When Oblivious is Not: Attacks against OPAM

Nirjhar Roy*
Indian Institute of Technology - Kanpur
nirjhar@iitk.ac.in

Nikhil Bansal
Indian Institute of Technology - Kanpur
nikhilba@iitk.ac.in

Gourav Takhar
Indian Institute of Technology - Kanpur
tgourav@cse.iitk.ac.in

Nikhil Mittal
Fortanix
nkmittal4994@gmail.com

Pramod Subramanyan†
Indian Institute of Technology - Kanpur
spramod@cse.iitk.ac.in

Abstract

Enclave platforms like Intel SGX, Sanctum and Keystone promise attractive security guarantees but have not always lived up to their billing, mostly due to side channel leaks in platform implementations. A particularly important side channel in these platforms has been the page-fault side channel. This side channel has proven to be particularly problematic because it is deterministic and controllable by a malicious operating system. This paper presents a new attack on the page-fault channel that works on the state-of-art proposal for secure demand paging in enclaves (InvisiPage, ISCA'19). The insight behind the attack is that even if the exact page-fault addresses are hidden, the adversary may be able to infer the interval between when a page is evicted from an enclave and when it is fetched back into the enclave. Our evaluation shows this leak is sufficient to: (i) identify which application is being executed in an enclave, (ii) infer confidential details about the inputs to the application, and (iii) function as a covert channel between an untrusted enclave application and a malicious operating system.

1 Introduction

Enclave platforms [6, 10, 13, 14, 17, 24, 29, 30, 33, 46] enable secure execution of applications in the presence of a privileged software adversary. Enclave platforms can, in principle, enable secure computing in the cloud even if co-located applications and the cloud provider's infrastructure are malicious. These platforms achieve this security guarantee by relying on trusted hardware primitives that isolate a region of memory such that only the code that lies within this region can read and write to it.

Despite providing attractive security guarantees, enclave platforms have been plagued by implementation vulnerabilities. Almost all of these are based on side channels violating confidentiality [11, 18, 28, 31, 42, 47, 49] that enable exfiltration of confidential data from enclaves. In this context, it is worth distinguishing two kinds of side channels: (i) software-visible hardware side channels such as caches, prefetchers, and branch predictors, and (ii) software side channels, like page tables. Significant progress has been made in addressing the first kind of side channel by isolating shared microarchitectural resources to ensure non-interference between protection domains (e.g. [8, 14, 39]).

Software side channels, especially page tables, have received less attention. In their seminal work on controlled-channel attacks, Xu et al. [49] showed that careful control of the page tables makes it possible to extract large amounts of confidential information from enclaves. They extracted image outlines from image processing applications and text documents from word processing tools. Their attack relies on the fact that platforms like SGX allow the operating system (OS) to retain control over the enclave applications page tables and use the OS to service page-faults. This control is exploited by the OS to cause page-faults within the enclave by marking all pages as not being present and learning the enclave's memory access pattern by observing which page-faults are triggered.

An initial defense against the controlled channel attack was T-SGX [41] which uses transactional memory extensions to detect if an attack is underway and aborts execution if one is detected. T-SGX was defeated by a stealthier attack from Van Bulck et al. [48] which analyzed page tables' accessed and dirty bits without causing page-faults. An approach that has proven more resilient has been the idea of making enclave page tables private. This approach was first taken by Iso-X [17], and was subsequently adopted by Sanctum [14], Apparition [16], Keystone [29, 30] and InvisiPage [5].

Simply making enclave page tables private is not enough, page fetches and evictions will be visible to the untrusted operating system (OS) because the OS continues to manage resource allocation. As a result, the OS can infer enclave secrets by observing the sequence of page-faults that an enclave experiences. Sanctum [14] side-stepped this problem by disabling demand paging of enclave memory by default. This

*This author is currently at Fortanix Inc.
†This paper is dedicated to the loving memory of our advisor, late Dr. Pramod Subramanyan.

is clearly secure: if page-faults are not adversary-observable, nothing can leak via page-faults.

However, disabling demand paging is undesirable as it restricts enclaves to fixed-size working sets. There is significant interest in enclave applications that operate on large amounts of data, e.g. databases [35], blockchains [12], machine learning [25, 34], data analytics [37, 51], to name just a few. Restricting these applications to small datasets limits applicability of enclave platforms. Currently these applications have either used demand paging while disregarding the page-fault side channel [25, 35] or have developed novel oblivious algorithms that do not leak information via the program's access pattern [34, 51]. Neither of these is a satisfactory state of affairs. Principled solutions to the page-fault side channel will deliver increased security for all enclave applications rather than just those which have been custom-developed.

In response to the above concerns, Aga and Narayanasamy introduced InvisiPage [5], which enables demand paging in an enclave processor through an oblivious page access module (OPAM). InvisiPage addresses some of the above concerns by hiding the page-faults experienced by an enclave through interposition of an oblivious RAM (ORAM) primitive [20, 44] between an enclave's page-faults and OS-visible page operations. Unfortunately, instead of using a "vanilla" ORAM primitive, InvisiPage introduces certain optimizations to Path ORAM [44] in order to improve performance. In this paper, we perform a careful security analysis of these optimizations and show that InvisiPage is vulnerable to a new type of side channel leak. It leaks the reuse distances of enclave pages – the number of page-faults between when a page is evicted and subsequently re-fetched into the enclave.

We show that this vulnerability can be exploited in three ways. First, we can train a convolutional neural network to fingerprint applications executing within an enclave. Second, we can use the same kind of fingerprinting to classify the confidential types of inputs an enclave is operating on. Finally, we show that the reuse distance attack can be used as covert channel that transmits up to 70 bits/second between an enclave application that is sandboxed using techniques like Ryoan [26] and a colluding operating system.

2 Background

This section provides a brief review of enclave platforms, demand paging in enclaves and tree-based oblivious RAMs.

2.1 Overview of Enclave Platforms

Enclave platforms provide trusted hardware primitives for secured execution of application code in the presence of a privileged software adversary. This is achieved by isolating a range of memory addresses such that only the code within that range of memory is allowed to read/write to the range. This form of isolation ensures confidentiality and integrity against direct attacks. Confidentiality also requires isolation of various microarchitectural side channels, and some implementations have adopted such isolation techniques [8, 14, 39]. Enclave platforms must also provide an attestation operation [9, 46] that produces a signed cryptographic digest, called a measurement, of the initial state of the enclave.

Life Cycle of an Enclave

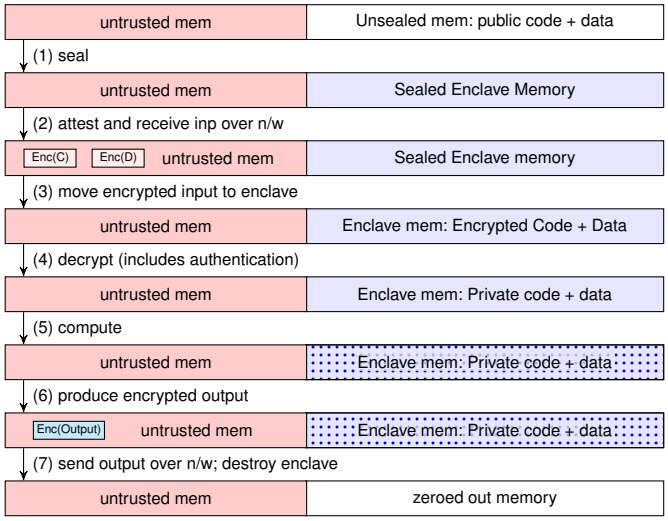

Figure 1: Overview of Enclave Execution. Red and white boxes show memory accessible to the untrusted OS. Blue boxes show trusted and sealed memory; this memory is only accessible to the enclave.

