InvisiPage: Oblivious Demand Paging for Secure Enclaves

Shaizeen Aga[*]
University of Michigan, Ann Arbor
shaizeen@umich.edu

Satish Narayanasamy
University of Michigan, Ann Arbor
nsatish@umich.edu

ABSTRACT

State-of-art secure processors like Intel SGX remain susceptible to leaking page-level address trace of an application via the page fault channel in which a malicious OS induces spurious page faults and deduces application's secrets from it. Prior works which fix this vulnerability do not provision for OS demand paging to be oblivious. In this work, we present InvisiPage which obfuscates page fault channel while simultaneously making OS demand paging oblivious. To do so, InvisiPage first carefully distributes page management actions between the application and the OS. Second, InvisiPage secures application's page management interactions with the OS using a novel construct which is derived from Oblivious RAM (ORAM) but is customized for page management. Finally, we lower overheads of our approach by reducing page management interactions with the OS via a novel memory partition. For a suite of cloud applications which process sensitive data we show that page fault channel can be tackled while enabling oblivious demand paging at low overheads.

CCS CONCEPTS

• **Security and privacy → Hardware-based security protocols**.

KEYWORDS

Intel SGX, ORAM, Page Fault Channel

ACM Reference Format:
Shaizeen Aga and Satish Narayanasamy. 2019. InvisiPage: Oblivious Demand Paging for Secure Enclaves. In *The 46th Annual International Symposium on Computer Architecture (ISCA '19), June 22–26, 2019, Phoenix, AZ, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3307650.3322265

1 INTRODUCTION

Our reliance on cloud computing only increases with time for it has myriad benefits to offer. Consequently, more and more sensitive data is being pushed to the cloud, thus bringing forth the security concerns users feel for their data. Preserving privacy of client's data in a cloud computing setup continues to be an open challenge for such a setup necessitates the assumption of a powerful adversary who could have full control over the system software and be even capable of launching physical attacks on machines in the data center.

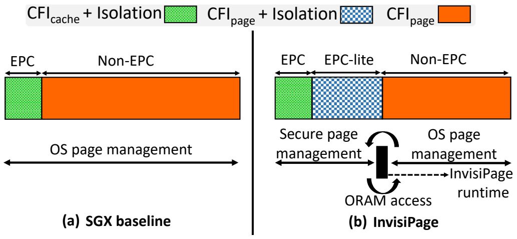


Figure 1: Memory organization under SGX (a) and under InvisiPage (b). CFI: Confidentiality, Freshness, Integrity

Intel Software Guard Extensions (SGX) [8, 26] is the latest commercially available offering which aims to answer this demand for privacy preserving remote computations. With SGX, a cloud user can designate parts of his application as private or sensitive (termed `enclave`) and the SGX-enabled processor will protect code and data of the enclave from the rest of the system, including the application's public functions, system software, and hardware peripherals. To provide this protection, SGX provisions checks which ensure that any sensitive code/data pages of an enclave *are kept confidential and are only read/written by the owning enclave.*

Memory under SGX is partitioned into two regions *EPC* (Enclave Page Cache) memory and *non-EPC* memory (Figure 1). Sensitive pages of an enclave are allocated in *EPC*, they can be spilled to *non-EPC* memory and then fetched back in *EPC* on an access by the enclave. SGX provides the core guarantees of `Confidentiality` (hide information in data - encryption), `Freshness` (read returns latest written value - nonces) and `Integrity` (prevent data corruption - MAC tag) for enclave's sensitive data both in *EPC* and *non-EPC* memory albeit at cache and page granularities respectively. In fact, provisioning these guarantees at finer granularity (cache block) is what limits *EPC* size (128MB in current SGX processors) for the performance and space overheads of these security guarantees increase as the memory size increases. Additionally, for data in *EPC*, SGX also provides `Isolation` (only owning enclave can issue reads/writes) by tracking ownership metadata for each *EPC* page.

While the security guarantees provided by SGX for sensitive pages of an enclave are strong, SGX leaves page management entirely to the OS. The OS can allocate pages in *EPC* to enclaves, de-allocate them at will and spill them to *non-EPC* memory. Furthermore, address translations (virtual to physical address mappings) are also under OS's purview. As such, OS handles page faults, accesses and updates page tables for enclaves.

As SGX leaves page management to the OS, a malicious OS can simply revoke page permissions to induce spurious page faults during an enclave's execution. Under SGX, the OS can also induce

[*] The author is currently at AMD Research.

ACM ISBN 978-1-4503-6669-4/19/06…$15.00
https://doi.org/10.1145/3307650.3322265

spurious page movements between *EPC* and *non-EPC* memory. Using either of these mechanisms, the OS can learn the address trace of an enclave. This vulnerability, termed page fault channel [46], can be used to recover sensitive inputs of an application [46]. Specifically, prior work [46] showed how the sensitive input image to an enclave can be completely recovered. This can have catastrophic consequences say for a medical image processing cloud application.

Current solutions to fix page fault channel either propose determinizing page access patterns [37] or require that an enclave's memory requirement be pre-determined and reserved [16, 36]. Both of these techniques are hard to realize for general programs without severely constraining them (no dynamic memory allocations, no recursion etc.). Further, reserving large swathes of memory for a single enclave precludes other enclaves from using the same machine and also disallows the OS from flexibly managing memory as a shared resource. Additionally, these proposals do not support oblivious demand paging: any access to a page de-allocated by the OS will leak the accessed address to the OS [16].

In this paper, we propose InvisiPage, a page fault channel defense in which unlike prior solutions we allow the OS to perform demand paging (allocate/deallocate pages) during an enclave's execution while preventing it from learning the address trace of the enclave. To do so, InvisiPage first provisions for collaborative page management wherein it carefully distributes page management actions between the enclave and the OS. Second, InvisiPage secures enclave's page management transactions with the OS via a novel construct we term *Oblivious Page Management* (OPAM). OPAM is based on Oblivious RAM (ORAM) [20], but is customized for page management context. As OPAM transactions are costly, we reduce their number by creating a new memory partition termed *EPC-lite* (Figure 1) which has similar guarantees to *EPC* but does not incur the overheads of actually increasing *EPC* size.

Collaborative page management: Unlike baseline SGX, under InvisiPage the OS does not handle all page management actions but shares them with the enclave. InvisiPage decouples memory resource management from its associated metadata management (page tables). While it leaves the former to the OS, the enclave securely manages/updates page table data for it's sensitive pages. Further, while OS still retains complete control over page allocations to both *EPC* and *non-EPC* memory, under InvisiPage, *EPC* deallocations and page movements between *EPC* and *non-EPC* memory are performed by the enclave in collaboration with the OS. Overall, this completely hides any *EPC* accesses by the enclave from the OS.

OPAM for securing interactions with OS: Under both baseline SGX and InvisiPage, the OS retains the ability to deallocate *non-EPC* memory pages and swap them to a backing store. A malicious OS can use this to revoke permissions on *non-EPC* memory pages and learn the address trace of an enclave. To prevent this, InvisiPage obfuscates addresses of pages moved between *EPC* and *non-EPC* memory via ORAM construct.

ORAM [20] is a cryptographic primitive which makes a memory access pattern computationally indistinguishable from a random access pattern of same length. We customize the traditional ORAM implementation for page management context and term the resultant implementation oblivious page management (OPAM). InvisiPage addresses several challenges that arise in using ORAM for page management. Unlike traditional ORAM implementations for secure processors, the memory size covered by the ORAM dynamically changes for our context. InvisiPage addresses this challenge efficiently while also avoiding frequent metadata updates as memory size grows. Further, InvisiPage exploits opportunities unique to using ORAM for page management. InvisiPage encodes metadata needed to support the ORAM algorithm inside existing page tables and saves performance and space overheads. It also decouples metadata and data updates to reduce the overheads of ORAM algorithm.

***EPC-lite* to reduce OPAM transactions**: While OPAM considerably reduces baseline ORAM algorithm overheads, each OPAM transaction is still costly. As these transactions are required only while accessing *non-EPC* memory, we could reduce their number by increasing *EPC*. Increasing *EPC* size, however, is challenging as SGX provides security guarantees for *EPC* pages at cache block granularity. As such, any increase in *EPC* size will incur additional performance and space overheads for maintaining and accessing the metadata needed for providing these guarantees. Instead, InvisiPage extends *EPC* without incurring metadata overheads by devising a memory partition which has all the security properties of *EPC* except at page-level. We term this novel memory partition as *EPC-lite*. InvisiPage can move pages between *EPC-lite* and *EPC* without needing OPAM transactions. We identify challenges in supporting *EPC-lite* region and propose simple solutions to address these challenges.

We model a suite of cloud applications: genome processing, vision applications, graph processing, and in-memory key value store which frequently process sensitive data including but not limited to medical images, genome sequences and social graphs. We demonstrate how page fault channel can be fixed for these applications at low overheads while enabling oblivious demand paging.

