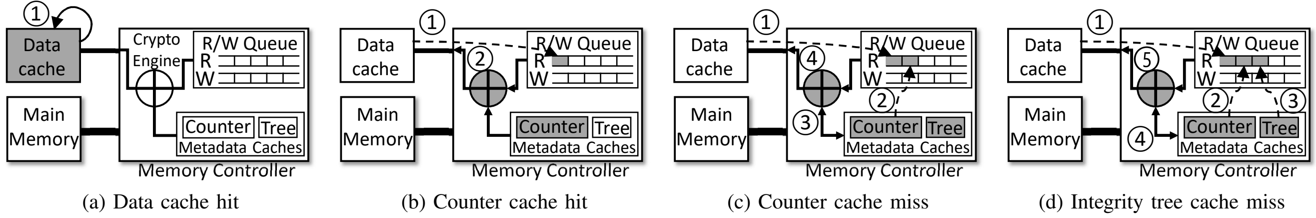


Fig. 5: Data access paths of memory reads associated with metadata accesses: a) data cache hit; b) data cache miss and counter cache hit; c) data/counter cache misses and integrity tree cache hit; and d) data/counter/integrity tree cache misses.

node and all its descendant node blocks' minor counters and major counters are reset and incremented, respectively. Accordingly, a *re-hashing* operation is performed to update the embedded hash in all the node blocks of this subtree. Intel SGX uses a variant of CT design, referred to as the SGX Integrity Tree (SIT) [11], [67], [87]. Different from SCT, SIT leverages only monolithic counters in tree node blocks [67].

Algorithm 2 illustrates the abstracted mechanism for the tree-based verification in secure processors. To verify the integrity of an attached block, all schemes require the procedure of *loading node blocks* to the metadata cache bottom up, which concludes at the first ancestor node on-chip. This exposes a new timing leakage scenario: the latency of the integrity verification path varies according to *tree node caching state*. More importantly, as the integrity tree is maintained globally, each tree node is shared across a group of attached blocks. For instance, in an 8-ary tree, a node block in the first and second levels is shared by 8 and 64 attached memory blocks, respectively. Ultimately, an arbitrary memory block $\mathcal{B}$ shares *at least one node* with other attached blocks in secure memory. In other words, tree node sharing is universal in secure processors and is independent of regular data sharing, which the TEE runtime could regulate. The sharing nature of the tree node, combined with its caching-dependent verification, creates a new side channel attack surface. Essentially, an adversary can exploit the timing of secret-dependent *integrity metadata access* (VUL-2), in contrast to the widely-understood secret-dependent *data access* that enables numerous microarchitectural attacks [2], [3], [4], [36]. Moreover, as discussed above, node updates in counter trees (e.g., SCT) can induce the handling of overflow, which introduces much higher latency. Hence, the non-constant write timing (as in encryption counters VUL-1) also exists for *tree counters*, which can vary at an even greater scale with the potentially larger block coverage.

V. Timing Characterization in Secure Processor Architectures

This section investigates latency variations in data read and write paths due to metadata management in secure processor designs. We primarily focus on the timing of software-hardware interactions that exercise the metadata management mechanism to assess their potential for information leakage. Figure 5 illustrates various access paths corresponding to the processor data read operation under different metadata

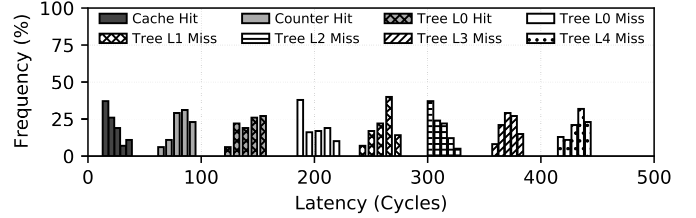


Fig. 6: Latency distribution across access paths (Simulation).

caching states. **Path-1**: the data read request results in cache hits (①), no action is triggered in any security component (Figure 5a). When a last-level cache miss occurs, the data block must be loaded from main memory, decrypted, and integrity verified. Therefore, **Path-2** represents the scenario when the data block's counter is cached on-chip (Figure 5b). In this case, the memory controller (MC) services the read request from its read queue (①) and issues it to main memory. The returned data block is authenticated using MAC and decrypted with the counter (②). **Path-3**: When the counter misses in the metadata cache, it needs to be read from memory first (②) and then integrity verified using the integrity tree (③) before decrypting the data (④). Figure 5d illustrates the execution path when the *tree leaf node* is cached on-chip, with which counter block integrity is verified. Lastly, **Path-4** is similar to Path-3, with the difference that one or multiple nodes leading to the tree root are not present on-chip (Figure 5d). As such, the integrity tree nodes themselves need to be read from memory (③) and verified from bottom up before verifying the integrity of the counter (④), which eventually leads to data decryption (⑤). This process introduces multiple memory accesses and hashing operations, resulting in considerable long and variant latency for the data accesses.

For data write operations, the counter must be present to encrypt the outgoing data, resulting in similar behavior as read operations. In addition to that, data write will also update the counters, subsequently requiring integrity tree node updates to keep the metadata up-to-date. Note that encryption counters are typically updated when data write is serviced by the MC. Differently, integrity tree nodes are only updated when the corresponding encryption counters are evicted from the cache. Typically, a *lazy update* scheme is employed where only the immediate tree node (e.g., leaf) is updated upon encryption counter writeback, and higher-level nodes are updated when

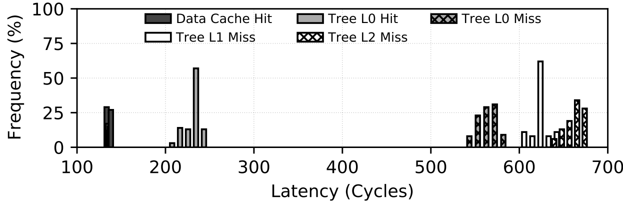


Fig. 7: Latency distributions across access paths (Intel SGX).

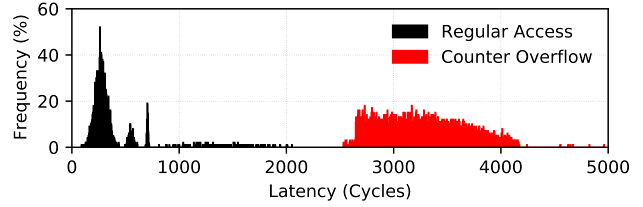


Fig. 8: Observable memory latency distributions impacted by counter overflow (Simulation).

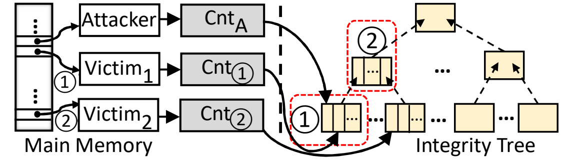


Fig. 9: Implicit sharing of integrity tree metadata.

dirty child tree nodes are evicted from the metadata cache.

