Aria: Tolerating Skewed Workloads in Secure In-memory Key-value Stores

1st Fan Yang 2nd Youmin Chen 3rd Youyou Lu 4th Qing Wang 5th Jiwu Shu*

Tsinghua University
{yangf17, chenym16, q-wang18}@mails.tsinghua.edu.cn, {youyoulu, shujw}@tsinghua.edu.cn

***Abstract*— The recent advent of the hardware trusted execution environment (TEE), e.g., Intel SGX, enables encrypted and integrity-verified in-memory key-value (KV) stores. However, due to the architectural limitations of the hardware, it is non-trivial to build a secure in-memory KV store with SGX without compromising the performance. The reason comes from (*i*) the limited memory capacity the SGX TEE provides, and (*ii*) being unaware of the access patterns of skewed workloads, which are commonly seen in the real world.**

In this paper, we present Aria, a secure in-memory KV store based on SGX. Our goal is to utilize the limited resource while still achieving high performance. Aria places KV pairs and index structures directly in the untrusted memory and introduces the security metadata in the TEE to conduct protection. The core component of Aria is *Secure Cache*, a software-based cache layer, which uses the limited memory resource to guarantee the confidentiality and integrity (including freshness) of Aria. *Secure Cache* keeps the frequently accessed security metadata in the TEE memory at fine-granularity and evicts rarely-used ones to the untrusted memory. With *Secure Cache*, we have the opportunities to explore strategies that are impossible in SGX implementation. By decoupling the security metadata management from the index structure, Aria supports various index schemes. We implement Aria with the indexes of both a hash table and a B-tree. Experiments show that Aria improves throughput by up to 104% compared to the state-of-the-art system.

***Index Terms*—Trusted Execution, Intel SGX, Key-value Store**

I. Introduction

In-memory key-value (KV) stores have become a fundamental component in data center infrastructures [1], [2]. While KV stores provide the simple abstraction of interfaces for a variety of online applications, they pose serious security threats for the users, especially when they are deployed on the third-party untrusted cloud infrastructure, whose operating systems and hardware are exposed to potentially malicious attackers [3]–[6]. In an untrusted environment, an attacker can compromise the confidentiality and integrity of the stored data. In addition, software bugs, configuration errors, and security vulnerabilities also pose serious threats to cloud systems [7]–[11].

The hardware-based trusted execution environment (TEE), such as Intel Software Guard Extensions (SGX), provides shielded execution for applications in an untrusted infrastructure. Shielded execution provides strong security properties, such as confidentiality and integrity (including freshness), using a hardware-protected secure memory region. With such shielded execution environments, SGX is considered a promising application isolation technology that allows users to run their applications securely on an untrusted infrastructure. Given the importance of security threats in the cloud, public cloud providers, such as Microsoft Azure [12] and IBM Cloud [13], offer SGX-capable computing platforms to support confidential computation. And there is a recent surge in leveraging SGX for shielded execution of applications in the untrusted infrastructure [7], [14]–[18].

However, building in-memory KV stores with strong security guarantees always comes with compromised performance. SGX constructs a secure hardware container (i.e., *enclave*) to protect applications inside it from malicious attacks. The available space (i.e., enclave page cache, EPC) of the enclave is about one hundred megabytes. To accommodate larger data beyond the physical limit, SGX adopts a secure paging mechanism: SGX divides memory spaces into two regions. One is the EPC that stores recently accessed pages and a non-EPC region that stores pages evicted from the EPC. EPC and non-EPC regions are both protected by SGX. Hardware encrypts pages before moving them to the non-EPC region. SGX moves pages from the non-EPC region into the EPC before accessing them. Unfortunately, an EPC miss incurs significant paging overhead (about 40K cycles [19]) compared to an EPC hit (around 200 cycles). Therefore, placing all data of a KV store in EPC can incur significant performance degradation due to the hardware secure paging overhead.

To eliminate the paging overhead, recent work tries to place a KV store directly in untrusted memory, and manually build security metadata (e.g., message authentication codes (MACs) and encryption counters) in EPC to protect the integrity and confidentiality of KV items [15], [16], [18]. For example, ShieldStore [15], a state-of-the-art in-memory KV store, builds security metadata utilizing the internal format of its hash table-based index. It maintains a Merkle Tree (MT) [20] for each hash bucket to enforce protection and only stores the root node of each MT in the EPC. Since the number of buckets determines the space consumption of root nodes, ShieldStore can always prevent root nodes from exceeding the EPC space by using a proper bucket size. However, avoiding secure paging doesn't always indicate high efficiency under real-world workloads, which typically exhibit skewed access patterns [2], [21]–[24]. In ShieldStore, hot and cold KV pairs are treated equally — a Put/Get operation to a hot key is still processed by traversing through the corresponding MT first (i.e., conduct expensive MT verification) since the inner nodes and leaf nodes are always placed in the untrusted memory.

*Jiwu Shu is the corresponding author. This work is supported by National Key Research & Development Program of China (Grant No. 2018YFB1003301), the National Natural Science Foundation of China (Grant No. 61832011, 61772300), and Research and Development Plan in Key field of Guangdong Province (Grant No. 2018B010109002).

In this paper, we address this problem by introducing *Secure Cache*, a software-based EPC space manager that aims to provide both security guarantee and high performance for KV stores. In contrast to the past work that completely avoids swapping in the EPC, the key idea underlying *Secure Cache* lies in that it allows the size of security metadata to grow beyond the EPC size. Specifically, it caches frequently accessed MT nodes (including both inner and leaf nodes) in the EPC, and actively evicts cold ones to untrusted memory before the EPC space is used up. Note that *Secure Cache* doesn't utilize the paging mechanism provided by the hardware directly to evict data, since a 4-KB page may contain MT nodes of both cold and hot KVs, leading to sub-optimal performance. Instead, it adopts a fine-grained approach to track each individual MT node and decide whether it should be cached. In this way, a hot KV pair can be verified by simply checking its MT leaf node, so long as it is already cached in the EPC. We also incorporate an existing technique [20] to further mitigate the verification overhead in a Merkle Tree (e.g., recursive checking and updating), where verification and updating of a MT node stop immediately once encountering its ancestor node that has already been cached in the EPC.

With *Secure Cache*, we further perform an in-depth exploration of optimizations considering hit and miss penalty and thus optimize the cache policy. First, exposing *Secure Cache* management to a software-based way brings us the opportunities to enable optimizations that cannot be implemented with SGX hardware paging. We introduce two such optimizations: swap out without encryption, and prevent writing back clean security metadata to the untrusted memory. Second, since the integrity verification of a KV pair is always performed from the MT leaf node to the first cached MT inner node (or the MT root), the inner nodes of higher layers in the MT are typically accessed more frequently. Hence, we directly pin them to the EPC to prevent them from being evicted. Third, based on the observation that large scale cache makes managing cache metadata in memory no longer feasible [25], and data operations in the EPC incur higher latency than that in untrusted memory [26], we utilize the first-in-first-out (FIFO) policy to avoid the tax of hits of other cache replacement policies. Fourth, the organization of the MT controlled by *Secure Cache* is based on the memory address, which benefits from the hardware prefetching and cache locality.

Based on *Secure Cache*, we implement a secure KV store named Aria. Unlike ShiedStore that is closely coupled with a hash table, Aria doesn't rely on a specific index structure since security metadata are managed separately from the index structure. In this regard, we implement two KV store variants, Aria-H and Aria-T, which are based on a hash table and a B-tree, respectively. We use both YCSB [22] and Facebook ETC workload [2] to demonstrate the efficiency of our design. With keyspace ranging from 2M (million) to 134M, Aria-H always performs better than ShieldStore under skewed workloads, and Aria-H improves performance by up to 104% with 134M keyspace. With a fixed 10M keyspace, Aria-H improves throughput by up to 41% compared to ShieldStore [15] under YCSB workloads. Specifically, Aria-H still improves performance by 14% with only 15 MB EPC occupation for *Secure Cache* compared to ShieldStore which uses fixed 64 MB EPC storing the MT roots. Under the Facebook ETC workload [2] with 10M keyspace, Aria-H improves throughput by 33% on average compared to ShieldStore. For tree-based KV store, Aria-T improves throughput by 205% under Facebook ETC workload on average compared to a naive tree-based KV store implementation with SGX.

II. Background

In this section, we first present the background of Intel SGX and the threat model in Aria. Then we describe the encryption and integrity verification method, and how Merkle Tree works.