This paper makes the following contributions:

- We propose a novel page fault channel defense InvisiPage which unlike prior works provisions for oblivious demand paging by the OS.
- InvisiPage carefully distributes page management actions between the enclave and the OS to allow the OS flexibility in managing memory as a resource yet hides application's address trace from it.
- To secure its page management transactions with the OS, InvisiPage uses a novel construct, Oblivious Page Management (OPAM) which is derived from ORAM but is customized for addressing challenges and exploiting opportunities that arise in the context of page management.
- We also propose a novel memory partition *EPC-lite* which reduces the OPAM transactions needed and brings down the overheads of our proposed solution further.
- For a suite of cloud applications we show that page fault channel can be mitigated while enabling oblivious demand paging at low overheads.

2 MOTIVATION AND BACKGROUND

2.1 Threat Model

We assume a secure processor with support for isolated execution like Intel SGX. Our attack model assumes a powerful adversary with full control over the operating system. We assume that the

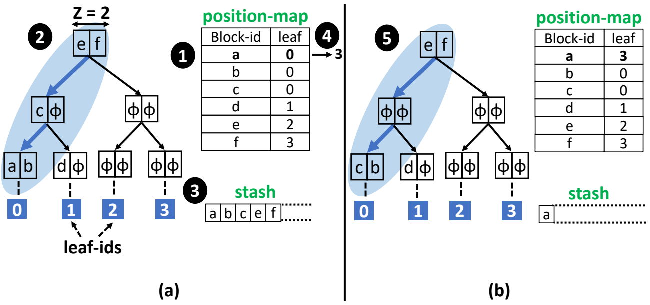


Figure 2: Accessing `block a` **involves read and write of path to leaf to which the block is mapped. (a) Depicts stash after end of path read. (b) Depicts stash after end of path write. Block with dummy data is represented as ϕ.**

entire application is bundled as an enclave (along with necessary libraries) and its interactions with external world are made secure i.e. the enclave is protected against attacks like Iago attacks [13].

An Intel SGX like system that we assume is susceptible to leaking the address trace of an application via varied side channels like cache [47], memory bus [6] and page faults [46]. While both cache and memory bus side channel attacks are important vulnerabilities, in this work, we focus on designing an efficient solution to address the page fault channel. We believe that there are orthogonal solutions [6, 16] which can be combined with our proposed solution to address these other vulnerabilities.

We consider side channels like power [23], thermal [29], program execution time [48] outside the scope of our work. They can be mitigated by using prior techniques [48] in tandem with our approach.

2.2 Dual Page Table Support

In this section, we discuss dual page table support presented in Sanctum [16] that we also assume and augment in this work. The key benefit of this support is to ensure that page faults (and TLB misses) to public or non-sensitive regions of an application do not incur any additional overheads in a system which obfuscates page fault side channel as compared to an unsecure baseline. This is realized by harnessing the fact that under SGX and other secure designs [19], the programmer explicitly demarcates a range of virtual addresses of an application as being private or sensitive and the rest as public or non-sensitive. Such demarcation aids in reducing overheads of SGX security guarantees. An example of data that could be placed in public partition is an array which is read serially at the beginning of program execution. Page faults to such data does not leak any sensitive information and hence could be left under the purview of the OS.

So as avoid incurring overheads for public pages, Sanctum [16] provisions for dual page tables: page table data for private pages is stored securely (in *EPC* in SGX parlance) and is only manipulated by the enclave, while a second, conventional page table is used for public pages which is under the control of OS. In such a system, TLB misses need to be appropriately directed to the right page table which is realized via a dual page table walk mechanism. This requires an additional page table base register which points to the physical address of enclave's private page table. Such support has been extensively studied by prior work [16] and can be adopted in our system. While Sanctum employs a security monitor which sets the page table base register appropriately, we assume that SGX is augmented to perform this function.

Note that such dual page table support alone is not enough to mitigate page fault channel attacks while not robbing the OS of its flexibility in managing memory. This is so as Sanctum does not support oblivious on-demand paging. While Sanctum does not allow the OS to forcibly reclaim a page from the enclave, once reclaimed, the OS can learn of any further accesses to this page by the enclave. InvisiPage on the other hand allows an enclave to access pages reclaimed by the OS without leaking information.

2.3 Path Oblivious RAM

Oblivious RAM (ORAM) [20] is a cryptographic construct which makes a memory access trace computationally indistinguishable from a random access trace of same length. Our work employs path-ORAM [38] which is the most practical implementation of this construct. In this section we explain the workings of path-ORAM algorithm (henceforth referred to simply as ORAM).

ORAM organizes memory as a binary tree and each node in the tree has fixed number of slots (`z`) each capable of storing a single data block. The tree also has associated `utilization` factor which indicates percentage of real blocks that can be stored in the tree; remaining blocks hold dummy data. The algorithm maps each real block to a leaf in the tree and these mappings are maintained in the `position-map` structure. All the blocks (real and dummy) are stored in encrypted form in memory. The tree also stores metadata for every block which includes its block-id (null for dummy blocks), its leaf and encryption related counter.

Figure 2 depicts steps involved while reading `block a`. The algorithm first looks up the `position-map` to find the leaf the block is mapped to (❶). The ORAM invariant is that `block a` will either be in the `stash` structure the algorithm maintains or in some slot along the path to its mapped leaf (`path-0`). Next, it accesses `path-0` and decrypts all blocks along the path (❷), storing only real blocks in the `stash` structure (❸). At this point, `block a` is read and remapped to a random leaf (❹). Next, as many blocks as possible are encrypted and written back to `path-0` and rest of the slots are filled with dummy blocks (❺). During path write, data blocks are pushed as close to leaf nodes as possible (`block c` moves to leaf node). Notice that `block a` is left behind in the `stash` for it is now mapped to leaf 3 and the only common node between `path-0` and `path-3` is the root which is currently full. Write operations proceed similarly except that blocks are also updated.

ORAM's security relies on two actions. First, mapping of blocks to random leafs on each access causes new set of blocks to be read each time a block is accessed. Second, each access causes re-encryption of all blocks accessed which makes it hard for an adversary to differentiate between real and dummy blocks and deduce which block was actually accessed.

Note that ORAM is susceptible to failures. Most secure processor implementations provision `stash` as an on-chip structure and a stash overflow due accumulation of blocks causes ORAM failure. So as to reduce this probability, on every ORAM access, data blocks are pushed as close to leaf nodes as possible to free up higher level nodes which can store blocks mapped to larger set of leaves.

3 INVISIPAGE DESIGN

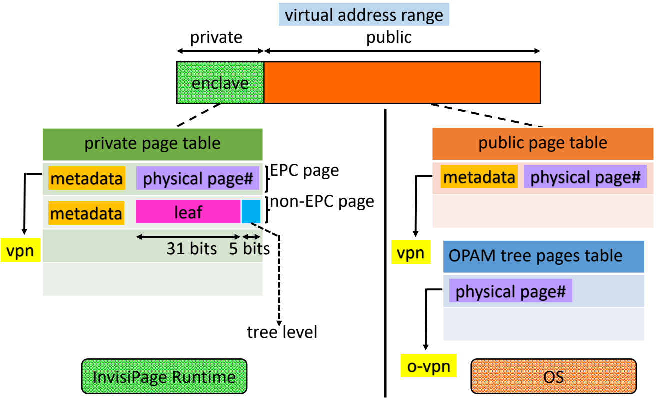


Figure 3: Page tables in InvisiPage.

3.1 InvisiPage Overview

We present an overview of InvisiPage in this section. The key design goal of InvisiPage is to prevent the OS from using page faults to learn an enclave's memory access trace while preserving the control the OS currently enjoys in allocating/deallocating memory to applications. To realize this goal, we observe that it suffices to carefully distribute page management actions between the enclave and the OS. InvisiPage provisions a careful division of page management which ensures that while the OS can control the *number* of pages allocated to a given enclave, it is unaware of which pages the enclave accesses or *what virtual address* a given physical page is mapped to. InvisiPage augments SGX's trusted runtime [45] to collaboratively perform page management actions with the OS (Section 3.2).

To ensure that the OS enjoys flexibility in managing memory as a resource, under InvisiPage, OS retains the ability to reclaim an *EPC* page from an enclave in collaboration with InvisiPage runtime. Further, the OS can also deallocate *non-EPC* memory pages and swap them to a backing store at will. To prevent the OS from learning of an enclave's address trace via its accesses to reclaimed *EPC* pages, InvisiPage obfuscates addresses of pages moved between *EPC* and *non-EPC* memory via ORAM construct. We adapt the ORAM construct for page management, identify and address challenges and exploit unique opportunities in doing so. We term this customized ORAM implementation (Section 3.3) as oblivious page management (OPAM).

While OPAM considerably reduces baseline ORAM algorithm overheads, each OPAM transaction is still costly. To further reduce the overheads of our proposed approach, we design a novel memory partition, *EPC-lite* (Section 3.4) which reduces the number of OPAM transactions necessary. We identify challenges in realizing *EPC-lite* and address them efficiently.

3.2 InvisiPage Runtime

We extend the trusted runtime system [45] of an enclave to take over some page management actions from the OS and term this augmented runtime as InvisiPage runtime. To this end, under InvisiPage, we first decouple resource management (allocation, deallocation of pages) from its associated metadata (virtual to physical address translations in page table). To realize this, enclave's sensitive pages are tracked in a private page table by the InvisiPage runtime and this table is isolated from the OS (Figure 3). A TLB miss to a sensitive page is directed to this private page table. Note that, non-sensitive pages are still managed by the OS (public page table in Figure 3) as is the case with prior dual page table designs (Section 3.3).