To understand the impact of metadata management on data access latency, we design a microbenchmark such that the memory data access path follows one of the paths from Figure 5. We perform this study in two configurations: i) simulate academic designs with SCT [67] and HT [12] (SCT by default), and ii) Intel SGX hardware with the SIT integrity tree (See Section VII for detailed experimental setup).

Memory Read Latency. Figure 6 shows the latency distributions corresponding to various access paths with the simulated academic design. We observe that different paths exhibit highly distinguishable access latencies (between the range of 30 to 400 cycles). Additionally, the same access path (i.e., Figure 5d) can have multiple additional latency levels depending on which level of the integrity tree is in cache prior to the data block access. Notably, 450 cycles is needed in case the integrity verification is missed at all levels of the tree. Note that we observe similar latency distributions in a simulated HT-based design. In addition to the simulated configurations, we also perform latency characterization on SGX-enabled processors (See Section VIII-B for detailed method). In particular, we perform latency characterization in Intel SGX by allocating 80MB EPC data in an enclave and time the read latency to blocks accessed in a stride pattern. Figure 7 presents the latency corresponding to data access paths in SGX. Specifically, we observe access latency varying between 150 to 700 cycles. In particular, around 250 cycles is needed for data read from memory when integrity tree leaf is in cache and about 650 cycles is required when tree node blocks are missed at all levels.

Memory Write Latency. The completion of data write from MC in secure processors can exhibit the same set of variations due to caching status of security metadata. Different from reads, write operations increment the encryption counters (and tree counter in CT), which can potentially lead to counter overflow. As discussed in Section IV, counter overflow handling is an expensive operation. To quantify the latency distributions associated with tree counter overflow, we design a microbenchmark to measure the latency of a read operation immediately following overflow. The benchmark performs $2^n - 1$ writes updating a tree counter node, where n represents the size of the tree minor counter. This saturates the counter, ensuring that any subsequent write updating this will lead to overflow. Subsequently, we perform either a write updating this counter (thus triggering overflow) or a write to an entirely different location (avoiding overflow). Concurrently, in another thread, we execute a timed memory read of a memory block whose address is in the same memory bank. In SCT, we collect 10,000 read latency samples for each case. As illustrated in Figure 8, observed memory read latency resides in two distinct latency bands that differ in 2000 cycles.

In summary, secure processor designs severely exacerbates latency profiles due to the complex interactions between the MC and main memory. Such latency differences associated with metadata maintenance can be exploited to perform novel information leakage attacks. Note that while the prior work [88] proposing optimized and compact metadata management has considered potential covert channels due to the sharing of metadata cache, our work is the first to explore microarchitecture security in the *design space* of secure processor architectures with *formulation of side channels* that can manifest in real-world applications.

VI. MetaLeak - Side Channels Exploiting Security Metadata

In this section, we present MetaLeak- a side channel attack framework that leverages metadata accesses in secure processors. We design two variants of MetaLeak: a) MetaLeak-T- which exploits the shared integrity tree to infer victim memory access patterns; and b) MetaLeak-C- which harnesses shared counters to infer victim program secrets.

A. MetaLeak-T: Exploiting Integrity Tree Sharing

In all secure processor architecture designs, integrity tree is inherently *one logical structure per memory controller* and shared across all running processes. This leads to a new aspect of memory sharing among processes by design. Figure 9 illustrates how different levels of integrity tree node lead to sharing across different regions of data. While physically contiguous memory pages may share tree leaf nodes (①), metadata sharing can be achieved among pages separated by large data regions through intermediate tree nodes (②). Since these different pages access the shared tree block for integrity

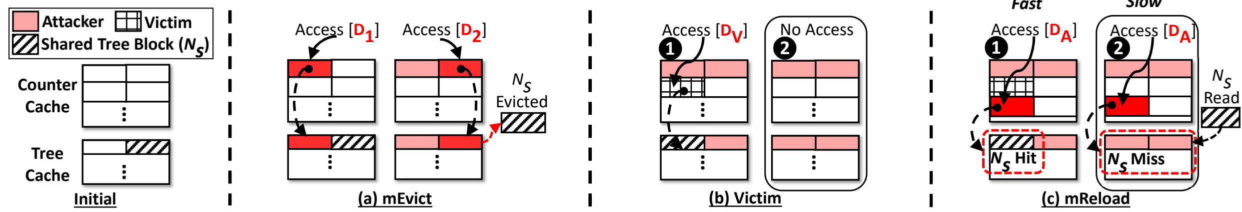


Fig. 10: MetaLeak-T attack steps: a) N_s eviction through indirect access of non-shared attacker page (D_1 and D_2); b) Victim's secret-dependent execution; c) Inferring victim secret by reloading N_s using non-shared D_A.

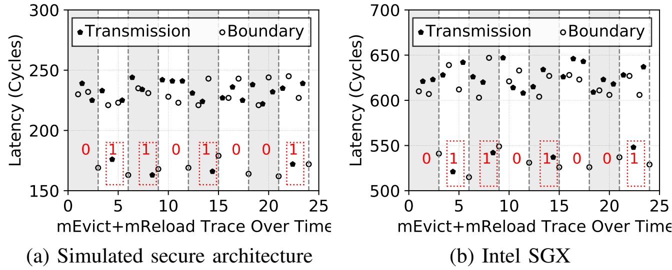


Fig. 11: Latency trace for transmitting `'01101001'` in the MetaLeak-T covert channel. Each band denotes one-bit transmission window (separated by a hit in the *boundary* set). A spy's *reload* hit of the shared metadata block in the *transmission* set is decoded as bit `'1'`, otherwise bit `'0'`.

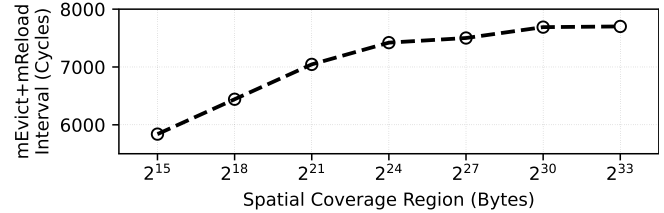


Fig. 12: mEvict+mReload operation and corresponding spatial coverage regions as exploited tree node level changes from leaf to top in the SCT tree.

verification, it is possible to exploit a timing attack in a shared memory fashion without actually sharing program data.

We formulate MetaLeak-T that exploits the aforementioned implicit memory sharing through security metadata. Specifically, MetaLeak-T observes victim page access activity through monitoring accesses to shared tree nodes. Compared to data cache attacks, implementing integrity tree cache attacks presents two major challenges: 1) Program execution can only directly initiate data access, not metadata access; 2) To observe tree cache activities, the attacker's process needs to enforce longer access paths (as shown in Figure 5d), which requires additional indirection through counter metadata. To tackle these challenges, we design a new exploitation technique–`mEvict+mReload`, with three main steps (Figure 10):

Step 1: mEvict—Evict Shared Integrity Tree Blocks. In this step, the attacker ensures that the shared integrity tree block for observing the victim's activity (i.e., N_s) is evicted from the metadata cache. Since the attacker cannot directly access integrity tree blocks, it instead creates a set of data blocks (i.e., D_1 and D_2 in Figure 10a) whose encryption counter blocks map to tree nodes corresponding to the same set as N_s. Note that the data block accesses must incur miss in data cache (and their corresponding encryption counters must also miss in the counter cache). As such, in order to verify the counter integrity, the corresponding tree nodes will be accessed, filling the metadata cache set containing N_s. This ensures that the N_s is evicted from the cache after the `mEvict` step.

