DR.SGX: Automated and Adjustable Side-Channel Protection for SGX using Data Location Randomization

Ferdinand Brasser
Technische Universität Darmstadt
ferdinand.brasser@trust.tu-darmstadt.de

Srdjan Capkun
ETH Zurich
srdjan.capkun@inf.ethz.ch

Alexandra Dmitrienko
University of Würzburg
alexandra.dmitrienko@uni-wuerzburg.de

Tommaso Frassetto
Technische Universität Darmstadt
tommaso.frassetto@trust.tu-darmstadt.de

Kari Kostiainen
ETH Zurich
kari.kostiainen@inf.ethz.ch

Ahmad-Reza Sadeghi
Technische Universität Darmstadt
ahmad.sadeghi@trust.tu-darmstadt.de

ABSTRACT

Recent research has demonstrated that Intel's SGX is vulnerable to software-based side-channel attacks. In a common attack, the adversary monitors CPU caches to infer secret-dependent data accesses patterns. Known defenses have major limitations, as they require either error-prone developer assistance, incur extremely high runtime overhead, or prevent only specific attacks.

In this paper, we propose data location randomization as a novel defense against side-channel attacks that target data access patterns. Our goal is to break the link between the memory observations by the adversary and the actual data accesses by the victim. We design and implement a compiler-based tool called DR.SGX that instruments the enclave code, permuting data locations at fine granularity. To prevent correlation of repeated memory accesses we periodically re-randomize all enclave data. Our solution requires no developer assistance and strikes the balance between side-channel protection and performance based on an adjustable security parameter.

CCS CONCEPTS

• **Security and privacy → Side-channel analysis and countermeasures**; **Trusted computing**.

KEYWORDS

SGX; side channel defense; data randomization

ACM Reference Format:
Ferdinand Brasser, Srdjan Capkun, Alexandra Dmitrienko, Tommaso Frassetto, Kari Kostiainen, and Ahmad-Reza Sadeghi. 2019. DR.SGX: Automated and Adjustable Side-Channel Protection for SGX using Data Location Randomization. In *2019 Annual Computer Security Applications Conference (ACSAC '19), December 9–13, 2019, San Juan, PR, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3359789.3359809

ACSAC '19, December 9–13, 2019, San Juan, PR, USA

1 INTRODUCTION

Intel Software Guard Extensions (SGX) [18, 35] enable execution of security-critical application code, called *enclaves*, in isolation from the untrusted system software. SGX was designed to ensure confidentiality of enclave data and integrity of enclave execution and is used in a number of academic works [4, 9, 14, 15, 20, 26, 41, 58, 63, 70].

Recent research has, however, demonstrated that SGX isolation can be violated using software-based side-channel attacks. In SGX, memory management, including paging, is left to the untrusted OS [18]. By monitoring page usage, the OS can learn coarse-grained enclave control flow or data access patterns [72, 77]. Enclave data can also be inferred by monitoring CPU caches that are shared between the enclave and the untrusted software, enabling more fine-grained information leakage [11, 31, 32, 51, 64]. Such attacks can defeat one of the main benefits of SGX—the ability to compute over private data on an untrusted (cloud) platform.

The problem of side-channel leakage has been studied extensively. Oblivious RAM (ORAM) [69] and Oblivious Execution [44, 45, 48] are well-known defensive techniques. Obfuscuro [1] implements those techniques for SGX enclaves, hiding all access patterns. The main drawback is an extremely high runtime overhead (83× on average and up to 220×). Another common defense is manual code hardening that is typically used by developers of cryptographic algorithms to make their implementations side-channel resilient [12]. This defense is not easily applicable to enclaves written by developers who are not security experts. Recent research has also proposed SGX-specific defenses. T-SGX [66] and Déjá Vu [17] use the processor's transactional memory features to prevent attacks that interrupt the victim enclave repeatedly. Such features are available only in a subset of SGX processors and the defense only protects against attacks that leverage interrupts. Cloak [33] and Raccoon [59] hide memory accesses to developer-annotated enclave data, but relying on the developer to mark all (possibly non-obvious) secret data correctly can be very error-prone. In summary, all known defenses either impose extremely high runtime overhead, rely on the developer, require functionality that is not available in all CPUs, or mitigate only specific side channels.

Our goals and approach. In this paper we focus on information leakage caused by *data access* monitoring. Our goal is to provide an *automated* tool that provides side-channel protection without

developer assistance and enables an *adjustable* trade-off between security and performance.

We focus on data accesses, as they are the target of many recent SGX attacks [11, 31, 51, 64]. Preventing control flow leakage is also important, but an orthogonal problem to our work. We build an automated tool because, similar to the development of other software, not all enclave developers are security experts and many would fail to correctly use solutions that require identification of potentially subtle sources of leakage for manual annotation. Instead, our primary goal is to strike the balance between provided protection and performance. While our tool can be configured to prevent all the leakage, this would incur a prohibitive performance penalty for most applications. Instead, we aim to give a means to enclave developers to get the best possible protection for a given application and performance overhead.

The main idea of our approach is to randomize all data locations in the enclave's memory at fine granularity. The enclave generates a secret randomization key and based on that computes a permutation for every memory address. As a result, the adversary cannot map the observed (permuted) memory address to the actual address, regardless of the channel he uses to make observations [11, 31, 32, 51, 64, 72, 73, 77]. Because all data is randomized without the need to understand its structure or semantics, we call our approach *semantic-agnostic data randomization.*

Randomization is a well-known hardening technique, but our approach is different from the existing solutions that randomize code by leveraging its known structure, such as functions or blocks. Due to the well-known difficulty of C/C++ code analysis and pointer tracking, no similar structure is available for data [6]. Indeed, existing randomization tools like SGX-Shield [65] focus on randomizing the code and do not tackle the problem of data randomization. Thus, they cannot prevent attacks that exploit data accesses, such as [11, 31, 51, 64].

Challenges and results. Secure and practical realization of our approach imposes a number of technical challenges. The first challenge is secure and efficient permutation computation under adversarial monitoring. If the adversary is able to derive information from the process of address permutation, he can revert the randomization. The second challenge is efficiency — computing a permutation for every data access is expensive and causes a high overhead. The third problem is information leakage through repeated memory accesses. Although an individual access is effectively hidden from the adversary, repetitive access patterns may allow (permuted) address correlation and leakage, i.e., correlation attacks.

In this paper, we tackle the above mentioned challenges and design and implement a compiler-based tool called DR.SGX (***D**ata Location **R**andomization for **SGX***) that instruments enclave code at compile time such that all memory locations used to store enclave data (in the heap) are permuted at cache-line granularity during run time. We realize the permutation securely using small-domain encryption [5] and leveraging the CPU's hardware acceleration units (AES-NI). To address correlation attacks, our tool allows periodic re-randomization of enclave data: more aggressive re-randomization rates hide repeated memory access patterns better at the cost of higher run-time overhead.

The basic runtime overhead of DR.SGX is 4.36× without re-randomization. Using different re-randomization rates, we measured an overhead approximately between 5× and 11×. We acknowledge that this is a significant performance penalty, but emphasize that our solution is at least one order of magnitude faster than complete ORAM schemes like Obfuscuro [1]. Additionally, we note that this overhead only applies to the SGX enclave, which handles just the security-critical part of an application.

Our security evaluation reveals that the protection provided by DR.SGX depends on the target enclave. Enclaves where predictable data access patterns, like initialization routines, are soon followed by secret-dependent data accesses, require aggressive re-randomization to prevent leakage, incurring higher overhead. In a corner case, our solution can prevent any leakage by re-randomizing enclave memory after every memory access, effectively functioning as an ORAM implementation. However, enclaves where secret-dependent accesses do not happen (soon) after predictable accesses can be strongly protected with much lower overhead.

Contributions. This paper makes the following main contributions:

- *Novel approach.* We propose a novel approach called semantic-agnostic data randomization as a defense against side-channel attacks on SGX.
- *New tool.* We design and implement a tool called DR.SGX that instruments code to permute an enclave's data memory locations at cache-line granularity and re-randomize them repeatedly.
- *Evaluation.* We evaluate the performance of our system, analyze possible leakage, and show how previous attack targets can be protected.

The paper is organized as follows: Section 2 defines our problem. Section 3 presents our approach and Section 4 details on our implementation. We evaluate DR.SGX's performance in Section 5 and analyze its security in Section 6. Section 7 reviews related work, Section 8 provides discussion and Section 9 concludes the paper.

2 PROBLEM STATEMENT

In this work we focus on systems that provide an isolated execution environment that is implemented as an execution mode of the main CPU. In particular, the CPU's shared resources, like caches, are used by all execution modes of the CPU and thus are shared between isolation domains. Our work is targeted towards Intel SGX, however, the same model also applies to other architectures like ARM TrustZone [2] and SANCTUARY [10] or software-based isolation solutions [49].

Problem space. Side-channel attacks on software in general, and SGX in particular, come in many different forms. Any kind of resource use that is influenced by the software's execution and can be observed by the adversary can serve as a side channel. For instance, the use of electricity as well as effects thereof like electro-magnetic emission, or the use of shared CPU caches. In this work we focus on *software* side channels, i.e., such that are observable by a software program running on the target machine, precluding physical or hardware side-channel attacks.

In the realm of software side-channel attacks a number of distinct variants exist. On one hand, different shared resources can be used as a side channel, like the different caches of the CPU, or the virtual memory management. On the other hand, side-channel attacks can target different information, including sensitive access patterns to data as well as secret dependent code execution paths.

In this work we focus on software attacks that target *data accesses* and consider attacks aiming to infer the control flow of a program as an orthogonal problem. Our rationale is two-fold. First, many side-channel attacks on SGX have been based on data access patterns [11, 31, 51, 64]. Furthermore, our solution can be combined with protections against control flow leakage attacks, for example with the Zigzagger approach proposed by Lee et al. [42].