Function	Operation
OS Interface	
a. get_epc_page()	Allocate a free *EPC* page.
b. get_non-epc_pages (n)	Allocate n free *non-EPC* memory pages.
c. opam_access (o-vpn[])	Keep *non-EPC* memory pages corresponding to list of o-vpns provided memory resident.
d. get_epc-lite_page()	Allocate a free *EPC-lite* page.
InvisiPage Runtime Interface	
e. free_epc_page()	Free an *EPC* page.
f. free_epc-lite_page()	Free an *EPC-lite* page.

Table 1: Interface between InvisiPage runtime and the OS

The interface of InvisiPage runtime with the OS is depicted in Table 1. The runtime interacts with the OS each time a sensitive page is allocated in *EPC* (Table 1, a). Unlike in the baseline SGX where the OS picks a victim *EPC* page while freeing *EPC* memory, under InvisiPage the runtime is responsible for picking a victim *EPC* page. As such, the OS interfaces with the runtime in order to deallocate an *EPC* page (Table 1, e). As *EPC* is limited, an enclave's sensitive pages can be spilled to *non-EPC* memory and such pages can later be re-accessed by the enclave and consequently fetched back into *EPC*. The runtime interfaces with the OS on such spills and fetches (Table 1, c) and further uses our OPAM construct (Section 3.3) to secure such transactions and hide the page address from the OS. Finally, the runtime also interfaces with the OS when it needs more *non-EPC* memory pages (Table 1, b).

Enclave's sensitive pages which are spilled to *non-EPC* memory are managed by the runtime via our OPAM construct. Recall from Section 2.3 that the underlying ORAM construct arranges such pages as a logical tree. The OS tracks such *non-EPC* memory pages separately in a new structure as depicted in Figure 3. Unlike the private page table and the public page table of an enclave, which are indexed by the virtual page number (vpn), this new structure is indexed by a novel number we construct and term oblivious virtual page number (o-vpn). The o-vpn is a unique-id which is statically assigned to each slot in the underlying ORAM tree (Section 2.3). For each spill to *non-EPC* memory and fetch from *non-EPC* memory, the runtime specifies to the OS a list of o-vpns to be accessed (corresponding to a path in the tree) and the OS ensures that all *non-EPC* memory pages corresponding to the provided list of o-vpns are memory resident for the duration of the spill/fetch. We provision mechanisms for the runtime to ensure this is the case (Section 4.1.1). Note that beside this, the OS has complete control over these *non-EPC* memory pages and it can swap them to backing store at will.

Under InvisiPage as in unsecure baseline, memory management (allocation, deallocation of pages) is transparent to the enclave and as such the enclave does not need to be modified. InvisiPage runtime interfaces with the OS to ensure memory management flexibility is maintained while not leaking memory access pattern of the enclave.

3.3 Oblivious Page Management

The limited amount of *EPC* space causes enclave's sensitive data to be spilled to *non-EPC* memory. InvisiPage runtime converts each access by an enclave to a *non-EPC* memory page into an ORAM access so as to hide the page address from the OS (Table 1, c). Recall from Section 2.3, that ORAM organizes memory as a binary tree and each ORAM access involves reading and writing all the blocks (pages) belonging to a certain path in the tree. Similarly, under InvisiPage, each ORAM access involves reading and writing *non-EPC* memory pages which belong to a given tree path. Section 3.3.1 discusses how these pages are identified. Next, we discuss how we customize the traditional ORAM construct (Section 2.3) to be more suitable for page management context. We term our customized construct Oblivious Page Management (OPAM).

3.3.1 Page table as Position Map. As in ORAM algorithm, InvisiPage organizes *non-EPC* memory pages as a binary tree. The ORAM algorithm provisions a `position map` which stores page address-leaf mappings which helps locate a page in the tree. This structure is both consulted and updated on each ORAM access and is one of the chief contributors to ORAM's performance and space overhead. InvisiPage does away with this overhead by encoding `position map` inside the page table. Page tables are already consulted on a TLB miss and by placing `position map` in page table entries, we do away with the performance and space overheads of `position map`. Figure 3 depicts this encoding in an enclave's private page table; an enclave's sensitive page could either be mapped to *EPC* or be mapped to *non-EPC* memory in which case the PTE holds its associated leaf. Assuming x86-based architecture, close to 36 bits are available of which we use 31 bits to store leaf information. The InvisiPage runtime uses the leaf number to identify the path in the tree to be accessed which in turn allows it to generate the oblivious virtual page numbers (`o-vpn`, Section 3.2) that need to be accessed. The `o-vpns` are then conveyed to the OS to perform the ORAM access.

3.3.2 Dynamic ORAM. In traditional ORAM, address obfuscation is provided for accesses to fixed size memory. This is undesirable in our context as predetermining number of *non-EPC* memory pages (and hence application memory footprint) is hard without severely constraining the program. Further, allocating large *non-EPC* memory space apriori is unwise as ORAM access overhead increases with larger memory size. As such, we choose to grow the ORAM tree gradually. While such *dynamic* ORAMs have been studied theoretically [27], we offer the first practical implementation by providing efficient solutions to identified challenges. Note that, OPAM tree can also be shrunk as enclave frees memory. However, we leave this to future work and focus on growing the tree efficiently in this work.

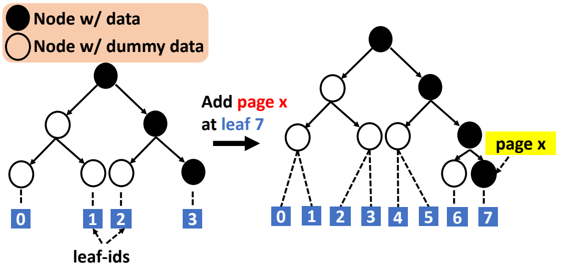


Figure 4: Smart growth. Spilling a page to a full tree (50% utilization). Naive growth adds (two) nodes gradually from left to right which will cause the page spill to fail as path to leaf 7 is full. Smart growth prioritizes adding nodes to path which is accessed. As a consequence, the spill succeeds.

Growth size: An interesting decision for dynamic ORAMs is when and by how much to grow the tree. Recall that ORAM has an associated `utilization` factor (Section 2.3) which dictates the fraction of pages in the tree that hold real data. We use this parameter to guide tree growth. At each spill to *non-EPC* memory, we add nodes (*non-EPC* memory pages, Table 1, b) such that the spill does not cause `utilization` factor to exceed (e.g. With utilization factor 50%, when adding a new page to a full tree, we will add two new pages to the tree).

Smart growth: While prior works [27] discuss gradual node addition to an ORAM tree, our work observes that *where* nodes get added can help reduce spill failures to ORAM. Figure 4 depicts a scenario wherein addition of page x to a full tree fails as naive growth (left to right node addition in the new level) fails to address the fact that path to leaf 7 is full (3 pages are already mapped to it). Instead, we take a different approach termed smart growth. Therein, whenever ORAM tree is grown, we first try to grow the path we are accessing. In the above example, this will cause nodes to be added to path to leaf 7 and addition of page x will succeed. If the path we are accessing is already grown, we fall back on naive growth. Our evaluation shows that prioritizing node addition to accessed path helps reduce spill failures considerably.

Security of smart growth: Our smart growth optimization does not leak any more information than the naive growth as it is independent of the current contents of ORAM tree. This is so, as smart growth deterministically grows either the accessed path (if not already grown) or falls back on naive growth. Hence, the adversary only learns which path is being accessed and that the tree is growing like in naive growth.

Avoiding frequent position map updates: A challenge in realizing dynamic ORAM is the overhead of updating `position map` caused by changes to the leaf a page maps to as the tree grows. The change in mapping is due to two reasons. First, as Figure 4 shows, path to leaf 3 is quite different in trees with three and four levels. Second, on each access to the tree, real pages on the accessed path are pushed as close to leaf as possible causing pages to be shuffled along the tree path. When the tree grows, two leafs (two paths) are created where one existed. As it is desirable to spread pages on both the new paths, the leaf a page maps to changes. In the extreme

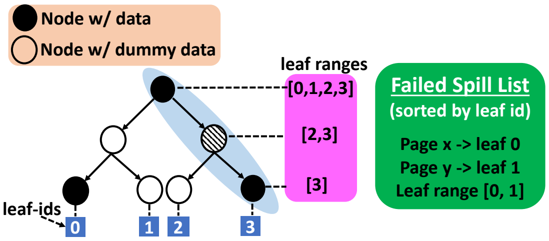


Figure 6: Fast failed spills processing. We store failed spills in sorted order (by leaf id). Figure highlights the path being accessed which has one empty node (striped) to which we can potentially spill a data block. We also show the leaf ranges that can be spilled to each node. Simple range checks of these ranges (e.g. range[2-3]) against available failed spills (range[0-1]) can help process past failed spills quickly.