Step 2: Idle. In this step, the attacker triggers victim execution and waits for her secret-dependent access to complete. As illustrated in Figure 10b, the victim's data access leads to either N_s being brought into the metadata cache (if D_V is accessed) ❶ or N_s not cached (if D_V is not accessed) ❷.

Step 3: mReload—Infer Victim Secret through Shared Tree Node Caching State. Finally, the attacker measures the access latency of a data block (D_A) whose counter block shares the same tree block as the victim's D_V (i.e., N_s). As demonstrated in Section V, the access latency of D_A can be used to infer the corresponding security metadata caching state. Figure 10c illustrates that ❶ fast access to D_A means victim access to D_V in Step 2, whereas ❷ slow access to D_A indicates no victim access to D_V. Based on the data access latency correlated with integrity metadata accesses, the victim secret can be inferred.

MetaLeak-T Covert Channel. We first demonstrate MetaLeak-T in a covert channel. In this attack, a trojan uses two tree nodes (i.e., two metadata blocks) mapped to two different cache sets for covert communication: one set to transmit bit information (i.e., access for bit '1', no access for bit '0'), and the other set to define a bit boundary. The spy monitors activity in these two sets using mEvict+mReload. Between each access in the *boundary* set by trojan, if trojan's access is detected in the *transmission* set (i.e., shorter latency in mReload), bit '1' is decoded ('0' otherwise). The covert channel transmits 1000 bits using this communication protocol. Figure 11 shows a snippet of the latency trace observed by the trojan. Overall, we observe 99.3% bit accuracy in SCT (Figure 11a) and 94.3% bit accuracy in SGX's SIT configurations (Figure 11b). Figure 12 shows the average mEvict+mReload intervals when using different levels of tree nodes as shared memory. Note that the temporal resolution of MetaLeak-T decreases with the increase in tree node level,

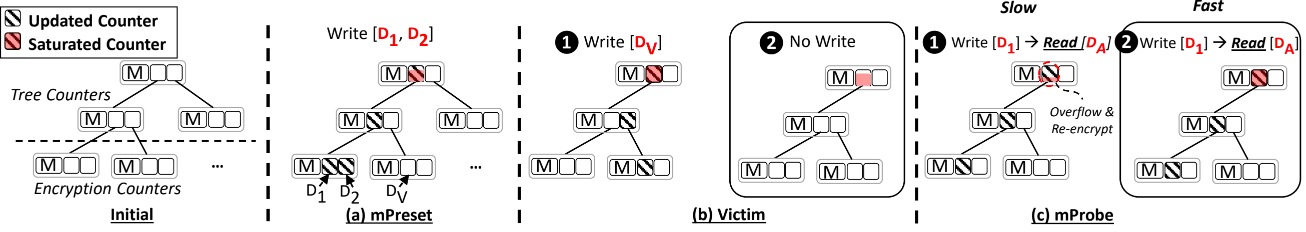


Fig. 13: MetaLeak-C attack steps: a) Preset the shared integrity tree counter by writing to attacker-controlled memory (D_1 and D_2); b) Victim's secret-dependent write; c) Inferring victim secret by performing *one* write to attacker-controlled block (D_1) and subsequently timing read latency of a block (D_A) not belonging to this sub-tree to detect counter overflow. Pattern-filled counters indicate the counters updated during each step. The color-filling in blocks denotes the counter saturating status.

but with a higher possibility of sharing due to the larger data coverage for a tree node (i.e., 32KB for leaf nodes), which increases exponentially as tree level increases.

B. MetaLeak-C: Exploiting Counters for Write Monitoring

As discussed in Section IV, when a minor counter overflows, all memory blocks corresponding to that counter sharing group (or sub-tree of the integrity tree) require re-encryption (or re-hashing in SCT), which results in highly observable latency variations in subsequent memory accesses. This observation can be maliciously utilized to observe victim write activity by an attacker. If the attacker manages to share a counter the victim (e.g., tree counters in SCT), then the counter can be preset such that one victim write will saturate the counter, allowing the attacker to detect victim write at a later stage by simply performing one addition write to trigger counter overflow. However, effectively observing victim writes through counter overflow has several challenges: i) To ensure that victim write saturates the target counter, the attacker needs to strategically set the state of the counter prior to victim execution, which is difficult since the attacker does not know the runtime state of the counter; ii) Ensuring the attacker's timed read gets delayed by the ongoing re-encryption process so that it can be observed. To tackle these challenges, we present MetaLeak-C- that exploits shared counters with the following three steps (illustrated in Figure 13):

Step 1: mPreset—Preset the State of Shared Counter. The key target of this step is to ensure that the counter is set to an attacker known state so that it can be probed later to detect victim's write activity. While the preset state can be any predetermined value, for simplicity, we assume that the attacker wants to preset the counter so that one victim write saturates the counter. To do that, the attacker first determines the counter state by resetting it. This can be done by the attacker continuously writing to many several data blocks that map to this counter (i.e., D_1, D_2, ...) and after each write, timing the memory read latency by loading a separate memory block (i.e., D_A). Note that D_A does not need to belong in the same sub-tree under the counter targeted for overflow. However, there are two issues that need to be addressed: i) writes are typically not performed immediately, rather, they are buffered at the memory controller's *write queue* and later serviced under the arbitration of the scheduling policy; ii) while writes are pending in the write queue, subsequent writes to memory blocks present in the queue can get merged together. These issues can impact both the perceived counter state during preset as well as hinder the attacker's visibility on counter overflow by reordering the timed read. To address these issues, we flush the write queue by performing redundant writes to blocks outside of the sub-tree coverage (similar to D_A). Once the attacker observes an overflow in the target counter, its state is now known and can be preset to any desired state. The attacker then performs $2^n - 2$ total writes (for an n-bit counter) to preset the counter so that it is one write short of saturation (Figure 13a).

Step 2: Idle. The attacker then waits for victim's execution. As illustrated in Figure 13b, if victim performs one write to D_V (that shares the same integrity tree counter) ❶, it will saturate the counter. Any further update to this counter will trigger an overflow. In contrast, if the victim does not perform any write utilizing this counter, it remains unchanged ❷.