Adversary model. The adversary's goal is to extract sensitive information from an isolated execution environment (enclave) [3, 8] through cache side-channel attacks (including CPU-internal caches like the translation look-aside buffer [32]) and/or paging side-channel attack [72, 77]. Sensitive data in this context are not limited to cryptographic keys, which are the "classical" targets of side-channel attacks. Instead, sensitive data have to be seen much broader, for instance, when processing privacy-sensitive data in the cloud [11].

The adversary can freely configure and modify all software of the system, including privileged software like the operating system (OS). He knows the initial memory layout of the enclave, i.e., the code and initial data of the enclave. Furthermore, we assume that the adversary can initiate the enclave arbitrarily often.

However, the adversary cannot directly access the memory of the enclave. The internal processor state (e.g., the CPU registers) is inaccessible to the adversary, in the event of an interrupt the state is securely stored in an isolated memory region. The adversary cannot modify the code or initial data of the enclave, as enclave's integrity can be verified using remote attestation.

We consider our work orthogonal to the recently discovered platform vulnerabilities Meltdown [43] and Spectre [39] that leverage transient execution to read secrets across isolation boundaries. Although these vulnerabilities apply to SGX enclaves as well [16, 71], Intel has already issued security updates for SGX that address such attacks [16]. Also, SGX platform keys from unpatched (and thus potentially compromised) platforms can be identified at the time of attestation and revoked [16]. The more general problem of data-access driven side-channels is much harder to solve in architectures like SGX. DR.SGX addresses this latter and more difficult problem.

We assume the position of the attacker to be as strong as possible and therefore we will assume him to have a noise-free cache side-channel and to be able to obtain a "perfect cache trace" of the enclave. This means that he can observe all memory accesses of an enclave, e.g., using a cache attack technique such as Prime+Probe [54]. He can precisely determine which cache line has been used by the enclave and also the order in which the cache lines have been accessed. The adversary cannot extract information which is more fine grained than accesses to cache lines, i.e., the offset inside a cache line is not observable to him (see Section 8 for a discussion of possible attacks with finer granularity). Additionally, for each memory access, the adversary can gain information about the accessed memory pages of an enclave [72, 77].

More formally, trace $t = \{c_1, p_1\}, ..., \{c_n, p_n\}$ is an ordered list of side-channel observation pairs that capture every memory access that the victim enclave makes. In each observation pair, c_i is the part of the memory address that determines the cache line the accessed address gets mapped to and p_i is the part of the address that determines the accessed memory page. On current Intel CPUs the cache line size is 64 bytes, thus, the last six bits of an address are oblivious to the adversary.

Design goals. General statements about which memory accesses of a program could leak information are hard to make in practice. All memory accesses must be assumed to potentially leak information if the attacker can associate them with relevant data elements or structures. For the adversary it is sufficient to distinguish two memory locations to learn one bit of information. Those memory locations could be two different data structures, e.g., two variables, or different elements within the same data structure, e.g., different entries in a table. To protect all possible programs, the data structures of a program and the elements within data structures both need to be randomized.

The goal of our work is to provide a protection mechanism against side-channel attacks that can be applied to *arbitrary enclave programs without developer assistance*. In particular, the developer must not be required to follow any rules or guidelines for programming his application or add annotations to the source code. While annotating "critical" data in general helps improving the performance of most solutions, it is also very error-prone: especially in non-cryptographic applications, it is not always obvious which accesses to data objects might leak sensitive information. This is crucial as most software developers are not security experts and cannot comprehensively identify data that could leak information.

The goal of DR.SGX is to provide a trade-off between security and cost in the design space reaching from unprotected processes, over plain SGX enclaves, enclaves with DR.SGX to oblivious RAM (ORAM) solutions. On the one hand, plain SGX enclaves provide basic data protection with little performance penalty; on the other hand, schemes like Obfuscuro [1] that implement ORAM for every memory access impose very high performance overheads (83× on average and up to 220×). DR.SGX strives to protect enclaves better than plain SGX while keeping the performance overhead at least one order of magnitude lower than systems like Obfuscuro. DR.SGX's security parameter (the re-randomization window w: see Section 3) allows it to be configured to cover the spectrum between plain SGX and full ORAM for data accesses. With $w = 1$ DR.SGX implements ORAM, admittedly in a costly way. On the other hand, $w = \infty$ only randomizes the initial memory layout of an enclave, which can be sufficient for some enclaves; we discuss this scenario in Section 6. For most enclaves a window size between those two extremes can be chosen. We evaluate different windows sizes in Section 5.

3 DR.SGX

Our core idea is to break the link between side-channel observations made by an attacker and the sensitive information processed by the victim. Side-channel attacks inherently rely on the correlation between an observable effect and the data the attacker aims to extract. Our defense obfuscates the link between memory locations and

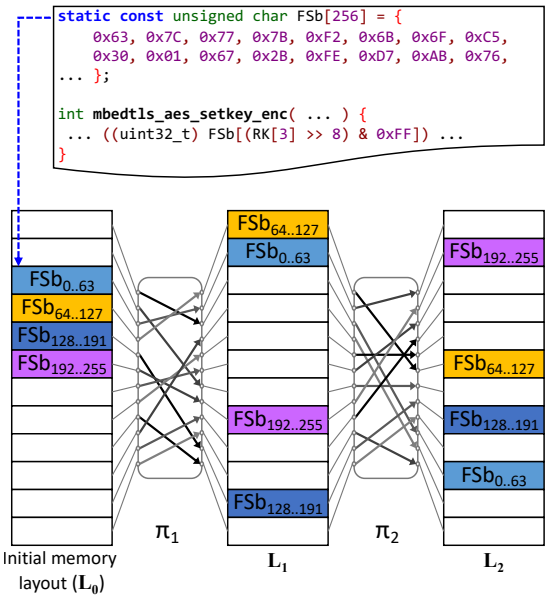

Figure 1: DR.SGX's memory block randomization splits large memory structures like arrays into small blocks and reorders them. During the run time of an enclave its memory layout are re-randomized using the permutation function π. Each memory block is the size of a cache line (64 B), i.e., the finest granularity observable by the adversary.

data elements. Data elements are located at randomized memory locations, so the adversary cannot deduce which data element was accessed from an observed memory access location. The adversary no longer learns *which* data element was accessed but only learns that *some* data element was accessed.

DR.SGX splits enclave memory into small blocks that are randomly reordered, resulting in an unpredictable memory layout from the adversary's point of view. Figure 1 illustrates the concept on the example of the S-box of an AES implementation. By default the S-box (FSb) is stored as an array in consecutive memory at a predictable location, shown on the left as initial memory layout L_0 in Figure 1. Through a cache side channel an adversary can observe which part of the S-box is accessed. Since the accesses to the S-box depend on the secret key the adversary can use this information to recover the key. However, the adversary cannot observe accesses to individual bytes of the S-box but only at the granularity of cache lines (64 bytes). DR.SGX divides *all* data memory of an enclave into blocks of cache line size, illustrated by the blocks forming L_0 in Figure 1. These blocks are reordered by a permutation function π_1, resulting in a randomized memory layout L_1. Throughout the runtime of an enclave the memory layout is constantly re-randomized, by applying a permutation function π_2 on L_1 a new and different memory layout L_2 is created. As a result, the memory locations and thus the cache lines corresponding to the S-box are frequently changing, hindering the adversary's ability to link observed (cache or paging) accesses to the S-box.

3.1 Requirements and Challenges

Below we describe the main challenges to tackle when implementing this idea.

Semantic gap. Providing side-channel protection through data randomization without developer assistance (e.g., code annotations) is a challenging task due to the semantic gap that is inherent to unsafe languages like C and C++. Currently C and C++ are the only programming languages officially supported in the software development kit (SDK) that Intel provides for the development of SGX enclaves.

Re-randomization. Randomizing the memory layout of a program once to prevent an adversary from learning which data has been accessed is not sufficient. The adversary can determine the relation of memory locations and data objects based on various information. For instance, the initialization of data structures can reveal data locations. In the example in Figure 1, the S-box is initialized during the creation of the enclave, however, other AES implementations initialize the S-box at run time which allows the adversary to learn the locations of all parts of the S-box array *after* the initial randomization of the memory layout. Similarly, access frequency can reveal the randomized location of data elements: if a particular object is accessed a predictable number of times the adversary can identify the object by finding the memory location that was accessed the expected numbers of times (frequency analysis). To thwart the adversary in recovering the randomized memory location of data objects, their locations need to be changed throughout the runtime, such that the adversary cannot link data accesses to data objects.

(Re-)randomization under attacker's observation. All memory-related actions of the attacked enclave can be observed by the adversary, including those required during the initial data randomization and during the re-randomization of the memory layout. The initial (un-randomized) memory layout is known to the adversary, i.e., he can monitor memory events while data is copied to its randomized locations. Similarly, if the adversary managed to recover information about the randomized memory layout L_n the adversary could link the re-randomization operations used to transfer data from L_n to L_{n+1} and thus also gain knowledge about the new layout L_{n+1}. Therefore, the randomization has to be done in such a way that its effects are not observable by the adversary.

3.2 DR.SGX Design

Our solution, a compiler-based tool called DR.SGX, addresses the design goals and challenges described above by randomizing *all* program data at fine granularity and re-randomizing the data continuously throughout the run time of the program.

