SG^{XL}: Security and Performance for Enclaves Using Large Pages

SUJAY YADALAM, University of Wisconsin–Madison
VINOD GANAPATHY and ARKAPRAVA BASU, Indian Institute of Science

Intel's SGX architecture offers clients of public cloud computing platforms the ability to create hardware-protected *enclaves* whose contents are protected from privileged system software. However, SGX relies on system software for enclave memory management. In a sequence of recent papers, researchers have demonstrated that this reliance allows a malicious OS/hypervisor to snoop on the page addresses being accessed from within an enclave via various channels. This page address stream can then be used to infer secrets if the enclave's page access pattern depends upon the secret and this constitutes an important class of side-channels.

We propose SG^{XL}, a hardware-software co-designed system that significantly increases the difficulty of any page address-based side-channels through the use of large pages. A large page maps address ranges at a much larger granularity than the default page size (at least 512× larger). SG^{XL} thus significantly lowers resolution of the leaked page address stream and could practically throttle all flavors of page-address based side-channels. We detail the modifications needed to SGX's software stack and the (minor) hardware enhancements required for SG^{XL} to guarantee the use of large pages in the presence of adversarial system software. We empirically show that SG^{XL} could be one of those rare systems that enhances security with the potential of improving performance as well.

CCS Concepts: • **Security and privacy → Side-channel analysis and countermeasures**;

Additional Key Words and Phrases: Intel SGX, enclaves, page-based side-channel attacks, large pages

ACM Reference format:
Sujay Yadalam, Vinod Ganapathy, and Arkaprava Basu. 2020. SG^{XL}: Security and Performance for Enclaves Using Large Pages. *ACM Trans. Archit. Code Optim.* 18, 1, Article 12 (December 2020), 25 pages.
https://doi.org/10.1145/3433983

1 INTRODUCTION

On public cloud computing platforms, cloud providers own and administer the system software (e.g., the BIOS, the OS, and/or the hypervisor) that manages the computing infrastructure. A malicious actor can leverage this system software to compromise the integrity and confidentiality of data and code of the clients of public computing infrastructures.

Work performed when the author was with the Indian Institute of Science.
We are grateful to the Ramanujan Fellowship scheme from the Government of India, Pratiksha Trust, Bangalore, and Intel India for research grants that financially supported this work.
Authors' addresses: S. Yadalam, University of Wisconsin—Madison, 1210 West Dayton Street, Madison, Wisconsin, 53706; email: sujayyadalam@cs.wisc.edu; V. Ganapathy and A. Basu, Indian Institute of Science, Bangalore, India, 560012; emails: {vg, arkapravab}@iisc.ac.in.

1544-3566/2020/12-ART12
https://doi.org/10.1145/3433983

Recent hardware advances aim to address this situation by allowing clients to protect their confidentiality and integrity even in the presence of adversarial system software with hardware-rooted guarantees [4, 5, 21, 30, 44]. This article focuses on the Intel SGX [4, 21, 30], a set of hardware and software extensions that provides support for *enclaves*, which can be used to build a number of novel security applications [6, 8, 23, 29, 38, 40, 43, 50]. SGX hardware ensures that client's code and data in the enclaves are integrity protected and are opaque to even privileged system software. Thus, from the client's perspective, the Trusted Computing Base (TCB) in the cloud would include *only* the processor, unlike traditional cloud platforms today.

However, SGX relies on system software to manage enclave memory even though the system software cannot access the clear-text contents of these pages. Thus, the OS/hypervisor is responsible for servicing page faults and setting up page table mapping from an application's virtual address to the physical address.

Several recent papers have demonstrated that this reliance on system software for managing enclave memory enables *page address-based side-channels* that can leak enclave secrets [51, 52, 55]. The key idea behind these attacks is that it is very often possible to infer properties of the data (secret) processed by an in-enclave victim program by simply observing the page access stream of the enclave. For example, a secret value used in a conditional branch within the enclave may determine the next set of instructions to be executed. If the instructions for the true and false branches lie on different pages, then the page access sequence could reveal some information about the secret being used in the conditional. Previous works have revealed a number of different attack vectors to leak page address streams.

Xu et al. [55] demonstrated that adversarial system software can induce page faults any time an enclave program accesses a new page by simply tinkering with the `present` bit in page table entries (PTEs). Page faults reveal page addresses being accessed by the enclave program to the OS/hypervisor. Van Bulck et al. [51] and Wang et al. [52] demonstrated an even more powerful attack vector that does not require generating page faults. Instead, adversarial system software can monitor (and periodically reset) the `accessed` and `dirty` metadata bits in PTEs to leak victim's page address access stream. Gras et al. [19] showed that even if the system software is not adversarial, translation lookaside buffer (TLB) side-channel can leak the page address stream. The attacker co-schedules a thread alongside a victim thread on the same hyperthreaded CPU core. The attacker can then extract the victim's page address stream via a timing attack on the TLB, to which the victim and attacker threads share access.