A. Intel SGX

Intel SGX is a set of x86 ISA extensions for TEE [27]. SGX allows applications to create a secure execution environment, called an *enclave*, to protect the user-level software in the enclave from being compromised by the environment, which includes the operating system, the virtual machine manager, the BIOS, and the hardware surrounding the CPU chip. An enclave includes Enclave Page Cache (EPC), a dedicated memory region protected by an on-chip Memory Encryption Engine (MEE) [28]. EPC can only be accessed by the owner enclave, as the virtual-to-physical mapping is protected by the hardware address translation logic [27]. Codes running in the enclave can access both un-protected memory and EPC. The data in the un-protected memory region are not protected by SGX. Every access to the EPC is protected by MEE at cache-line granularity. The data in the on-chip cache is in plaintext, and are encrypted and integrity-protected when they are written back to the EPC. For confidentiality, every evicted data from the CPU cache to the EPC is encrypted. When the evicted data is fetched from the EPC to the processor, they are decrypted. For integrity, SGX keeps a hash value for each cache-line and verifies an evicted cache-line by comparing it with the stored hash value of that address. To prevent replay attacks, SGX maintains a variant of the MT on the hash values, and the root is always stored in a safe area [27], [29], [30].

Limitations of SGX. There are mainly two performance considerations for SGX. One is the limited EPC resource. In the current SGX, approximate 94 MB [16] are available to the user (the recently released server supports 168 MB EPC). To allow the creation of enclaves with a size beyond that of EPC, SGX features a secure paging mechanism. If the size of memory pages used in an enclave exceeds the EPC limit, the OS can evict EPC pages from the EPC region to an un-protected memory region using SGX instructions at 4 KB granularity. The evicted pages are encrypted. When they are reaccessed, they are decrypted, and their integrity is checked. Bringing evicted pages back requires a secure page swap mechanism, incurring significant performance overhead (2x-2000x) [15], [19]. Besides, Microsoft Windows is yet to support such paging, limiting the enclave memory only to

the EPC size. This limitation will affect the performance and feasibility of applications based on SGX.

The other is the SGX Edge Calls. Intel SGX supports a call-gate mechanism to control entry into and exit from the TEE. After the secure-enclave is initialized, the only way for untrusted codes (outside the enclave) to start executing trusted codes (inside the enclave) is by invoking *ECALL*. Besides, the TEE created by SGX has only user-level privileges, it has no access to hardware or other OS resources. Privileged instructions, such as system calls, are not allowed to execute inside the enclaves. Before executing a system call, the enclave code has to make an *OCALL* to exit the enclave to the untrusted code. *ECALL* and *OCALL* are considered edge functions, as they cause execution to cross security boundaries [26]. The parameters of Edge calls are copied between protected memory and un-protected memory and this procedure needs security checks. The cost of crossing the enclave boundary is expensive because of security checks, TLB flushes [19], and L1 cache flushes [31]. The overhead of an *ECALL/OCALL* is about 8000-14000 CPU cycles [26], which is much more expensive than a regular system call.

B. Threat Model

The Trusted Computing Base (TCB) of Aria consists of the processor chip, the codes executed inside the enclave, and data stored in the EPC. Thus Aria is resilient to malicious privileged attacks and certain physical attacks. With the reduced TCB, Aria does not rely on the security of the operating system managed by cloud providers. Furthermore, it is also resilient to direct conventional physical attacks such as cold boot attacks [32], or bus probing [33]. Aria can protect cloud user data from such compromised operating systems and physical attacks by malicious staff. However, due to the lack of protection for side-channel attacks in the SGX [34], we do not consider this kind of attack. Separate software or hardware techniques have been proposed to provide protection [35]–[37]. Besides, Aria does not consider availability attacks similar to ShieldStore [15]. Note that since SGX libraries provide essential implementations to establish the initial secure connection between the client and server in the enclave via remote attestation [15], [38], we assume that network communication between clients and Aria is protected through this protection.

C. Encryption and Integrity Verification

AES_CTR counter mode encryption (CME) and message authentication code (MAC) are widely used ways to conduct encryption and ensure integrity respectively [15], [29], [39]. CME introduces counters for encryption and each counter is associated with one item (e.g., one KV pair) and the counter is incremented for every encryption cycle. Message authentication code is a cryptographic checksum on data (e.g., KV pair) that uses a session key to detect both accidental and intentional modifications of the data.

However, the above encryption and integrity verification method is unable to detect replay attacks that corrupt data

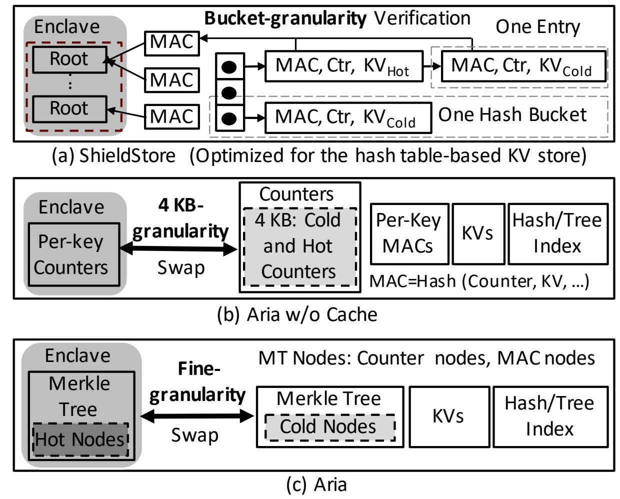

Fig. 1. Three types of design schemes. (a) shows the state-of-the-art in-memory KV store with SGX optimized for hash table-based index. ShieldStore [15] stores the whole KV store in untrusted memory and builds a MT for each hash bucket. Ctr means counter. (b) shows an intuitive approach. It only places counters inside the enclave and stores the whole KV store in untrusted memory. (c) shows the architecture of Aria. It adopts a fine-granularity swap in a software-based hotness-aware way.

freshness. Attackers can replay encrypted KV pair, its associated counter, and MAC to their old value. Merkle Tree is a widely used structure to detect replay attacks [29], [30], [39], [40]. MACs are hierarchically organized as a tree structure. The protected data is placed in its leaf nodes. The leaf nodes can be protected since a parent MAC can check the integrity of multiple child MACs, and the root node is always placed in a safe place, which cannot be damaged or replayed by adversaries. To access a data block, we check its integrity by verifying the corresponding ancestor nodes, including the root node in the Merkle Tree. These ancestor nodes are updated whenever the leaf node is updated.

III. MOTIVATION

In this section, we first present existing design schemes for secure in-memory KV stores and then compare them to motivate our design.

A KV store typically consists of an index structure, with which we can find a KV pair by passing the key, and a storage manager that keeps actual KV pairs. There are two commonly used index schemes, which are the hash-based (e.g., chained hashing, cuckoo hashing, etc.) and tree-based index (B-tree, etc.). The hash table-based index provides a simple and fast point query, while the tree-based index supports range query by keeping its items ordered. Using a hardware-based way to protect an in-memory KV store requires careful refactoring of data into trusted and untrusted components. In the following part, we describe and analyze existing approaches that build secure in-memory KV stores.

KV stores are naturally executed in the enclave because most of their codes are dedicated to manipulating private data in response to client requests. However, since KV stores are typically larger than the EPC size, the performance is significantly impacted. As shown in Figure 2, the performance

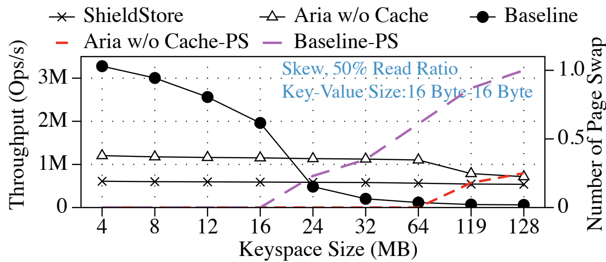

Fig. 2. Performance of different design schemes. We use fixed 16-byte keys. Keyspace means the total different keys in the KV store and keyspace size means the total size of all keys (e.g., 16 MB keyspace size means 1048576 keys). PS means the number of secure paging compared to that of Baseline at 128 MB keyspace size. (1) *ShieldStore*: a state-of-the-art in-memory hash table-based KV store with SGX. (2) *Aria w/o Cache*: only putting the counters in the enclave. (3) *Baseline*: putting the whole KV store in the enclave.

of Baseline sharply decreases at 24 MB keyspace size when secure paging starts to occur (line Baseline-PS).

Instead, some researches maintain security metadata (i.e., encryption counters and MACs) in the EPC to conduct protection among KV pairs [15], [16], [18]. SGX protects security metadata in the EPC from attacks, indirectly guarding KV pairs. Since security metadata takes up less space, the secure paging overhead can be largely reduced. As shown in Figure 1(a), ShieldStore [15], a state-of-the-art secure in-memory KV store with SGX, puts security metadata, KV pairs, and hash table in untrusted memory. Security metadata and KV pairs are stored inside the hash table. It builds a MT for each (or more than one) hash table bucket and only stores the MT roots in the EPC. Since the number of MT roots is controllable, ShieldStore only places 4M roots (64 MB) in the EPC for avoiding secure paging, achieving the best performance.