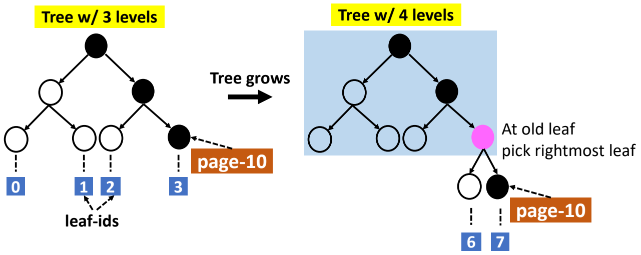


Figure 5: Finding page 10 in new tree which was mapped to leaf 3 in old tree. We employ deterministic remapping; even addresses get remapped to rightmost paths and odd addresses to leftmost paths. We also remember tree level with the mapping. To find the page, we find leaf in tree with #old levels and traverse the tree in relevant direction based on address.

case, one could update all affected entries in the `position map` on each access to the tree. As `position map` data is stored in the page table, this can lead to severe overheads for it will lead to random page table walks.

InvisiPage uses two strategies to avoid frequent updates to `position map`. First, while inserting an entry in `position map`, we store current #levels in the tree (private page table, Figure 3) along with leaf. Second, we use deterministic remapping during page shuffling: we remap even (odd) addresses to rightmost (leftmost) paths. Together these strategies help us locate a page in the (new) grown tree using its old mapping based off the (old) smaller tree obviating the need to update the `position map` when the tree grows.

Figure 5 depicts an example. In the older tree (left) with 3 levels, page 10 is mapped to leaf 3. Note that, this path in the grown tree (right) corresponds to two leafs: 6 and 7. Our deterministic page shuffling will only place page 10 (even address) along path to leaf 7 (rightmost leaf). Consequently, to find page 10 when only its mapping in the old tree is available (leaf = 3, levels = 3), we first find the node which would have mapped to leaf 3 in the tree with 3 levels. From this node, to find page 10, we pick the right-most path as 10 is an even address.

3.3.3 *Decoupling ORAM Metadata and Data to Reduce Page Copies between EPC and non-EPC memory.*

Traditional ORAM implementations employ a `stash` structure in secure space wherein pages read on each ORAM access are decrypted and stored. To complete the access, as many pages from the `stash` as possible are re-encrypted and written back to the tree. Implementing this as is in our system will lead to severe overheads as it will require allocation of `stash` in *EPC* memory and increase by several times the number of pages copied between *EPC* and *non-EPC* memory.

We use two observations to overcome this challenge. First, the net effect of an ORAM access is that one page is moved between *EPC* and *non-EPC* memory and rest of the pages are merely shuffled in *non-EPC* memory. Second, only ORAM metadata needs to be inspected to deduce page shuffle decisions. Recall that every page in ORAM tree has associated metadata (virtual address of page, leaf, level). To harness these observations, we decouple ORAM metadata and store it in a separate mirrored tree structure in *non-EPC* memory. InvisiPage runtime first accesses this metadata tree to deduce page shuffling needed. Post that, the runtime performs page shuffles in *non-EPC* memory and only moves a single page across *EPC* and *non-EPC* memory.

This decoupling requires the OS to ensure that the *non-EPC* memory pages needed for an ORAM access are resident in memory during the ORAM access. We provision mechanisms for the runtime to check that it is reading (writing) the right *non-EPC* memory pages (Section 4.1.1).

Security of Page Copy Reduction: Moving a single page between *EPC* and *non-EPC* memory and shuffling the remaining pages in *non-EPC* memory is secure as the OS does not learn the address of the moved page. Unlike in baseline SGX, the InvisiPage runtime performs page movements via OPAM construct wherein entire accessed path is written and one page is moved between *EPC* and *non-EPC* memory. The OS can merely scan *non-EPC* memory pages to *only* learn that the entire path was written by the runtime.

3.3.4 *Dataless Stash.*

With traditional ORAM, the `stash` structure holds memory pages which could not be spilled to the tree. Traditional hardware implementation stores this structure on-chip for security purposes. As such, its size limits the number of failed spills the ORAM implementation can bear. To avoid ORAM failure (Section 2.3), dummy accesses (read/write of a random ORAM path) are employed with the hope that they might spill some stash pages to the tree which further worsen ORAM overheads.

Under InvisiPage, we utilize the fact that secure space (*EPC*) is fully associative allowing us to simply pick another victim *EPC* page which can be evicted to *non-EPC* memory on a spill failure. This allows InvisiPage to employ a `dataless stash` which only holds metadata associated with failed spill (virtual address of page, leaf, level). This stash is stored securely (in *EPC*) and is maintained by InvisiPage runtime. Dataless (lower footprint) and in-memory nature (stored in *EPC*) of our stash allows us to track larger number of spill failures.

On every ORAM access, attempt is made to spill data in stash to the tree (background spill processing). So as to push data as close to leaf nodes as possible, entries in stash are sorted based on the leaf currently being accessed. Tracking large number of entries in stash can cause high overheads for this sorting. In our design, we avoid this sorting. Instead we maintain a sorted list (by leaf-id) of available entries in stash which reduces stash processing to simple range checks. If an accessed path has a free slot, we compare the leaf range that can be spilled to this slot against leaf range available in stash. Only on an overlap, do we check the stash and do so for overlapped range only. Figure 6 shows an example.

3.3.5 Realizing Thin Nodes. As the tree node width (z) of ORAM tree decreases, the number of pages on the path decrease, reducing page shuffling and overhead per access. However, as z decreases, the probability of spill failures also increases as the #options available to spill a page reduces (slots available on a path). Recall from Section 3.3.4 that such spill failures are tracked in stash and traditional implementations rely on dummy accesses to keep stash occupancy low. In fact, for tree node width 1, an order of magnitude of dummy accesses could be required making this configuration infeasible [30].

However, in InvisiPage, a confluence of optimizations makes thin nodes feasible. First, our stash is in-memory and is dataless making stash footprint very low. As such, we can track large number of spill failures. Second, as discussed in Section 3.3.4 we also have an efficient mechanism to process stash. Finally, smart growth optimization (Section 3.3.2) keeps spill failures low. Together these help us realize thin nodes and reduce ORAM access overheads.

3.4 EPC-lite to reduce OPAM reliance

While our OPAM construct reduces ORAM algorithm overheads considerably, each OPAM access causes several pages to be shuffled in *non-EPC* memory and incurs high overheads. Increasing *EPC* size can reduce such OPAM transactions required. However, as some of the *EPC* security guarantees are at cache block granularity, increasing *EPC* size increases overheads per cache block access to *EPC* to update and lookup metadata (upto four metadata memory accesses for 128MB EPC, upto five memory accesses for 256MB). To overcome this challenge, we make the key observation that in addition to securing the page table entries belonging to sensitive pages we simply need the isolation guarantee of *EPC* to prevent the OS from learning about enclave's accesses to *EPC* pages. To this end, we design a separate memory partition which we term *EPC-lite*, which has the same page-level isolation guarantees as *EPC* but provides other security guarantees (confidentiality, freshness and integrity) at page-level and not cache-block level (as is the case for *non-EPC* memory pages in baseline SGX). As such, *EPC-lite* does not incur the increase in metadata that larger *EPC* incurs. Finally, page movements between *EPC-lite* and *EPC* are done by the InvisiPage runtime and do not require use of the OPAM construct.

When needing a physical page for sensitive data, InvisiPage runtime first tries to get an *EPC* page from OS (Table 1, a). If that fails, it then tries to get an *EPC-lite* page (Table 1, d). Only when both of these fail, the runtime spills an *EPC* page to *non-EPC* memory and incurs an OPAM access (Table 1, c).

Supporting *EPC-lite* requires extending baseline SGX's isolation mechanism to also cover some *non-EPC* memory pages. To support isolation for *EPC* pages, baseline SGX maintains a map structure (Enclave Page Cache Map, EPCM) which tracks metadata per *EPC* page (page permissions, the virtual address of page, owner enclave id). SGX hardware uses this metadata to ensure that only owning enclave reads/writes to a given *EPC* page. To support isolation for pages in *EPC-lite*, we propose to track them similarly in EPCM. Also, just like baseline SGX, which marks *EPC* as no-DMA at memory controller, we also need similar support for *EPC-lite*.

Unlike *EPC* which is fixed at boot time, we propose that the *EPC-lite* be *dynamic*; the number of pages in *EPC-lite* grow and shrink over time. To realize this, we augment baseline SGX to periodically collaborate with the OS (based on *EPC* usage and *EPC* to *non-EPC* memory page movements seen) to extend/shrink *EPC-lite*. This, however, raises the question as to which *non-EPC* memory pages can be added to *EPC-lite*. If we provision support for any *non-EPC* memory page to be added to *EPC-lite*, every memory access needs to check EPCM which will add overheads while accessing non-sensitive pages. Under SGX, *EPC* is a contiguous chunk of physical memory. As such, checking if an address falls in *EPC* is a simple range check. Similarly, we constrain the *EPC-lite* region to be a contiguous memory chunk following the *EPC*. This preserves the single range check currently supported in baseline SGX.