Unfortunately, previously proposed defenses for page address-based side-channels fall short. They either defend only against a subset of the attack vectors listed above [2, 9, 17, 42] or provide partial support, e.g., by protecting only code pages but not data pages [17]. Some defenses [17, 49] make assumptions that have subsequently been shown to be unrealistic (or broken [42]). We discuss the drawbacks of previously proposed defenses in further detail in Section 7 of this article; see, in particular, Table 7.

In theory, page address-based side-channels can be closed by ensuring page-access obliviousness (PAO) [47]. A program that satisfies PAO will always generate the same page access stream, irrespective of the input to the program. Because the page access stream is always the same, no individual execution of the program reveals any information about the secret being processed. Sinha et al. [47] develop a compiler to ensure PAO that adds dummy memory accesses into the enclave program to obscure its page access stream. However, their PAO-compiler places tight restrictions on the programming model of the enclave, e.g., it disallows loops predicated on sensitive values. Consequently, it has only been applied to small benchmarks, and it is likely to add significant performance overhead due to the large number of dummy accesses when applied to larger applications (more in Section 6).

In this article, we propose *SG^{XL}*, a hardware-software co-designed approach that provides a practical defense against all known page-address-based attack vectors without introducing runtime performance overheads and without programming restrictions. SG^{XL} defends against page-address based side-channels by reducing the resolution of the page access stream and consequently, reducing the effectiveness of the side-channel. We propose to use *large pages* for enclave code and/or data, which vastly reduces the number of distinct page addresses in the stream observable to adversarial system software.

Applications (inside or outside the enclave) use the default page size of 4 KB to map virtual addresses to physical addresses in x86-64 based processors, including those from Intel. However, modern processors allow system software the choice of mapping memory at a larger granularity of 2 MB or 1 GB (called *large pages*) to reduce address translation overheads for applications with large memory footprint [7]. Large pages map significantly larger chunks of the virtual address space, e.g., 512× for 2-MB pages than with default 4-KB pages to reduce address translation overheads.

While large pages have traditionally been used for better performance, we argue that they can also be used to defeat page-address based attacks. Mapping enclave memory with large pages reduces the resolution of page address stream by at least 512× (262144× for 1-GB pages). We empirically demonstrate in Section 6 that this is typically enough to defeat any known incarnation of page-address based attacks as an attacker is left with little information with which to infer secrets.

It is important to note that SG^{XL} does not *theoretically* guarantee defense against page address-based attacks by itself, i.e., it does not guarantee the PAO property. Instead, it *approximates* PAO by reducing the resolution of the page address stream and does so without any new programming restrictions or performance overheads. That said, for the applications considered in our empirical evaluation, SG^{XL} very nearly achieves PAO. We also demonstrate in our evaluation that SG^{XL} can complement theoretically sound approaches that guarantee PAO, such as Sinha et al.'s PAO-compiler, by reducing the number of dummy memory accesses that the compiler has to insert, thus improving the performance of PAO-compiled enclave applications, if a theoretical guarantee is desired.

A key security challenge in realizing SG^{XL} is that the enclave application may request a large page but the SGX device driver, which runs as part of the untrusted and potentially adversarial system software, cannot be trusted to provide one. Therefore, the *hardware needs to guarantee* to the application that a given memory region is *truly* mapped using a large page. We observe that the hardware while walking the page table in the course of performing the address translation, learns the page size used to map a given virtual address region. We thus propose a simple extension to Intel's SGX hardware for this purpose. Specifically, the semantics of a handful of SGX instructions needs to be extended and minor updates to the hardware page table walker and page fault generator are needed (see Section 5). To accompany these hardware changes, we also modify the Intel SGX SDK and the associated SGX device driver to support large pages.

Historically, security mechanisms often come at the cost of performance. However, SG^{XL} can aid performance while also significantly reducing the effectiveness of any page address-based side-channels. We expect SG^{XL} will likely be even more relevant for future iterations of SGX, which are speculated to have much larger enclave memory [25]. This would allow applications with bigger memory requirement such as in-memory databases to benefit from SGX [38]. These are the very applications that often suffer from address translation-based performance bottlenecks outside the enclave, and they will directly benefit from the use of large pages in SG^{XL}. SG^{XL} therefore represents a new point in the design space of defenses—it significantly reduces page-address-based side-channels, is not tied to any particular attack strategy, and is effective against all currently known variants of page-address-based side-channel attacks, all without hurting application performance.

In short, this article makes the following contributions:

- We build the software stack for SG^{XL} and demonstrate the effectiveness of using large pages to thwart page-address based side-channels on Intel SGX.
- We detail the necessary enhancements to the SGX hardware to reliably enforce the use of large pages within the enclave.
- Our evaluation of enclave-based applications shows that SG^{XL} can also aid their performance.

2 BACKGROUND

This section provides background material on three topics central to our work—Intel's SGX architecture, large pages, and the interplay between virtual memory and SGX.

2.1 Intel SGX

SGX enables clients of remote cloud platforms to ensure the confidentiality and integrity of their sensitive code and data. It does so by allowing sensitive portions of an application to run inside a hardware-protected execution environment on the cloud platform, called an *enclave*. The hardware ensures that enclave contents are protected even in presence of malicious privileged system software in the cloud.