Following the above design scheme, we present an intuitive design called Aria w/o Cache in the paper. Figure 1(b) shows the data layout of Aria w/o Cache. We use one encryption counter to encrypt one KV pair. The KV pair and its corresponding counter are combined to generate one MAC which protects the integrity of that KV pair. The MACs are stored in untrusted memory to reduce EPC usage. Since all encryption counters are stored in the EPC, we can always trust the counter value. **Any attacks to the KV pairs or MACs in untrusted memory will cause a mismatch between the MAC computed from its corresponding counter and KV pair with the MAC stored in untrusted memory.** Figure 2 presents the throughput of Aria w/o Cache. The throughput of Aria w/o Cache is steady until keyspace size reaches 119 MB, the performance drops because the counter size is beyond the EPC capacity, and the secure paging degrades the performance. However, it still shows better performance than ShieldStore since the hardware secure paging is hotness-aware.

Table I compares the aforementioned design schemes in four aspects. The first two aspects impact the system **performance**. First, Aria w/o Cache relies on the SGX secure paging to protect security metadata that is swapped out to the non-EPC region. Since secure paging conducts lots of actions including OS context switch, data copy, encryption, and SGX integrity tree update, it is heavy for security metadata protection. For security metadata, since it can be stored in plaintext [39], it

	Protection Granularity	KV Hotness-aware	Index Schemes	EPC Occupation
ShieldStore	Hash Bucket	Unaware	Hash	Low
Aria w/o Cache	Page (4 KB)	Aware	Hash/Tree	Medium
Aria	KV pair	Aware	Hash/Tree	Low

TABLE I
COMPARISON BETWEEN DIFFERENT DESIGN.

is only necessary to ensure the integrity of them. Besides, 4 KB granularity is too coarse for security metadata which is usually a few bytes. ShieldStore conducts a bucket-granularity verification which incurs read and verification amplification. For every KV operation (Put/Get), it needs to read the whole bucket's MAC values, and then compute and verify the MAC value with the corresponding root stored in the EPC. Besides, it has to update the root for Put requests. Since applications access a KV store by putting/getting individual KV pair, the protection granularity of both Aria w/o Cache and ShieldStore mismatch with the semantic of the KV store. Second, Real-world workloads commonly exhibit highly skewed access patterns [1], [2], [21], [22]. Here, a small fraction of hot items receive disproportionately more requests than the remaining items. Many such workloads can be modeled using Zipfian access distributions [2], [24], [41], [42]; recent work has shown that some real workloads exhibit unprecedented skew levels (e.g., Zipf distributions with $\alpha > 1$) [23], [24]. Though the secure paging of SGX can maintain frequently accessed pages in the EPC, SGX can swap some pages (4 KB) out, in which there are security metadata of both hot and cold KV pairs, incurring secure paging overhead when hot keys are accessed again. ShieldStore is unaware of such hotness and hot KV pair may reside in a bucket with a long list, exacerbating the side effect of the bucket-granularity verification.

The other two factors present the **usability** of the KV store. ShieldStore is designed to be closely coupled to chained hashing, which prevents them from being widely adopted. While Aria w/o Cache and Aria can support various index schemes. Finally, since the EPC is a limited resource and is usually shared by multiple tenants in the cloud environment, it is better to use it as little as possible. Aria w/o Cache consumes medium EPC space since it puts all counters inside EPC. When the number of KV pairs increases, the EPC occupation grows proportionally. ShieldStore consumes low EPC occupation since it only stores a fixed number of MT roots in the EPC. However, with growing keyspace, ShieldStore suffers significant verification overhead due to the long bucket.

To this end, we propose Aria (in Figure 1(c)) to address the above problems simultaneously. Compared to them, Aria achieves KV semantic-aware protection with fine-granularity swap for security metadata and supports both hash table-based index and tree-based indexes. The EPC occupation of Aria is low and even, it achieves higher performance than ShieldStore with fewer EPC occupation, which is beneficial to tenants who purse the price-performance ratio in the cloud environment.

IV. DESIGN OF *Secure Cache*

A. *Overall Architecture*

Figure 3 shows the overall architecture of *Secure Cache*. Since we place encrypted KV pairs in untrusted memory, we

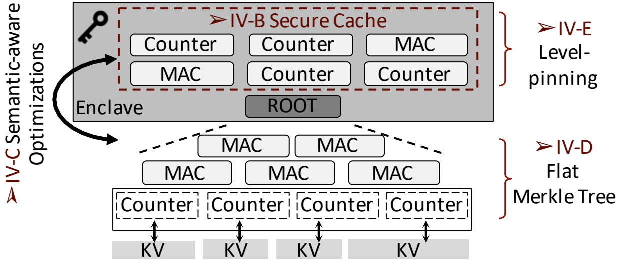

Fig. 3. Overall Architecture of *Secure Cache*.

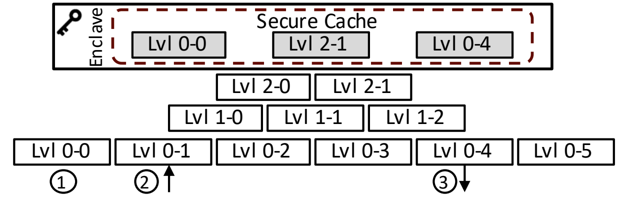

Fig. 4. *Caching* and *eviction* of *Secure Cache*. The MT verification stops at the first cached nodes. If the counter (leaf node) is cached in *Secure Cache*, we can directly use it to decrypt the KV pair without MT verification from the leaf to the root, achieving the KV pair granularity protection.

place counters in the EPC to indirectly protect KV pairs from replay attacks. To avoid the hardware-based secure paging, we build a MT among encryption counters and introduce *Secure Cache* (IV-B) to manually manage the swap of MT nodes between the EPC and untrusted memory in a software-controlled way. Specifically, *Secure Cache* caches the frequently accessed MT nodes in it. The advantage of such caching is that once we find the counters in *Secure Cache*, we can eliminate the MT verification overhead for Put/Get requests, achieving the integrity protection granularity at KV pair.

Secure Cache management in software provides several benefits beyond the hardware secure paging. In particular, it enables the semantic-aware optimizations we discuss in IV-C. To reduce the software-based swap overhead, we propose a flat MT structure (IV-D) to balance the verification overhead against the cache hit ratio. We further optimize Aria by reducing the hit and miss penalty of *Secure Cache* (IV-E). We propose a pinning mechanism to pin top-K levels of the MT in *Secure Cache* to mitigate the MT verification overhead when misses occur. And we revisit the cache eviction policy of *Secure Cache* and adopt FIFO policy to reduce the hit penalty.

B. Secure Cache Mechanism

Secure Cache stores the frequently accessed MT nodes inside the EPC, and swap out cold ones to untrusted memory. It processes a read/write access by first checking whether the requested data is cached. The item is read or written directly if it is already cached (e.g., ① in Figure 4). Since the cached items are protected by SGX, we can always trust its value.

Then, we describe two actions that help with achieving KV pair granularity protection while still preventing replay attacks: 1) swap a MT node from untrusted memory into EPC (i.e., *Caching*); and 2) evict the cold ones to the untrusted memory area (i.e., *Eviction*). Figure 4 shows a cache state at some point in time. For simplicity, we describe a 2-ary MT (every parent tree node contains two MACs of its child nodes) with 4 layers. Lvl m-n means the n-th node in the m-th level (lvl-0 stores the counters). Currently, *Secure Cache* caches three tree nodes.

Caching. *Secure Cache* needs to check the integrity of the uncached item before placing it in the EPC. The corresponding MT nodes are verified during the verification. The following shows the steps of caching a new item (say, Lvl 0-1) in detail.

Since Lvl 0-1 (i.e., (②) in Figure 4) is not cached in the enclave, Aria first reads it from the untrusted memory and computes a MAC, and then reads its parent node (i.e., Lvl 1-0) to compare with the computed MAC. Since Lvl 1-0 is not cached in enclave, a MAC of Lvl 1-0 is computed to compare with Lvl 2-0. The ancestor nodes are verified recursively till the root which always resides in the EPC. After verifying the Lvl 0-1, we put it in the EPC, and any following verification procedure can stop at the cached MT nodes. This is valid because the node is now protected by SGX hardware, and thus no longer needs to be protected by its parent node, which reduces the cost of subsequent access to it.

Eviction. If the *Secure Cache* is full, we need to evict one item (e.g., Lvl 0-4) to accommodate the newly inserted cache item. Before evicting, we need to update the MAC of its parent so as to ensure the correct verification for following requests.