While *EPC-lite* is interesting from a security standpoint, it does have associated limitations. Similar to *EPC*, deallocation of an *EPC-lite* page requires the request to be routed via InvisiPage runtime (Table 1, f). As such, a larger *EPC-lite* region implies that the OS has lower control in moving pages to the backing store and has to rely on an owning enclave to give up an *EPC-lite* page. Also, constraining the *EPC-lite* to be a contiguous memory chunk as we do can lead to memory fragmentation as enclaves give up *EPC-lite* pages. Further, enforcing contiguity can also force hot pages in *EPC-lite* to be freed while shrinking *EPC-lite* size. Techniques which address these limitations (memory compaction to reduce fragmentation, InvisiPage runtime controlled swapping of *EPC* and *EPC-lite* pages) are possible. We leave investigating them to future work.

3.5 InvisiPage Security Analysis

Page fault side channel comprises of addresses accessed by the program, the access type (read, write), number of page faults and their timing. InvisiPage hides addresses accessed along with their access type from a malicious OS. By isolating the page translations of sensitive *EPC* pages from the OS, the OS has no visibility in which *EPC* pages are accessed. Further, by using the OPAM construct, InvisiPage's runtime hides the addresses of pages moved between *EPC* and *non-EPC* memory. Consequently, for a fixed *non-EPC* memory size, InvisiPage's security guarantees are similar to prior ORAM proposals [30].

Further, like these prior proposals, InvisiPage does not hide timing of page faults (specifically timing of page moves between *non-EPC* memory and *EPC*) and their number. To the best of our knowledge, no attack exploits page fault trace length and timing *only* to deduce application secrets. Further, without the address and access type, the amount of information that the trace length or timing can expose is quite limited. Finally, well known prior timing channel solutions employed for other side channels like having a static page move rate [6] or a more dynamic scheme with bounded leakage [48] can be easily adapted in our system to address this leak.

One aspect wherein InvisiPage differs from prior ORAM proposals is that *non-EPC* memory can grow dynamically. This, introduces a new form of leak - memory footprint size. To the best of our knowledge, no attack exploits this leak. Additional mechanisms like static rate of memory allocation can be used to plug this leak.

4 INVISIPAGE IMPLEMENTATION

4.1 InvisiPage Metadata

We discuss in this section the changes we need to the metadata that SGX already provisions, additional metadata needed for InvisiPage and how we update/access them independently.

4.1.1 Changes to SGX Metadata. For every *EPC* page that is spilled to *non-EPC* memory, SGX guarantees confidentiality (encryption), integrity (MAC tag) and freshness (8-byte nonce). It also creates metadata per spilled page which includes among other things virtual address of the page and ID of the owner enclave. The MAC tag SGX generates is over both page data and metadata. As all of this metadata is stored in *non-EPC* memory, a malicious OS can easily read it to learn pages being accessed by the enclave. As such, under InvisiPage, this metadata is encrypted.

Recall that the ORAM tree in *non-EPC* memory has real and dummy pages which need to be indistinguishable to the adversary. Consequently, we create SGX metadata (dummy) for dummy pages in ORAM tree and similarly encrypt it.

A malicious OS can swap two nodes (*non-EPC* memory pages) in the ORAM tree and we need mechanisms to thwart this. To this end, we create the MAC tag part of SGX metadata over not only page data and metadata but also over the page's oblivious virtual page number `o-vpn` (Section 3.2). With this, InvisiPage runtime can easily detect malicious node swaps.

4.1.2 OPAM Metadata. We add metadata per *non-EPC* memory page (real and dummy) to support OPAM construct, termed OPAM metadata. This metadata comprises of page's virtual address (VA), its leaf and level of OPAM tree. Storing VA helps us identify the page that needs to be moved from *non-EPC* memory to *EPC*. Storing leaf and level aid in page shuffling (push real data close to leaf) on each access without accessing position map. As OPAM metadata is inspected independent of SGX metadata (Section 3.3.3), we not only encrypt it but also store a separate MAC tag (also calculated considering `o-vpn`) for it.

Like baseline SGX, we also need to ensure freshness for InvisiPage metadata. Simply adding an additional nonce will double space overheads for nonces. Instead, we observe that both InvisiPage and SGX metadata are updated at each page access albeit sequentially. To harness this, we use the same nonce for both metadata: while InvisiPage metadata uses available SGX nonce, page data and SGX metadata use nonce+1.

4.2 OPAM Implementation

In this section we discuss our OPAM implementation, hardware support we need and an optimization we employ.

We implement an OPAM tree with 50% utilization factor and evaluate different node widths(z). Each slot in our tree stores a *non-EPC* memory page of size 4KB. We support three tree operations: `spill` (*EPC* to *non-EPC* memory page move), `fetch` (*non-EPC* memory to *EPC* page move) and `increase` (add *non-EPC* memory pages to the tree). Our OPAM tree is exclusive; a given page is either in the OPAM tree or in *EPC*. Finally, our implementation does not hide OPAM operation type (spill, fetch etc.).

4.2.1 Additional Paging Primitives. So as to decouple metadata access from page access (Section 3.3.3) we rely on new paging primitives. We describe these primitives and the hardware support needed for them.

Baseline SGX performs page moves between *EPC* and *non-EPC* memory while guaranteeing confidentiality, integrity and freshness. Similarly, we need a primitive which copies a *non-EPC* memory page to another while ensuring these guarantees in order to shuffle *non-EPC* memory pages without having to copy them first in *EPC*.

Further, we also need a primitive which moves data between *EPC* and *non-EPC* memory at granularities smaller than a page while also performing integrity and freshness checks. We use this primitive to read and write OPAM metadata. Prior works like Eleos [28] also propose having such primitives (sub-page access) so as to avoid moving page worth of data between *EPC* and *non-EPC* memory when locality is lacking.

4.2.2 Page Copy Unit. During an OPAM access *non-EPC* memory pages that merely get shuffled are never accessed by the enclave post the access and bringing such pages in caches can only pollute caches. Instead, we perform these copies at memory controller via a copy unit in hardware. The page copy unit helps copy a *non-EPC* memory page to another while ensuring integrity and confidentiality. Our threat model assumes that execution of enclave and its data in the processor (caches, on-chip structures etc.) is secure and isolated from other computation. Several prior studies [35] have discussed solutions for ensuring this property in a multi-core processor with shared hardware structures such as our copy unit.

4.2.3 Spill-ahead Optimization. A free *EPC* page is needed each time an enclave needs a new stack/heap page or wants to access a previously spilled page. Unless the enclave maintains free *EPC* pages, each such demand will add a page spill to the critical path to create a free *EPC* page. Instead, InvisiPage runtime spills an *EPC* page eagerly to maintain one free *EPC* page at all times. Such eager spills can be done in a parallel thread. We term this as spill-ahead optimization and also evaluate it. Note that this does not (directly) help with reducing read/write overheads of an OPAM event.

5 APPLICATIONS AND SECURITY CONTEXT

This section discusses the cloud applications we study and outlines scenarios where they manipulate sensitive data.

- **Genome Processing:** We study PRIMEX [24] which creates metadata to aid fast searches over a genome sequence.
  Security Context: Genome data is highly sensitive as it can be used to identify a person, ancestry information and more. Given genome processing deals with large scale of data, cloud computing is often employed.
- **Graph Processing:** We study the following kernels from GraphMat [40].
  PageRank: PageRank orders web pages based on some metric like popularity.
  Breadth First Search (BFS): BFS takes a graph and an initial vertex and computes the distance (number of edges) to all reachable vertices from the initial vertex.

Benchmark	Instructions	CPI	Benchmark	Instructions	CPI
primex			**pagerank**		
yeast	6.2	1.42	amazon	32.6	0.67
worm	13.6	1.39	flickr	72.2	0.77
gorilla	24.8	1.41	wiki	178.1	0.75
redis			**bfs**		
4k1800s	4.8	1.67	amazon	12.6	0.86
4k3600s	7.8	1.68	flickr	32.7	0.93
4k7200s	12.1	1.68	wiki	71.9	0.92
sgd			**sssp**		
netflix_1	11.0	1.92	amazon	12.6	0.85
netflix_2	10.8	1.92	flickr	32.6	0.93
netflix_5	10.9	1.93	wiki	71.6	0.92
mser			**sift**		
hd	3.2	0.61	hd	20.1	0.41
sun	6.7	0.77	saturn	24.8	0.42
dog	18.2	1.39	sun_1	38.9	0.42
kme	26.8	1.26	sun_2	71.9	0.43

Table 2: Instructions (in billions) and CPI for unsecure baseline (native execution).

Single Source Shortest Path (SSSP): SSSP takes a weighted graph and computes the minimum distance of all vertices from a given vertex.

Collaborative Filtering (SGD): SGD is used by recommender systems [31] to deduce an user's rating for an item based on incomplete set of (user, item) ratings.

Security Context: Wide range of sensitive data is expressed as graphs. Social network analysis [22] manipulates social graphs (containing sensitive information like political or personal views of people). Graphs are also used in bioinformatics to capture functional relationships between entities like genes and proteins.

- **Image Processing:** We study the following kernels from the San Diego Vision Benchmark Suite [43]:
  - **Scale Invariant Feature Transform (SIFT):** SIFT extracts features from images which are robust to scaling, rotation and noise.

    Security context: SIFT is widely used in medical image analysis; an important step in diagnosis and subsequent treatment of diseases [32, 34].
  - **Maximally Stable Extremal Regions (MSER):** MSER is a method to detect blobs in images.