Figure 2 shows the system view of DR.SGX. The trusted computing base (TCB) of an SGX enclave includes the CPU package and an isolated section of the main memory (RAM). However, the CPU caches, translation look-aside buffer (TLB) and the page tables are observable by the adversary. The data cache of the CPU can be used to observe memory access patterns of an enclave. On the other hand, the paging mechanism can be exploited in different ways to learn about memory reads and writes by an enclave. By observing cache conflicts in the TLB, the adversary learns which memory

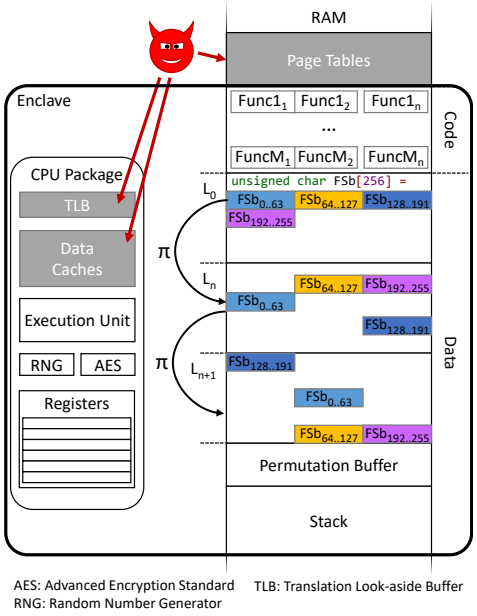

Figure 2: DR.SGX's system design. The main memory of an enclave is not directly accessible by the adversary, however, the adversary can observe memory access indirectly through cache and paging side channels. The CPU's internal state stored in registers and/or special function units (e.g., the AES engine) are not observable by the adversary.

pages were used. Additionally, the adversary has control over the page tables also allowing him to learn which memory pages an enclave accessed.

However, an SGX enclave also includes components that cannot be attacked through a software side channel. The CPU's registers and accesses to them cannot be observed by the adversary.[1] Also the execution unit and special function units, like the random number generator (RNG) or the AES engine, are secure when operating over registers. DR.SGX combines these parts and function units of SGX that are secure against side channels, to obfuscate main memory accesses to the adversary.

DR.SGX performs randomization at granularity of cache lines, the finest granularity at which the adversary can distinguish memory accesses (Section 2). Figure 2 shows how DR.SGX uses a random permutation function π to reorder the program's data in memory. Since the adversary cannot identify individual elements within a single cache line, accesses to the first array element (`FSb[0]`) and the 64th element (`FSb[63]`) are indistinguishable for the adversary. The randomization is based on secret values which are generated and only accessible *inside* the enclave and only processed by the hardware AES engine of the CPU. The CPU's AES engine holds all state and intermediate results in registers which are not observable by the adversary, hence, the adversary cannot learn about π through cache or paging side channels.

[1]The LazyFP [68] attack cannot be used on SGX enclaves, since the register state is cleaned by the processor before exiting the enclave.

DR.SGX randomizes global variables and the heap. The stack cannot be easily randomized, since the hardware expects it to be contiguous. Thus, variables on the stack larger than a cache line are moved to the heap, and replaced by a pointer on the stack. The remaining variables are protected using multiple memory layouts: for every function n variants are created (`Func1`$_1$, `Func1`$_2$, ..., `FuncM`$_n$ in Figure 2), all with different stack memory layouts. On every invocation of a function one of its n variants is chosen randomly.

The size of the memory region (heap) for the enclave's data is a parameter of the permutation function π (see Section 4).

Memory access instrumentation. DR.SGX performs randomization on cache line granularity for two reasons: (a) randomizing at finer granularity provides no security advantages, and (b) randomizing in a data structures aware fashion is impractical due to the semantic gap. Our randomization requires that *all* memory accesses are instrumented, which we ensure using a compiler pass. The program code determines the memory location (i.e., address) of the data in the original, un-randomized layout. Then, before the access is performed, the randomized location of that address is calculated. The data is then accessed in its new, randomized location.

As we will elaborate in later sections, the cost of performing the randomization calculation for *every* memory access is significant. We overcome this problem by implementing a "permutation buffer". The permutation buffer, similar to an address translation cache, holds the randomized locations of recently used data. Hence, for data locations stored in the permutation buffer the function π does not need to be recalculated. However, accesses to the permutation buffer itself must be protected from leaking information. Therefore the buffer is accessed in an oblivious way.

Initial randomization. The initial randomization of the enclave's data needs to be done in a way that cannot be observed by the adversary, to keep him from learning the randomization function π or the new memory layout. In particular, if the adversary can observe a read operation from the un-randomized initial memory layout and a subsequent write operation to a randomized address, he can link data structures to the randomized memory locations.

A general approach to break this linkage is to load a set of data into CPU registers (register operations cannot be tracked by the adversary) and write the data in a random fashion to their new locations. This approach, however, is limited in the amount of the data that can be loaded at once into registers, enabling the adversary to learn partial information about the randomized memory layout.

DR.SGX uses a randomization method which hides fine-grained (cache-line granularity) memory locations from the adversary. Specifically, we use *non-temporal writes* [36] that evade the CPU's caches, therefore the adversary cannot observe memory addresses written during the initial randomization. Although the non-temporal writes prevent accesses to the new memory layout L_1 from being cached, the adversary can still observe the written memory locations through the more coarse-grained paging side-channel (that is, the adversary's trace contains a page event p_i, but no cache event c_i for the non-temporal write). This allows him to know, for each memory block read from the previous memory layout L_0, to which memory page it was written in L_1. However, multiple cache lines are written to each page: assuming 4 KB pages, 64 cache-line-sized memory blocks will be written to the same page.

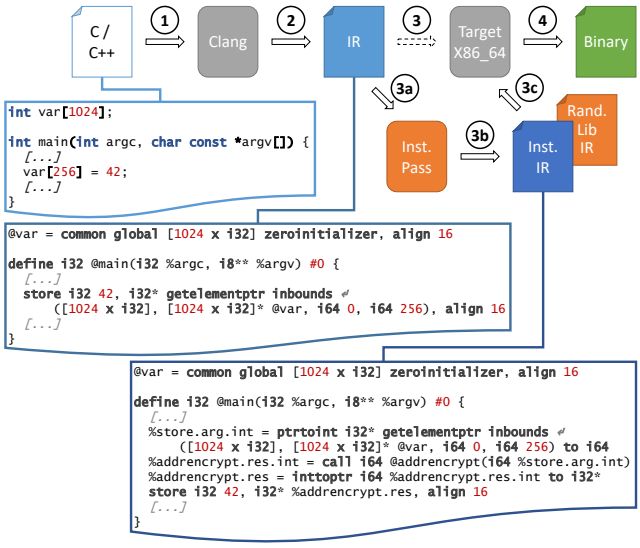

Figure 3: Code instrumentation with DR.SGX. Before each memory access the randomized memory address is calculated. The calculation is done by a function provided by the DR.SGX library (Rand. lib), which can be written in C / C++ and is included in the instrumented binary. The snippets show the instrumentation of a sample store instruction.

To hide this access pattern the initial randomization of DR.SGX accesses *all* memory pages of L_1 for each memory block that is moved, see Section 6.2.

DR.SGX continuously re-randomizes the memory layout. Starting from the initial memory layout L_0 a random permutation function π_1 is applied to derive the first randomized layout $L_1 = \pi_1(L_0)$. After a configurable window w the memory layout is re-randomized, applying π_2 to derive $L_2 = \pi_2(L_1)$.

Like with the initial randomization, the adversary (who can observe reads from L_n and writes to L_{n+1}) could link those operations to learn the relation between those memory layouts. Again, DR.SGX uses non-temporal writes to hide this information. In Section 6 we explain how a small number of re-randomization rounds hides the location of the element from the adversary completely.

4 DR.SGX IMPLEMENTATION

This section provides further details of DR.SGX. We explain how we implemented the key-components of DR.SGX: access instrumentation, permutation computation, initial randomization, permutation buffering, and re-randomization. Throughout this section we will refer to *data* memory regions or *data* memory accesses simply as memory regions and accesses (omitting *data*).

4.1 Memory Access Instrumentation

DR.SGX randomizes the memory locations of an SGX enclave's data. The enclave, however, has been developed targeting a linear (virtual) memory model. Therefore, each memory access of an enclave has to be instrumented to determine the correct randomized memory location of the data element that is meant to be accessed.

We extended the LLVM compiler [47] to instrument the enclave code, working at the intermediate representation (IR) level. Figure 3 shows on the top the high-level compile process of LLVM. A source file on the left is translated by the compiler front-end ①, Clang in the case of C/C++, into a LLVM intermediate representation (IR) ②. The IR is then translated by the back-end ③ into target architecture specific binary code ④, which in our case is Intel x86 64-bit. With DR.SGX the IR file is processed by a compiler pass (3a) that instruments all memory access instructions (instrumentation pass) before it is translated into machine code (3c). Furthermore, DR.SGX adds a small library (3b), which contains functions used to perform the randomization. This library can be written in a high-level language like C/C++ and is translated into IR as well.

Additionally, the instrumentation pass examines all allocations on the stack and transforms those which are larger than a single cache line into heap allocations. A pointer to the heap allocation is placed on the stack and the code is modified to access the heap allocation instead of accessing the stack.

Instrumentation example. Figure 3 illustrates the instrumentation of a write access to an array. The code snippet in the C file shows a write access to the 257-th element of an integer array `var`. The code snippet in the middle shows the intermediate representation (IR) of the write operation. The array is accessed by calculating the pointer to the 257-th element of the array, using the LLVM function `getelementptr`. The value 42 is then stored into this memory location. The instrumented IR is shown in the bottom code snippet. Again, a pointer to the 257-th element of `var` obtained using `getelementptr` and stored in the variable `store.arg.int`. However, before storing the value 42, `store.arg.int` is passed to the permutation function `addrencrypt`. The function returns the permuted location of the 257-th element of `var`, which gets cast from an integer value to a pointer value (`inttoptr`). The value 42 is then stored to the permuted location `addrencrypt.res`.