Aria first computes the MAC of the to-be-evicted cache item, say, Lvl 0-4 in ③, then writes the computed MAC to their parent node (Lvl 1-2). Since Lvl 1-2 is not cached, *Secure Cache* first needs to swap in this node. After verifying the integrity of the node Lvl 1-2, it is placed in *Secure Cache*. Then, the computed value from Lvl 0-4 is written to the cached node (Lvl 1-2). Finally, the evicted cache item (Lvl 0-4) is directed written to their original places in untrusted memory.

A secure initialization of the MT determines the correctness of the runtime integrity verification. At the initialization phase, we assign a random value to each counter first. Then, we compute MAC values of leaf nodes and store the computed MAC in their parent nodes. We recursively conduct this procedure from the leaf till the root. Finally, a consistent MT is generated. The initialization is executed inside the enclave.

Proof Sketch of *Secure Cache*. The MT can detect replay attacks on leaves because ① the root is secure and ② contains the newest information of all leaves (i.e., a change to a leaf requires that all the nodes between it and the root be updated) [20]. *Caching* process guarantees the integrity of the node fetched from untrusted memory, and then SGX guarantees its security. Hence we can use the fetched node as the root of a new smaller MT. To maintain ②, *Secure Cache* propagates changes on a leaf from the leaf to the first cached node (e.g., if the leaf is cached, we can directly update it and stop propagating). If an updated cached node is evicted, *Secure Cache* propagates its update to its parent nodes, ensuring that the newest information of a leaf node always resides in at least one node in the EPC. This *Eviction* process will finally propagate updates of leaves to the root. Thus *Secure Cache* prevents replay attacks on counters. Since we compute a MAC for each KV pair and its counter, and the integrity of all counters is guaranteed, we prevent replay attacks on KV pairs [39]. The confidentiality and integrity of KV pairs are guaranteed using

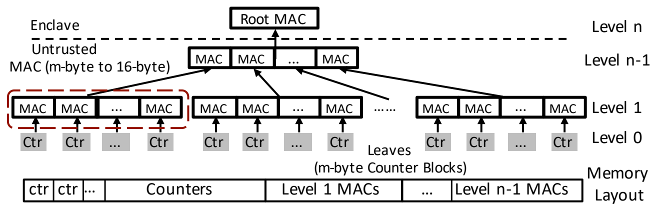

Fig. 5. Merkle Tree layout. We organize MT nodes in continues untrusted memory space and the MT root always stay in the EPC. Ctr means counter.

the CME and MAC discussed in SectionII-C.

C. Semantic-aware optimizations

Secure Cache enables optimizations that are not yet available in SGX. We show two of such optimizations: swap out without encryption and avoiding write-back for clean items.

Eliminating Encryption. Considering the semantic of security metadata (i.e., as auxiliaries to protect KV pairs), its plaintext is meaningless, and it is only necessary to ensure the integrity of them, which is enough for the protection of KV pairs. Therefore we avoid the encryption overhead which must be included in SGX secure paging when conducting swapping.

Avoiding Write-back for Clean Cache Items. Eviction is a heavy operation on the critical path in large memory footprint workloads. However, if the item to be evicted is already present in untrusted memory and has not been modified since the previous eviction, the write-back is unnecessary and the item can be discarded. This optimization is not implemented in SGX and it is unclear whether such implementation is at all possible. This is because the only SGX memory eviction instruction (EWB) forces the page being evicted to be written to the backing store regardless of whether it has been modified [14]. In *Secure Cache*, we maintain a one-bit tag for each cache item in *Secure Cache* to mark the dirty or clean state. Clean cache items in *Secure Cache* can be directly discarded.

D. Flat Merkle Tree

Integrity verification of the leaf nodes needs to go through all levels of the MT till the first cached nodes. Since a counter (in the leaf node) is associated with one unique key (of the KV store), the height of the MT increases drastically for a large number of KV pairs, resulting in lots of MAC computation overhead during MT verification. Besides, the verification procedure is conducted sequentially instead of in parallel, which further exacerbates the cost. Hence, we must focus on limiting the number of verification (i.e., MAC computation and comparison) times.

To reduce the verification overhead, we need to flatten the MT structure. We achieve this by increasing the input length of the MAC computation function, making the fixed-size MAC hash value cover more bytes at its child level. As shown in Figure 5, the dashed box represents one MT node and the input length equals the node size. Longer input length forms a shorter tree, so the number of verification times is reduced. However, longer input length causes higher MAC computation overhead. In addition, Aria conducts MAC computation inside the enclave, so we need to copy the MT nodes from untrusted memory to the EPC before proceeding the MAC computation. Longer input length causes higher MT nodes copy overhead. In Section VI-D3 we analyze the trade-off between the input length and integrity verification overhead.

E. Optimizing Secure Cache Hits and Misses

Level-pinning. Retrieving a counter from the *Secure Cache* requires multiple integrity verification if the branch of the MT containing this counter doesn't exist in the EPC. In the worst case, O(h-1) (h is the height of the MT) MAC computation and comparison is required to verify the counter. In order to mitigate the worst-case overhead, Aria adopts level pinning. The idea of the level pinning is straightforward. If the MT has h levels, Aria keeps MT nodes for top-k levels ($k \leq h-1$) in the EPC. With level-pinning, the number of the worst-case MT verification is reduced to O(h-k-1), and *Secure Cache* only manages the swap of nodes in levels that are not pinned.

Level-pinning requires small amounts of EPC capacity. In the MT, an upper level (L_i) is T times smaller than a lower level (L_{i+1}). Assuming a keyspace of ten million (which requires at least ten million counters), we construct a 5-level MT and the amount of the EPC required to pin for L1, L2, L3, and L4 is 0.75 KB, 1.5 KB, 68.25 KB, and 3.18 MB, respectively. Thus, the total size of the top-4 levels only consumes about 3.25 MB EPC. Note that after the MT initialization, it is enough to only pin the L4 in the EPC. However, we still pin the top-3 level for low-overhead dynamically resizing. When we unpin the L4, we directly compute MACs of L4 and update L3 in the EPC without copying overhead from untrusted memory to the EPC.

Revisiting the Hit Penalty. During the cache hit process in *Secure Cache*, Aria needs to update the metadata of *Secure Cache* for picking out the suitable item in the next eviction process. However, this update requires many memory operations when the cache size is large, which incurs high overhead. Worse, memory operations in the EPC lead to longer latency than that for untrusted memory [26], which further increases the penalty of such an update. Therefore, we adopt a simple yet efficient replace policy, FIFO in *Secure Cache* to reduce the metadata update overhead. We present the performance advantage of such a policy over LRU in Figure 12.

Stopping Swap. Considering the high miss penalty, we stop the swap when the hit ratio of *Secure Cache* is below a certain threshold (e.g., 70% in Aria). However, we still use the level-pinning mechanism to reduce the integrity verification overhead. *Secure Cache* chooses to pin the upper layers of the MT according to its current size, eliminating secure paging (e.g., for 10M keyspace, Aria pins MT nodes except L0). The stopping swap process is as follows: 1) during caching, we verify all nodes and only move nodes chosen to be pinned to the EPC; 2) during eviction, we only evict nodes that don't reside in the pinning layer (e.g., L0).

V. Implementation of Aria

In this section, we present the implementation detail of Aria. Aria uses the `sgx_aes_ctr_encrypt` counter mode

encryption and `sgx_rijndael128_cmac` to guarantee the confidentiality and integrity of each KV pair respectively. Both of them are in the current Intel SGX SDK [34]. Counters used for `sgx_aes_ctr_encrypt` and MACs generated by `sgx_rijndael128_cmac` are both 16-byte in Aria. All encryption, decryption, and MAC computation are conducted inside the enclave, while the encrypted KV pairs and security metadata are stored in untrusted memory. Aria builds a MT in continuous memory space to increase the cache locality (V-A) and utilizes a heap allocator (V-B) which eliminates OCALLs on every untrusted memory allocation for KV pairs. To support both hash table-based and tree-based index schemes, we decouple the MT organization from the index structure and introduce a *redirection layer* to index the counters (V-C). Finally, we summarize Aria by walking through an example of Put and Get operations (V-D).

A. Continuous Merkle Tree

Figure 5 shows the overall memory layout of the MT in Aria. MT nodes are stored in untrusted memory in continuous memory space which benefits from CPU perfecting and cache locality. The address of a MT node can be calculated directly using the offset of its child node. We use a fixed input length (m-byte in Figure 5) when calling the MAC computation function, and the output of the MAC hash function is a 16-byte MAC value. The size of one MT node equals to the input length of the MAC computation function. As shown in Figure 5, the dashed box represents one node that contains multiple MAC values of the corresponding child nodes (ctr (counter) in the figure). The root MAC must be stored inside the EPC, to ensure the safety of the whole tree [20], [39]. For MT expansion, we construct a new MT structure if there is no free counter for the newly inserted KV pair. Aria reserves a new MT using a background thread when the number of used counters reaches the threshold.