Step 3: mOverflow—Infer Victim Secret through Counter Overflow. Once victim execution is complete, the attacker probes the state of the counter to infer if the victim has indeed performed any write. As we set the counter to $2^n - 2$ in `mPreset`, if the victim performs one write operation, then one attacker write (i.e., to D_1) will overflow the counter (Figure 13c). Conversely, if overflow is not observed, the victim did not perform any write. We use a timed read latency to D_A to determine if the counter has overflown after performing one write to D_1.

While we monitor for exactly one victim write here, these steps can be generalized to infer upto x victim writes by presetting the counter to $2^n - x + 1$ and observing the number of writes needed to induce an overflow in the `mOverflow` step. In addition, compared to MetaLeak-T, MetaLeak-C does not require any metadata eviction and can naturally incur without requiring data block eviction (such as in persistent memory applications where critical sections are written back to memory immediately). Note that the temporal resolution of MetaLeak-C can be limited due to the set of writes needed to overflow a counter. However, under TEE, the attacker can perform fine-grained stepping of victim execution [18], [20], [25], achieving attack synchronization even with long attack steps.

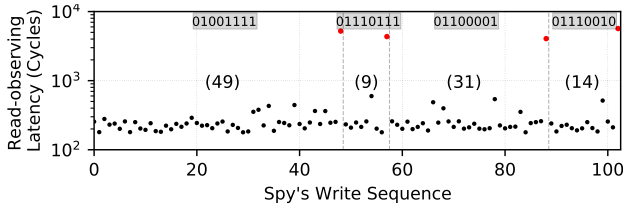


Fig. 14: Spy observed write latency traces from SCT for 4 transmission window (inferred transmission data is in inset).

MetaLeak-C Covert Channel. To demonstrate the practicality of MetaLeak-C, we design a covert channel where a trojan process transmits secret symbols by encoding values through the number of writes to a selected memory block (i.e., modulating *minor counter* values). A separate spy process infers the trojan-encoded symbols by observing additional writes required before the targeted counter overflows. Since the processor contains 7-bit minor counters in the integrity tree (detailed in Section VII), MetaLeak-C can transmit a 7-bit symbol at a time for each counter modulation. Specifically, in the `mPreset` step, the spy presets the minor counter. Subsequently, the trojan issues s writes, which corresponds to the 7-bit symbol to transmit. Finally, the spy decodes the symbol by checking the number of additional writes for the overflow. If the spy needs m additional writes to trigger the overflow, then a secret symbol with value $2^7 - m$ is decoded. Notably, `mOverflow` also *resets* the counter to 0. As a result, the `mPreset` step is essentially not needed after initial setup. We perform experiments with multiple runs of 1000-symbol transmissions. Figure 14 shows a snippet of the trace for 4 consecutive transmissions. Our results show that MetaLeak-C can manifest as a highly accurate covert channel with an average 99.7% transmission accuracy.

VII. Experimental Setup

To systematically evaluate microarchitecture security in secure architectures, we investigate representative designs in both academic works as well as real-world hardware with SGX. In particular, for SGX, we setup the system on an Intel Core i7-9700K processor and run the attacker and victim processes inside enclaves. Due to the lack of commercial prototypes for academic proposals, we extensively model state-of-the-art secure architectures [12], [14] using the cycle-level gem5 simulator [89] under full system simulation. We model a combination of secure architecture configurations considering mainstream encryption and integrity checking schemes (with SCT [14] and HT [12]). The key architecture parameters in the simulator and Intel SGX are listed in Table I.

VIII. Case studies of MetaLeak Attacks

In this section, we perform case studies on real-world applications to evaluate the proposed MetaLeak attacks in secure processor architecture designs.

Hardware	Configurations
Simulated architecture configuration	
Processor	Quad-core OoO x86 CPU
L1 I/D-Cache	Private, 32KB, 8-way, 1-cycle hit
L2 Cache	Private, 1MB, 4-way, 10-cycle hit
L3 Cache	Shared, 8MB, 16-way, 40-cycle hit
Mem. Ctrl.	64 RD & WT queue, FR-FCFS, open-row 8-way 256KB counter & Tree cache
Main Memory	64GB, dual channel, 2 ranks/channel
Crypto engine	20-cycle AES latency
Encryption	Counter-mode encryption, SC (64-bit major, 7-bit minor counters)
Integrity tree	**HT:** Hash tree, 8-ary BMT, 6-level tree [12] **SCT:** Split-counter tree, 32-ary L0, 16-ary L1-L5 [14], [15] Leaf level counters: 56-bit major, 7-bit minor
SGX hardware configuration	
Processor	Intel Core i7-9700K, 93.5MB EPC
Encryption	Counter-mode encryption, 56-bit monolithic counters
Integrity tree	**SIT:** 56-bit monolithic counters, 8-ary 4-level tree, leaf tree node (L0) covers one data page [67], [87]

TABLE I: Simulated secure processors (Top) and the SGX hardware configurations (Bottom).

```
1  encode_one_block (...) {
2   ...
3   /* Encode the coefficients */
4   for (k = 1; k < DCTSIZE2; k++) {
5    if (block[jpeg_natural_order[k]] == 0) {
6     r++;
7    } else {
8    ...
9     /* Check for out-of-range coefficient */
10   if (nbits > MAX_COEF_BITS) { ... }
11  }
12 }
```

Listing 1: Exploited gadget in `libjpeg`.

A. Attacks on Simulated Secure Processor Designs

We demonstrate MetaLeak side channels against an image processing application in the secure processor architecture with a split-counter tree design (SCT) over encryption counters. The application uses the open-source `libjpeg` library, to convert input images to compressed JPEG format. The algorithm first generates blocks of AC coefficients for input image blocks. A per-block entropy is computed by applying the Huffman coding over the sets of AC coefficients, which represents *changes in image patterns across blocks*. The entropy blocks are leveraged to derive new blocks of compressed coefficients, which are utilized to generate the output JPEG image. Listing 1 shows the code snippet of the `encode_one_block()` function from `libjpeg` that updates the compressed coefficient blocks by scanning the entropy array (i.e., `block[][]`). An attacker observing victim accesses to lines 6 and 10 in Listing 1 can obtain the execution trace and learn about the status of the entropy (i.e., coefficient change), which contains information about discernible features in the original image (i.e., sharp color gradient changes). Afterwards, the attacker can launch a local image conversation pipeline with `libjpeg` starting from a blank image. During this process, the inferred entropy information is used to guide the generation of the compressed coefficients, which can potentially yield a local image similar to the original one.