Figure 1 shows an example of an enclave executing a private program on a private dataset. First, the application loads the public components of its code and data into memory via the operating system. Since the OS is untrusted, there is no guarantee that the code and data loaded into memory are what was intended. Therefore, before the enclave becomes usable, a *seal* operation is required. This makes the initial code and data inaccessible to all *other* software on the platform. Immediately and atomically after sealing, the platform computes the enclave's measurement and stores the measurement in a trusted processor register. An attestation is a hardware-signed statement including this measurement.

In step 2, this attestation is provided (possibly over a network connection) to the trusted first-party who requested the creation of the enclave. If the attestation is valid, the first party and the enclave will establish a secure communication channel and the secret components of the code and data will be sent to the enclave. Note that enclave itself does not need to directly perform these I/O operations (and is not allowed to perform I/O in the case of Intel SGX), these are proxied through the untrusted OS and cryptography provides confidentiality and integrity against a malicious OS. In steps 4 to

6, the enclave decrypts this private code and data, computes and produces encrypted output that is sent to the first-party.

Finally, in step 7, the enclave is destroyed. The platform ensures that enclave memory is zeroed out at the time of destruction to prevent the OS from extracting secrets from the enclave after destruction.

2.2 Page Fault Handling in Enclaves

Intel SGX places the page tables under the control of the host OS. SGX uses special hardware protections to prevent the OS from changing the contents of the enclave's pages while still allowing the OS to perform demand paging. This is ideal from a resource-management perspective but opens up the possibility of page table-based attacks [42, 48, 49], which allow a malicious OS to infer confidential information about the enclave's execution. Sanctum [14] addresses this problem by maintaining a separate page table in enclave memory. While the page table's initial setup is done by the OS, it becomes private to the enclave after sealing (Step 2 in Figure 1).[1]

Sanctum allocates contiguous regions of virtual and physical memory to the enclave. The allocated memory is divided into two compartments, one compartment for the enclave runtime and the other for the application. These two compartments are isolated from each other via standard page table permission-bits. The mapping between these virtual and physical memory regions is controlled via the page tables which are stored in enclave runtime memory. When a page-fault occurs, execution is redirected to a page-fault handling routine in the runtime. The default runtimes in Sanctum and Keystone kill the enclave when a page-fault occurs; recent versions of Keystone provide an optional page-fault handler that simply encrypts page data.

Overview of Secure Demand Paging Approaches

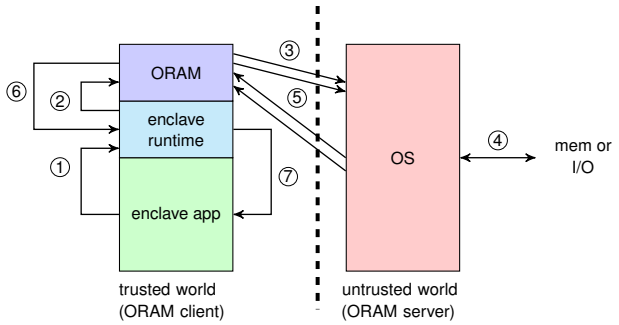

Figure 2: Secure demand paging using ORAMs.

[1]Note that by enclave, we are referring to both the enclave application as well the runtime libraries which are linked with it.

Oblivious RAM (ORAM) primitives enable secure demand paging in enclaves by interposing an ORAM between enclave accesses and OS-visible operations. This approach, first suggested by Sanctum [14] and implemented by InvisiPage [5] is shown in Figure 2. The steps shown in the figure are described below.

1. The page-fault occurs and is handled by the runtime.
2. The runtime determines whether this is a demand fault and if an eviction is required before the demand fetch.
3. If an eviction is needed, the victim page is identified and an ORAM client write is performed. If no eviction is needed, only an ORAM client read is performed. ORAM client reads and writes correspond to page fetch and eviction requests respectively.

4-5) In typical tree-based ORAMs, a client read or write involves several server reads and writes. In our specific case, server reads and writes correspond to OS-visible page fetches and page evictions respectively. Data in these pages is protected using randomized authenticated encryption. The ORAM orchestrates the reads and writes such that the OS learns nothing about the page addresses accessed by the application.

6) Eventually the page of interest is brought inside the enclave, decrypted and made available to the runtime.

7) The runtime now maps this page into the page table and returns control back to the enclave application.

The interface functions between the different components involved in the above process are shown in Table 1.

Table 1: App/RT/OS/ORAM Interfaces for Demand Paging.

Interface	Function	Description
App/RT	`handlePF`	Page-fault handling entrypoint.
RT/ORAM	`accessORAM`	ORAM access (page eviction/fetch).
ORAM/OS	`pageRead` `pageWrite`	Read page from untrusted memory. Write page to untrusted memory.
RT/OS OS/RT	`allocPages` `reclaimPages`	Get additional physical memory. Reclaim allocated memory.

Most functions in Table 1 have been described above. The new entrants are `allocPages` and `reclaimPages` which are used by the enclave to request additional physical memory from the OS, and by the OS to request that an enclave free some of its memory and return it to the OS, respectively.

2.3 Overview of Path ORAM

Oblivious RAM (ORAM) [20, 21] is a cryptographic primitive that allows a client with a small amount of trusted storage

to access a much large untrusted server storage while maintaining computational indistinguishability of usage access patterns. Most recent implementations of ORAMs have used the tree-based ORAM structure introduced by Shi et al. [40] and Path ORAM [44], introduced by Stefanov and colleagues is perhaps the best-known exemplar of tree-based ORAMs.

2.3.1 Path ORAM Data Structures

The two data structures maintained by Path ORAM in the enclave are a *stash*: a temporary array containing recently accessed pages and the *position map* which stores a mapping between enclave virtual addresses (trusted) and OS identifiers (untrusted addresses). In the untrusted OS, Path ORAM stores a binary tree comprised of buckets. Each bucket typically contains 4 encrypted pages.

2.3.2 Path ORAM Invariant

Each page in the ORAM tree is mapped to some leaf by the position map and the page is stored somewhere along the path from the root to this leaf in the tree. The invariant explains why Path ORAM tree nodes are buckets containing multiple pages. Some leaves may be oversubscribed and the bucket structure prevents failure in these cases.[2] Some buckets may also be undersubscribed and if so "dummy" pages containing random data are stored in them.