B. User-space Heap Allocator

If we want to allocate untrusted memory by `malloc/free` in the enclave, OCALLs are needed to go out of the enclave. To eliminate OCALLs on every untrusted memory allocation, we implement a user-space heap allocator in Aria which can be directly used inside the enclave.

Aria maintains an untrusted memory pool and cuts the untrusted memory space into 4 MB chunks. Further, the 4 MB chunks are cut into different sizes of data blocks, and the data blocks in the same chunk are of the same size. The size of data blocks in a chunk is recorded for that chunk when it is ready for allocation. A bitmap is also maintained for that chunk to track the used and unused data blocks. This metadata is stored in the EPC to prevent corruption on the allocator metadata. For fast allocating, we also maintain a free list. However, we put it in untrusted memory to eliminate EPC usage. Upon receiving an allocating request, the allocator first chooses a proper memory chunk where the data block size best fits the allocating request size, allocates a free data block from the free list, and checks the allocated address with the corresponding offset in the bitmap. Finally, we modify the bitmap. Since the initial address of each chunk is 4 MB-aligned, the offset of an allocated memory block in the chunk can be calculated directly using its address and its initial address. Hence the bitmap update would add minimum overhead to Aria. For an allocation with a size larger than 4 MB (which is less likely to happen in a KV store), we directly assign it with one or multiple contiguous chunks.

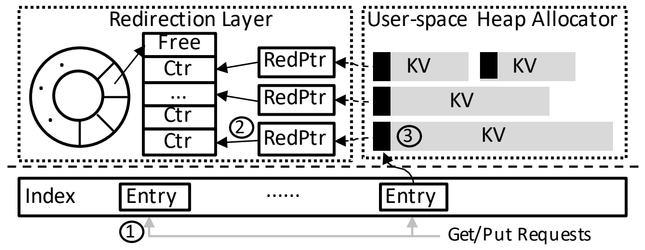

Fig. 6. Decoupled structure. Aria decouples the index from the security metadata management so as to support various index schemes.

C. Decoupled Structure and Index Protection

Decoupled Structure. Aria adopts a decoupled design to support various index schemes such as hash table and B-tree. To enable Aria to support multiple index schemes, we decouple the index from the security metadata management and build security metadata only on KV pairs. Then we introduce a *redirection layer* to map each KV pair to its associated security metadata and to manage the space storing security metadata. Figure 6 shows the relationship among index, redirection layer, and user-space heap allocator. The redirection layer consists of lots of redirection pointers (i.e., RedPtr) each of which points to a unique encryption counter. Each RedPtr is associated with only one KV pair. When a new item is inserted into the KV store (①), Aria fetches a free counter from the *redirection layer* and assigns it to the RedPtr (②). Aria places RedPtr (black area in the figure) along with each KV pair, and the KV pair is managed by a user-space heap allocator (③).

Counter Area Management. We maintain a circular buffer in untrusted memory to record the offset of the freed counters and a bitmap in the EPC to record the occupation of corresponding counters. The head pointer and tail pointer of the circular buffer are placed in the EPC. When serving a counter fetch operation, a free counter is returned which is pointed by the head pointer, and the counter's occupation is checked using the bitmap. If it is used, we assert that an attack happens. The bitmap is updated when fetching/freeing a counter. This updating is fast since we already know the offset of the counter. If the counter area is used out, Aria will apply for memory allocation from the user-space heap allocator and build a new MT over the newly allocated counter area.

We present two different index design (a hash table and a B-tree) based on Aria. The index entrance such as the hash table pointer and the B-tree root pointer is stored in EPC, ensuring that we can always find the correct place storing the index.

Aria-H is a hash table-based index structure (Aria-H) with a chaining linked-list to resolve collisions similar to Shield-

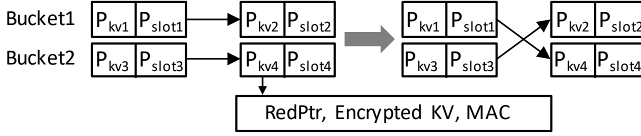

Fig. 7. Attacks to the connections of the index structure. It succeeds since the connection (pointer) is not protected by the security metadata.

Store [15]. It implements a key hint in the data entry to reduce the cost of searching encrypted keys. The key hint is a hash value of the plaintext key and is stored in each data entry. The hint is used to find candidates in the hash bucket for the key (of Put/Get) without decrypting the encrypted KV pair [15].

Aria-T is a B-tree-based index structure. For every KV request, Aria-T needs to decrypt the encrypted KV pair and than compare the targeted key with the decrypted key to find the correct tree branch.

Index Protection. A KV store includes KV pairs and an index structure that connects KV pairs. Since the security metadata is built above KV pairs, the index is not protected by the MT. Attacks to the connections of the index structure and unauthorized deletion cause missing the targeted key in KV store though it exists. Figure 7 shows the attack that exchanges two slot pointers in the hash table without being detected (B-tree is similar). Note that the availability attacks (e.g., corrupt the pointer value causes the crash of the system) is beyond the scope of this paper and we discuss this in Section VII.

We observe that both linked-list hash table and B-tree are connected by a series of pointers. If we guarantee the security of the index entrance (which is stored in the EPC) and protect the connection between the adjacent element, we are able to detect such attacks. We add an additional field to the input for the generation of KV pair's MAC. For Aria-H, the additional field includes the address of the forward node. For example, for KV pair pointed by P_{kv4}, the address of P_{slot3} is included when generating the MAC. For Aria-T, the additional field includes the address of the pointer that points to this node.

For unauthorized deletion, attackers can deliberately clear the content in slots. Thus, we need to record some metadata inside the enclave. For Aria-H, we record the number of data entries in each bucket in the enclave. For Aria-T, we record the number of tree nodes from the root node to each leaf node in the enclave. We use these metadata to detect unauthorized deletion once we can't find the key in the KV store.

D. Putting It All Together

We summarize the design of Aria by walking through an example of putting a new KV pair into a KV store and then getting it. Note that we focus on operations at the server-side since they are responsible for the confidentiality and integrity of KV pairs. The protection between the clients and the server is beyond the scope of this paper and can be applied using the existing remote attestation technique SGX provides [15], [38].

Put(Key, Value) request is processed with the following steps:

1) Aria first retrieves the index with the given key to find the actual place that serves the Put operations.

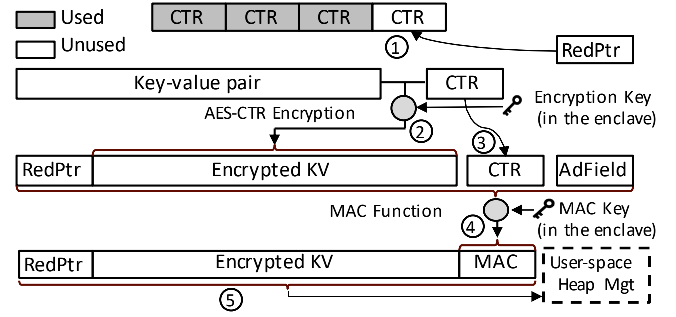

Fig. 8. Description of a Put operation in Aria. CTR means counter.

2) Then Aria creates a RedPtr for the newly inserted key and assigns a free counter fetched from the *redirection layer* to it (in Figure 8 ①). Before the counter can be used for encryption, the integrity of the counter value is verified by *Secure Cache*. If there exists the same key in the KV store, we reuse the original counter for encryption.

3) After verifying the counter, Aria conducts encryption and integrity protection over the inserted KV pair shown in Figure 8. The key and value are concatenated and encrypted by counter mode encryption with a 128-bit global secret key and the corresponding counter (in ②). Before the encryption, the counter corresponded to that RedPtr is incremented. After obtaining the encrypted KV pair, we combine the RedPtr, the encrypted KV, the counter value, and the AdField (additional field is discussed in Section V-C) in a continuous area (in ③). Then Aria computes a keyed hash value for integrity (MAC) (in ④). The computed MAC is attached to the encrypted KV pair. Finally, the combined item is delivered to the user-space heap allocator for management (in ⑤).

4) Aria allocates a data block from the user-space heap allocator and copies the record, computed MAC and RedPtr with the format of (RedPtr, k_len, key, v_len, value, MAC) into the allocated space.

5) Finally Aria updates the corresponding entry in the index to point to this allocated space.

Get(Key) request is served with the following steps:

1) Aria first locates a KV item by referring to the index with the given key to find the actual encrypted KV item. This procedure is dependent on the index scheme. For hash table-based index, Aria will decrypt every ciphertext that meets the requirement of the hash function and then compare the decrypted key with the targeted key (of Get) until they match. For tree-based index, it needs to decrypt every encountered tree nodes and choose the correct branch until it meets the targeted key.