4.2 Random Permutation

DR.SGX uses run-time data randomization, which is required for both the unobservable initial randomization as well as the re-randomizations. This means that the randomized location of data must be recovered dynamically. Using a purely random permutation would require storing extensive meta-data, which would then need to be accessed in an unobservable way.[2] Therefore, DR.SGX uses a pseudo-random permutation function to determine the random location of data. This approach has two advantages: (1) collisions, i.e., different element mapped to the same location, are inherently avoided, and (2) randomized locations can be computed based on a non-secret algorithm and a key, which is *small* compared to the meta-data in the naive approach. However, the permutation function itself must be resilient against side-channel attacks, otherwise the adversary can learn the randomization secret and disclose the accessed memory locations.

We use small-domain encryption for our random permutation function. The domain size must be in the order of memory size used by the enclave employing DR.SGX (divided by the size of a

[2]The need to maintain meta-data is one of the main problems when using ORAM to protect SGX enclaves from side-channel attacks targeting the enclave's main memory accesses.

cache line). In particular, we use the FFX Format-Preserving Encryption scheme, which is based on a 10-round Feistel network [5]. As the underlying block cipher for FFX we used AES, for which the hardware acceleration extension AES-NI [36] is available in all SGX-enabled CPUs. AES-NI provides both good performance and resiliency against cache-based side-channel attacks.

Our implementation only supports single-threaded enclaves. However, standard software-engineering techniques can be employed to extend the support to multi-threaded enclaves. Only the re-randomization operations need to be synchronized between threads.

4.3 Initial Randomization

The initial randomization is particularly challenging since the adversary knows the initial memory layout of an enclave. If we used standard write operations to copy data from the initial data section L_0 to the randomized section L_1, the adversary would be able to learn the randomized layout.

In DR.SGX we use non-temporal write instructions to tackle this problem [36]. Non-temporal write instructions provide the processor with the meta-information that the data will not be used again soon by the program and it is not necessary to store them in the cache. On current Intel processors memory write operations using this instruction immediately affect the DRAM and are not buffered in the CPU's cache,[3] i.e., they are invisible to the adversary. Page-granularity side-channels information is hidden by accessing all heap memory pages for each block.

The secret keys we need as input to our random permutation are generated by the hardware random number generator *inside* the enclave. We use `rdseed` to obtain true random numbers from the CPU [36]. This way the adversary cannot influence or obtain the secret key.

4.4 Stack Randomization

DR.SGX uses the stack only for data elements that are smaller than a cache line, all other data are moved to the heap where they are subject to (re-)randomization. For the remaining data elements on the stack we use an approach inspired by the code randomization method introduced by Crane et al. [19]. The stack layout of each function is randomized by reordering the local variables on the stack. At compile time n variants of each function with different stack layouts are generated. At run time one function variant is chosen at random every time it is invoked. DR.SGX uses $n = 10$ variants for each function, as the empirical evaluation [19] suggests.

4.5 Permutation Buffer

Performing the calculation for the pseudo-random permutations is costly and needs to be performed for each memory access. To improve the performance we introduced a buffer for memory translations (Permutation Buffer in Figure 2). Permutation is performed at cache line granularity, i.e., all bytes in one cache line in L_0 are mapped as a single block. When this block is moved to L_1 it will, with high probability, be mapping to a different cache line, and to yet another cache line in L_2, and so on. On recent x86 processors a cache line is 64 bytes, thus, by storing the result no extra calculations are necessary for memory accesses that fall within the same cache line. Our buffer is currently 1 KB which allows for a direct-mapped storage of permutation results for 256 translations. To prevent leakage through our permutation buffer we access it in a way which is oblivious to the adversary. For each read operation to the buffer we simply access *all* CPU cache lines in our permutation buffer. Moreover, we randomize the location of the items in the permutation buffer by performing an xor operation with a randomly-generated value before determining which buffer item to use. The random value changes and the buffer is invalidated every time a re-randomization happens.

4.6 Re-Randomization

DR.SGX constantly re-randomizes the memory layout of an enclave. Figure 2 shows the overall memory layout. The blocks are copied from L_n to L_{n+1} in the same order as they appear in L_n, so the adversary only observes reads to every block in L_n, in order. Like in the initial permutation, non-temporal write operations are used to hide fine-grained writes.

For each cache-line-sized memory block in L_n, DR.SGX needs to compute the corresponding addresses in L_n and in L_{n+1}. Hence, the cost of re-randomization primarily comes from the permutation calculations required. However, the pipelining of AES instructions in the CPU makes encrypting multiple addresses together faster than encrypting them sequentially. This reduces the cost for the re-randomization and leads to better overall performance of DR.SGX.

5 PERFORMANCE EVALUATION

We evaluated the performance of DR.SGX using the benchmark suite Nbench [13].[4] We use Nbench because it has been previously used to analyze SGX performance [65], it relies only marginally on the file system, and it is relatively simple (5217 LoC), so it can easily be adapted to run inside an SGX enclave. The original version relies on timestamps to run each benchmark for an equal amount of time; since timestamps are not available in SGX enclaves we manually chose for each benchmark the lowest number of iterations that yielded a run time greater than 100 ms. We measured the run time of the benchmarks by briefly switching to the non-SGX mode and reading the hardware time stamp counter. We measured the overhead due to this mode switch and it is negligible compared to the overall run time. Our test system is equipped with an Intel Skylake i7-6700 processor clocked at 3.40 GHz, 128 MB Enclave Page Cache, running Ubuntu 14.04.4.

Memory overhead. The memory overhead of DR.SGX is mainly due to (1) heap randomization and (2) stack randomization. For the heap randomization two memory areas as large as the heap need to be reserved while the re-randomization is in progress. In our evaluations the heap size was set to values between 512 KB and 4 MB. Whenever the re-randomization is ongoing an additional 100% for the heap size is required. Stack randomization is based on providing n variants for each function. This increases the memory required

[3] We verified this behavior on a Skylake test system by issuing a non-temporal write followed by a read from the same cache line, and verifying that the read generates a cache miss on all three cache levels.

[4] Benchmarking SGX code can be challenging, since well-known benchmark suites rely on a number of features, including system calls, timestamps, and the file system, which are not directly available in SGX.

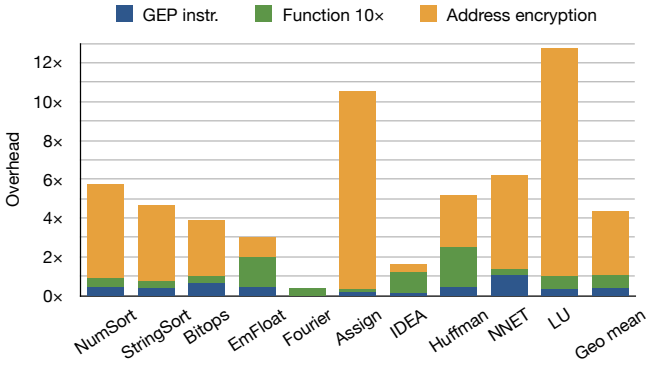

Figure 4: Overhead of each benchmark, using various subsets of DR.SGX.

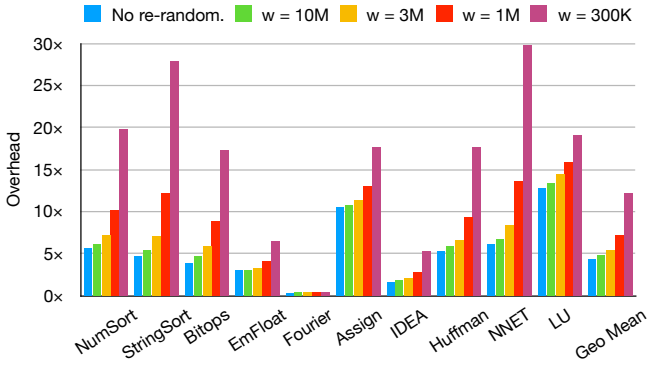

Figure 5: Overhead of each benchmark, with heap size h = 4 MB, without re-randomization and with various re-randomization windows w.

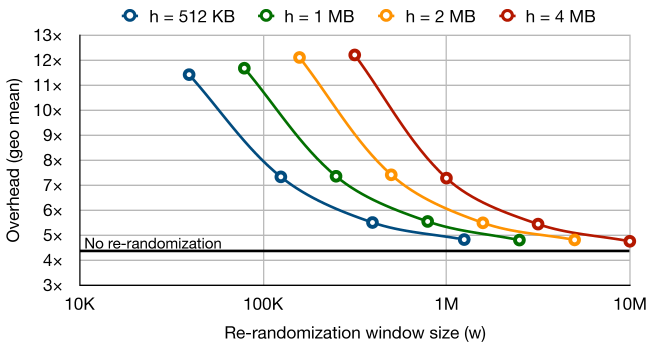

Figure 6: Geometric mean of the overheads, for various heap sizes h and various window sizes w. The black line represents the overhead without re-randomization.

for the code by factor n. We chose $n = 10$, thus the overhead is 10×. The size of the stack itself does not increase. For each invocation at run time only one of the function variants is used, i.e., the number of stack frames to be stored on the stack does not increase.

Runtime overhead of DR.SGX modifications. We evaluated DR.SGX with different subsets of its components active. To get a better understanding of the impact DR.SGX's individual components we ran all benchmarks multiple times activating one more components for every repetition. A breakdown of each component's overhead is shown in Figure 4.

We first tested our mechanism to move large stack allocations to the heap, i.e., replacing allocations on the stack larger than 63 bytes with calls to `malloc`. We measured a negligible overhead well below 1%, which is too small to be visible in Figure 4. Then, we tested the instrumentation of reads and writes (LLVM instruction `getelementptr`). In DR.SGX, instances of this instruction are followed by a call to our permutation function, unless the argument to the instruction is on the stack. In this test, the identity permutation function was used, which returns immediately. Therefore overhead reflects the impact of the instrumentation alone. We measured overheads between 0 and 102%, with a geometric mean of 39% (*GEP instr.* in Figure 4). Next, we added our stack randomization using function duplication. The geometric mean of the additional overhead is 46%, while the maximum is 135% (*Function 10×* in Figure 4). Finally, we tested our complete system (without periodic re-randomization). Overheads range between 0.39× and 12.77×, with a geometric mean of 4.36×. The benchmarks *Assign* and *LU* have the biggest overheads, 10.56× and 12.77× respectively, due to high miss rates in our permutation buffer (their miss rates are ~ 13× higher).