2.3.3 Path ORAM Access Algorithm

Algorithm 1 Path ORAM Pseudocode

```
1:  procedure ACCESS(vpn, op, data*)
2:      leaf ← positionMap[vpn]
3:      positionMap[vpn] ← UNIFORMRANDOM(1, N)
4:      stash ← stash ∪ READPATH(leaf)
5:      data ← stash[vpn]
6:      if op = evict then stash[vpn] ← data*
7:      end if
8:      WRITEPATH(stash, leaf)
9:      return data
10: end procedure
```

Pseudocode for Path ORAM is shown in Algorithm 1. The algorithm is remarkably simple and elegant, but has a number of subtleties and pitfalls which we discuss in the next subsection. Each ORAM client access is handled in the following way. First, we look up the position map of the leaf node to which the requested page is mapped to (line 2). Second, we *remap* the page to a new leaf (line 3). Now the entire path from the root to this leaf is read in from the untrusted server into the client-side stash (line 4). After each bucket is read in, it is also authenticated and decrypted. Authentication ensures that the server has not modified the data. Decryption is needed because items in the tree are encrypted. Next, the requested page is fetched from the stash (line 5). If the access is an eviction, the stash is updated to contain the new data (line 6). Finally, we write all items in the stash that can possibly be stored on this path to the leaf back to the server (line 8).

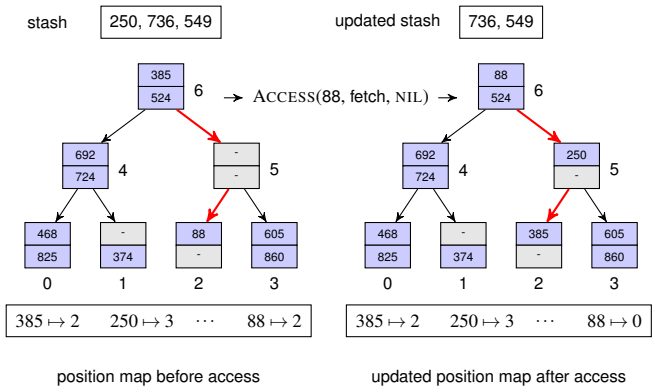

Figure 3: Example Path ORAM access. The accessed path is shown in red thick arrows. Nodes labeled with a hyphen (-) contain dummy pages.

Figure 3 shows an example of how Path ORAM's data structures are updated with an example access. An access is made to vpn=88 and this page is remapped to leaf 0. The ORAM fetches all buckets along the path to the leaf labeled 2. This brings in pages with vpn 88, 385 and 524 into the stash. Now all pages which can be written along this path in the stash are spilled from the stash back to the ORAM. This allows us to spill vpn 250 from the stash to the ORAM, vpn 385 is "pushed down" while 88 can only be stored at the root node because it has now been mapped to leaf 0. *From the perspective of the untrusted OS, it only sees the enclave read in encrypted data stored in buckets labeled 6, 5 and 2 and write re-encrypted data back to the same buckets. The specific information about which pages were moved, which pages stayed in the same location, which new pages were spilled to the ORAM, etc. are all opaque to the OS due to randomized (re-)encryption.*

2.3.4 Understanding Security of Path ORAM

Why is Path ORAM secure? First, let us consider the access pattern of the algorithm. Each access is a sequence of reads and writes to all buckets on the path from the root to some leaf. Specifically which leaf is accessed depends on the position map. All entries in the position map are obtained from a cryptographically secure pseudorandom number generator and are independent of the enclave program's page-faults.

[2] The ORAM "fails" when an ORAM access cannot be completed because client-side storage is full and the corresponding path on the server is also full.

How about the actual data itself? Path ORAM requires that all blocks stored in untrusted memory be (computationally) indistinguishable to the adversary (OS). This means that the OS should not be able to distinguish between dummy pages and real pages. It must also not be able to distinguish between pages that were read into the stash and the (possibly different, possibly identical) pages that are written back from the stash. Further, among the many pages that are read into the enclave, the OS must not be able to distinguish between the real page that was requested, unrelated real pages, and dummy pages. Authentication, randomized encryption and re-encryption of all pages on a particular path with each access are needed to ensure that the above security properties are satisfied.

Path ORAM brings in a whole path from the root to some leaf in the tree. The application itself requires only the page it has requested. If Path ORAM did not authenticate all the pages which were fetched, the untrusted OS could tamper with some subset of pages and check whether the tampering is detected. If no error is raised, the OS can infer that tampered pages were not accessed; this breaks indistinguishability.

Second, we need randomized encryption as well as re-encryption of all pages on the path with each ORAM access. With each access, the host OS sees that the ORAM fetches a number of pages into the enclave along a path from the root to some leaf, and writes a number of pages back into the tree along the same path. Suppose identical data is fetched into the enclave and then written back with the exception of a single page that was modified. This lets the OS determine exactly which page was accessed, this also breaks indistinguishability.

In practice, these pitfalls are easily avoided by the use of authenticated encryption modes (e.g. Galois Counter Mode) with a fresh initialization vector (IV) for each (re)-encryption.

2.4 Oblivious Page Access Module (OPAM)

Aga and Narayanasamy introduced the Oblivious Page Access Module (OPAM) as part of InvisiPage [5]. OPAM builds on a Path ORAM substrate and introduces several optimizations to improve performance in the context of demand paging for enclaves. We do not review all the details of OPAM due to a lack of space and instead discuss its most important security-relevant optimizations. OPAM stores additional metadata corresponding to each tree node. Each access fetches this metadata inside the enclave and analyzes it to determine exactly which level of the tree contains the page of interest and fetches/evicts only this particular page in/out of the enclave. The other pages, which may need to be re-organized in order to prevent ORAM failure are shuffled *without re-encryption* using an enclave-specific key. This optimization reduces the cost of each ORAM access from reading, decrypting, re-encrypting and writing $lg\ N$ pages to reading and writing $lg\ N$ pages but decrypting/encrypting only one page.

Security Analysis of OPAM

OPAM retains the same access template as Path ORAM in that a randomly-chosen path from the root to leaf in the tree is accessed with each client operation. Therefore, the path itself does not leak any information. In contrast to Path ORAM, OPAM does allow the host OS to distinguish between page evictions (client writes) and page fetches (client reads). This leaks one bit of information with each access and does not appear to be large enough to be exploitable. The most significant leak in OPAM is due to the fact that only one page is fetched into the enclave and all pages of others along this particular path are merely shuffled by the untrusted host OS. Since the entire path is not re-encrypted with each access, the untrusted host OS can distinguish between pages which are moved up/down the ORAM tree, pages which are read into the enclave, and pages which are newly spilled from the enclave to the ORAM. The primary difference in terms of obeservability of vanilla PATH ORAM and Invisipage is that in PATH ORAM, the adversary is able to observe a random set of pages(fully encrypted) getting read and written, whereas in Invisipage the adversary can see only one page getting read or written. So this leak information about the page getting read or written.

We refer to the number of page-faults that occur between a page eviction and a subsequent re-fetch of the page as the page's reuse distance. *The above analysis shows that OPAM allows an honest-but-curious host OS to learn the reuse distances of all pages evicted from the enclave.* While this may appear to be a benign leak, our experiments show that reuse distances are effective in exfiltrating secrets from enclaves.