There are two key pieces to SGX. First, the remote hardware (e.g., an SGX-capable processor) needs to gain the confidence of the client by using a remote attestation protocol. The remote hardware verifies the integrity of secrets (sensitive code and/or data) to the client though signature matching. Second, the remote hardware ensures that an application's secrets provisioned within the enclave are opaque and integrity-protected against any malicious software, including the system software.

The SGX SDK provides the client with a trusted `signing_tool` to compute an attestation, which is a digitally signed hash of the application's binary/data. This signature is sent to the remote hardware over a secure channel as part of a structure called `SIGSTRUCT`. This is later used by the client to verify the integrity of its binary/data on the remote hardware.

The second piece of SGX involves the remote cloud platform with SGX-capable processors. On a system startup, a contiguous physical memory region, referred to as the Processor Reserved Memory (PRM) is set aside for enclaves. The PRM region is split mainly into two parts—the Enclave Page Cache (EPC) and the Enclave Page Cache Map (EPCM). The client application's sensitive code and data are loaded into the EPC before enclave execution starts. The EPCM contains the metadata (e.g., ownership, addresses, type, permissions) to track the contents of the EPC at 4-KB page granularity. The EPCM region is accessible only to the hardware.

At the client side, an application writer specifies the enclave configuration that includes the enclave size and attributes. This enclave metadata is contained in a a structure called SGX Enclave Control Structure (`SECS`). `SECS`, enclave code and data, and the `SIGSTRUCT` are bundled into an enclave file that is sent to the remote platform for execution.

Figure 1 depicts the major components of an enclave and the steps involved in enclave creation at the remote platform. ① An untrusted SGX driver running on the remote server issues an SGX-specific `ecreate` instruction to set up the initial environment. The enclave configuration stored in the `SECS` is passed along with the `ecreate` instruction to the hardware. Further, the `ecreate` instruction initializes the `MRENCLAVE` field. `MRENCLAVE` is a register in SECS that keeps the measurement of the enclave build process. ② Then, `eadd` instructions are issued to copy the application's enclave code and data into the EPC at the granularity of 4-KB pages. It also adds an entry in the EPCM with the virtual address and the corresponding security attributes. As part

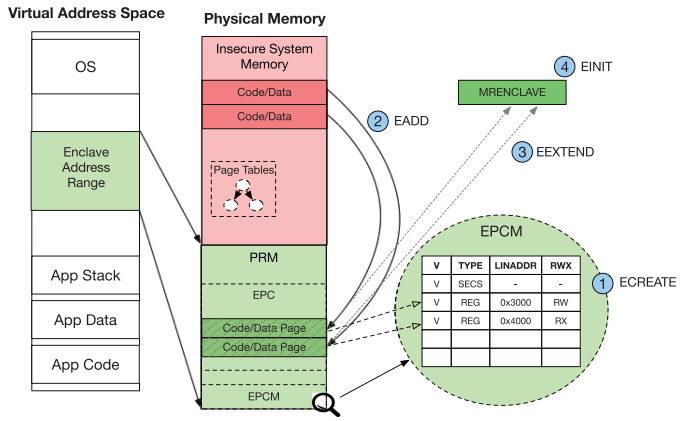


Fig. 1. Components of SGX's enclave creation at the remote platform. Green portions are secured by SGX.

of the eadd instruction, the virtual address and its security attributes are hashed and augmented into MRENCLAVE. ③ The eextend instruction is then used to compute a cryptographic hash of the contents copied into the EPC pages. This hash is then augmented to the content of MRENCLAVE. ④ Finally, on issuing the einit instruction the hardware compares the content of the MRENCLAVE with the signature sent by the client as part of SIGTRUCT. A match guarantees the integrity of client application's code and data in the enclave.

After enclave initialization, enclave execution can be started, resumed or exited by issuing of SGX-specific eenter, eresume, and eexit commands.

Future generations of Intel SGX. The current implementation of SGX requires all of the enclave code and data pages to be loaded into the EPC before the enclave execution starts. Future generations of the SGX promise to overcome this drawback by allowing dynamic addition of pages to EPC, on demand [53]. For that purpose, it introduces eaug instruction that allows an application to add a page to an already initialized enclave. The page is added in a pending state that can later be accepted by the enclave using eaccept or eacceptcopy instructions. It has also been speculated that the size of the PRM may increase from the 128 MB in future editions [25].

2.2 Large Pages and PTE Metadata Bits

System software (i.e., OS/hypervisor) is responsible for maintaining virtual to physical address translations in per-process page tables. By default, page tables keep translations at the granularity of the base page size of 4 KB (e.g., in x86-64 processors). However, applications with large memory footprint often suffers significant performance loss due to overheads of virtual to physical address translation [7, 26, 32].

Large pages are the most widely used technique for reducing address translation overheads [11, 26, 32, 36, 37]. A *large page* maps a much larger contiguous virtual address range to a contiguous physical address range (e.g., 2 MB or 1 GB compared to default 4 KB) and can thus reduce translation overheads by enabling a single TLB-entry to map larger amount of memory. Modern x86-64 processors from Intel and AMD support two large page sizes—2 MB and 1 GB.