1) Exploitation using MetaLeak-T: We observe that when `encode_one_block()` function executes lines 6 or 10,

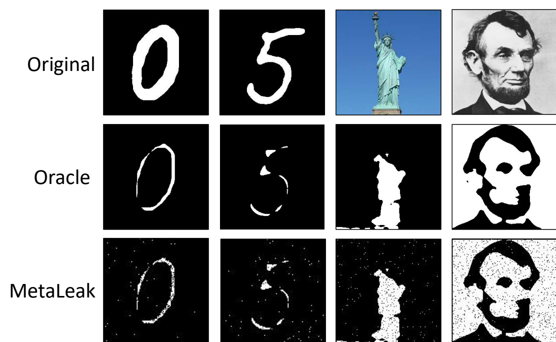


Fig. 15: Image reconstruction from the libjpeg program.

either variable `r` and `nbits` is loaded, which corresponds to two different pages by default. Access to `nbits` indicates a non-zero change of coefficient and access to `r` is used to delineate each loop iteration. To obtain tree node sharing with `r` and `nbits`, we exploit the per-core free page management mechanism in the Linux operating system to precisely map victim pages to the attacker-controller page frames [58], [90]. The two data pages are positioned with their corresponding encryption counter blocks attached sufficiently apart in the SCT. The attacker then carefully allocates two data blocks D_{A1} and D_{A2} such that their encryption counter blocks share the *leaf-level* tree node block with those of variable `r` and `nbits`, respectively. Once the desired sharing is achieved, the attacker launches mEvict+mReload to observe the victim's access. Figure 15 illustrates the original input images and the reconstructed images based on the observed access patterns. To highlight the maximum possible leakage using this function, we include the *Oracle* case, which reconstructs the image based on page accesses detected through code instrumentation. With an overall accuracy of **94.3%**, we observe that the reconstructed images are close to the oracle cases, and retain considerable details of the original images.

2) Exploitation using MetaLeak-C: We further mount MetaLeak-C against the vulnerable code in `libjpeg` (Listing 1). Specifically, as shown in line 6, the variable `r` is also updated if the corresponding coefficient change of the block in the image is zero. Such write activity can be potentially observed using the MetaLeak-C that manipulates counter overflow to monitor memory writes. With MetaLeak-C, the attacker achieves a minor counter sharing at the 2nd level of the tree in the verification path of `r`, using the same technique in Section VIII-A1. Following the attack formulation in Section VI-B, the attacker performs `mPreset` on the shared tree counter. After victim execution, the attacker performs `mOverflow` to determine the number of additional writes required for overflow, thereby detecting if the victim performed any write to `r`. If `mOverflow` needs only one attacker write, this means the victim performed a write to `r`. Note that attacker writes to update tree counter (during both `mPreset` and `mOverflow`) are distributed across different data blocks to avoid overflowing counters below the target level (i.e., encryption counter). Overall, MetaLeak-C can successfully recover zero-elements in the entropy blocks with **97.2%** accuracy.

B. Attacks on Systems with the SGX Processor

Intel's TEE solution (SGX) incorporates one of the most comprehensive implementation of secure processor architectures in the industry. Particularly, SGX (i.e., SGX client) integrates counter-mode encryption and tree-based integrity verification for data stored in a protected memory region, called Enclave Page Cache (EPC). Based on prior studies [67], [87], Intel SGX managed by the Memory Encryption Engine (MEE) maintains an 8-ary 4-level counter tree. Each level denoted as $L0$, $L1$, $L2$, and $L3$. $L3$ is the root level secured on-chip. Each 64B tree node block consists of eight 56-bit monolithic counters and a 64-bit hash. At the leaf level ($L0$), each tree counter covers 8 encryption counter blocks, collectively mapping to one page of EPC data blocks. As a result, for an EPC page (with index p) and a specific integrity tree level (l), the set of EPC pages sharing a tree block with page p at the lth level can be derived as: $\{\lceil\frac{p-1}{A^l}\rceil \cdot A^l + x | x \in \{1, 2, ..., A^l\}\}$. Given a victim's physical block or page to monitor, an attacker can determine the EPC pages sharing the same integrity tree block with the target at the desired level. Specifically, a group of 1, 8, and 64 consecutive EPC pages share the same tree block in $L0$, $L1$, and $L2$, respectively. Note that unlike the MetaLeak-T attack demonstrated in SCT (Section VIII-A1), the attacker cannot exploit the leaf level (i.e., $L0$) in SGX since one $L0$ SGX tree block maps to only one physical page, which would not be shared between two domains (no explicit data sharing). As a result, we target the shared tree block in $L1$ by allocating attacker's physical page such that it is within the same 8-page group as the victim's target page. Finally, given the large size of counters used in SGX's integrity tree (56-bit monolithic), triggering counter overflow is impractical under MetaLeak-C. The related SGX parameters are shown in Table I. In this section, we demonstrate cryptographic secret leakage from SGX using MetaLeak-T.

Attack Setup. Before mounting MetaLeak, the attacker must be able to fulfill two prerequisites: i) obtain latency profile corresponding to different levels of metadata availability on-chip (Figure 7), and ii) share specific integrity tree block with the victim. As a privileged attacker with control over OS, attacker can directly observe and control which EPC frame is provided for victim enclave page allocation through OS intervention, allowing attacker to create integrity tree co-location at preferred level. Additionally, attacker can use the SGX-step framework [25] to monitor and frequently interrupt victim enclave execution to perform `mEvict` operation. In particular, we interrupt enclave execution every 500 cycles (by setting the APIC timer interrupt frequency) to ensure mEvict+mReload is performed at each required victim iteration.

1) Attacking Modular Exponentiation in `libgcrypt`: We now demonstrate MetaLeak-T on vulnerable cryptographic algorithm implementations. The RSA algorithm in `libgcrypt` 1.5.2 uses the square-and-multiply arithmetic (code snippet shown in Listing 2). In this algorithm, either the

```
1   gcry_mpi_powm (...) {
2    ...
3    /* Main loop */
4    for (;;) {
5     /* Square operation */
6     _gcry_mpih_sqr_n_basecase(...);
7     /* Check if exponent bit is 1 */
8     if ( (mpi_limb_signed_t)e < 0 ) {
9      /* Multiplication operation */
10     _gcry_mpih_mul_karatsuba_case(...);
11   }
12  }
```

Listing 2: Exploited gadget in `libgcrypt`.

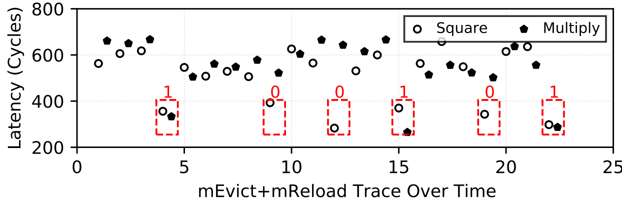


Fig. 16: Latency observation traces with mEvict+mReload for secret exponent bit sequence `'100101'`.

square (line 6) or both square and multiplication operations (lines 6 and 10) are performed based on the secret exponent bit value (i.e., '0' or '1' correspondingly). We find that the `_gcry_mpih_sqr_n_basecase()` (i.e., square) and `_gcry_mpih_mul_karatsuba_case()` (i.e., multiply) functions reside in separate pages (i.e., when compiled using `./configure -disable-asm`). We set up an enclave running encryption with this `libgcrypt` library. The attacker first achieves integrity tree sharing with victim EPC pages containing the square and multiplication functions (as discussed in attack setup). Once integrity tree node sharing is established, the attacker launches mEvict+mReload to monitor the aforementioned two functions and utilizes the access pattern to deduce the secret exponent. Figure 16 shows attacker traces observing data read latency of blocks sharing the same tree block as either the square or multiply function. From our evaluation, we observe an overall accuracy of 91.2% in recovering the secret exponent from an SGX enclave with MetaLeak-T. We additionally run this experiment in the simulated secure architecture with SCT (Section VII), which achieves 95.1% accuracy.