2) For every decryption process, Aria uses the RedPtr to fetch the corresponding counter. Then *Secure Cache* verifies the integrity of the counter. After that, we compute the MAC value using the stored RedPtr, encrypted KV pair, counter value and AdField. Then we compare the computed MAC with the stored MAC value. If they mismatch, we assert an attack happens. After verifying the integrity of the encrypted KV pair, we use the verified counter to conduct decryption.

3) Finally, we compare the decrypted key with the targeted

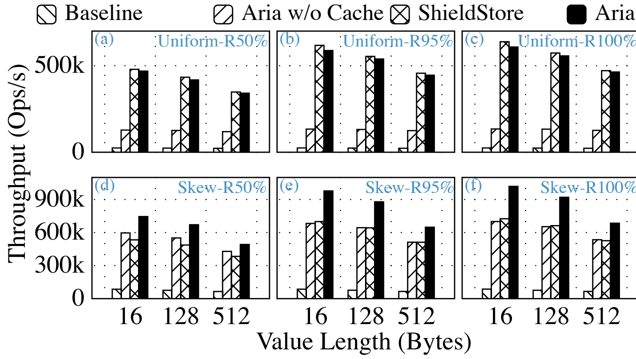

Fig. 9. Overall performance with hash table-based index.

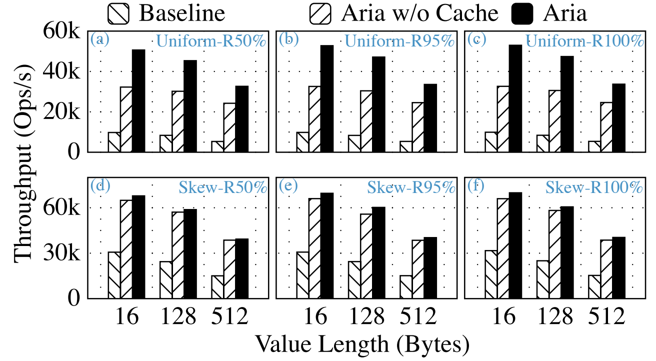

Fig. 10. Overall performance with B-tree-based index.

key. If they mismatch, we continue index retrieving, verification, and decryption procedure.

VI. EVALUATION

We evaluate Aria on a machine with an Intel Core i7-7700 processor. The processor has 32KB instruction and data caches, 256KB L2 cache, and 8192KB shared L3 cache. The machine has two memory channels with two 16 GB DIMM modules and runs Ubuntu 18.04.1 LTS 64bit with linux kernel 5.4.0-48. We run the SGX driver and SGX at version 2.6 [34]. Since the machine we use only supports 91 MB EPC, we set the *HeapMaxSize* in Enclave.config.xml as 91 MB. This setting avoids the occurrence of hardware secure page swap. The content of *Secure Cache* is set as large as possible.

In the experiment, we focus on the performance of the KV store with SGX, and KV requests are generated by the server without network components. This setup (same as ShieldStore) enables us to explore the CPU-memory performance impact with shielded execution under different design schemes. Unless specified, all experiments use a single thread.

Compared Schemes. Here are the different design schemes we mainly use in the evaluation:

1) Baseline puts the whole KV store in the EPC without any modification to the KV store.

2) *Aria w/o Cache* only places all counters in the EPC. Swapped out counters are protected by SGX hardware.

3) *ShieldStore* [15] is a state-of-the-art hash table-based secure KV store. Since it is built at SGX and driver version 1.8, we transplant it to version 2.6 for consistency.

4) *Aria* is our proposed design and adopts a fine-granularity swap with KV hotness-aware.

A. YCSB Microbenchmark

Workloads. We first evaluate the performance of Aria by varying the value size, read ratio and skewness with YCSB workloads [22]. All of them have the key range (a.k.a, keyspace) of 10 million and the key size of 16 bytes. The skewed workload uses a zipfan distribution of key popularity (skewness of 0.99, the default setting in YCSB), and the uniform workload generates keys with a uniform distribution. We evaluate three different read ratio, RD_50 (50% *Get*), RD_95 (95% *Get*), and RD_100 (100% *Get*), and three different value sizes, small (16-byte), medium (128-byte), and large (512-byte). We initially set 10 million KV pairs for each test similar to ShieldStore [15]. All of the working sets in our experiments are beyond the capacity of EPC size (91 MB).

Aria-H. The experimental results of hash table-based KV store are shown in Figure 9 and we make the following observations: First, Aria-H improves performance on average by 40%, 38%, and 28% for the small, medium, and large data sets respectively over ShieldStore under skewed workloads. The performance gain comes from the *Secure Cache*. It absorbs lots of security metadata access under skew distribution, avoiding most MT verification overhead. Since we can directly use the cached counter for decryption and MAC comparison, we achieve KV-granularity protection in Aria. Second, Aria w/o Cache scheme shows comparable performance with ShieldStore because under skewed workloads, the hardware secure paging mechanism is still KV pair hotness-aware. However, since the granularity of the swap is 4 KB, it is possible that the swapped out pages contain both hot and cold data, and the following access to the hot data incurs secure paging, degrading the performance. Thus Aria-H performs better than Aria w/o Cache under skew workloads. Third, ShieldStore performs slightly better than Aria-H under uniform workloads. Since uniform workloads present random access characteristics, Aria stops the swap mechanism and only uses level-pinning. Thus Aria needs to conduct one MT verification for every Put/Get requests which hurt the performance. However, when the keyspace is large, Aria still shows better performance than ShieldStore under uniform workloads shown in Figure 13.

Aria-T. Since ShieldStore can't support tree-based index, we only test Aria, Aria w/o Cache and Baseline. Figure 10 shows results of B-tree-based KV stores. The throughput of all schemes is much lower than hash table-based KV store. B-tree-based index reduces throughput by about 10x. Since for every query operations, it needs to decrypt every node encountered during indexing, significantly degrading the performance. Instead, we use a key hint in hash table-based KV store to eliminate decrypting every item encountered.

B. Facebook ETC Workload

We emulate the ETC pool at Facebook [2] as the production workload to evaluate the behavior of different design schemes. We use fixed 16-byte keys and make the value size variable. It has three kinds of KV pair distribution. The value size can be tiny (1-13 bytes), small (14-300 bytes), or large (larger than

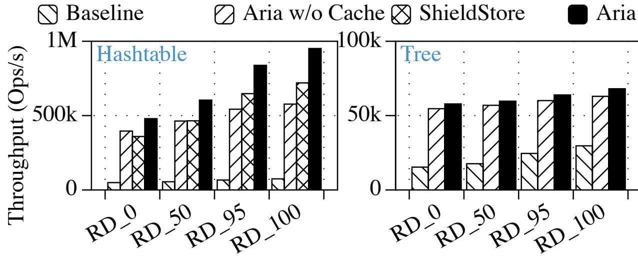

Fig. 11. Throughput with Facebook ETC.

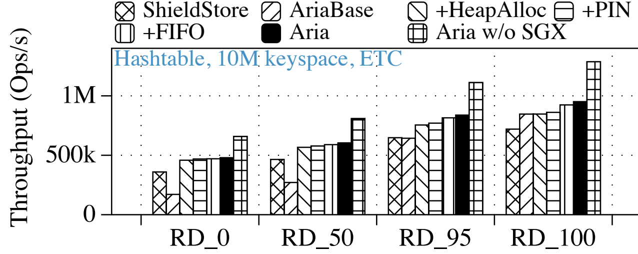

Fig. 12. Effects of different optimizations and the overhead of SGX.

300 bytes). Out of the keyspace (10 million), 40% KV pairs correspond to tiny pairs, 55% KV pairs correspond to small pairs, and the remaining 5% to large ones. We use zipfian distribution (skewness of 0.99) on the tiny and small KV pairs. Large items are chosen uniformly at random. We consider four read ratios (0%*Get*, 50%*Get*, 95%*Get*, and 100%*Get*). The evaluation results are shown in Figure 11 and we have the following observations:

First, Aria performs better than all other schemes with both hash table-based and B-tree-based index under all read ratios. Specifically, Aria improves performance by 32% compared to ShieldStore on average. *Secure Cache* provides performance advantages under production workloads since lots of counter access hit in it. Second, Aria w/o Cache performs better than ShieldStore under 0% read ratio. For every put request, ShieldStore not only needs to compute and compare MACs from the bucket (leaves of the MT) to the root, it also needs to update the root from the bucket, incurring extra overhead. While for Aria w/o Cache, the hotness-aware of the hardware secure paging brings performance advantage. As read ratio increases, the MT root updating overhead is reduced and ShieldStore shows better performance than Aria w/o Cache.