Runtime overhead of re-randomization. Next, we assessed the impact of various window sizes w and heap sizes h on the run time overhead and re-randomization window duration. We chose our heap size $h \in \{4\text{ MB}, 2\text{ MB}, 1\text{ MB}, 512\text{ KB}\}$ but other values are also possible. We measured the time required to perform a re-randomization by dividing the CPU cycles required by the processor's nominal speed, 3.4 GHz. The re-randomization requires 7.31 ms, 4.07 ms, 2.26 ms, and 1.26 ms respectively for $h = 4$ MB, $h = 2$ MB, $h = 1$ MB, $h = 512$ KB.

We first measured the run time overheads for $h = 4$ MB, $w \in \{10\text{ M}, 3\text{ M}, 1\text{ M}, 300\text{ K}\}$. In Figure 5, the left-most bars in each group represent the overhead without re-randomization (like Figure 4), with a geometric mean of 4.36×. Re-randomization every 10 million accesses ($w = 10$ M) increases the overhead slightly (geometric mean of 4.76×). Reducing the window to 3 M, 1 M, and 300 K brings the geometric mean of the overhead to 5.45×, 7.29×, and 12.21×.

We then measured the overhead for smaller heap sizes. We expected that halving both the heap size and the window size, i.e., re-randomizing a heap half as big twice as often, would yield similar performance results. Figure 6 shows the overhead depending on the window size for various heap sizes and confirms our intuition. Each line refers to a heap size twice as big as the line to its left. The black line at 4.26× is the overhead measured in the case without re-randomization and represents $\lim_{w\to\infty}$ of the overhead; in other words, increasing the values of w further would bring diminishing results.

Summary. The performance of our solution depends heavily on the user parameters. For example, the overhead is 4.8× for parameters h = 1MB and w = 2.5 M.

Developers and system administrators can adjust the parameters of DR.SGX based on the memory needs of their application

and the available computing resources. For example, if the deployment scenario requires 1 MB of heap memory and allows up to 8× overhead, the window size w can be set to 250 K for maximal re-randomization rate and security (see Figure 6). We consider this task of parameter tuning feasible for most developers. A typical developer may not be able to assess subtle sources of information leakage for correct source code annotation, but usually the developer knows the application's performance requirements and can set h and w accordingly.

Finally, we emphasize that in many SGX application scenarios, the overhead of the enclave (imposed by DR.SGX) is not directly the overhead of the entire application. For example, SGX-based applications that perform networking or database queries spend most of their time in the unprotected part of the application, and therefore the slowdown of the enclave represents only a minor part of the application's performance. Thus, in many cases, a high enclave overhead can still be acceptable for the overall performance.

6 SECURITY ANALYSIS

In this section we analyze the security of DR.SGX. We focus on the security properties of our novel heap data protection mechanism. Our stack data protection follows a known approach, evaluated in [19].

The goal of the adversary is to recover secret data from the victim enclave based on secret-dependent (heap) data access patterns to data. Recall that we consider a powerful adversary that gets a perfect trace of all cache and page events. Since all known attacks [11, 31, 51, 64] exhibit significant noise in the cache channel, this is an over-approximation of the capabilities of today's attackers and allows us to reason about the effectiveness of our solution against more powerful future adversaries.

In a data-driven side-channel attack, the adversary leaks information by monitoring secret-dependent access patterns. We model this as follows. The targeted victim enclave has secret data s of any length. The secret could be a cryptographic key, medical data, financial information or sensitive machine learning training sets. The enclave has a data structure d that consists of n elements $(e_1, ..., e_n)$ and is accessed based on s. The data structure could be a look-up table, S-box, index, or in-memory database. The size of each element e_i is the cache line size (smaller elements cannot be attacked, larger elements can be modeled as multiple elements). Based on the value of s, the enclave makes k accesses to different elements of d. Such access pattern determines the value of s. The enclave may also make predictable accesses to d (e.g., iterate through it during initialization).

6.1 Finding Attack Position in Trace

We start our analysis by explaining how the adversary can find the "attack position" in the side-channel trace, i.e., the position where (permuted) secret-dependent data accesses take place. The adversary can compile the victim enclave without DR.SGX protection and instrument those parts of the enclave where the secret-dependent accesses to d happen. The adversary can then run the instrumented enclave, monitor side-channels, and based on the instrumentation learn the position in the trace where the secret-dependent accesses are located. After that, the adversary can run the victim enclave that is protected with DR.SGX using the same inputs and again monitor side-channels. Assuming a deterministic enclave,[5] the adversary obtains a protected trace that includes additional randomization events to the trace (see Figure 7). Next, the adversary can filter out all randomization events. Since we use non-temporal (NT) writes that bypass the cache for randomization writes, the adversary finds each page event p_i that has no corresponding cache event c_i in the trace. For each such randomization write, the previous event in the trace is a read due to the randomization. The adversary removes all randomization events. The known attack position in the non-protected trace corresponds to the same position in the filtered protected trace.

6.2 Inferring Secret Enclave Data

Once the attack position is known, the adversary can attempt to infer secret data s from the permuted memory accesses in the attack trace. The adversary's success depends on the type of the victim enclave.

No predictable accesses. We first consider enclaves that make *no* predictable accesses to d (i.e., the enclave accesses d only based on a pattern that is derived from the secret data s). For such enclaves, DR.SGX provides strong protection due to its initial randomization that is illustrated in Figure 7. The enclave's data is copied from the known, original memory layout L_0 to a new randomized memory layout L_1 in blocks of cache line size using NT writes. For each block, the initial randomization process performs one read access to the original memory layout and NT writes to all memory pages. Because NT writes hide the accessed address at cache-line granularity, the adversary gains no knowledge of the new location in L_1. The same process is repeated for every memory block and in the end the location of each block in L_1 is equally likely for the adversary.

By observing the permuted side-channel trace, the adversary may infer execution characteristics such as frequencies of accesses to the same memory address (e.g., address a was accessed x times). However, because permuted addresses a can refer to any actually accessed addresses, such frequency analysis does not help the adversary to infer the secret data s, unless the enclave exhibits predictable access patterns which we discuss below.

Assuming no predictable access patterns, the best option for the adversary is a guessing attack. The adversary knows the permuted addresses of k secret-dependent accesses. For each access, every address, and thus every data structure element e_i, is equally likely. After observing k distinctive accesses to n elements, the number of possible alternatives will be given by an arrangement of k from n: $A_n^k = \frac{n!}{(n-k)!}$. For example, a data structure of $n = 50$ elements and any number of secret-dependent accesses resulting in 25 distinctive accesses to the data structure, the amount of arrangements is 1.96×10^{39}, which gives the chance of a random guess of approximately 2^{-131}. We conclude that DR.SGX provides strong protection for enclaves that have no predictable accesses to the data structure d.

[5] We consider a deterministic enclave, because that is the best case the for adversary for building the tracking tree. Thus, the following analysis based on this assumption represents the best case for the adversary regarding finding the attack position in the trace.

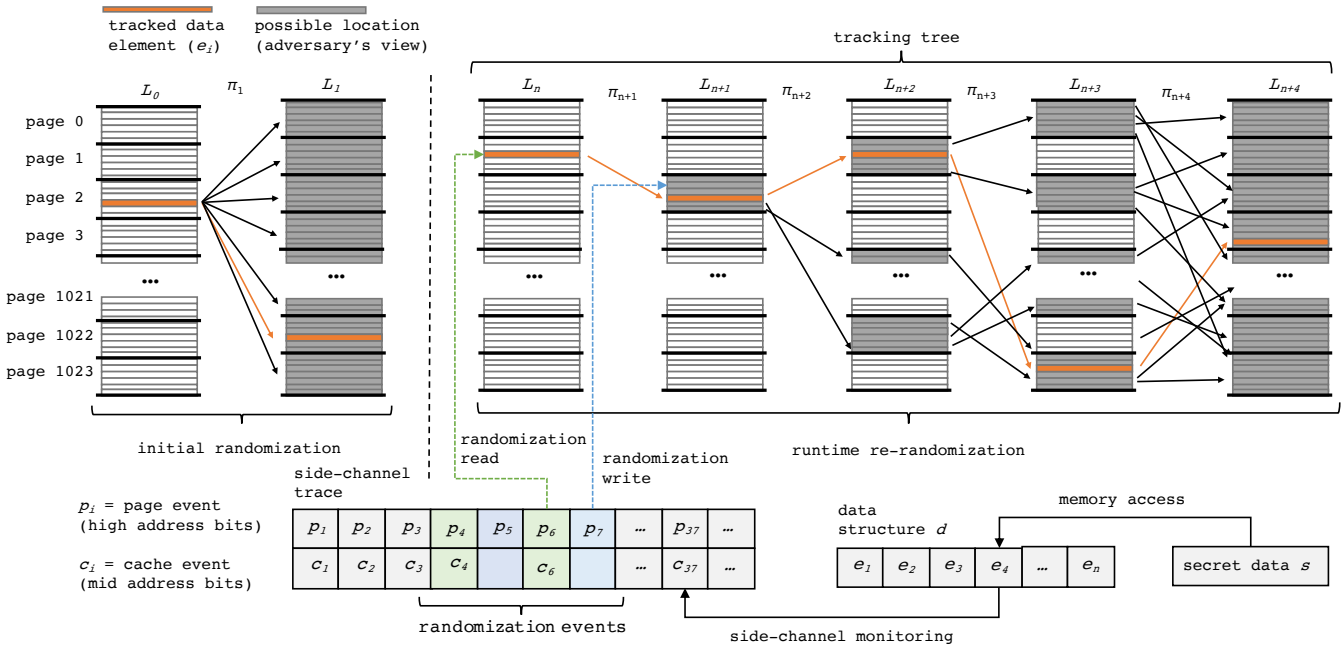

Figure 7: Location tracking. By identifying missing cache events in the trace, the adversary can learn the re-randomization writes (marked in blue) and preceding re-randomization reads (marked in green). The initial randomization hides destination addresses completely. The adversary can build a tracking tree, where the source address of each re-randomization is known with cache-line granularity and the destination address with page granularity. After few re-randomization rounds, the tracked memory location can reside in any memory location.