3 Attacks on InvisiPage/OPAM

We now describe the main attack introduced by this paper: the page reuse distance attack. We start with a description of the threat model used in this section. We then provide an intuitive description of why re-use distances leak information and describe details of the attack.

3.1 Threat Model

We assume the standard enclave threat model corresponding to a software attacker. Only the enclave itself and the hardware platform are trusted [5, 45]. All other software on the system, including privileged software such as OS and other enclaves, are untrusted. The adversary can observe all invocations of untrusted code made by the enclave and read and modify all the untrusted code and data stored in untrusted memory. The adversary can also observe all the pages that are evicted by the enclave to the untrusted memory and read from the untrusted memory back to the enclave memory.

We do not consider microarchitectural side channels (e.g. cache-based side channels [27, 32, 38]). These can be miti-

gated by isolating shared microarchitectural resources across protection domains as done in Sanctum [14], MI6 [8], etc. We do not consider the timing side channel for page-faults; that is we do not consider attacks based on measuring the wall-clock time between successive page-faults. The timing channel can be blocked by making ORAM accesses at a fixed rate [19].

3.2 Why are Reuse Distances Leaky?

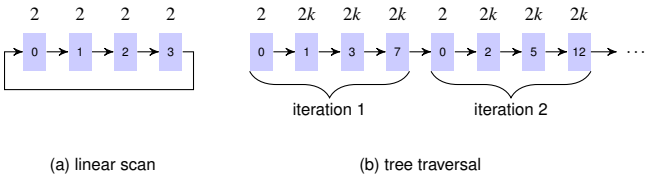

Figure 4: Illustrating relation between reuse distances and access patterns. The numbers within the boxes are page addresses while the labels above the boxes show the reuse distance for that particular address.

Figure 4 shows two kinds of access patterns. In Figure 4(a), the program repeatedly scans over an array. This results in the linear access pattern shown where four pages labeled from 0 to 3 are repeatedly accessed. Suppose the enclave has only two pages in its working set and the LRU page replacement algorithm is used. When the access to page 2 is made, page 0 will need to be evicted. Two page-faults later, when page 0 is accessed again, it will be fetched back into the enclave. For this access pattern, the reuse distance of every page is 2. In contrast, Figure 4(b) shows the access pattern for a program that repeatedly traverses a binary tree. For the tree traversal, the reuse distance of the root is 2 because the root node (labeled 0) is accessed in every iteration. The reuse distances for non-root pages are multiples of 2 because these may or may not be accessed in successive iterations.

Therefore, if we have two programs running with an enclave – one performing a linear scan repeatedly and the other performing a tree traversal – we can distinguish between them by observing the reuse distances of pages evicted from the enclave. *The reuse distance attack exploits this insight by training a convolutional neural network sequence classifier to fingerprint victim enclaves using their page reuse distances.*

3.3 Attacking OPAM Using Reuse Distances

The OPAM/untrusted memory interface consists of three operations: page evictions, page fetches and page shuffles. For simplicity, we assume the OS can directly intercept calls to `pageRead` and `pageWrite` in order to track the reuse distances of all pages as shown in Algorithm 2. The attack is not restricted to direct interception as discussed in the next subsection. The algorithm has two main procedures:

Algorithm 2 Tracking Reuse Distances

```
    global reuseDist, trace
1:  procedure OnPageWrite(addr, data)
2:      call IncrDist()        ▷ increment all reuse distances
3:      k ← SHA256(addr || data)
4:      reuseDist[k] ← 0
5:  end procedure
6:  procedure OnPageRead(addr, data)
7:      call IncrDist()        ▷ increment all reuse distances
8:      k ← SHA256(addr || data)
9:      trace ← trace ++ reuseDist[k]
10:     delete reuseDist[k]
11: end procedure
```

OnPageWrite and OnPageRead. These procedures are called by the adversarial OS whenever calls to `pageWrite` and `pageRead` respectively are intercepted. OnPageWrite initializes the reuse distance of a freshly evicted page to be zero. Both procedures increment the reuse distance of all pages currently in the ORAM when a page is read/written. The array `trace` contains a sequence of reuse distances and OnPageRead adds the reuse distance of the page being fetched to `trace` when that page is fetched from the OS into the enclave. This array is fed to a sequence classifier in order to exfiltrate secrets from the enclave application.

Why are page evictions/fetches adversary observable? The obvious (and perhaps cleanest) implementation of the `pageRead` and `pageWrite` operations is to invoke them by calling into the untrusted OS. In this case, they will obviously be visible to adversarial OS which can intercept and redirect these calls. Even if these operations are performed from within the enclave, their effect can be observed using the flush+reload techniques described in [48, 50]. Enclave side channel protections (e.g. [8, 14, 39]) *cannot* prevent this leakage. These protect *enclave* memory; but the backing store is *not* part of the enclave and necessarily OS-accessible.

3.4 The Level Tracking Attack

Can InvisiPage/OPAM be made secure by just re-encrypting all pages upon each access? Since OPAM has dataless stash, re-encrypting a page would need bringing the page into the enclave, re-encrypting it and writing it back. On the other hand, the page being accessed will either be read or written (depending on access type) but not both. Thus, re-encryption does not help preventing hiding of access type. In this section, we consider only the level of the accessed page in the OPAM tree and show that it alone is enough to leak information about the application.

It turns out that level at which data is stored in an ORAM tree loosely correlates with the age (aka reuse distance) of the data. Why does the level leak information? Each ORAM access (at least in principle) brings in the entire path into the

stash, re-encrypts each of these pages and writes data from the stash back greedily as close to the leaf as possible. In OPAM, these operations are performed without necessarily reading the path into the enclave, but the effective movement of data is the same. This results in "pushing" older pages towards lower (closer-to-leaf) levels of the tree.

Thus, the level of the page accessed in the OPAM tree leaks information about the age of the accessed page. This leads to a straightforward extension of the reuse distance attack, where instead of collecting a trace of reuse distance, we collect a trace of access levels. [3] We show that this trace is sufficient to identify secret applications, albeit with lower confidence.

4 Covert Channels Using Reuse Distances

In this section we show that reuse distances can be used as a covert channel to transmit information from a sandboxed enclave to the outside world. Platforms like Ryoan [26] and /CONFIDENTIAL [37, 43] aim to isolate untrusted enclave applications using a trusted enclave runtime. For example, Ryoan can be used for private genomic data analysis within an untrusted enclave and guarantees that only a binary output corresponding to the result of the analysis is revealed. These security guarantees are enforced using ideas similar to software fault isolation by restricting enclave access to non-enclave memory, and enclave OCALLs (external calls) to pre-defined trusted API functions.

The reuse distance leakage of OPAM provides such applications a covert channel to leak secret information (e.g. the input genome data) by carefully engineering their access patterns to cause a particular sequence of page-faults.

4.1 Threat Model

Our threat model here is similar to the one in Section 3.1 with the additional qualification that enclave application tries to collude with the host operating system to leak sensitive input data to the enclave The host OS is aware of the encoding used by the enclave application to transmit information via the reuse distance cover channel. Figure 5 shows attack model.