2) Attacking Private Key Loading in `mbedTLS`*:* Now we discuss how an attacker can exfiltrate private exponents of RSA from mbedTLS v3.4.0. Security of RSA key depends on the secrecy of its private components: two large prime numbers (p and q) and a private exponent (d), which is computed from p and q (i.e., $d = e^{-1} mod((p-1)(q-1))$). e is a *public* exponent. The private key loading implementation in mbedTLS library [91] uses modular inversion to compute d. Specifically, $(p-1)(q-1)$ is computed using two basic arithmetic operations [92], *right shifts* and *subtractions* (through `mbedtls_mpi_sub_mpi()` and `mbedtls_mpi_shift_r()` functions, respectively). The secret exponent d can be computationally recovered given that access pattern traces for both of these functions are available for each iteration [91], [93],

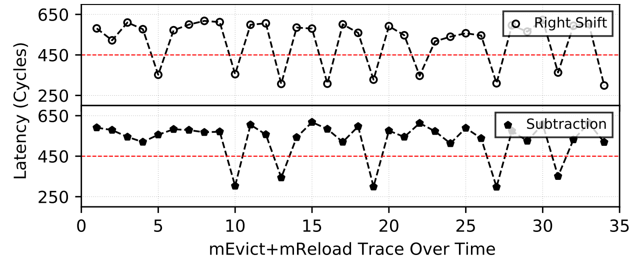


Fig. 17: mEvict+mReload traces to detect access to `mbedtls_mpi_shift_r()` (top) and `mbedtls_mpi_sub_mpi()` (bottom).

[94]. These two functions (corresponding to two different EPC pages) are placed under two different sub-trees of the integrity tree. We use mEvict+mReload to recover access patterns to each of these functions by exploiting tree sharing in $L1$. Based on the observation from Figure 7, we set 600 cycle latency as the timing threshold for integrity tree leaf hit (and miss). Figure 17 shows mEvict+mReload traces for both *Shift* and *Sub* operations. Overall, we observe 90.7% accuracy in detecting *Shift* and *Sub* accesses.

IX. Discussions

A. Implication of MetaLeak on Microarchitecture Security

Firstly, MetaLeak unveils a *new source of leakage vulnerability*. Since TEE (e.g., SGX) excludes side channels from the threat model, there have been many works demonstrating side channels inside TEE [18], [20], [95], [96]. However, most of such prior studies utilize *known* microarchitectural vulnerabilities already exploited in non-privileged settings (e.g., conflicts on caches [2]), with the objective of harnessing adversarial privileged access to achieve fine-grained victim execution control (e.g., zero stepping [25]). In contrast, MetaLeak exploits metadata and the highly distinguishable memory access timing due to metadata management to mount side channels, which exposes a new vulnerability *uniquely tied* to the design of modern secure processor architectures.

Secondly, while prior defenses including obfuscation and partitioning [28], [30], [31], [43], [97], [98] can successfully mitigate conventional cache attacks (e.g., Prime+Probe [2]), they are ineffective against MetaLeak. The key reason is that *mainstream protection schemes assume there is no sharing of data*, both readable and writable, across untrusted domains, which is plausible in secure processors as metadata is shared and updated at runtime. Specifically, randomized cache techniques [28], [43], [44], [46], [54] perform randomization for address-to-cache mapping to disrupt the creation of eviction set. However, MetaLeak does not rely on set-based eviction for sensing conflicts (a property similar to FLUSH+RELOAD [3]). In particular, in MetaLeak-T, the attacker only monitors the *reload* latency of the shared metadata block with `mReload`. To mitigate cache side channels on shared data, secure cache proposes to duplicate shared read-only memory among untrusted domains. However, duplication

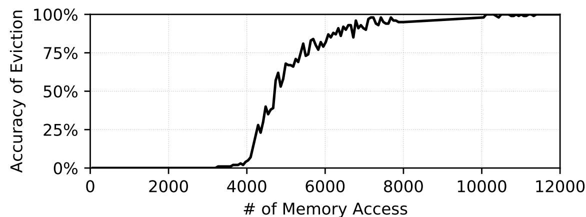


Fig. 18: Accuracy of eviction with cache randomization.

is problematic over writable data due to cache coherence issues [28]. For the same reason, although existing cache partitioning solutions [30], [31] can defeat non-data sharing side channels, they do not defend against MetaLeak. Notably, the root cause of MetaLeak is that the metadata mechanism is designed without consideration of side channels, indicating the need to rethink secure architecture design.

Scope of MetaLeak Attacks. MetaLeak mainly manifests in secure architectures employing counter-mode encryption and integrity tree-based verification, which offers strong data protection. Alternative to counter-mode encryption, several TEE implementations (e.g., AMD SEV and SGX-server [99], [100]) utilize other encryption modes such as AES-XEX and AES-XTS [101], [102]. These adopt an address-dependent nonce/tweak derived from the block address, which eliminates the use of counters and integrity trees with the sacrifice of security guarantees to memory systems. MetaLeak does not apply to those secure mechanisms. Note that recent studies have shown compromises of those commercial solutions (e.g., recent AMD SEV-SNP), including exfiltration of register values [101], complete plaintext recovery from TEE [103], and arbitrary code execution [101]. We believe the study of MetaLeak could help guide future designs of secure processors for microarchitecture security.

B. Effectiveness of Microarchitectural Attack Mitigation

Since MetaLeak-C exploits counter overflow handling timing, not cache timing, it is not impacted by cache related defensive schemes. To validate the applicability of MetaLeak-T when state-of-the-art cache defense [28] deployed, we quantify the metadata eviction accuracy corresponding to different numbers of additional cache block accesses. Using the open-source repository of MIRAGE [104], we first access a target memory block to bring it into cache, followed by accessing many random memory blocks, and subsequently *reload* the target memory block to determine its eviction accuracy (i.e., for the MetaLeak approach). In our experiments, we adopt the default configuration of MIRAGE shown to be secure against *conflict-based cache attacks* (i.e., Prime+Probe) by the authors [28]. We use a *two-skew* cache with *six additional ways* per skew (8 + 6 ways per skew). As demonstrated in Figure 18, around 7000 random block accesses are sufficient to evict the target block with more than 90% accuracy (assuming a 16-way 256KB metadata cache). This indicates that existing defenses, in particular cache randomization schemes, could not effectively stop MetaLeak attacks.