Predictable accesses. The second case that we consider is a victim enclave that exhibits predictable access patterns to d, e.g., the enclave may initialize d in an order that is known to the adversary. The enclave may also access elements of d a predictable number of times. Such predictable accesses in the trace will disclose the current permuted memory addresses for each accessed element e_i.

Figure 7 illustrates an example scenario, where the permuted address of element e_i is revealed to the adversary in memory layout L_n. The next re-randomization round moves the data of that element to a new location in layout L_{n+1}. Since the move operation is implemented using NT writes, the adversary learns the new page in L_{n+1}, but not the fine-grained location. The leakage of the target page allows the adversary to construct a *tracking tree* for element e_i.

The expansion of the tracking tree depends on the size of the used memory in the victim enclave. For example, if the victim enclave uses 2 MB memory (out of total 4 MB address space), each memory page contains on the average 32 blocks. On the next re-randomization round, each of these blocks are moved to new memory locations in layout L_{n+2}. Because the adversary does not know the exact location of element e_i in L_{n+1}, he cannot distinguish when the element is moved from the set of 32 move operations that use the same page as the source.[6] From the adversary's point of view, after two re-randomization rounds, the element can reside in 32 pages with high probability. After four re-randomization rounds, the adversary must track $32^3 = 32,768$ re-randomization moves. Although some of the moves may write to the same target pages, the tracking tree covers all 1,024 memory pages in L_{n+4} with high probability, and thus all memory locations are equally likely for the adversary. For enclaves with smaller heap size (512 KB), similar effect can be achieved after three rounds. The shortest re-randomization window we tested in Section 5 lasted 0.37 ms, in which case the required three (or four) re-randomization rounds would be performed after 1.1 ms (or 1.5 ms) of enclave execution. We conclude that enclaves with predictable accesses can leak information. If the secret-dependent access happens after the predictable access and before a sufficient number of re-randomization rounds, the secret may be leaked to the adversary. By touching additional memory pages on every re-randomization write, the window can be reduced to fewer rounds. Alternatively, re-randomization rounds can be performed more frequently. Both approaches increase runtime overhead.

[6] Our implementation randomizes 8 blocks at once which makes tracing even more difficult for the adversary.

7 RELATED WORK

Previous research has proposed various side-channel defenses. In this section we review them and compare existing defenses to DR.SGX.

ORAM and Oblivious Execution. Oblivious RAM (ORAM) [28–30, 60, 69, 76] refers to schemes that hide the memory access pattern of a trusted client (e.g., CPU or network client) to an untrusted and encrypted memory (e.g., DRAM or server) by introducing fake

accesses and shuffling the encrypted memory elements such that the observable access pattern is independent of the actual access pattern. Oblivious execution architectures [44, 45, 48] attempt to hide all observable effects of program execution, including both memory accesses (code and data) and timing information. Implementing ORAM for every enclave memory access is extremely expensive. Obfuscuro [1], a program obfuscation system, implements both ORAM and oblivious execution, with performance overheads of 83× on average and up to 220×. DR.SGX's performance overhead is at least one order of magnitude lower than Obfuscuro.

Sinha [67] proposes a compiler-based tool to protect code written in their custom language from paging-based side-channel attacks. In contrast, DR.SGX works with existing code in C/C++ and also mitigates cache-based side-channel attacks.

Raccoon [59] is a system that provides oblivious data access only for developer-annotated enclave data, thus reducing the overhead. Memory accesses are hidden by either using ORAM or by streaming over the entire data structure. In contrast, DR.SGX does not rely on developers to identify and annotate data that might leak.

ZeroTrace [62] is an oblivious data structure framework for SGX that runs on top of a software memory controller. ZeroTrace is designed to hide memory access to resources *outside* of an enclave, e.g., to the hard disk drive. Importantly, it is not designed to make *all* memory accesses of an enclave to its own main memory oblivious, like DR.SGX does. Furthermore, ZeroTrace requires the developer to use the memory controller interface for all access that should be protected. DR.SGX does not require similar developer assistance.

Ohrimenko et. al. propose data-oblivious machine learning algorithms [53] and a side-channel resilient MapReduce framework [52] for SGX. Fuhry et. al. propose a page-fault side-channel secure database [27]. Such defenses are tailored to specific enclaves and algorithms, while DR.SGX applies to arbitrary enclaves.

Transactional memory. Some of the known SGX side-channel attacks interrupt the victim enclave repeatedly [77]. A corresponding defense is to enable the victim enclave to detect interruption and take counteractive measures, such as stopping its execution. T-SGX [66] leverages the Intel Transactional Synchronization Extension (TSX) to detect asynchronous enclave exits, e.g., due to interrupts of page faults. Déjá Vu [17] monitors the execution time of an enclave to detect a slowdown caused by frequent interrupts. These defenses do not prevent attacks that work without interrupts [11, 31, 64]. DR.SGX is applicable to such attacks.

Cloak [33] uses TSX to preform atomic memory operations that hide sensitive memory accesses. Before sensitive memory is accessed, all cache lines are touched (primed) by the enclave, and thus the adversary learns nothing about the enclave's sensitive accesses. Cloak relies on the developer to annotate sensitive data structures that should be protected from side-channel attacks and requires TSX, which is not supported by all SGX processors. DR.SGX does not require similar developer assistance and works on all SGX processors.

Software diversity. Crane et al. [19] propose to apply dynamic software diversity, an effective countermeasure against code reuse attacks and reverse engineering, to defend against cache-based side-channel attacks. The approach is to create multiple copies of code and choose one of them at the time of execution. We apply this technique to protect stack data. However, the solution by Crane et al. is specifically targeting protection of cryptographic algorithms. In contrast, DR.SGX can protect non-cryptographic enclaves.

Randomization. Address Space Layout Randomization (ASLR) [57] is a common defensive technique against memory corruption attacks such as ROP [61]. ASLR hides the locations of memory regions (code and data) by randomizing their offsets at load time. More fine-grained solutions randomize code (but not data) at function [38], block [21, 75], or instruction [34, 56] level.

Such randomization techniques are insufficient as a side-channel defense for SGX. Offset-based ASLR is not effective since the privileged attacker is responsible for memory management and thus learns the "secret" randomized offsets. Code randomization, as implemented in SGX-Shield [65], is not complete [7] and does not prevent attacks that monitor data accesses [11, 64, 77].

New cache architectures. Cache-based side channels can be addressed by changes in the cache architecture. The two common approaches are (i) cache partitioning [23, 24, 55, 74], dividing the cache into partitions that are not shared between processes, and (ii) cache access obfuscation [22, 37, 40, 46, 74], where the goal is to obfuscate the obtainable side-channel information, either by introducing noise or by randomizing the address to cache line mapping. Such defenses require hardware changes and are limited to cache attacks. DR.SGX works on current processors and applies to additional side-channels (e.g., page faults).

8 DISCUSSION

Fine-grained leaks. Recent works [50, 78] have investigated the possibility of leaking information through a side-channel with a granularity smaller than a cache line. However, they are not applicable in our case.

CacheBleed [78] exploits cache *bank conflicts* to leak fine-grained information. This attack does not apply to SGX CPUs due to an updated cache design. We verified this experimentally.

MemJam [50] uses read-after-write false dependencies to introduce latency when a victim program reads data with a specific page offset. By measuring the run time of the victim program a high number of times while *jamming* different page offsets, the attacker can infer which offsets are read more often by the victim. This attack can leak information with a four byte granularity, but requires an extremely high number of runs (*50 million runs* for an attack against a simple and deterministic SGX enclave). However, with DR.SGX, the page offsets of data change between different runs, making the correlation of timing information for different runs exponentially more involved. Moreover, the accesses due to DR.SGX's own code generate a significant amount of noise, which complicates the matter further. Finally, the code of DR.SGX itself was designed to not be vulnerable to MemJam attacks, e.g., by randomizing the permutation buffer layout (see Section 4.5).

Leakage quantification. Quantification of cache-based information leakage has been studied in previous works. For example, CacheAudit [25] is a well-known static analysis framework that given an x86 binary and a cache configuration yields an upper

bound on the amount of information leakage via cache- and time-based side-channels. The information leakage is quantified based on the number of side-channel observations an attacker can obtain.

CacheAudit, and similar existing tools, are not applicable to our scenario for two main reasons. First, in the model of CacheAudit, randomly permuted observations contribute to the total number of observations, even though the attacker may not learn any useful information from such accesses. Second, CacheAudit does not consider information leakage through other channels, such as page faults, that can be correlated with cache observations. Therefore, CacheAudit cannot be used to quantify informations leakage of DR.SGX.

9 CONCLUSION

In this paper we have proposed semantic-agnostic data randomization as a new defensive approach against side-channel attacks on SGX. We have designed and implemented DR.SGX, which allows to instrument enclave code such that all data locations in enclave memory are permuted at cache-line granularity and re-randomized at runtime. Unlike previous defenses, our solution allows non-expert developers to harden their enclaves against various data-driven attack strategies with an adjustable security-performance trade-off.