4.2 Encoding information in Reuse Distances

Assume we have an n-bit message to transmit and we are using a k-ary encoding for this message. Since the enclave is colluding with the OS, we assume the number of physical pages allocated to the enclave is $P = \lceil n/log_2(k) \rceil$; P is the length of a k-ary encoding of the message. The enclave allocates a buffer of size $B = 2 \times P \times k$ pages and accesses all the pages in this buffer in sequence. This causes the eviction of $P \times (2 \times k - 1)$ pages from the enclave. We can now leak a k-ary encoded message (k and the message length

[3] Please see Appendix A for details on how the levels are leaked

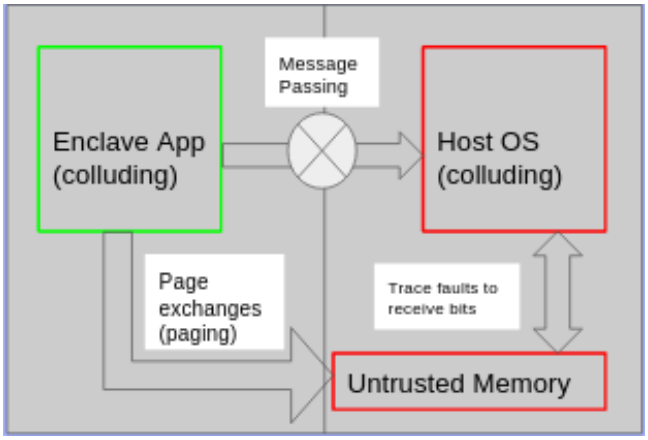

Figure 5: Covert Channel Attack Model.

is known to host OS) as follows. Each k-ary digit d is encoded by accessing a page with virtual page numbers in the range $[(k-d-1)*2P, (k-d)*2P)$, which corresponds to the reuse-distance $[d*2P, (d+1)*2P)$.

The host OS can trace the sequence of reads, writes and reuse distances by the enclave and later process it to extract the message. We note that this is one possible encoding and depending on the data to be leaked, much more efficient encodings can be used. We verified that this attack works with replacement policies like FIFO, LRU, LFU, CLOCK.

Example Encoding: Suppose the application wants to transmit a message 1011 using a binary encoding. The length $n = 4$ and $k = 2$. The enclave will allocate a buffer with $B = 2 \times 4 \times 2 = 16$ virtual pages. The runtime uses FIFO for page replacement and has been allocated $P = 4$ enclave physical pages to the application. First all the pages are accessed in sequence which generates OS observable sequence of page evictions to pages 1 to 12 respectively. To transmit the first bit of the message, which is 1, the application accesses a page with reuse distance in range $[8, 16)$. To transmit the second bit, the enclave accesses a page with reuse distance in the range $[0, 8)$. The third and fourth bit are also transmitted similarly, resulting in the access sequence $\langle 1, 8, 2, 3 \rangle$. This can be decoded by the OS as being the message 1011.

5 Evaluation

This section presents our experimental evaluation of the reuse distance and level-tracking attacks, and the reuse distance covert channel.

5.1 Methodology

This subsection briefly describes the experimental infrastructure, methodology and the benchmarks used in this paper.

5.1.1 Experimental Infrastructure

Our experiments were conducted on the open source Keystone Enclave Platform [3, 30] where we implemented InvisiPage/OPAM. The Keystone runtime is written in C and our modifications are ≈ 2800 SLoC. We used RISC-V QEMU [3] to conduct these experiments in emulation mode. We have verified via experiments with a HiFive Unleashed board [1] that results on emulation are representative of results on the actual hardware. To perform the reuse distance attack, we instrumented the untrusted components of InvisiPage and the encrypted address handlers to collect the reuse distance trace as described in Algorithm 2.[4] We used both FIFO and CLOCK replacement in our experiments.We intend to open source the full experimental framework at the time of publication.

5.1.2 Benchmarks

We created a benchmark suite consisting of five applications and three microbenchmarks chosen from diverse domains. We picked applications satisfying the following criteria. Applications amenable to enclaved execution, usually as privacy-preserving versions of cloud services. Applications that work in batch mode: they only receive input once at the beginning of execution and produce output at the end of execution. Since enclaves must proxy all their input/output via the host OS, it is much easier to identify applications that interact with the OS multiple times in different ways by just fingerprinting these I/O interactions. We focus on the more challenging scenario where the encrypted input is given only once and encrypted output is also produced only once.

Applications used were the following.

1. `avl`, `sha` and `mat_mul`: The latter two are from Keystone's source code. `avl` is from [36].

2. `picosat` is a solver for propositional satisfiability by Biere et al. [7]. SAT solvers are at the core of modern formal verification techniques and we use `picosat` as a proxy for privacy-preserving cloud-based verification platforms.

3. `lbm`: This fluid dynamics application is taken from SPEC CPU 2017. It is representative of proprietary scientific applications being executed on untrusted clouds.

4. `Recommender`: This is a product recommendation engine built using collaborative filtering by Hamrouni et al. [22]. It as an exemplar of the class of privacy-preserving cloud-based machine learning frameworks.

5. `libjpeg`: This application performs JPEG encoding. It is interesting because it is an example of a privacy-preserving image processing application implemented using enclaves. This application was also studied by Xu et al. [49].

6. `xz`: This is also taken from SPEC CPU 2017 and is an example of data compression application that is likely to be commonly used in cloud-based services.

We had to modify the benchmarks for the Keystone environment. These changes consist of about 1200 lines of code.

5.2 Attack Evaluation

How can the leakage of reuse distances and levels accessed be used to infer confidential applications? There are two possibilities: (a) given a secret application running within an enclave, the OS may use reuse distances or level access sequence to try to identify what application this is, and (b), given a known application operating on secret data, an attacker can attempt to use reuse distances or level access sequence to infer some properties of the secret input to the application.

5.2.1 Experimental Setup

We executed these applications with a large number of (approximately 50-200) different inputs and collected traces of (a) reuse distances as described in Algorithm 2, or (b) tree level access data as described in Section 4.2. Data was divided into training and test sets in a 3:1 ratio. To remove any bias in the classifier due to the specific choice of training and test datasets, the accuracy evaluation is repeated 10 times with the random splits of the data into training and test datasets. The reuse distance trace is used as the input feature to a CNN (Convolutional Neural Network) Sequence Classifier. The classifier was implemented in Python 3 using the `sklearn` [4] and `keras` [2] frameworks and is only a few hundred lines of code. The traces are collected by observing reuse distances for the first F OPAM accesses and then feeding this sequence to the classifier. We experiment with three values of F: 5k, 50k and 100k.

5.2.2 Secret Application Identification With Reuse Distances

We evaluated the accuracy of the classifier in identifying applications given only a trace of reuse distances obtained when

[4] Note that our implementation of InvisiPage/OPAM does not include the EPC-lite region of memory. EPC-lite is proposed by InvisiPage as a region of memory that provides page-level memory encryption as opposed to cache-block level memory encryption implemented in the EPC. The former has lower overheads for encryption and authentication assuming accesses occur only at the granularity of pages. The Freedom U540 does not have hardware support for memory encryption and our threat model does not include a hardware attacker, so separating EPC and EPC-lite does not reduce overheads.