C. Future Secure Architecture Designs

The underlying vulnerability of MetaLeak attack is the use of a logically global integrity tree spanning the entire memory. Intuitively, isolation of the tree can be done such that *mutually distrusted domains do not share non-root tree nodes at any level.* While resource isolation is a promising mechanism to ensure security against side channels, partitioning the integrity tree can be particularly challenging [88]. Isolation techniques that support only a limited number of security domains with fixed tree sizes are subject to low efficiency and limited scalability. To mitigate MetaLeak, future secure architecture can leverage *isolated and dynamic* integrity trees. Specifically, a *per-domain* dynamic integrity tree can be allocated, and the coverage of the tree in each domain can grow *on-demand*. However, unlike flat resources such as cache, partitioning integrity trees and dynamically managing them at runtime can involve non-trivial overheads (e.g., due to chained rehashing and node re-positioning [105]), which is on the critical path of program execution. Overall, a practical mitigation should provide complete isolation among isolated trees, flexible partitioning to prevent memory stranding and low additional overhead for runtime metadata management.

To thwart encryption counter overflow-based attacks, it is possible to either: i) ensure previous counter states are cleared when counters are reassigned to different security domains; or ii) associate counters with virtual address space, thereby making temporal sharing across security domains impossible. Note that these mitigations are exclusive to encryption counters. Security issues concerning tree counter overflow in CT cannot be resolved using the aforementioned approaches. Instead, tree counter nodes must also be protected similar to the classical integrity tree structure.

X. Conclusion

In this paper, we perform an extensive investigation of microarchitecture security in the design space of secure processor architectures. Our work identifies the unique properties in the metadata management mechanisms in secure processors, which creates new attack vectors for information leakage. We present MetaLeak, an end-to-end side channel framework that exploits security metadata to exfiltrate program secrets in secure processors. In particular, MetaLeak-T manipulates the shared integrity tree blocks using `mEvict+mReload`, and MetaLeak-C observes secretive write activity by modulating counter states with `mPreset+mOverflow`. Evaluation on real-world victim programs shows that the attack is highly successful in both state-of-the-art academic design and the SGX processors. Our study further indicates that the identified vulnerabilities are uniquely tied with the design of secure processors, and highlights the need to rethink secure architectures for microarchitecture security.

XI. Acknowledgements

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

REFERENCES

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

[2] 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.

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

[4] 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.

[5] 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.

[6] D. Evtyushkin, R. Riley, N. C. Abu-Ghazaleh, ECE, and D. Ponomarev, "Branchscope: A new side-channel attack on directional branch predictor," in *ACM ASPLOS*, 2018.

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

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

[9] J. Ravichandran, W. T. Na, J. Lang, and M. Yan, "Pacman: attacking arm pointer authentication with speculative execution," in *IEEE ISCA*, 2022.

[10] Y. Tobah, A. Kwong, I. Kang, D. Genkin, and K. G. Shin, "Spechammer: Combining spectre and rowhammer for new speculative attacks," in *IEEE S&P*, 2022.

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

[12] 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.

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

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

[15] 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.

[16] 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.

[17] 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.

[18] I. Puddu, M. Schneider, M. Haller, and S. Čapkun, "Frontal attack: Leaking control-flow in sgx via the cpu frontend," in *USENIX Security*, 2021.

[19] C. Disselkoen, D. Kohlbrenner, L. Porter, and D. Tullsen, "Prime+abort: A timer-freehigh-precision l3 cache attack using intel tsx," in *USENIX Security*, 2017.

[20] J. Van Bulck, F. Piessens, and R. Strackx, "Nemesis: Studying microarchitectural timing leaks in rudimentary cpu interrupt logic," in *ACM CCS*, 2018.

[21] W. Wang, G. Chen, X. Pan, Y. Zhang, X. Wang, V. Bindschaedler, H. Tang, and C. A. Gunter, "Leaky cauldron on the dark land: Understanding memory side-channel hazards in sgx," in *ACM CCS*, 2017.

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

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

[24] W. Hua, M. Umar, Z. Zhang, and G. E. Suh, "Mgx: Near-zero overhead memory protection for data-intensive accelerators," in *IEEE ISCA*, 2022.

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

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

[27] W. Xiong and J. Szefer, "Leaking information through cache LRU states," in *IEEE HPCA*, 2020.

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

[29] Z. Lin, U. Mathur, and H. Zhou, "Scatter-and-gather revisited: High-performance side-channel-resistant AES on GPUs," in *ACM GPGPU*, 2019.

[30] 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.

[31] F. Liu, Q. Ge, Y. Yarom, F. Mckeen, C. Rozas, G. Heiser, and R. B. Lee, "Catalyst: Defeating last-level cache side channel attacks in cloud computing," in *IEEE HPCA*, 2016.

[32] 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.

[33] 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.

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

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

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

[37] P. Pessl, D. Gruss, C. Maurice, M. Schwarz, and S. Mangard, "Drama: Exploiting dram addressing for cross-cpu attacks," in *USENIX Security*, 2016.

[38] 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.

[39] V. R. Kommareddy, B. Zhang, F. Yao, R. Ewetz, and A. Awad, "Are crossbar memories secure? new security vulnerabilities in crossbar memories," in *IEEE CAL*, 2019.

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

[41] M. H. I. Chowdhuryy, M. R. H. Rashed, A. Awad, R. Ewetz, and F. Yao, "Ladder: Architecting content and location-aware writes for crossbar resistive memories," in *IEEE MICRO*, 2021.

[42] M. Schwarz, M. Schwarzl, M. Lipp, J. Masters, and D. Gruss, "Netspectre: Read arbitrary memory over network," in *Springer ESORICS*, 2019.

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

[44] C. Hunger, M. Kazdagli, A. Rawat, A. Dimakis, S. Vishwanath, and M. Tiwari, "Understanding contention-based channels and using them for defense," in *IEEE HPCA*, 2015.

[45] A. Purnal, L. Giner, D. Gruss, and I. Verbauwhede, "Systematic analysis of randomization-based protected cache architectures," in *IEEE S&P*, 2021.

[46] 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.

[47] P. W. Deutsch, W. T. Na, T. Bourgeat, J. S. Emer, and M. Yan, "Metior: A comprehensive model to evaluate obfuscating side-channel defense schemes," in *IEEE ISCA*, 2023.

[48] J. Szefer, "Survey of microarchitectural side and covert channels, attacks, and defenses," *Springer Journal of Hardware and Systems Security*, 2018.

[49] M. H. I. Chowdhuryy, Z. Zhang, and F. Yao, "Beknight: Guarding against information leakage in speculatively updated branch predictors," in *IEEE ICCAD*, 2023.

[50] M. Yan, B. Gopireddy, T. Shull, and J. Torrellas, "Secure hierarchy-aware cache replacement policy (sharp): Defending against cache-based side channel atacks," in *IEEE ISCA*, 2017.

[51] J. Chen and G. Venkataramani, "Cc-hunter: Uncovering covert timing channels on shared processor hardware," in *IEEE MICRO*, 2014.