A PTE also contains metadata bits such as the accessed and the dirty bits that are set by the hardware on an access to a page and on a store to a page, respectively. These bits are accessible to the system software and are typically used for page replacement and writeback decisions.

2.3 Interplay of SGX and Virtual Memory

When a user application creates an enclave, the SGX driver on the remote platform maps that enclave's virtual address pages to physical page frames drawn from the EPC. Since the driver is not trusted, it is essential for the hardware to ensure that a malicious driver has not altered the address mappings. This goal is achieved in two steps. First, as mentioned earlier, the translation information is added to the EPCM as soon as a page is added to the EPC by executing the `eadd` instruction. The software cannot access the contents of EPCM. Second, on every TLB access, and before filling a new TLB entry after a page table walk, the SGX hardware checks the integrity of the translation (i.e., a PTE) against the corresponding entry in the EPCM. On a mismatch, the SGX hardware exits the enclave.

While the SGX driver cannot compromise an enclave execution by altering address mappings, hardware allows it to modify the metadata in a PTE. For example, it can toggle the `present` bit that indicates the presence of data pointed by the PTE in the memory. A page fault to the OS is triggered if the `present` bit is unset. This is necessary for enabling the driver to manage EPC memory by swapping in/out data across EPC and non-EPC memory, under memory pressure. However, this flexibility allows the driver to induce page faults during enclave execution. The stream of falsely induced page fault addresses could be exploited by a malicious driver to infer secrets. Furthermore, metadata bits such as the `accessed` and `dirty` bits of a PTE are under driver control and can be also used to infer secrets (Section 3).

3 THREAT MODEL

In our threat model, the enclave must be protected from potentially adversarial system software. The OS/hypervisor are assumed to be under the control of the adversary. The enclave can therefore only trust the processor. The adversary can observe all page table walks in enclave mode without interrupting enclave's execution. By controlling privileged system software, the adversary can arbitrarily modify page-table entries that map enclave memory.

SGX ensures that the adversary cannot observe the cleartext contents of enclave pages, because the hardware encrypts their contents whenever they exit the processor package. However, the adversary's ability to interrupt enclave execution by modifying PTEs and/or observe meta-bits in PTEs is a powerful side-channel. This family of attacks (first described by Xu et al. [55]) uses the side-channel to observe the sequence (or set) of (virtual) page addresses accessed by the enclave, and leverages this to infer secrets processed by the enclave.

Example attack. To illustrate how this side-channel attack works, consider `FreeType`, which is a library that renders TrueType fonts onto bitmaps. `FreeType` executes within an enclave, and the goal of the attacker is to infer the secret text being rendered by `FreeType` by simply observing the page accesses it makes.

For each input character that `FreeType` processes, the logic used depends on the specific properties of the character. The glyph for each character is a collection of lines, curves, and hints, and the code path used depends on these parameters. The specific code path executed determines the number of different code pages touched in the process. Prior work showed that this code page address access stream is enough to uniquely identify the character (secret) being rendered [55]. For example, the snippet in Figure 2 shows how the number of contours in a glyph is used to select the processing logic—the structure `loader` is derived from the input character. The key property to note is that the code that executes on the true branch resides on a different set of pages than the code that executes on the false branch.

For `FreeType`, it turns out that even observing the total number of page faults during enclave execution is sufficient to determine the input character that was processed. We reconstructed Xu

```
    <src/truetype/ttgload.c> (Version 2.5.3)

1609: if (loader->n_contours > 0) {
        ...
1618:   TT_Process_Simple_Glyph(loader);
        ...
        }
1631: else if {
        ...
1642:   face->read_composite_glyph(loader);
        ...
        }
```

Fig. 2. Code snippet from Freetype.

Table 1. Number of Page Faults Observed by an Adversary for Various Lowercase Characters Processed by FreeType

Char	#Faults	Char	#Faults
a	2,044	n	1,817
b	1,129	o	1,551
c	1„471	p	1,398
d	1404	q	1,346
e	2,486	r	1,104
f	1,644	s	2,671
g	3,192	t	1,048
h	1,714	u	1,398
i	1,871	v	3,897
j	1,536	w	5,401
k	3,136	x	3,963
l	1,544	y	3,176
m	2,462	z	1,977

et al.'s attack on `FreeType`'s rendering of `TrueType` fonts. Table 1 shows the number of page faults observed by the adversary as different lower-case English characters were processed by the enclave. As the table shows, each character has a unique "signature" of the number of page faults observed, which is therefore sufficient to reconstruct the input character. Although not necessary for FreeType, the adversary also has access to the precise sequence of page faults observed for code pages as well as for data pages, which provide even more information for the adversary to reconstruct the input.

Known attack vectors. There are three broad classes of attack vectors in the research literature that rely on the same principle of observing page address access stream to infer secrets:

① **Page-faults.** Xu et al. [55] describe this attack vector in which the adversarial system software clears the `present` bit for PTEs mapping enclave memory. Consequently, if the enclave attempts to execute code or access data residing on such pages, it will trigger a page fault even if the corresponding physical page resides in the EPC. Adversarial system software is invoked to service the page fault, and in doing so, it can observe the virtual page number that triggered the fault. It sets the `present` bit back, allowing execution to continue. When the enclave executes code or accesses data on another page, the adversary clears the `present` bit for the current page so that a fault is triggered if the code/data on this page is executed/accessed again. It then proceeds to service the page fault for the new page. The attacker can thus trace the code/data accesses of the victim at page granularity.