C. Effects of Optimizations and The Overhead of SGX

This section presents the effectiveness of different optimizations with *Secure Cache* in Aria using ETC workloads. LRU is the default cache replacement policy for *Secure Cache*. We test the following schemes: *AriaBase* is the proposed KV store without any optimizations. *+HeapAlloc* uses user-space heap allocation. *+PIN* utilizes level-pinning. *+FIFO* removes the level-pinning optimization but adopts FIFO. *Aria* adopts all optimizations. Except for *AriaBase*, all other schemes adopt user-space heap allocation. Figure 12 shows the comparison and we make the following observations:

First, OCalls significantly decrease the performance. Since it may incur one OCall for a write request to allocate untrusted memory for the newly inserted KV item, AriaBase decrease the performance by 62.7% compared with +HeapAlloc with 0% read ratio. Since AriaBase won't generate allocation operations for 100% read ratio workload, it shows the same performance as +HeapAlloc. Second, level-pinning and FIFO work under various read ratio. Thus Aria improves the performance by 5%, 7%, 11%, and 13% under 0%, 50%, 95%, and 100% read ratios compared with +HeapAlloc. Third, LRU is not suitable for *Secure Cache* since Aria with FIFO shows higher performance than that with LRU (i.e., +HeapAlloc). This is because LRU includes lots of memory operations when updating the LRU information, which incurs high overhead when *Secure Cache* is large. Fourth, Aria reduces the performance by 25.7% compared to Aria w/o SGX on average. Since Aria eliminates secure paging and OCalls, the performance drop mainly comes from the protection overhead of SGX when data are transferred between the EPC and the LLC.

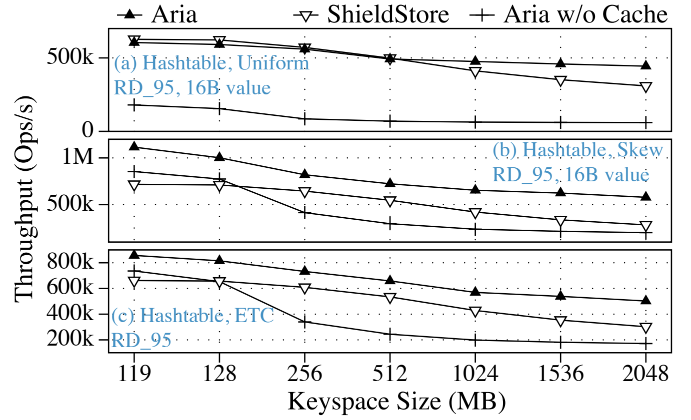

Fig. 13. Performance on various keyspace size. The number of keys ranges from 7766016 (119MB) to 134217728 (2GB). Each key is 16-byte.

D. Sensitivity Test

1) Working Set Size: Figure 13 presents the throughput comparison of Aria, ShieldStore, and Aria w/o Cache with various keyspace size from 119 MB to 2 GB and we make the following observations:

First, with increasing keyspace, the throughput of all design schemes decreases. However, Aria is less affected by the large keyspace compared to the other two design schemes. For skewed and ETC workloads, Aria benefits from *Secure Cache* since it absorbs lots of MT verification. That is, if a counter of a requested KV item exists in *Secure Cache*, we directly use it for encryption/decryption and eliminate MT verification process. ShieldStore has to conduct MT verification for every KV request. Worse, the length of the hash bucket increases as the keyspace increases, which amplifies the read and verification cost for each KV operation in ShieldStore. Therefore, Aria improves performance by 104% under the skewed workload and 67% under the ETC workload at 2 GB keyspace size compared to ShieldStore. For uniform workload, though Aria stops swap, it still shows better performance than ShieldStore when the key space size is larger than 256 MB. This is because the overhead for one verification is

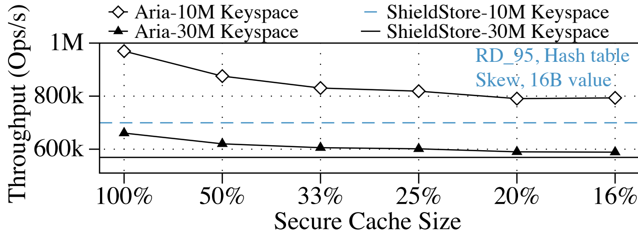

Fig. 14. Performance on different size of *Secure Cache*. 100% means using as much EPC for *Secure Cache* as possible.

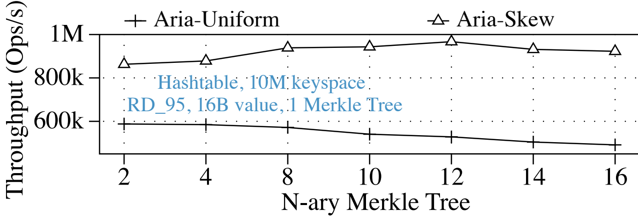

Fig. 15. Performance on different branch number of the MT.

fixed in Aria due to its fixed MT nodes in continuous MT layout and level-pinning mechanism. While for ShieldStore, since the number of buckets is fixed, larger keyspace leads to a longer bucket list, thus increasing the MT verification overhead. Hence, Aria improves performance by 44% at 2 GB keyspace over ShieldStore. Second, Aria w/o Cache shows higher performance than ShieldStore when the keyspace size is smaller than 128 MB and shows worse performance at larger keyspace size. Because with larger keyspace, the side effect of secure paging overwhelms the benefit of hotness-aware hardware secure paging.

2) Secure Cache Size: Figure 14 shows the performance of Aria-H with various *Secure Cache* size using skewed workloads under 10 million and 30 million keyspace. 100% means using as much EPC for *Secure Cache* as possible. We make the following observations:

The throughput of Aria-H drops when the *Secure Cache* size decreases. Because smaller size can contain fewer security metadata and reduce the hit ratio. However, the downward trend of the throughput is gradually flattening, and the throughput only reduces about 9% and 18% when the *Secure Cache* size reduces to 50% (45 MB EPC) and 16% (15 MB EPC) respectively with 10 million keyspace. Specifically, Aria with only 15 MB EPC occupation for *Secure Cache* shows higher performance than ShieldStore which occupies fixed 64 MB for storing the MT roots. This means that though with limited EPC resources, Aria is still effective.

3) N-ary MT: In this experiment, we use both uniform (Aria-U) and skewed (Aria-S) workloads (95% read, 16-byte value size, and one MT). Figure 15 shows the throughput comparison of Aria-H for different branch number of tree and we make following observations:

First, with increasing branch number, the performance of Aria rises under skew workloads. A MT tree node gets bigger with a larger branch number. Since the capacity of the *Secure Cache* is limited to 91 MB, we need to reduce the size of metadata of the *Secure Cache* as much as possible. Bigger tree nodes make the space utilization rate

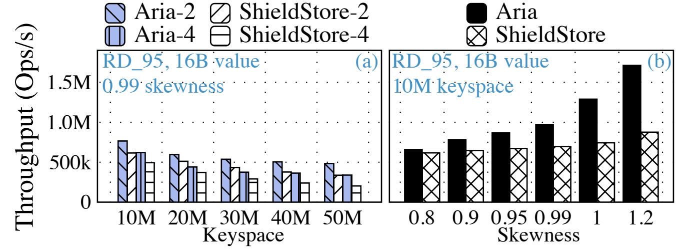

Fig. 16. (a) Multi-tenant results (2 tenants and 4 tenants) with different keyspace and (b) performance on different skewness.

$(cached\ data\ size/cache\ metadata\ size)$ of the *Secure Cache* higher. Thus it can cache more MT nodes, increasing the cache hit ratio. Second, when the branch number of the MT gets too large, the performance decreases because a larger branch number means longer input length for the MAC computation function, and the overhead of computing MAC increases. Besides, bigger tree nodes will incur more memory copy overhead in Aria because when a node is not cached, it has to be moved from the untrusted memory to the EPC first before verifying it. Third, under uniform workloads, since Aria stops swap, the verification overhead increases with the increasing size of one MT node. Thus Aria-U's performance decreases with the increasing degree of the MT.

4) Memory Consumption Analysis: We analyze the memory usage by giving the size of additional memory needed for each newly inserted KV item. Considering the security metadata, we maintain a 16-byte counter, a 16-byte MAC, and an 8-byte RedPtr for each KV item. Ten million keyspace means there are 10M different keys in the KVS, leading to about 152 MB counters. The size of each tree layer is presented in Section IV-E (Level-pinning). The whole memory usage of the MT built above the counters is about 385 MB for 10M keyspace. Considering the index metadata, Aria-H uses a 4-byte hash value for the key, 2-byte value length and a pointer for each KV item and one tree node of Aria-T consists of a 2-byte length and a pointer pointing to the child. Considering the allocator metadata, one bit in the bitmap and a 16-byte free list entry are required for each KV item.