[52] 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.

[53] B. Gras, K. Razavi, H. Bos, and C. Giuffrida, "Translation leak-aside buffer: Defeating cache side-channel protections with tlb attacks," in *USENIX Security*, 2018.

[54] 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.

[55] Y. Zhu, Y. Cheng, H. Zhou, and Y. Lu, "Hermes Attack: Steal DNN Models with Lossless Inference Accuracy," in *USENIX Security*, 2021.

[56] 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.

[57] Y. Kim, R. Daly, J. Kim, C. Fallin, J. H. Lee, D. Lee, C. Wilkerson, K. Lai, and O. Mutlu, "Flipping bits in memory without accessing them: An experimental study of dram disturbance errors," in *IEEE ISCA*, 2014.

[58] 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.

[59] R. Zhang, C. H. Center, L. Gerlach, D. Weber, L. Hetterich, Y. Lü, A. Kogler, and M. Schwarz, "Cachewarp: Software-based fault injection using selective state reset," in *USENIX Security*, 2024.

[60] R. Elbaz, D. Champagne, C. Gebotys, R. B. Lee, N. Potlapally, and L. Torres, "Hardware mechanisms for memory authentication: A survey of existing techniques and engines," *Springer Transactions on Computational Science IV*, 2009.

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

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

[63] V. Rijmen and J. Daemen, "Advanced encryption standard," *FIPS*, 2001.

[64] P. C. Kocher, "Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems," in *Springer Advances in Cryptology—CRYPTO*, 1996.

[65] T. Ichikawa, T. Kasuya, and M. Matsui, "Hardware evaluation of the aes finalists." in *AES candidate conference*, 2000.

[66] 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.

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

[68] D. McGrew and J. Viega, "The galois/counter mode of operation (gcm)," *NIST*, 2004.

[69] 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.

[70] W. E. Hall and C. S. Jutla, "Parallelizable authentication trees," in *Springer LNSC*, 2006.

[71] S. Na, S. Lee, Y. Kim, J. Park, and J. Huh, "Common counters: Compressed encryption counters for secure gpu memory," in *IEEE HPCA*, 2021.

[72] M. Umar, W. Hua, Z. Zhang, and G. E. Suh, "Softvn: Efficient memory protection via software-provided version numbers," in *IEEE ISCA*, 2022.

[73] S. Lee, J. Kim, S. Na, J. Park, and J. Huh, "Tnpu: Supporting trusted execution with tree-less integrity protection for neural processing unit," in *IEEE HPCA*, 2022.

[74] 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.

[75] R. Paccagnella, L. Luo, and C. W. Fletcher, "Lord of the ring (s): Side channel attacks on the cpuon-chip ring interconnect are practical," in *USENIX Security*, 2021.

[76] J. Izraelevitz, J. Yang, L. Zhang, J. Kim, X. Liu, A. Memaripour, Y. J. Soh, Z. Wang, Y. Xu, S. R. Dulloor *et al.*, "Basic performance measurements of the intel optane DC persistent memory module," *arXiv preprint arXiv:1903.05714*, 2019.

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

[78] 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.

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

[80] J. Yang, Y. Zhang, and L. Gao, "Fast secure processor for inhibiting software piracy and tampering," in *IEEE MICRO*, 2003.

[81] G. E. Suh, D. Clarke, B. Gassend, M. Van Dijk, and S. Devadas, "Aegis: Architecture for tamper-evident and tamper-resistant processing," in *ACM ICS*, 2003.

[82] W. Shi, H.-h. S. Lee, M. Ghosh, C. Lu, and A. Boldyreva, "High efficiency counter mode security architecture via prediction and pre-computation," in *IEEE ISCA*, 2005.

[83] S. Chhabra and Y. Solihin, "i-NVMM: A secure non-volatile main memory system with incremental encryption," in *IEEE ISCA*, 2011.

[84] M. Ye, C. Hughes, and A. Awad, "Osiris: A low-cost mechanism to enable restoration of secure non-volatile memories," in *IEEE MICRO*, 2018.

[85] 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.

[86] X. Wang, D. Talapkaliyev, M. Hicks, and X. Jian, "Self-reinforcing memoization for cryptography calculations in secure memory systems," in *IEEE MICRO*, 2022.

[87] I. Anati, F. McKeen, S. Gueron, S. Gueron, S. Johnson, R. Leslie-Hurd, H. Patil, C. Rozas, and H. Shafi, "Intel Software Guard Extensions (SGX)," 2015. [Online]. Available: https://community.intel.com/legacyfs/online/drupal_files/332680-002.pdf

[88] 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.

[89] 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.

[90] A. Kwong, D. Genkin, D. Gruss, and Y. Yarom, "Rambleed: Reading bits in memory without accessing them," in *IEEE S&P*, 2020.

[91] K. Q. Ye, M. Green, N. Sanguansin, L. Beringer, A. Petcher, and A. W. Appel, "Verified correctness and security of mbedtls hmac-drbg," in *ACM CCS*, 2017.

[92] C. P. García, S. Ul Hassan, N. Tuveri, I. Gridin, A. C. Aldaya, and B. B. Brumley, "Certified side channels," in *USENIX Security*, 2020.

[93] O. Acıiçmez, S. Gueron, and J.-P. Seifert, "New branch prediction vulnerabilities in openssl and necessary software countermeasures," in *Springer IMACC*, 2007.

[94] A. Cabrera Aldaya, R. Cuiman Marquez, A. J. Cabrera Sarmiento, and S. Sánchez-Solano, "Side-channel analysis of the modular inversion step in the rsa key generation algorithm," *International Journal of Circuit Theory and Applications*, 2017.

[95] 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.

[96] 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.

[97] 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.

[98] 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.

[99] A. Sev-Snp, "Strengthening vm isolation with integrity protection and more," *White Paper, January*, 2020.

[100] I. Corporation, "Intel software guard extensions (intel sgx) - key management on the 3rd generation intel xeon scalable processor," 2021.

[101] L. Wilke, J. Wichelmann, M. Morbitzer, and T. Eisenbarth, "Sevurity: No security without integrity: Breaking integrity-free memory encryption with minimal assumptions," in *IEEE S&P*, 2020.

[102] Intel, "Runtime Encryption of Memory with Intel® Total Memory Encryption–Multi-Key (Intel® TME-MK)." [Online]. Available: https://www.intel.com/content/www/us/en/developer/articles/news/runtime-encryption-of-memory-with-intel-tme-mk.html

[103] M. Li, Y. Zhang, H. Wang, K. Li, and Y. Cheng, "Cipherleaks: Breaking constant-time cryptography on amdsev via the ciphertext side channel," in *USENIX Security*, 2021.

[104] gururaj s, "gururaj-s/mirage." [Online]. Available: https://github.com/gururaj-s/mirage

[105] J. Szefer and S. Biedermann, "Towards fast hardware memory integrity checking with skewed merkle trees," in *IEEE HASP*, 2014.