② **Accessed and dirty bits.** Van Bulck et al. [51] and Wang et al. [52] show that an adversary can simply observe the `accessed` and `dirty` bits in the PTEs to identify the set of pages accessed/modified, even without inducing page faults. They also show that an adversary that periodically resets these bits as an enclave executes (using a separate thread) can also identify the order in which the enclave accesses the pages.

③ **Tapping the TLB.** Gras et al. [19] show that it is possible to infer an application's page access patterns by directly tapping into the TLB. Two hyperthreads—the victim and the attacker—executes on the same processor core while sharing the TLB. The attacker infers the victim's page access patterns by implementing a timing side-channel using methods

similar to those implemented for cache side-channels. Although their attacks were not demonstrated on SGX, Gras et al. speculate that it would work for SGX enclaves as well.

The methods above yield the sequence/set of pages accessed by an enclave. The adversary can then use various techniques, ranging from simple inference to machine learning [19], to reverse-engineer the secrets processed by the enclave.

Our threat model precludes an adversary with physical access to the memory bus of the machine. While an adversary that controls the system software can observe the sequence of page numbers, an adversary with physical access can observe *full address* (including the page offset) of *every* enclave access. Defenses against such physical adversaries are beyond the scope of this article. Prior work [1] has proposed hardware enhancements to protect against such attacks.

4 THE DESIGN OF SG^{XL}

SG^{XL} is a hardware-software co-designed system that uses large pages to reduce the effectiveness of page access patterns that malicious system software may observe. The use of large pages to map code and/or data of an enclave process decreases the resolution of the page-address stream that an attacker can observe by more than two orders of magnitude. For example, in Intel processors, a large page is at least 2 MB while the default page size is 4 KB. A 512× reduction in the granularity at which an attacker can observe page accesses vastly reduces the effectiveness of drawing inferences about enclave secrets based on page access pattern observations. Furthermore, the next larger page size is 1 GB, another 512× larger than a 2-MB page size. An over-cautious application can even use 1 GB pages to further defend against page-address based side-channel attack.

There are two key challenges in realizing the aforementioned idea. First, Intel's SGX SDK and the SGX driver fundamentally assume the use of only the default page size of 4 KB. They must be modified to allow enclaves to support multiple page sizes. Second and more importantly, the hardware needs to provide the guarantee that the SDK and driver are *truly* providing large pages as desired by the client. This is critical, since the driver is not trusted in SGX's model.

We address these challenges by enhancing SGX in the following ways. In our proposed execution model, the application writer can specify which segments (e.g., code, heap) of the enclave's virtual address space are to be mapped using large pages. For example, if an application is potentially vulnerable to page-address based attacks on the code pages then the code segment can be mapped using large pages inside the enclave. Similarly, if secrets could be inferred from observing page faults on data pages, then the application writer can configure the enclave to map heap or data segment using large pages. Given this information, the modified `signing_tool` at the client's computer creates a signature by augmenting the hash of the page contents with the hash of the page sizes. This signature is then sent to the remote computer for later verification over a secure channel. Beyond the signature, the virtual address offsets of each of the program segments to be mapped using large page is also sent to remote platform.

On the remote computer (i.e., the SGX-enabled cloud platform), the enhanced SGX driver is expected to map the enclave's data and/or code using large pages as requested by the client. During the enclave initialization, the SGX hardware is responsible for *guaranteeing* that the page sizes used to map enclave's virtual addresses are as desired, since the driver is untrusted. This is achieved by augmenting the hardware-computed hash of the enclave's contents with that of the corresponding page size used for mapping the enclave. This enhanced hash is then compared against the signature supplied by the client. A match guarantees that correct page sizes are used for mapping enclave's addresses.

The hardware further needs to ensure that the page sizes used for mapping enclave's virtual addresses are not altered *after* enclave initialization. Incidentally, the SGX hardware already performs

similar checks during virtual to physical address translation to ensure that the system software has not altered the mapping between enclave's virtual address and the physical address in the EPC. The check is extended to also ensure that page sizes used for mapping a given virtual address range has not been altered by the system software. Further, before raising a page fault exception to the system software, current SGX hardware masks out 12 bits of page offset to hide the exact faulting virtual address. SG^{XL} extends this by checking if the faulting address is within one of the segments that was to be mapped using large pages. If yes, then it masks the lower 21 bits (for 2-MB pages) instead of 12 bits.

SG^{XL} does not introduce any new components in the TCB. SG^{XL} is effective even in the presence of malicious software at the cloud platform, because the hardware guarantees the use of large pages to the enclave application.