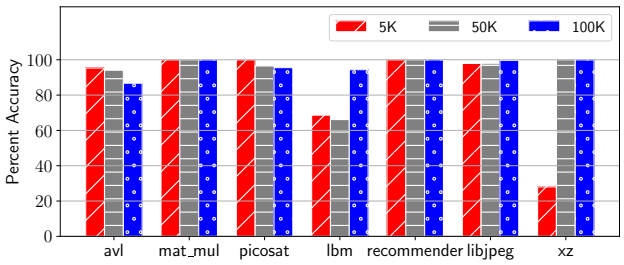

Figure 6: Classification accuracy using reuse distances for OPAM for varying trace lengths.

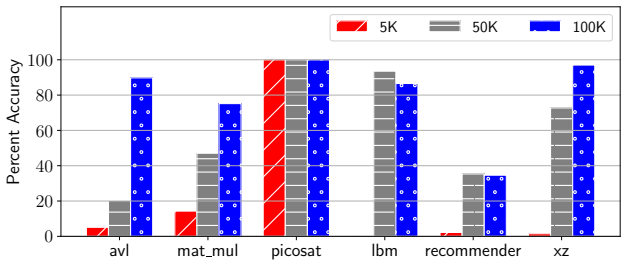

Figure 7: Classification accuracy using levels for OPAM for varying trace lengths.

executing one of the applications the CNN has been trained on for a new input that the classifier has never seen before. Figure 6 shows the classification accuracy for each of the seven applications while observing the first 5k, 50k and 100k page-faults. Average classification accuracy for these three values is 85%, 94% and 96% respectively. Average classification accuracy usually increases with trace length – this is expected because with more page-faults, we have more information about reuse distances, and hence access patterns.

5.2.3 Secret Data Identification using Reuse Distances

To see how effective the reuse distance attack is in identifying secret data within a known application, we ran `picosat` on two sets of formulas from benchmarks in the main and random tracks of the SAT Competition 2017 [23]. Using similar experiments as in the previous subsection, we evaluated how effective the classifier is in predicting the source of the benchmark for the reuse distances obtained from a new formula given to `picosat`. This could be predicted with $\geq 84\%$ accuracy. SAT research has shown that random instances produce different behavior in SAT solvers than instances generated from "real-world" problems [15]. This difference in behavior leaks through reuse distances.

5.2.4 Secret Application Identification Using Levels

We evaluated the accuracy of the classifier in identifying applications given only a trace of levels. We train a CNN sequence classifier on the traces obtained during application execution. Figure 7 shows the classification accuracy for each of the six applications on previously unseen traces for traces of first 5k, 50k and 100k page-faults.

Average classification accuracy for these three values is 19.82%, 62.45% and 85% respectively. Classification accuracy generally increases with increase in trace length. For 5k traces, the classification accuracy is just 19.82%, almost similar to what one could get with random guessing. For most applications the memory required is greater than the memory available within the enclave, during the initialization, the applications allocate pages and the pages that could not fit in the enclave memory are evicted to the OPAM tree. As a result most of the accesses in the beginning are writes to the lower levels of the tree (as the tree is initially empty), irrespective of the application. Thus, all the applications show similar access patterns (when only level of access is considered) for first few page-faults. As a result the classifier performs poorly for traces of length 5k. However, as the trace lengths increase the access pattern starts diverging and the classifier shows improved classification accuracy. However, Figure 6 and 7 shows that some applications (like lbm) experience slightly reduced identification accuracy for increasing trace lengths. This is because some applications have an intermediate phase of execution with similar access patterns (e.g. initializing a large array); this occasionally confuses the classifier.

5.2.5 Setup For Covert Channel Using Reuse Distances

We generated random binary messages of various lengths(n), used k-ary encoding and transmitted from within the enclave to the host OS as described in Section 4. Figure 8 shows results for five different values of k:2, 4, 8, 16 and 32. The corresponding values of n are: 256, 512, 768, 1024, 1280 respectively. In each case, we calculate the wall clock time and the number of bits leaked. A comparison is done between the leaked message and the original message to ensure accuracy.

5.2.6 Reuse Distance Covert Channel Analysis

Figure 8 shows how the bandwidth changes with the arity of the message. We see a peak bandwidth with arity 4. As we increase k, more data (bits) is transmitted with each page fault, but the number of page-faults required to setup the algorithm also increases. Our experiments show that for our setup, choosing k = 4 gives highest bandwidth. As we increase k, the overheads associated with increased number of initial page-faults/evictions (given by $P \times (2 \times k - 1)$ and 12 in section 4.2) dominate and we see a steady decline in transmission

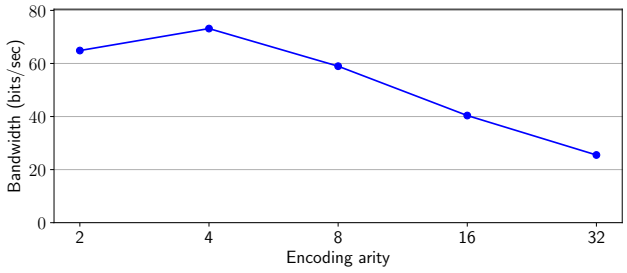

Figure 8: Bandwidth vs Message arity.

bandwidth. The total number of bits transmitted in the entire process is independant of the encoding.

6 Related Work

Page-fault based attacks on enclaves were introduced by the seminal work of Xu et al. [49]. Shinde et al. [42] also used the page-fault side channel to attack OpenSSL. Mitigations to this include T-SGX [41] and Apparition [16]. T-SGX uses transactional memory primitives to determine whether an attack is underway by counting the number of page-faults that occur within a transaction. T-SGX was defeated by Van Bulck et al. [48] who observed accesses by monitoring the accessed/dirty bits of the page table directly, as opposed to marking the page not present and then waiting for a fault to occur. Sanctum [14], Keystone [29, 30] and InvisiPage [5] have attempted to address the page-fault side channel by keeping enclave page tables private.

7 Conclusion

This paper introduced a new side channel attack called the reuse distance attack on demand paging for enclave platforms. Our attack is able to infer confidential information about an enclave's execution by examining its page-faults and is applicable to the state-of-the-art approach to secure paging in enclaves: InvisiPage. Further, we showed that fixing InvisiPage/OPAM is not as simple as just re-encrypting all path accesses. InvisiPage explicitly leaks the level of the tree from which an access is made, and as we showed with the level tracking attack, this leak alone is sufficient to defeat InvisiPage. Although we do not show results due to a lack of space, our experiments with Path ORAM showed that although a vanilla Path ORAM is secure, it also about an order of magnitude slower than InvisiPage. All of the above implies that performant and secure demand paging in enclaves remains an open problem.