    Security Context: MSER is widely employed in visual surveillance [33], traffic analysis, vehicular tracking and medical image segmentation [49].
- **Redis:** Redis [4] is an open source in-memory key-value data structure store which is widely used (Amazon's SimpleDB, Google's AppEngine).

  Security Context: Key-value stores are often employed as caches for frequent computations like complex SQL queries over traditional databases. As such, they also manipulate a breadth of sensitive data from commercial (stock quotes, people location services) to military sectors.

6 EVALUATION

In this section we demonstrate the efficacy of our proposed design and implementation. To do so, we answer several questions:

- What performance overheads does InvisiPage incur to guard against page fault channel (Section 6.3.1)?
- How successful is *EPC-lite* optimization in reducing performance overheads of InvisiPage (Section 6.3.2)?
- How does Smart Tree Growth help (Section 6.4)?
- What are the benefits of realizing Thin Nodes (Section 6.5)?

6.1 Methodology

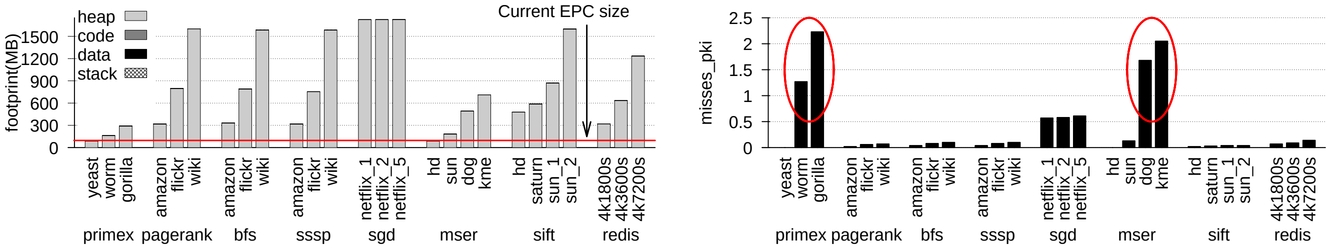
Application Inputs: For genome processing application `primex`, we use genome sequences from the Ensembl genome database [1] with varying sequence lengths. For graph applications, we run real-world graph datasets (Amazon, Flickr and Wikipedia) from the University of Florida Sparse Matrix collection [17] and Netflix challenge for collaborative filtering [10]. For image processing applications (`mser, sift`) we use the largest dataset (full hd) from the San Diego Vision Benchmark Suite [43] and also use images from MIT-Adobe fivek dataset [3] to get larger memory footprints. We run `redis` using Memtier [5], a traffic pattern generator for key value stores with 4096 bytes objects and vary the length of runs.

Note that we chose the application inputs (e.g. object size and increasing run lengths for `redis`) with the aim to exercise memory footprints which increasingly exceed available *EPC* size (Figure 7a, left to right per application). We do so to show how overheads change as working set size increases. Table 2 lists the instruction counts and CPI for unsecure baseline [2] for the applications and the different input sizes.

Execution Model: We generate instruction level memory traces using PIN tool [25] and infer instruction and data TLB misses from the trace. We model a 128 entry 4-way instruction TLB and a 64 entry 4-way data TLB. We use this TLB miss trace to infer *EPC* hits and misses. For *EPC* misses we infer the OPAM events incurred. We model 96MB of *EPC* [1] based on current Intel SGX processors [21] and employ clock algorithm [11] for page replacement.

Performance Model: We use the OPAM events from the execution model to infer performance overheads. Our analytical model assumes the application is stalled to tackle OPAM event and as such available memory bandwidth can be used to process the OPAM event (read, decrypt, encrypt pages etc.). Both page movements and OPAM algorithm contribute to performance overhead. We set the page movement (copy) cost assuming a standard memory system with 12.8 GB/s/channel and four channels. The OPAM algorithm cost involves reading metadata blocks and making page movement decisions. This cost is low at program start (short tree, small *non-EPC* memory size) and increases slowly with tree height (*non-EPC* memory increases). Our model for this cost assumes the worst case for number of metadata blocks (tallest OPAM tree we observe) being read and inspected. We assume that OPAM algorithm cost is two times page movement cost. A small portion of this cost is for read/updates of metadata blocks and rest is assumed for making page movement decisions. Note that the algorithm cost is dwarfed by the cost of page movements per OPAM event. Finally, we also assume that a parallel thread executes the OPAM events. This helps

[1] While actual *EPC* size is 128MB, only 96MB is usable and rest is for metadata.



(a) **(b)**

Figure 7: (a) Memory footprint (accessed) of applications. (b) *EPC* misses per kilo instructions.

us exploit the spill-ahead optimization (Section 4.2.3) which takes *EPC* spills off the critical path when possible.

6.2 Memory Footprint of Applications

We depict in Figure 7a the memory footprint of various applications we model for different input sizes. We track unique pages accessed by the application to deduce this footprint and also break it down into code, data, stack and heap buckets. The memory footprint of applications we model is dominated by heap pages. Per application, we pick inputs with varying memory footprints to evaluate how the overhead of mitigating page fault channel changes as memory footprint increasingly exceeds *EPC* size. As an example, the footprint for `pagerank` varies from 3X for `amazon` to 17X for `wiki` with respect to available *EPC* size. For `sgd` we do observe that changing the number of input movie files does not change memory footprint.

Larger memory footprints are more likely to cause page movements across *EPC* and *non-EPC* memory boundary and as such could cause larger overheads. As our optimizations (OPAM and *EPC-lite* memory) reduce both the number of the page movements and cost of making them secure, the benefits of our optimizations will be more pronounced for larger memory footprints. The memory footprints we study in this work are largely limited by the simulation time needed to get traces for the entire application and the enormous storage needed for the resultant traces.

6.3 InvisiPage Performance

6.3.1 InvisiPage without Enclave-lite. Figure 8 depicts performance overheads of InvisiPage. With existing *EPC* size (96MB) and no *EPC-lite* optimization, the average overhead of InvisiPage is 3.54X. First, note that our current performance model conservatively assumes the application is stalled to process OPAM events. More sophisticated models which use page access prediction to process OPAM events a priori can further lower these overheads. Next, as expected, we observe that the larger the delta between *EPC* size and the memory footprint of the application, more are the OPAM events incurred leading to increased overhead. However, the increase in overhead is not commensurate to memory footprint of the application. As an example, `sgd` has the highest memory footprint of all applications but not the highest performance overhead. This is also true for several other interesting workloads (`sift`, graph workloads) whose working sets far exceed combined capacity of *EPC* and *EPC-lite* but they end up with low overheads under InvisiPage.

We observe that the OPAM events incurred by an application are more a property of its memory access behavior than its footprint. Figure 7b depicts the *EPC* miss rates observed per kilo instructions for applications under study. Some of the applications (e.g. `primex, mser`) exhibit very high miss rates which in turn cause high performance overheads as depicted in Figure 8.

We depict in Figure 9c that in absence of the workloads with high *EPC* miss rates (`worm, gorilla, dog, kme`), the average overhead is much lower. Specifically for `z` = 1, overhead drops from 3.54X (`all`) to a mere 1.7X (`subset`) with existing *EPC* size (96MB).

6.3.2 InvisiPage with Enclave-lite. We discussed in Section 3.4 how *EPC-lite* optimization helps us reduce the number of OPAM transactions needed. Figure 8 depicts performance overheads of InvisiPage as we increase *EPC-lite* memory size for a four channels memory system. As expected, the performance overheads drop as *EPC-lite* memory size increases as the number of OPAM events drop and also as some of the workload's memory footprint fits within available memory. At 768MB memory, we see a performance overhead of mere 16% to fix page fault channel with memory footprint of fifteen of the available twenty-six workloads fits inside available memory.

6.4 Evaluation of Smart Tree Growth

Figure 9a depicts the comparison of smart tree growth and its naive counterpart. We show the maximum spill failures (pending spill failures averaged across all applications) for different tree node widths (`z`). On a page eviction from *EPC* we randomly pick a path in the OPAM tree to spill this page. The lower the value of `z`, lower the options available along the chosen path and hence higher the chances of spill failures as is seen in Figure 9a. While smart growth also depicts this behavior there is several orders of magnitude of reduction in the number of pending spill failures as compared to naive growth for higher values of `z` and close to an order of magnitude of reduction for `z = 1`. By prioritizing accessed paths, smart tree growth adds space to the OPAM tree where it is most needed and as a consequence far less failures need to be tracked and considered on each OPAM access.