We must be clear that SG^{XL} does not *theoretically* guarantee a defense against page-address-based attacks. Doing so would require SG^{XL} to provide the PAO property, which requires that the sequence of page accesses remains same irrespective of the input (secret). SG^{XL} only approximates PAO. However, as we will demonstrate in Section 6, for the applications we considered in our security evaluation, SG^{XL} can essentially achieve PAO in practice. We will further show that SG^{XL} can *complement* the compiler-based approach to achieve PAO [47], if theoretical guarantee is absolutely needed. This PAO-compiler inserts dummy accesses at various points in the program to ensure that the page access stream along various paths is the same. We show in Section 6 that when run on top of SG^{XL} , we can dramatically reduce the number of dummy accesses that have to be inserted. Thus, if desired, the PAO-compiler can be used together with SG^{XL} to reduce performance overheads of achieving PAO.

Qualitative security evaluation. While the use of large pages does not theoretically guarantee defense against any page-address based attacks, it typically renders them useless. To realize this, consider the `FreeType` example referenced in Section 3. With SG^{XL} , all the code of `FreeType` is loaded into one large (2-MB) page. Thus, an adversary will observe only that single page's address in the access stream, irrespective of the input processed. With respect to number of page faults, an observer will observe only 1 page fault for all the characters as opposed to the varying number of page faults observed when 4-KB pages are used as shown in Table 1.

SG^{XL} is equally effective against all three attack vectors discussed in in Section 3. For example, attacks that rely on `accessed` and `dirty` bits in the PTE to obtain the page address stream are similarly weakened as the metadata bits are maintained at the page-size granularity. Thus, a given memory region mapped using 2-MB pages will have 512× fewer number of `accessed` and `dirty` bits as compared to 4 KB mappings. TLBs cache address translations at the granularity of the page size used for mapping the given address range. Thus, under SG^{XL} , the attack vector that directly taps into the TLB would be able to observe the victim's page addresses only at the much lower resolution of large pages.

5 IMPLEMENTING SG^{XL}

As depicted in Figure 3, implementing SG^{XL} requires modifications to the SGX SDK, linker scripts, SGX driver and a few minor enhancements to the hardware. All the changes needed in the software stack are implemented in the SG^{XL} SDK and the SG^{XL} driver, hosted at https://github.com/csl-iisc/SGXL.git.

The hardware enhancements are limited to extending few SGX instructions and minor micro-architectural extensions to existing SGX functionality. We detail all necessary enhancements. We study the performance impact of the proposed hardware enhancements using HSPICE simulations [10]. The state, area, and power overheads of the proposed modifications are discussed in

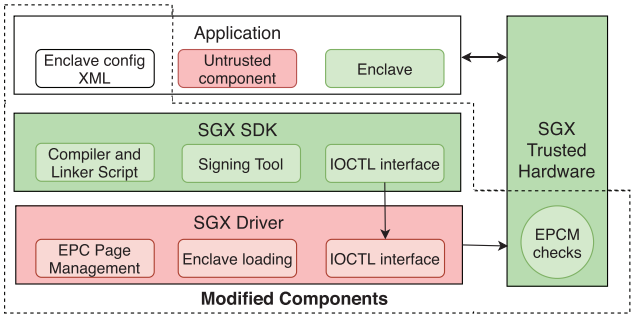


Fig. 3. SGX components to be modified to support SG^{XL}. Modified portions are within dotted lines.

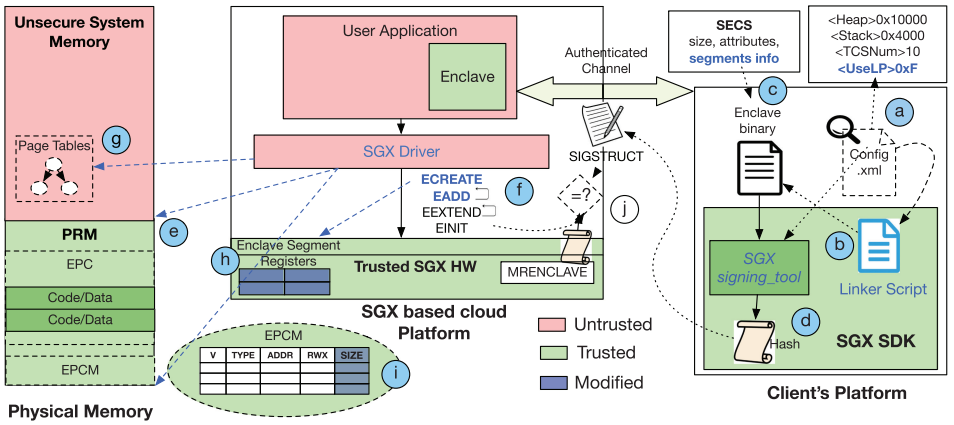


Fig. 4. Proposed modifications for SG^{XL}. Modifications are highlighted in blue.

Section 6.3. Our experiments affirm our intuition that minor hardware modifications introduced by SG^{XL} have negligible performance impact.

5.1 Modifications in the Local Client System

The first part of the implementation modifies the SDK in client's local computer (which is trusted, since the client is the code and data owner). Our implementation enables the client to specify which segment(s) of an enclave's virtual address space should be mapped using large pages, as depicted in the step ⓐ of Figure 4. For example, one could specify the code segment and/or the heap segment in the enclave be mapped using large pages. One can also specify all segments to be mapped using large pages. SG^{XL} 's enhanced parser then parses this file and augments the large page usage information to the enclave binary.