ACKNOWLEDGMENTS

The authors would like to thank Urs Müller for his feedback in the initial discussions that led to this work.
This work has been supported by the German Research Foundation (DFG) as part of projects HWSec, P3 and S2 within the CRC 1119 CROSSING, by the German Federal Ministry of Education and Research (BMBF) and the Hessen State Ministry for Higher Education, Research and the Arts (HMWK) within CRISP, by BMBF within the projects iBlockchain and CloudProtect, and by the Intel Collaborative Research Institute for Collaborative Autonomous & Resilient Systems (ICRI-CARS).

REFERENCES

[1] Adil Ahmad, Byunggill Joe, Yuan Xiao, Yinqian Zhang, Insik Shin, and Byoungyoung Lee. 2019. Obfuscuro: A Commodity Obfuscation Engine on Intel SGX. In *Network and Distributed System Security Symposium*.
[2] ARM Limited. 2009. ARM Security Technology – Building a Secure System using TrustZone Technology. http://infocenter.arm.com/help/topic/com.arm.doc.prd29-genc-009492c/PRD29-GENC-009492C_trustzone_security_whitepaper.pdf.
[3] Lejla Batina, Patrick Jauernig, Nele Mentens, A-R Sadeghi, and Emmanuel Stapf. 2019. In Hardware We Trust: Gains and Pains of Hardware-assisted Security. (2019).
[4] Andrew Baumann, Marcus Peinado, and Galen Hunt. 2014. Shielding Applications from an Untrusted Cloud with Haven.
[5] Mihir Bellare, Phillip Rogaway, and Terence Spies. 2010. *The FFX Mode of Operation for Format-Preserving Encryption.* Technical Report.
[6] David Bigelow, Thomas Hobson, Robert Rudd, William Streilein, and Hamed Okhravi. 2015. Timely rerandomization for mitigating memory disclosures. In *ACM SIGSAC Conference on Computer and Communications Security*. ACM.
[7] Andrea Biondo, Mauro Conti, Lucas Davi, Tommaso Frassetto, and Ahmad-Reza Sadeghi. 2018. The Guard's Dilemma: Efficient Code-Reuse Attacks Against Intel SGX. In *27th USENIX Security Symposium*.
[8] Ferdinand Brasser, Lucas Davi, Abhijitt Dhavlle, Tommaso Frassetto, Sai Manoj Pudukotai Dinakarrao, Setareh Rafatirad, Ahmad-Reza Sadeghi, Avesta Sasan, Hossein Sayadi, Shaza Zeitouni, et al. 2018. Advances and throwbacks in hardware-assisted security: special session. In *Proceedings of the International Conference on Compilers, Architecture and Synthesis for Embedded Systems*. IEEE Press, 15.
[9] Ferdinand Brasser, Tommaso Frassetto, Korbinian Riedhammer, Ahmad-Reza Sadeghi, Thomas Schneider, and Christian Weinert. 2018. VoiceGuard: Secure and Private Speech Processing. In *Interspeech 2018*. International Speech Communication Association (ISCA), 1303–1307.
[10] Ferdinand Brasser, David Gens, Patrick Jauernig, Ahmad-Reza Sadeghi, and Emmanuel Stapf. 2019. SANCTUARY: ARMing TrustZone with User-space Enclaves. In *26th Annual Network & Distributed System Security Symposium (NDSS)*.
[11] Ferdinand Brasser, Urs Müller, Alexandra Dmitrienko, Kari Kostiainen, Srdjan Capkun, and Ahmad-Reza Sadeghi. 2017. Software Grand Exposure: SGX Cache Attacks Are Practical. In *USENIX Workshop on Offensive Technologies*.
[12] E. Brickell, G. Graunke, and J.-P. Seifert. 2006. Mitigating cache/timing attacks in AES and RSA software implementations. In *RSA Conference 2006, session DEV-203*.
[13] BYTE Magazine and Uwe F. Mayer. 1995-2011. BYTEmark benchmark (nbench), port to Linux. Original address http://www.tux.org/~mayer/linux/bmark.html, now archived at https://web.archive.org/web/20151215162836/http://www.tux.org/~mayer/linux/bmark.html.
[14] Luigi Catuogno, Alexandra Dmitrienko, Konrad Eriksson, Dirk Kuhlmann, Gianluca Ramunno, Ahmad-Reza Sadeghi, Steffen Schulz, Matthias Schunter, Marcel Winandy, and Jing Zhan. 2009. Trusted Virtual Domains – Design, Implementation and Lessons Learned. In *International Conference on Trusted Systems*.
[15] Swarup Chandra, Vishal Karande, Zhiqiang Lin, Latifur Khan, Murat Kantarcioglu, and Bhavani Thuraisingham. 2017. Securing Data Analytics on SGX with Randomization. In *European Symposium on Research in Computer Security*.
[16] Guoxing Chen, Sanchuan Chen, Yuan Xiao, Yinqian Zhang, Zhiqiang Lin, and Ten H. Lai. 2018. SgxPectre Attacks: Stealing Intel Secrets from SGX Enclaves via Speculative Execution. arXiv:arXiv:1802.09085v3
[17] Sanchuan Chen, Xiaokuan Zhang, Michael K. Reiter, and Yinqian Zhang. 2017. Detecting Privileged Side-Channel Attacks in Shielded Execution with Déjá Vu. In *ACM Symposium on Information, Computer and Communications Security*.
[18] Victor Costan and Srinivas Devadas. 2016. *Intel SGX Explained.* Technical Report. Cryptology ePrint Archive. Report 2016/086. https://eprint.iacr.org/2016/086.pdf.
[19] Stephen Crane, Andrei Homescu, Stefan Brunthaler, Per Larsen, and Michael Franz. 2015. Thwarting Cache Side-Channel Attacks Through Dynamic Software Diversity. In *Network and Distributed System Security Symposium*.
[20] Poulami Das, Lisa Eckey, Tommaso Frassetto, David Gens, Kristina Hostáková, Patrick Jauernig, Sebastian Faust, and Ahmad-Reza Sadeghi. 2019. FastKitten: Practical Smart Contracts on Bitcoin. In *28th USENIX Security Symposium*.
[21] Lucas Davi, Alexandra Dmitrienko, Stefan Nürnberger, and Ahmad-Reza Sadeghi. 2013. Gadge Me If You Can - Secure and Efficient Ad-hoc Instruction-Level Randomization for x86 and ARM. In *ACM Symposium on Information, Computer and Communications Security*.
[22] Ghada Dessouky, Tommaso Frassetto, and Ahmad-Reza Sadeghi. 2020. HybCache: Hybrid Side-Channel-Resilient Caches for Trusted Execution Environments. In *29th USENIX Security Symposium*.
[23] Leonid Domnitser, Aamer Jaleel, Jason Loew, Nael Abu-Ghazaleh, and Dmitry Ponomarev. 2012. Non-monopolizable caches: Low-complexity mitigation of cache side channel attacks. *ACM Transactions on Architecture and Code Optimization (TACO)* 8, 4 (2012).
[24] Leonid Domnitser, Aamer Jaleel, Jason Loew, Nael Abu-Ghazaleh, and Dmitry Ponomarev. 2012. Non-monopolizable Caches: Low-complexity Mitigation of Cache Side Channel Attacks. *ACM Transactions on Architecture and Code Optimization* (2012). https://doi.org/10.1145/2086696.2086714
[25] Goran Doychev, Boris Köpf, Laurent Mauborgne, and Jan Reineke. 2015. Cacheaudit: A tool for the static analysis of cache side channels. *ACM Transactions on Information and System Security (TISSEC)* 18, 1 (2015).
[26] Tommaso Frassetto, David Gens, Christopher Liebchen, and Ahmad-Reza Sadeghi. 2017. JITGuard: Hardening Just-in-time Compilers with SGX. In *24th ACM Conference on Computer and Communications Security (CCS)*.
[27] Benny Fuhry, Raad Bahmani, Ferdinand Brasser, Florian Hahn, Florian Kerschbaum, and Ahmad-Reza Sadeghi. 2017. HardIDX: Practical and Secure Index with SGX. In *Conference on Data and Applications Security and Privacy (DBSec)*.
[28] Oded Goldreich. 1987. Towards a theory of software protection and simulation by oblivious RAMs. In *Annual ACM Symposium on Theory of Computing*. ACM.
[29] Oded Goldreich and Rafail Ostrovsky. 1996. Software Protection and Simulation on Oblivious RAMs. *J. ACM* (1996).
[30] Michael T Goodrich, Michael Mitzenmacher, Olga Ohrimenko, and Roberto Tamassia. 2012. Privacy-preserving group data access via stateless oblivious RAM simulation. In *Annual ACM-SIAM symposium on Discrete Algorithms*. Society for Industrial and Applied Mathematics.
[31] Johannes Götzfried, Moritz Eckert, Sebastian Schinzel, and Tilo Müller. 2017. Cache Attacks on Intel SGX. In *European Workshop on Systems Security*.
[32] Ben Gras, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. 2018. Translation Leak-aside Buffer: Defeating Cache Side-channel Protections with TLB Attacks. In *27th USENIX Security Symposium*. https://www.usenix.org/conference/usenixsecurity18/presentation/gras
[33] Daniel Gruss, Julian Lettner, Felix Schuster, Olya Ohrimenko, Istvan Haller, and Manuel Costa. 2017. Strong and Efficient Cache Side-Channel Protection using Hardware Transactional Memory. In *26th USENIX Security Symposium*.

[34] Jason D. Hiser, Anh Nguyen-Tuong, Michele Co, Matthew Hall, and Jack W. Davidson. 2012. ILR: Where'd My Gadgets Go?. In *IEEE Symposium on Security and Privacy*.