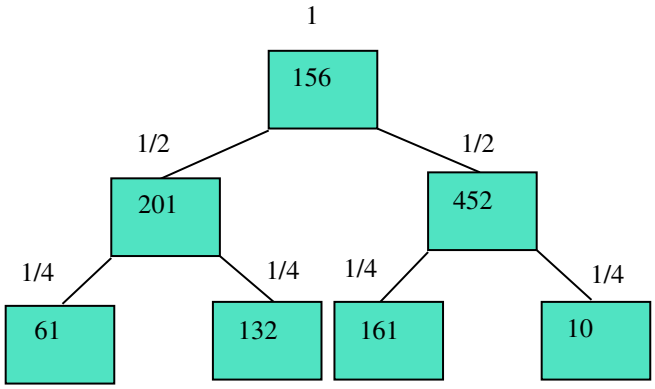

Figure 9: Sample OPAM tree to illustrate level leakage. The numbers inside the nodes are the o-vpns assigned to it and the label denote the fraction of paths from the root to leaves containing the node.

8 Acknowledgements

We thank Dr. Daniel Gruss and the anonymous reviewers for their valuable comments. This work is partially supported by Science and Engineering Research Board, Department of Science and Technology with sanction letter number sb/s2/rjn-057/2018.

Appendix

A Level Leakage

Figure 9 shows an OPAM tree. OPAM assigns static o-vpns to each node in the tree. We assume that the o-vpns assigned to each node are random and not correlated with the node's position in the tree. On every access, the OPAM runtime passes a list of o-vpns (corresponding to all nodes from root to a leaf) to the OS, one of which is read/written while the others are shuffled. Let's assume that the OPAM makes 5 access during the runtime and gives the following o-vpns to the OS: (156, 61, 201), (452, 156, 161), (10, 156, 452), (201, 132, 156) and (10, 156, 452). The list of o-vpns can be used to determine the mapping between o-vpns and the tree nodes.

We remove all duplicate sequences and get the following 4 sequences: (156, 61, 201), (452, 156, 161), (10, 156, 452) and (201, 132, 156). These four sequences correspond to different paths from the root to each of the leaves. Since the root node lies in all the paths, the o-vpn accessed in all the sequences must correspond to the root. This means that o-vpn 156 is the root. Similarly, the o-vpns accessed in only one path must correspond to the leaves. So 61, 132, 161, and 10 must be the leaves. Using a similar analysis, o-vpns corresponding to the other nodes can be found out. Once the attacker finds out the mapping, it can be used to find out the level of the tree nodes that are accessed.

References

[1] HiFive Unleashed, 2019. https://www.sifive.com/boards/hifive-unleashed.

[2] Keras: The Python Deep Learning Library, 2019. https://keras.io/.

[3] Keystone Enclave: QEMU + HiFive Unleashed, 2019. https://github.com/keystone-enclave/keystone.

[4] scikit-learn, 2019. https://scikit-learn.org/.

[5] Shaizeen Aga and Satish Narayanasamy. Invisipage: Oblivious demand paging for secure enclaves. In *Proceedings of the 46th International Symposium on Computer Architecture*, ISCA '19, page 372–384, New York, NY, USA, 2019. Association for Computing Machinery.

[6] Ittai Anati, Shay Gueron, Simon P Johnson, and Vincent R Scarlata. 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*, 2013.

[7] Armin Biere. Lingeling, plingeling, picosat and precosat at SAT Race 2010. In *FMV Report Series Technical Report*, volume 10, Aug 2010.

[8] Thomas Bourgeat, Ilia Lebedev, Andrew Wright, Sizhuo Zhang, Arvind, and Srinivas Devadas. MI6: secure enclaves in a speculative out-of-order processor. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, page 42–56, 2019.

[9] Ernie Brickell, Jan Camenisch, and Liqun Chen. Direct anonymous attestation. In *Proceedings of the 11th ACM Conference on Computer and Communications Security*, CCS '04, page 132–145, New York, NY, USA, 2004. Association for Computing Machinery.

[10] David Champagne and Ruby B Lee. Scalable architectural support for trusted software. In *2010 IEEE 16th International Symposium on High Performance Computer Architecture (HPCA)*, pages 1–12. IEEE, 2010.

[11] Guoxing Chen, Sanchuan Chen, Yuan Xiao, Yinqian Zhang, Zhiqiang Lin, and Ten H Lai. SGXpectre attacks: Stealing Intel secrets from SGX enclaves via speculative execution. *arXiv preprint arXiv:1802.09085*, 2018.

[12] Raymond Cheng, Fan Zhang, Jernej Kos, Warren He, Nicholas Hynes, Noah Johnson, Ari Juels, Andrew Miller, and Dawn Song. Ekiden: A platform for confidentiality-preserving, trustworthy, and performant smart contracts. In *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 185–200. IEEE, 2019.

[13] Victor Costan and Srinivas Devadas. Intel SGX Explained. Cryptology ePrint Archive, Report 2016/086, 2016. http://eprint.iacr.org/2016/086.

[14] Victor Costan, Ilia Lebedev, and Srinivas Devadas. Sanctum: Minimal hardware extensions for strong software isolation. In *Proceedings of the 25th USENIX Conference on Security Symposium*, SEC'16, page 857–874, USA, 2016. USENIX Association.

[15] James M. Crawford and Andrew B. Baker. Experimental Results on the Application of Satisfiability Algorithms to Scheduling Problems. In *Proceedings of the Twelfth National Conference on Artificial Intelligence (Vol. 2)*, AAAI'94, pages 1092–1097, Menlo Park, CA, USA, 1994. American Association for Artificial Intelligence.

[16] Xiaowan Dong, Zhuojia Shen, John Criswell, Alan L. Cox, and Sandhya Dwarkadas. Shielding software from privileged side-channel attacks. In *Proceedings of the 27th USENIX Conference on Security Symposium*, SEC'18, page 1441–1458, USA, 2018. USENIX Association.

[17] Dmitry Evtyushkin, Jesse Elwell, Meltem Ozsoy, Dmitry Ponomarev, Nael Abu Ghazaleh, and Ryan Riley. Iso-X: A flexible architecture for hardware-managed isolated execution. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 190–202, 2014.

[18] Dmitry Evtyushkin, Ryan Riley, Nael CSE Abu-Ghazaleh, Dmitry Ponomarev, et al. Branchscope: A new side-channel attack on directional branch predictor. In *ACM SIGPLAN Notices*, volume 53, pages 693–707. ACM, 2018.

[19] C. W. Fletchery, L. Ren, X. Yu, M. Van Dijk, O. Khan, and S. Devadas. Suppressing the oblivious ram timing channel while making information leakage and program efficiency trade-offs. In *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*, pages 213–224. IEEE, 2014.

[20] Oded Goldreich. Towards a theory of software protection and simulation by oblivious rams. In *Proceedings of the nineteenth annual ACM symposium on Theory of computing*, pages 182–194. ACM, 1987.

[21] Oded Goldreich and Rafail Ostrovsky. Software protection and simulation on oblivious RAMs. In *Journal of the ACM (JACM)*, volume 43. ACM, 1996.

[22] Ghassen Hamrouni. Recommender: A C library for product recommendations/suggestions using Collaborative Filtering, 2019. https://github.com/GHamrouni/Recommender.

[23] Marijn Heule, Matti Järvisalo, and Tomas Balyo. SAT competition 2017. *SAT*, 2017.