Next, we modify linker scripts to align the starting virtual addresses of each segment to be mapped using large pages at a large page (2-MB) boundary (step ⓑ of Figure 4). Note that the linker script need not be modified for the heap segment. Instead, the enhanced SDK's enclave creator aligns the heap's starting address at a large page boundary. The enclave SECS is augmented with segment start and end addresses, and if a segment is to be mapped with large pages (step ⓒ).

The third step is to create an augmented cryptographic hash that takes into account the desired page size. This hash is later matched with the hash generated by SG^{XL} -capable processor in the remote cloud platform. This is necessary to guarantee to the client that the system software at the remote platform is indeed using large pages. For implementing this, we extended

Intel's signing_tool to append the page sizes to the virtual address offsets while computing the hash. Specifically, the enhanced signing_tool computes a hash (SHA-256) over address space entries, which includes virtual address ranges with corresponding page sizes, along with other security/permission attributes. It augments the hash to the SIGSTRUCT data structure (step ⓓ). SIGSTRUCT is sent to the remote computer for it to be verified against the hardware-computed hash.

5.2 Modifications in the Remote Cloud Platform

The second part of the implementation is geared towards the software and the hardware at the remote cloud platform. The objective here is to be able to provide a guarantee to the client that the remote platform is *truly* using large pages. This guarantee is rooted in the trusted processor as is the case with the entire SGX philosophy. Most of the modifications for this step, however, are limited to the software stack with minor extensions in the hardware.

Software modifications. The most significant software modification happens to the SGX driver. The driver is responsible for mapping certain virtual address ranges with large pages and performs the associated management of multiple page sizes for the EPC memory.

The driver's ioctl interface is extended to include page size information. The driver itself is then enhanced in three ways: ① The driver now maintains a list of 2-MB contiguous physical page frames (besides the default 4-KB page frames) in the EPC for mapping large pages. On a request for a large page, the driver returns a physical memory region from the 2-MB list, and from the 4-KB list otherwise. ② The driver shall now issue an enhanced eadd instruction (detailed later) that accepts a page size argument. Since the enhanced eadd is not readily available in the current hardware, our present implementation of the driver emulates it by issuing eadd in a loop with 512 iterations to add a large page to the enclave. Similarly, the driver issues (unmodified) eextend instructions in a loop to compute the cryptographic hash of large page contents. ③ Finally, the driver creates page table entries for the enclave's virtual address ranges to be mapped using large pages. These steps are depicted as ⓔ, ⓕ and ⓖ of Figure 4.

Hardware and SGX ISA modifications. We now specify the changes in both the semantics and/or interface of SGX instructions and detail necessary micro-architectural enhancements. There are two sub-tasks. First, SG^{XL} needs to ensure that enclave is properly created with program segments mapped with large pages as desired by the client (conveyed through augmented SECS). Second, during enclave execution, the processor needs to ensure that malicious system software does not alter page sizes.

To accomplish the first sub-task, we extend ecreate to compute the hash of the segment information stored in the SECS, and augment it to the MRENCLAVE. This ensures that the segment information in the SECS has not been tampered with. When the eenter instruction is issued to transfer execution to the enclave, the enclave's segment information that is stored in the SECS is copied to the *Enclave Segment Registers* or ES registers (step ⓗ). There are eight ES registers (two for each segment) per-core that hold the the page sizes that each of the currently executing enclave segments are mapped to, along with the start and end addresses. Later in the section, we will discuss how these registers are used to determine the number of low bits to be cleared during a page fault.

Next, we extend the eadd instruction to include page size information. The eadd instruction, if specified with 2-MB page size, will be responsible for copying 2-MB memory region into the enclave memory. The instruction ensures that it obtains the lock for the entire 2-MB region before it start copying. Further, the eadd instruction will add page size information in the corresponding EPCM entries. For this purpose, we propose to extend each EPCM entry by 2 bits (step ⓘ) to hold page size information as depicted in the bottom part of Figure 4. eadd will also append the

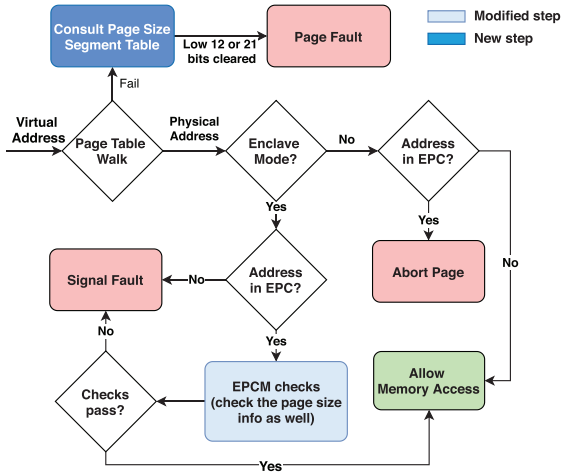


Fig. 5. Access control in SG^{XL}.

page size along with the newly added virtual address while computing the hash and augment that to the MRENCLAVE. During the enclave initialization (i.e.,when einit is issued), the content of the MRENCLAVE is matched against that provided in SIGSTRUCT information generated by the signing_tool at the client (step ⓙ), as done by today's SGX hardware. Since SG^{XL} takes into account the page size in computing the hashes, a match ensures that remote cloud platform is using large pages as desired.