6.5 Benefits of Thin Nodes

Thin nodes are interesting in that they help reduce performance cost of each OPAM event. As (`z`) decreases, while the performance cost of each OPAM event reduces (reduction in page moves needed), the spill failures also increase (Section 6.4) needing ability to track these failures. In order to deal with increased spill failures prior

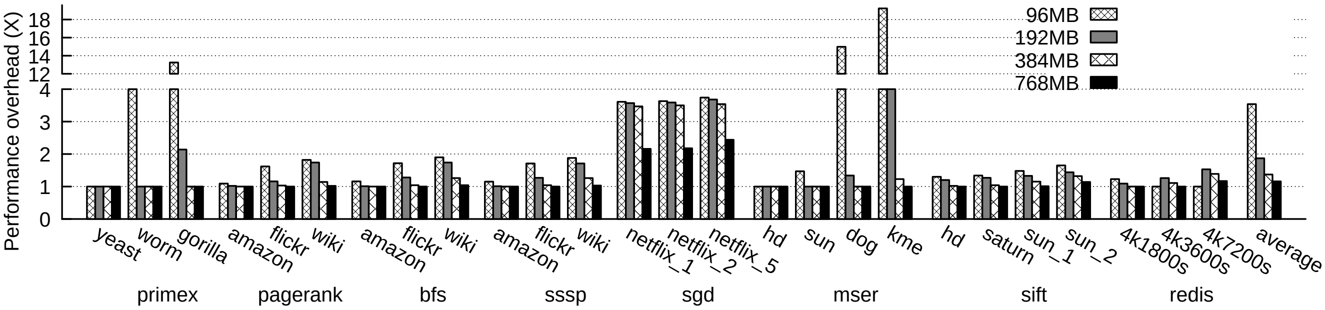
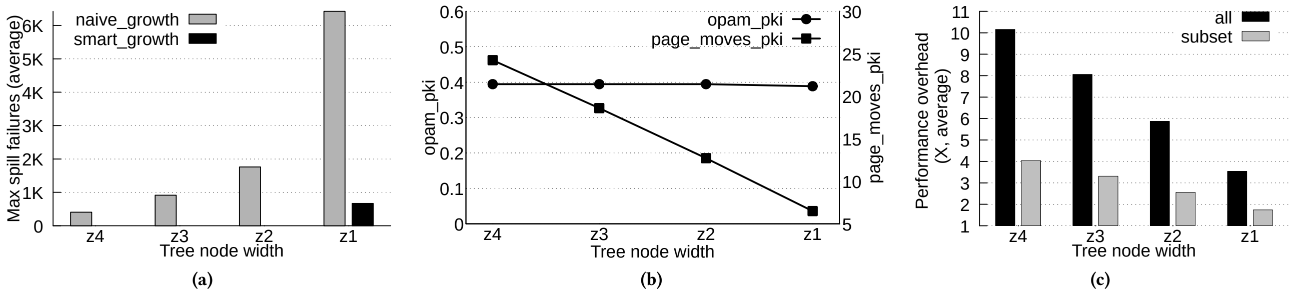


Figure 8: Performance overhead with increasing enclave-lite memory size (total isolated memory (usable) is depicted).



(a)

(b)

(c)

Figure 9: (a) Smart tree growth considerably reduces spill failures by prioritizing accessed paths while adding space. (b) Benefit of thin nodes: page moves reduce while OPAM events do not increase. (c) Performance overhead for existing *EPC* size (96MB).

works [30] incur an order of magnitude increase in OPAM events at lower values of (z) making such configurations infeasible. As discussed in Section 3, while smart growth helps us keep spill failures in check, dataless stash helps us keep the overhead of tracking spill failures low. Together they help us realize thin nodes.

Figure 9b depicts both the OPAM events and page moves per kilo instructions for different values of z averaged across all benchmarks. As we reduce z, the page moves needed per OPAM access reduces which will cause commensurate reduction in performance cost of each OPAM event. At the same time, the OPAM events needed do not increase as (z) decreases. Together, this leads to lower overheads at lower values of z as depicted in Figure 9c. Therein, we see decreasing average performance overhead for InvisiPage with decreasing node widths as compared to an unsecure baseline.

7 RELATED WORK

Note that we describe InvisiPage in the context of SGX owing to its commercial availability and the fact that it is already hardened against several attacks. Other secure hardware proposals beside Intel SGX do exist [12, 39, 42]. Such prior proposals are susceptible to page fault channel attack and could benefit from InvisiPage.

7.1 Prior Page Fault Channel Mitigations

Like InvisiPage, Sanctum [16] secures address translations of enclave's sensitive pages but requires that an enclave's memory requirements are known a priori which is unrealistic. Further, accesses to sensitive pages reclaimed by OS also leak information. T-SGX [36] relies on Intel Transactional Synchronization Extensions (Intel TSX) to get notified on a page fault and assumes any page fault is a potential attack. This necessitates that enclave's entire memory footprint fit inside *EPC*. Given the small size of *EPC* on current SGX processors, this is also unrealistic. Also, this leaves no provision for the OS to reclaim sensitive pages. Deja Vu [14] detects a privileged attack when enclave's execution differs widely from a reference clock it constructs. While a formidable technique to detect anomalous executions, this does not prevent information leak via page faults present in benign executions. In [37], the authors propose two approaches: determinizing page access pattern and reliance on the processor to ensure no page faults occur to a set of pre-declared pages. While deterministic page access patterns have high overheads, pre-declaring pages that a section of program will access is hard for general programs. More recently, Apparition [18] prevents the OS from manipulating page table entries belonging to sensitive pages but requires *all* sensitive pages to be swapped in on an access to *any* swapped out sensitive page by the application. This limits the memory size available for sensitive data severely (*EPC* size in SGX parlance).

7.2 Extending SGX

Proposals which extend SGX capabilities [19, 41] do exist. However, they are susceptible to page fault channel and as such could benefit from InvisiPage. VAULT [41] extends *EPC* to cover (possibly) entire physical memory space using a variable arity integrity tree. Similar to *EPC-lite* optimization, VAULT's extension will help reduce page movements across *EPC* and *non-EPC* memory boundary.

However, VAULT provides security guarantees at finer granularity (cache block instead of page like in InvisiPage) albeit at the cost of metadata checks even for non-sensitive pages. Further, while VAULT mechanisms do increase *EPC*, not only is it hard to predetermine an enclave's memory footprint, but also, reserving large *EPC* for a single enclave precludes sharing memory amongst different enclaves. This necessitates on-demand paging which is unsecure under VAULT.

Unlike SGX, Iso-X [19] enables on-demand isolation of memory pages which, however, takes away ability of OS to manage memory as a resource as more pages get isolated. To tackle the latter problem, Iso-X proposes OS managed page swapping using which the OS can learn the page access pattern. InvisiPage secures such OS page management actions using OPAM construct thus restoring OS's ability to serve as a resource arbiter of memory pages. Obliviate [7] obfuscates the file system access patterns of an enclave but does not address the page fault channel for the enclave's execution.

7.3 Optimizing SGX Performance

Prior works which optimize SGX performance exist and can be adapted in our system. Hotcalls [44] and Eleos [28] offload system call processing to a separate thread to reduce their overheads. In addition, Eleos uses Intel Cache Allocation Technology [15] to reduce LLC pollution due to system calls. SCONE [9] uses asynchronous system calls to enable secure containers with low overheads.

8 CONCLUSION

This paper presents InvisiPage, a page fault channel defense that enables oblivious OS demand paging: the OS preserves its flexibility to manage memory pages yet does not learn an application's address trace via page fault channel. To do so, InvisiPage carefully distributes page management actions between the application and the OS. Further, InvisiPage uses a novel Oblivious Page Management (OPAM) construct to make an application's page management transactions with the OS secure. Finally, InvisiPage employs a novel memory partition to lower interactions with the OS and further reduce overheads. Our results demonstrate that InvisiPage fixes page fault channel while enabling oblivious demand paging at low overheads for a suite of cloud applications.

9 ACKNOWLEDGMENTS

We thank anonymous reviewers for helping improve this paper. This work was supported by the NSF under the CAREER-1149773, SHF-1527301 and SHF-1703931 awards and by C-FAR, one of the six SRC STARnet Centers, sponsored by MARCO and DARPA.

REFERENCES

[1] 2017. Ensembl genome browser. "http://www.ensembl.org/index.html". (2017).

[2] 2017. IntelÂő CoreâĎć2 Quad Processor Q6600. "https://ark.intel.com/content/www/us/en/ark/products/29765/intel-core-2-quad-processor-q6600-8m-cache-2-40-ghz-1066-mhz-fsb.html". (2017).

[3] 2017. MIT-Adobe fivek dataset. "http://groups.csail.mit.edu/graphics/fivek_dataset/". (2017).

[4] 2017. Redis. "http://redis.io/". (2017).

[5] 2017. Redis Labs. Memtier Benchmark. "https://github.com/RedisLabs/memtierbenchmark". (2017).

[6] Shaizeen Aga and Satish Narayanasamy. 2017. InvisiMem: Smart Memory Defenses for Memory Bus Side Channel. In *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA '17)*.

[7] Adil Ahmad, Kyungtae Kim, Muhammad Ihsanulhaq Sarfaraz, and Byoungyoung Lee. 2018. OBLIVIATE: A Data Oblivious Filesystem for Intel SGX. In *25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, February 18-21, 2018*.

[8] Ittai Anati, Shay Gueron, Simon P Johnson, and Vincent R Scarlata. 2013. Innovative technology for cpu based attestation and sealing. In *Proceedings of the 2Nd International Workshop on Hardware and Architectural Support for Security and Privacy (HASP '13)*.

[9] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, Andre Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Dan O'Keeffe, Mark L. Stillwell, David Goltzsche, David Eyers, Rüdiger Kapitza, Peter Pietzuch, and Christof Fetzer. 2016. SCONE: Secure Linux Containers with Intel SGX. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI'16)*.