[35] Intel. 2015. Intel Software Guard Extensions. Tutorial slides. https://software.intel.com/sites/default/files/332680-002.pdf. Reference Number: 332680-002, revision 1.1.

[36] Intel. 2016. Intel 64 and IA-32 Architectures Software Developer's Manual. http://www.intel.com/content/www/us/en/architecture-and-technology/64-ia-32-architectures-software-developer-manual-325462.html.

[37] G. Keramidas, A. Antonopoulos, D. N. Serpanos, and S. Kaxiras. 2008. Non deterministic caches: A simple and effective defense against side channel attacks. *Design Automation for Embedded Systems* (2008).

[38] Chongkyung Kil, Jinsuk Jun, Christopher Bookholt, Jun Xu, and Peng Ning. 2006. Address Space Layout Permutation (ASLP): Towards Fine-Grained Randomization of Commodity Software. In *Annual Computer Security Applications Conference*.

[39] Paul Kocher, Jann Horn, Anders Fogh, , Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. 2019. Spectre Attacks: Exploiting Speculative Execution. In *40th IEEE Symposium on Security and Privacy (S&P'19)*.

[40] Jingfei Kong, Onur Aciiçmez, Jean-Pierre Seifert, and Huiyang Zhou. 2009. Hardware-software integrated approaches to defend against software cache-based side channel attacks. In *IEEE International Symposium on High Performance Computer Architecture*. IEEE.

[41] Kubilay Ahmet Küçük, Andrew Paverd, Andrew Martin, N. Asokan, Andrew Simpson, and Robin Ankele. 2016. Exploring the Use of Intel SGX for Secure Many-Party Applications.

[42] Sangho Lee, Ming-Wei Shih, Prasun Gera, Taesoo Kim, Hyesoon Kim, and Marcus Peinado. 2017. Inferring Fine-grained Control Flow Inside SGX Enclaves with Branch Shadowing. In *USENIX Security Symposium*.

[43] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. 2018. Meltdown: Reading Kernel Memory from User Space. In *27th USENIX Security Symposium (USENIX Security 18)*.

[44] Chang Liu, Austin Harris, Martin Maas, Michael Hicks, Mohit Tiwari, and Elaine Shi. 2015. Ghostrider: A hardware-software system for memory trace oblivious computation. *ACM SIGARCH Computer Architecture News* 43, 1 (2015).

[45] Chang Liu, Michael Hicks, and Elaine Shi. 2013. Memory trace oblivious program execution. In *IEEE Computer Security Foundations Symposium*.

[46] Fangfei Liu and Ruby B. Lee. 2014. Random Fill Cache Architecture. In *47th Annual IEEE/ACM International Symposium on Microarchitecture*.

[47] LLVM Foundation. 2019. The LLVM Compiler Infrastructure. https://llvm.org.

[48] Martin Maas, Eric Love, Emil Stefanov, Mohit Tiwari, Elaine Shi, Krste Asanovic, John Kubiatowicz, and Dawn Song. 2013. Phantom: Practical oblivious computation in a secure processor. In *ACM SIGSAC Conference on Computer and Communications Security*.

[49] Jonathan M. McCune, Yanlin Li, Ning Qu, Zongwei Zhou, Anupam Datta, Virgil Gligor, and Adrian Perrig. 2010. TrustVisor: Efficient TCB Reduction and Attestation. In *IEEE Symposium on Security and Privacy*.

[50] Ahmad Moghimi, Thomas Eisenbarth, and Berk Sunar. 2018. MemJam: A False Dependency Attack Against Constant-Time Crypto Implementations in SGX. In *Topics in Cryptology – CT-RSA 2018*, Nigel P. Smart (Ed.). Springer International Publishing.

[51] Ahmad Moghimi, Gorka Irazoqui, and Thomas Eisenbarth. 2017. *CacheZoom: How SGX Amplifies The Power of Cache Attacks.* Technical Report. arXiv:1703.06986 [cs.CR]. https://arxiv.org/abs/1703.06986.

[52] Olga Ohrimenko, Manuel Costa, Cédric Fournet, Christos Gkantsidis, Markulf Kohlweiss, and Divya Sharma. 2015. Observing and preventing leakage in MapReduce. In *ACM SIGSAC Conference on Computer and Communications Security*. ACM.

[53] Olga Ohrimenko, Felix Schuster, Cedric Fournet, Aasthaa Meht, Sebastian Nowozin, Kapil Vaswani, and Manuel Costa. 2016. Oblivious Multi-Party Machine Learning on Trusted Processors. In *USENIX Security Symposium*.

[54] Dag Arne Osvik, Adi Shamir, and Eran Tromer. 2006. Cache Attacks and Countermeasures: The Case of AES. In *The Cryptographers' Track at the RSA Conference on Topics in Cryptology*.

[55] D. Page. 2005. Partitioned Cache Architecture as a Side-Channel Defence Mechanism. In *IACR Eprint archive*.

[56] Vasilis Pappas, Michalis Polychronakis, and Angelos D. Keromytis. 2012. Smashing the Gadgets: Hindering Return-Oriented Programming Using In-Place Code Randomization. In *IEEE Symposium on Security and Privacy*.

[57] PaX Team. [n.d.]. PaX address space layout randomization (ASLR). http://pax.grsecurity.net/docs/aslr.txt.

[58] Bernardo Portela, Manuel Barbosa, Guillaume Scerri, Bogdan Warinschi, Raad Bahmani, Ferdinand Brasser, and Ahmad-Reza Sadeghi. 2017. Secure Multiparty Computation from SGX. In *Financial Cryptography and Data Security*.

[59] Ashay Rane, Calvin Lin, and Mohit Tiwari. 2015. Raccoon: Closing Digital Side-channels Through Obfuscated Execution. In *USENIX Security Symposium*. http://dl.acm.org/citation.cfm?id=2831143.2831171

[60] Ling Ren, Christopher W Fletcher, Albert Kwon, Emil Stefanov, Elaine Shi, Marten Van Dijk, and Srinivas Devadas. 2015. Constants Count: Practical Improvements to Oblivious RAM. In *USENIX Security Symposium*.

[61] Ryan Roemer, Erik Buchanan, Hovav Shacham, and Stefan Savage. 2012. Return-oriented programming: Systems, languages, and applications. *ACM Transactions on Information and System Security* 15, 1 (2012).

[62] Sajin Sasy, Sergey Gorbunov, and Christopher Fletcher. 2017. ZeroTrace: Oblivious Memory Primitives from Intel SGX. *IACR Cryptology ' Archive* Report 2017/549 (2017).

[63] Felix Schuster, Manuel Costa, Cedric Fournet, Christos Gkantsidis, Marcus Peinado, Gloria Mainar-Ruiz, and Mark Russinovich. 2015. VC3: Trustworthy Data Analytics in the Cloud Using SGX.

[64] Michael Schwarz, Samuel Weiser, Daniel Gruss, Clémentine Maurice, and Stefan Mangard. 2017. Malware Guard Extension: Using SGX to Conceal Cache Attacks. In *Detection of Intrusions and Malware, and Vulnerability Assessment*.

[65] Jaebaek Seo, Byounyoung Lee, Seongmin Kim, Ming-Wei Shih, Insik Shin, Dongsu Han, and Taesoo Kim. 2017. SGX-Shield: Enabling Address Space Layout Randomization for SGX Programs. In *Network and Distributed System Security Symposium*.

[66] Ming-Wei Shih, Sangho Lee, Taesoo Kim, and Marcus Peinado. 2017. T-SGX: Eradicating Controlled-Channel Attacks Against Enclave Programs. In *Network and Distributed System Security Symposium*.

[67] Rohit Sinha, Sriram Rajamani, and Sanjit A. Seshia. 2017. A compiler and verifier for page access oblivious computation. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering - ESEC/FSE 2017*. ACM Press. https://doi.org/10.1145/3106237.3106248

[68] Julian Stecklina and Thomas Prescher. 2018. LazyFP: Leaking FPU Register State using Microarchitectural Side-Channels. *CoRR* abs/1806.07480 (2018). arXiv:1806.07480 http://arxiv.org/abs/1806.07480

[69] 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 *ACM SIGSAC Conference on Computer and Communications Security*.

[70] Chia-che Tsai, Donald E. Porter, and Mona Vij. 2017. Graphene-SGX: A Practical Library OS for Unmodified Applications on SGX. In *USENIX Annual Technical Conference*.

[71] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. 2018. Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution. In *27th USENIX Security Symposium*.

[72] Jo Van Bulck, Nico Weichbrodt, Rüdiger Kapitza, Frank Piessens, and Raoul Strackx. 2017. Telling Your Secrets without Page Faults: Stealthy Page Table-Based Attacks on Enclaved Execution. In *26th USENIX Security Symposium*.

[73] Stephan van Schaik, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. 2018. Malicious Management Unit: Why Stopping Cache Attacks in Software is Harder Than You Think. In *27th USENIX Security Symposium*. https://www.usenix.org/conference/usenixsecurity18/presentation/van-schaik

[74] Zhenghong Wang and Ruby B. Lee. 2008. A Novel Cache Architecture with Enhanced Performance and Security. In *Annual IEEE/ACM International Symposium on Microarchitecture*.

[75] Richard Wartell, Vishwath Mohan, Kevin W. Hamlen, and Zhiqiang Lin. 2012. Binary Stirring: Self-randomizing Instruction Addresses of Legacy x86 Binary Code. In *ACM SIGSAC Conference on Computer and Communications Security*.

[76] Peter Williams and Radu Sion. 2012. Round-optimal access privacy on outsourced storage. In *ACM Conference on Computer and Communications Security*.

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

[78] Y. Yarom, D. Genkin, and N. Heninger. 2016. *CacheBleed: A timing attack on OpenSSL constant time RSA.* Technical Report. Cryptology ePrint Archive. Report 2016/224. https://eprint.iacr.org/2016/224.pdf.