[24] Matthew Hoekstra, Reshma Lal, Pradeep Pappachan, Vinay Phegade, and Juan Del Cuvillo. Using Innovative Instructions to Create Trustworthy Software Solutions. In *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy, HASP*, 2013.

[25] Tyler Hunt, Congzheng Song, Reza Shokri, Vitaly Shmatikov, and Emmett Witchel. Chiron: Privacy-preserving machine learning as a service. *arXiv preprint arXiv:1803.05961*, 2018.

[26] Tyler Hunt, Zhiting Zhu, Yuanzhong Xu, Simon Peter, and Emmett Witchel. Ryoan: A distributed sandbox for untrusted computation on secret data. *ACM Transactions on Computer Systems (TOCS)*, 35(4):13, 2018.

[27] G. Irazoqui, T. Eisenbarth, and B. Sunar. S$a: A shared cache attack that works across cores and defies vm sandboxing – and its application to aes. In *2015 IEEE Symposium on Security and Privacy*, pages 591–604, 2015.

[28] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom. Spectre attacks: Exploiting speculative execution. In *2019 IEEE Symposium on Security and Privacy*, pages 1–19, 2019.

[29] Ilia Lebedev, Kyle Hogan, Jules Drean, David Kohlbrenner, Dayeol Lee, Krste Asanović, Dawn Song, and Srinivas Devadas. Sanctorum: A lightweight security monitor for secure enclaves. In *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1142–1147. IEEE, 2019.

[30] Dayeol Lee, David Kohlbrenner, Shweta Shinde, Dawn Song, and Krste Asanović. Keystone: An Open Framework for Architecting TEEs. *arXiv preprint arXiv:1907.10119*, 2019.

[31] Sangho Lee, Ming-Wei Shih, Prasun Gera, Taesoo Kim, Hyesoon Kim, and Marcus Peinado. Inferring fine-grained control flow inside SGX enclaves with branch shadowing. In *Proceedings of the 26th USENIX Conference on Security Symposium*, SEC'17, page 557–574, USA, 2017. USENIX Association.

[32] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee. Last-level cache side-channel attacks are practical. In *2015 IEEE Symposium on Security and Privacy*, pages 605–622, 2015.

[33] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V. Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday R. Savagaonkar. Innovative Instructions and Software Model for Isolated Execution. In

Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy, HASP, 2013.

[34] Olga Ohrimenko, Felix Schuster, Cedric Fournet, Aastha Mehta, Sebastian Nowozin, Kapil Vaswani, and Manuel Costa. Oblivious multi-party machine learning on trusted processors. In *Proceedings of the 25th USENIX Conference on Security Symposium*, SEC'16, pages 619–636, USA, 2016. USENIX Association.

[35] Christian Priebe, Kapil Vaswani, and Manuel Costa. Enclavedb: A secure database using SGX. In *2018 IEEE Symposium on Security and Privacy*, pages 264–278, 2018.

[36] The Crazy Programmer, 2019. https://www.thecrazyprogrammer.com/2014/03/c-program-for-avl-tree-implementation.html.

[37] Felix Schuster, Manuel Costa, Cédric Fournet, Christos Gkantsidis, Marcus Peinado, Gloria Mainar-Ruiz, and Mark Russinovich. Vc3: Trustworthy data analytics in the cloud using SGX. In *2015 IEEE Symposium on Security and Privacy*, pages 38–54, 2015.

[38] Michael Schwarz, Samuel Weiser, Daniel Gruss, Clémentine Maurice, and Stefan Mangard. Malware Guard Extension: Using SGX to Conceal Cache Attacks. *arXiv preprint arXiv:1702.08719*, 2017.

[39] Ali Shafiee, Akhila Gundu, Manjunath Shevgoor, Rajeev Balasubramonian, and Mohit Tiwari. Avoiding information leakage in the memory controller with fixed service policies. In *Proceedings of the 48th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 89–101, 2015.

[40] Elaine Shi, T-H Hubert Chan, Emil Stefanov, and Mingfei Li. Oblivious ram with o ((logn) 3) worst-case cost. In *International Conference on The Theory and Application of Cryptology and Information Security*, pages 197–214. Springer, 2011.

[41] Ming-Wei Shih, Sangho Lee, Taesoo Kim, and Marcus Peinado. T-SGX: Eradicating Controlled-Channel Attacks Against Enclave Programs. In *Proceedings of the 2017 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, February 2017.

[42] Shweta Shinde, Zheng Leong Chua, Viswesh Narayanan, and Prateek Saxena. Preventing page faults from telling your secrets. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*, ASIA CCS '16, page 317–328, New York, NY, USA, 2016. Association for Computing Machinery.

[43] Rohit Sinha, Manuel Costa, Akash Lal, Nuno P. Lopes, Sriram K. Rajamani, Sanjit A. Seshia, and Kapil Vaswani. A Design and Verification Methodology for Secure Isolated Regions. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13-17, 2016*, pages 665–681, 2016.

[44] Emil Stefanov, Marten Van Dijk, Elaine Shi, T.-H. Hubert Chan, Christopher Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. Path ORAM: An Extremely Simple Oblivious RAM Protocol. In *Journal of the ACM (JACM)*, volume 43. ACM, 2018.

[45] Pramod Subramanyan, Rohit Sinha, Ilia Lebedev, Srinivas Devadas, and Sanjit A. Seshia. A formal foundation for secure remote execution of enclaves. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, page 2435–2450, New York, NY, USA, 2017. Association for Computing Machinery.

[46] G Edward Suh, Dwaine Clarke, Blaise Gassend, Marten Van Dijk, and Srinivas Devadas. AEGIS: architecture for tamper-evident and tamper-resistant processing. In *Proceedings of the 17th annual international conference on Supercomputing*, pages 160–171. ACM, 2003.

[47] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. Foreshadow: Extracting the keys to the intel SGX kingdom with transient out-of-order execution. In *Proceedings of the 27th USENIX Conference on Security Symposium*, SEC'18, page 991–1008, USA, 2018. USENIX Association.

[48] Jo Van Bulck, Nico Weichbrodt, Rüdiger Kapitza, Frank Piessens, and Raoul Strackx. Telling your secrets without page faults: Stealthy page table-based attacks on enclaved execution. In *Proceedings of the 26th USENIX Conference on Security Symposium*, SEC'17, page 1041–1056, USA, 2017. USENIX Association.

[49] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *2015 IEEE Symposium on Security and Privacy*, pages 640–656, 2015.

[50] Yuval Yarom and Katrina Falkner. Flush+reload: A high resolution, low noise, l3 cache side-channel attack. In *Proceedings of the 23rd USENIX Conference on Security Symposium*, SEC'14, page 719–732, USA, 2014. USENIX Association.

[51] Wenting Zheng, Ankur Dave, Jethro G. Beekman, Raluca Ada Popa, Joseph E. Gonzalez, and Ion Stoica. Opaque: An oblivious and encrypted distributed analytics platform. In *Proceedings of the 14th USENIX Conference on Networked Systems Design and Implementation*, NSDI'17, page 283–298, USA, 2017. USENIX Association.