[10] J Bennett and S Lanning. 2007. The Netflix Prize. In KDD Cup and Workshop at ACM SIGKDD, 2007. (2007).

[11] Richard W. Carr and John L. Hennessy. 1981. WSCLOCK&Mdash;a Simple and Effective Algorithm for Virtual Memory Management. In *Proceedings of the Eighth ACM Symposium on Operating Systems Principles (SOSP '81)*.

[12] D. Champagne and R. B. Lee. 2010. Scalable architectural support for trusted software. In *High Performance Computer Architecture (HPCA), 2010 IEEE 16th International Symposium on*.

[13] Stephen Checkoway and Hovav Shacham. 2013. Iago Attacks: Why the System Call API is a Bad Untrusted RPC Interface. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '13)*.

[14] Sanchuan Chen, Xiaokuan Zhang, Michael K. Reiter, and Yinqian Zhang. 2017. Detecting Privileged Side-Channel Attacks in Shielded Execution with DéJà Vu. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security (ASIA CCS '17)*.

[15] Intel corp. 2015. Improving Real-Time Performance by Utilizing Cache Allocation Technology. Intel White paper. (2015).

[16] Victor Costan, Ilia Lebedev, and Srinivas Devadas. 2016. Sanctum: Minimal Hardware Extensions for Strong Software Isolation. In *Proceedings of the 25th USENIX Conference on Security Symposium (SEC'16)*.

[17] T Davis. 2017. The University of Florida Sparse Matrix Collection. "http://www.cise.ufl.edu/research/sparse/matrices". (2017).

[18] Xiaowan Dong, Zhuojia Shen, John Criswell, Alan L. Cox, and Sandhya Dwarkadas. 2018. Shielding Software From Privileged Side-Channel Attacks. In *27th USENIX Security Symposium (USENIX Security 18)*.

[19] D. Evtyushkin, J. Elwell, M. Ozsoy, D. Ponomarev, N. A. Ghazaleh, and R. Riley. 2018. Flexible Hardware-Managed Isolated Execution: Architecture, Software Support and Applications. *IEEE Transactions on Dependable and Secure Computing* (2018).

[20] Oded Goldreich and Rafail Ostrovsky. 1996. Software Protection and Simulation on Oblivious RAMs. *J. ACM* (1996).

[21] S. Gueron. 2016. Memory Encryption for General-Purpose Processors. *IEEE Security Privacy* (2016).

[22] Mohsen Jamali and Hassan Abolhassani. 2006. Different Aspects of Social Network Analysis. In *Proceedings of the 2006 IEEE/WIC/ACM International Conference on Web Intelligence (WI '06)*.

[23] Paul Kocher, Joshua Jaffe, Benjamin Jun, and Pankaj Rohatgi. 2011. Introduction to differential power analysis. *Journal of Cryptographic Engineering* (2011), 5–27. https://doi.org/10.1007/s13389-011-0006-y

[24] Matej Lexa and Giorgio Valle. 2003. PRIMEX: rapid identification of oligonucleotide matches in whole genomes. *Bioinformatics* (2003).

[25] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. 2005. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '05)*.

[26] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V. Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday R. Savagaonkar. 2013. Innovative Instructions and Software Model for Isolated Execution. In *Proceedings of the 2Nd International Workshop on Hardware and Architectural Support for Security and Privacy (HASP '13)*.

[27] Tarik Moataz, Travis Mayberry, Erik-Oliver Blass, and Agnes Hui Chan. 2014. Resizable Tree-Based Oblivious RAM. Cryptology ePrint Archive, Report 2014/732. (2014). http://eprint.iacr.org/2014/732.

[28] Meni Orenbach, Pavel Lifshits, Marina Minkin, and Mark Silberstein. 2017. Eleos: ExitLess OS Services for SGX Enclaves. In *Proceedings of the Twelfth European Conference on Computer Systems (EuroSys '17)*.

[29] Jean-Jacques Quisquater and David Samyde. 2002. Side Channel Cryptanalysis. In *Workshop on the Security of Communications on the Internet (SECI)*.

[30] Ling Ren, Xiangyao Yu, Christopher W. Fletcher, Marten van Dijk, and Srinivas Devadas. 2013. Design Space Exploration and Optimization of Path Oblivious RAM in Secure Processors. In *Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA '13)*.

[31] Francesco Ricci, Lior Rokach, and Bracha Shapira. 2011. *Introduction to Recommender Systems Handbook*. Springer US.

[32] Y. Sa. 2015. Medical Image Registration Algorithm Based on Compressive Sensing and Scale-Invariant Feature Transform. In *2015 8th International Conference on Intelligent Computation Technology and Automation (ICICTA)*.

[33] E. Salahat, H. Saleh, A. S. Sluzek, B. Mohammad, M. Al-Qutayri, and M. Ismail. [n. d.]. Novel MSER-guided street extraction from satellite images. In *2015 IEEE International Geoscience and Remote Sensing Symposium (IGARSS)*.

[34] L. A. Salazar-Licea, C. Mendoza, M. A. Aceves, J. C. Pedraza, and A. Pastrana-Palma. 2014. Automatic segmentation of mammograms using a Scale-Invariant Feature Transform and K-means clustering algorithm. In *2014 11th International Conference on Electrical Engineering, Computing Science and Automatic Control (CCE)*.

[35] Ali Shafiee, Akhila Gundu, Manjunath Shevgoor, Rajeev Balasubramonian, and Mohit Tiwari. 2015. Avoiding Information Leakage in the Memory Controller with Fixed Service Policies. In *Proceedings of the 48th International Symposium on Microarchitecture (MICRO-48)*.

[36] Ming-Wei Shih, Sangho Lee, Taesoo Kim, and Marcus Peinado. 2017. T-SGX: Eradicating Controlled-Channel Attacks Against Enclave Programs. In *24th Annual Network and Distributed System Security Symposium, NDSS 2017, San Diego, California, USA, February 26-March 1, 2017*.

[37] Shweta Shinde, Zheng Leong Chua, Viswesh Narayanan, and Prateek Saxena. 2016. Preventing Page Faults from Telling Your Secrets. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security (ASIA CCS ’16)*.

[38] Emil Stefanov, Marten van Dijk, Elaine Shi, Christopher Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. 2013. Path ORAM: An Extremely Simple Oblivious RAM Protocol. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer &#38; Communications Security (CCS ’13)*.

[39] G. Edward Suh, Dwaine Clarke, Blaise Gassend, Marten van Dijk, and Srinivas Devadas. 2003. AEGIS: Architecture for Tamper-evident and Tamper-resistant Processing. In *Proceedings of the 17th Annual International Conference on Supercomputing (ICS ’03)*.

[40] Narayanan Sundaram, Nadathur Satish, Md Mostofa Ali Patwary, Subramanya R. Dulloor, Michael J. Anderson, Satya Gautam Vadlamudi, Dipankar Das, and Pradeep Dubey. 2015. GraphMat: High Performance Graph Analytics Made Productive. *Proc. VLDB Endow.* (2015).

[41] Meysam Taassori, Ali Shafiee, and Rajeev Balasubramonian. 2018. VAULT: Reducing Paging Overheads in SGX with Efficient Integrity Verification Structures. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS ’18)*.

[42] David Lie Chandramohan Thekkath, Mark Mitchell, Patrick Lincoln, Dan Boneh, John Mitchell, and Mark Horowitz. 2000. Architectural Support for Copy and Tamper Resistant Software. In *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS IX)*.

[43] S. K. Venkata, I. Ahn, D. Jeon, A. Gupta, C. Louie, S. Garcia, S. Belongie, and M. B. Taylor. 2009. SD-VBS: The San Diego Vision Benchmark Suite. In *2009 IEEE International Symposium on Workload Characterization (IISWC)*.

[44] Ofir Weisse, Valeria Bertacco, and Todd Austin. 2017. Regaining Lost Cycles with HotCalls: A Fast Interface for SGX Secure Enclaves. In *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA ’17)*.

[45] Bin Cedric Xing, Mark Shanahan, and Rebekah Leslie-Hurd. 2016. Intel&Reg; Software Guard Extensions (Intel&Reg; SGX) Software Support for Dynamic Memory Allocation Inside an Enclave. In *Proceedings of the Hardware and Architectural Support for Security and Privacy 2016 (HASP 2016)*.

[46] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. 2015. Controlled-Channel Attacks: Deterministic Side Channels for Untrusted Operating Systems. In *2015 IEEE Symposium on Security and Privacy*.

[47] Yuval Yarom and Katrina Falkner. 2014. FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-channel Attack. In *Proceedings of the 23rd USENIX Conference on Security Symposium (SEC’14)*.

[48] Danfeng Zhang, Aslan Askarov, and Andrew C. Myers. 2011. Predictive Mitigation of Timing Channels in Interactive Systems. In *Proceedings of the 18th ACM Conference on Computer and Communications Security (CCS ’11)*.

[49] Haijiang Zhu, Junhui Sheng, Fan Zhang, Jinglin Zhou, and Jing Wang. 2016. Improved Maximally Stable Extremal Regions Based Method for the Segmentation of Ultrasonic Liver Images. *Multimedia Tools Appl.* (2016).