5) Multi-tenant Test: Considering the data protection in the cloud scenario where multiple tenants share the platform resources, we present the effects caused by the parallel use of SGX resources. In the cloud environment, multiple requests may belong to different users. From the security perspective, it can offer better isolation with separated memory address space. Therefore, we use multiple separated enclaves based on the multi-process design. We reduce the EPC occupation by reducing the *Secure Cache* size for Aria and the number of MT roots for ShieldStore as the number of tenants increases, eliminating secure paging. Figure 16 (a) shows the average throughput with the different number of KVS instances each of which runs an individual Aria/ShieldStore. We make the following observation: Aria is less sensitive to the number of tenants compared to ShieldStore. The experiments show that the performance gap between Aria and ShieldStore becomes larger as the number of tenants and keyspace increase. Aria

outperforms ShieldStore by 24% and 26% with 2 and 4 tenants respectively under 10M keyspace and 44% and 67% under 50M keyspace.

6) Skewness: Figure 16 (b) shows the effect of different skewness in the YCSB skewed distribution. Since recent work has shown that some real workloads exhibit unprecedented skew levels (e.g., Zipf distributions with skewness $\geq$ 1) [23], [24]. We also test the skewness of 1.2. Since a more skewed distribution increases the hit ratio of *Secure Cache*, the performance improvement of Aria over ShieldStore increases as the skewness increases and reaches 96% at 1.2 skewness.

VII. Discussion

Security Vulnerabilities. Aria ensures confidentiality via cryptography operations and integrity through MT verification for KV pairs. Yet, with the encrypted data stored outside the enclaves, malicious adversaries can get the operation types, key access frequencies, hashed-key distributions for Aria-H, and the size relationship between two encrypted keys for Aria-T. We consider these security flaws side-effects of trading-off between security and performance. How to resolve these security vulnerabilities while remaining high performance is our future work. Besides, Aria store pointers used for index connection in untrusted memory. Although untrusted pointers may permit corruption and thus compromise the availability of Aria, we still ensure the integrity and confidentiality of the data. The above vulnerabilities also exist in ShieldStore.

Supporting for B+-tree-based Index. We only implement B-tree index in Aria. However, Aria can also support B+-tree-based index by encrypting key and value respectively. We leave it our future work to incorporate B+-tree into Aria.

VIII. Conclusion

In this paper, we present Aria, a secure in-memory KV store for untrusted hosts. Aria targets 1) guaranteeing the confidentiality and integrity of KV pairs, 2) supporting various index schemes as well as providing efficient query operations. We base the design of Aria on hardware-assisted shielded execution leveraging Intel SGX. To achieve these properties while overcoming the architectural limitations of SGX, we propose a software-based semantic-aware swap mechanism inside the enclave, which is called *Secure Cache*. We implement Aria based on hash table and B-tree indexes and evaluate it using YCSB and ETC workloads. Our experimental evaluation shows that the design of Aria is effective under real-world workloads and tolerates large keyspace and limited EPC size.

References

[1] R. Nishtala *et al.*, "Scaling memcache at facebook," in *USENIX NSDI*, 2013.

[2] B. Atikoglu *et al.*, "Workload analysis of a large-scale key-value store," in *ACM SIGMETRICS*, 2012.

[3] N. Santos *et al.*, "Towards trusted cloud computing," in *Proceedings of the 2009 Conference on Hot Topics in Cloud Computing*, USA, 2009.

[4] L. Bouganim *et al.*, "Chip-secured data access: Confidential data on untrusted servers," in *VLDB*, 2002.

[5] A. Arasu *et al.*, "Secure database-as-a-service with cipherbase," in *ACM SIGMOD*, 2013.

[6] H. Hacigümüş *et al.*, "Executing sql over encrypted data in the database-service-provider model," in *ACM SIGMOD*, 2002.

[7] A. Baumann, M. Peinado, and G. Hunt, "Shielding applications from an untrusted cloud with haven," in *USENIX OSDI*, 2014.

[8] H. S. Gunawi *et al.*, "What bugs live in the cloud? a study of 3000+ issues in cloud systems," in *SOCC '14*, 2014.

[9] S. Tu, M. F. Kaashoek, S. Madden, and N. Zeldovich, "Processing analytical queries over encrypted data," *Proc. VLDB Endow.*, 2013.

[10] S. Bajaj and R. Sion, "Trusteddb: A trusted hardware-based database with privacy and data confidentiality," *IEEE TKDE*, 2013.

[11] Y. Tang *et al.*, "Outsourcing multi-version key-value stores with verifiable data freshness," in *IEEE ICDE*, 2014.

[12] "Microsoft azure confidential computing," https://azure.microsoft.com/en-us/solutions/confidential-compute/, 2020.

[13] "Ibm cloud data shield," https://www.ibm.com/cloud/data-shield, 2020.

[14] M. Orenbach *et al.*, "Eleos: Exitless os services for sgx enclaves," in *ACM Eurosys*, 2017.

[15] T. Kim *et al.*, "Shieldstore: Shielded in-memory key-value storage with sgx," in *ACM Eurosys*, 2019.

[16] M. Bailleu *et al.*, "SPEICHER: Securing lsm-based key-value stores using shielded execution," in *USENIX FAST*, 2019.

[17] C. Priebe, K. Vaswani, and M. Costa, "Enclavedb: A secure database using sgx," in *IEEE Symposium on Security and Privacy (SP)*, 2018.

[18] L. Chen *et al.*, "Enclavecache: A secure and scalable key-value cache in multi-tenant clouds using intel sgx," in *Middleware Conference*, 2019.

[19] S. Arnautov *et al.*, "{SCONE}: Secure linux containers with intel {SGX}," in *USENIX OSDI*, 2016.

[20] B. Gassend *et al.*, "Caches and hash trees for efficient memory integrity verification," in *IEEE HPCA.*, 2003.

[21] Z. Cao *et al.*, "Characterizing, modeling, and benchmarking rocksdb key-value workloads at facebook," in *USENIX FAST*, 2020.

[22] B. F. Cooper *et al.*, "Benchmarking cloud serving systems with ycsb," in *ACM SoCC*, 2010.

[23] J. Chen *et al.*, "Hotring: A hotspot-aware in-memory key-value store," in *USENIX FAST'20*.

[24] J. Yang *et al.*, "A large scale analysis of hundreds of in-memory cache clusters at twitter," in *USENIX OSDI'20*.

[25] O. Eytan *et al.*, "It's time to revisit LRU vs. FIFO," in *12th USENIX Workshop on Hot Topics in Storage and File Systems*, Jul. 2020.

[26] O. Weisse *et al.*, "Regaining lost cycles with hotcalls: A fast interface for sgx secure enclaves," in *ACM/IEEE ISCA*, 2017.

[27] V. Costan and S. Devadas, "Intel sgx explained," *IACR Cryptology ePrint Archive*, vol. 2016, p. 86, 2016.

[28] S. Gueron, "A memory encryption engine suitable for general purpose processors," *IACR Cryptology ePrint Archive*, vol. 2016, p. 204, 2016.

[29] M. Taassori *et al.*, "Vault: Reducing paging overheads in sgx with efficient integrity verification structures," *ACM SIGPLAN Notices*, 2018.

[30] G. Saileshwar *et al.*, "Morphable counters: Enabling compact integrity trees for low-overhead secure memories," in *IEEE/ACM Micro*, 2018.

[31] O. Weisse *et al.*, "Foreshadow-ng: Breaking the virtual memory abstraction with transient out-of-order execution," 2018.

[32] J. A. Halderman *et al.*, "Lest we remember: Cold-boot attacks on encryption keys," *Commun. ACM*, vol. 52, no. 5, pp. 91–98, May 2009.

[33] P. Pessl *et al.*, "DRAMA: Exploiting DRAM addressing for cross-cpu attacks," in *USENIX Security*, 2016.

[34] "Intel software guard extensions sdk for linux os," https://download.01.org/intel-sgx/linux-2.6/docs/Intel_SGX_Developer_Reference_Linux_2.6_Open_Source.pdf, 2019.

[35] Y. Xu *et al.*, "Controlled-channel attacks: Deterministic side channels for untrusted operating systems," in *IEEE SP*, 2015.

[36] O. Oleksenko *et al.*, "Varys: Protecting SGX enclaves from practical side-channel attacks," in *USENIX ATC*, 2018.

[37] S. Eskandarian *et al.*, "Oblidb: Oblivious query processing for secure databases," *VLDB*, 2019.

[38] "Remote attestation," https://software.intel.com/content/www/us/en/develop/topics/software-guard-extensions/attestation-services.html, 2020.

[39] B. Rogers *et al.*, "Using address independent seed encryption and bonsai merkle trees to make secure processors os- and performance-friendly," in *IEEE/ACM MICRO*, 2007.

[40] F. Yang *et al.*, "No compromises: Secure nvm with crash consistency, write-efficiency and high-performance," in *DAC*, 2019.

[41] B. Berg *et al.*, "The cachelib caching engine: Design and experiences at scale," in *USENIX OSDI*, 2020.

[42] P. Bodik *et al.*, "Characterizing, modeling, and generating workload spikes for stateful services," in *ACM SoCC*, 2010.