To accomplish the second sub-task, i.e., that of ensuring page sizes are not altered during enclave execution, SG^{XL} minimally extends the checks that current SGX hardware performs, as depicted in Figure 5. On a TLB lookup, SGX hardware checks the integrity of the translation by validating it against the corresponding EPCM entry (Section 2). Similarly, before populating a new entry to the TLB after a page walk, the hardware validates it against EPCM. SG^{XL} extends these checks to include page size information too. A mismatch will trigger an error. This way, SG^{XL} ensures that a malicious OS cannot change the page size used for mapping during enclave execution. Note that the page walk handler can infer the page size information from a bit in the PTE (PSE bit), thus our design does not require any changes in the page table format or in the page table walker.

During a virtual address translation inside an enclave, if the page table walker fails to find a valid PTE or there is insufficient permission in the PTE, it raises a page fault exception to the system software. Currently, the hardware masks the lower 12 bits (page offset) of the faulting virtual address. This reveals only faulting virtual page number but not the entire address to potentially malicious system software. However, for a large page, the 21 least significant bits need to be masked for the same purpose. To determine the page size corresponding to the faulting address, the hardware consults the ES registers as shown in Figure 5. If the faulting virtual address falls within a virtual address range covered by one of segment registers that is to be mapped with large pages, then the lower 21 bits are cleared instead of lower 12 bits.

Finally, the eremove, ewb, etrack, eblock, emodpr, and eldu/eldb instructions have to be enhanced to handle large pages. The semantics of these instructions need not be modified significantly, because they are not directly involved in ensuring that the enclave is using the correct page sizes. These instructions need to be enhanced to manage 2-MB pages along with 4-KB pages, which they already do. For instance, the eremove instruction needs to be able to un-associate a large page from the SECS.

In summary, SG^{XL} proposes extensions to existing SGX instructions such as ECREATE and EADD to ensure that the enclave is mapped with the required size pages. To prevent a malicious OS from altering the page sizes after enclave initialization, i.e., during enclave execution, SG^{XL} augments the page table walker to verify the size of the page being accessed. SG^{XL} does not introduce new instructions or add new structures to the hardware. Since the hardware is already a part of the TCB in the SGX threat model, SG^{XL} assumes that an attacker cannot tamper with SGX instructions or the access checks performed during address translation. Therefore, an adversary cannot mislead an enclave into using the wrong page sizes.

Extending SG^{XL} for dynamic memory allocation (SGX2). While we focus on the currently available version of Intel's SGX, SG^{XL} can easily be extended to SGX2 that allows applications to dynamically map an enclave virtual page to a page frame, on demand (e.g., on first access) [53]. SGX2 provides enclave support for dynamic heap management, stack expansion, and thread context creation. Although it adds support for these dynamic features, the enclave still has to declare maximum sizes for the expandable components (i.e., heap and stack) during enclave creation. When the enclave makes a dynamic memory allocation request, SGX uses the newly introduced eaug and eaccept instructions to add pages. The eaug instruction allocates a new EPC page to the enclave after it is initialized. The eaccept instruction verifies that the specified attributes passed along with the instruction match the values in the EPCM to ensure page was initialized correctly.

To extend large page support to SGX with dynamic memory allocation, the eaug instruction should be extended to accept page size information, and be able to add a page of the desired size to the EPC. The semantics of the eaccept instruction needs to be extended to check the page size information in the corresponding entry in the EPCM. Any mismatch indicates a possible manipulation of page sizes and thus should exit enclave execution with an exception. Note that even with these enhancements for dynamic memory allocation, SGX requires the maximum sizes of the expandable components to be declared and the virtual address space to be reserved at enclave creation. From SG^{XL}'s perspective, the main difference is that the request will be satisfied with a large page. We do expect this to have performance implications, e.g., in enclave creation time, when pages are loaded (see Table 8), and in the time needed to handle page faults. To our knowledge, there is no publicly available study on the implications of dynamic memory allocation in enclaves, especially as relates to the relatively small amount of EPC memory currently available for use. Future work is required to comprehensively study the performance implications of using large pages in this context.

6 EVALUATION

We present empirical evidence that SG^{XL} 's efficacy in defeating page address based attacks by first recreating the attacks as described by Xu et al. [55] and then defending against them. We then evaluate the cost of proposed hardware modifications and present performance implications of SG^{XL} and discuss how it would scale to larger EPC sizes. We quantitatively show that, in practice, SG^{XL} comes very close to to achieving a theoretically guaranteed defense such as that provided by the PAO-compiler [47]. Finally, for applications where a *guarantee* of PAO is necessary, we show that SG^{XL} reduces the number of dummy memory accesses that the PAO-compiler would have to insert.

6.1 Methodology

We performed the software development on a system with Intel core i7-8700 processor that supports SGX (v1). SGX (v1) allows an EPC of maximum 128 MB. The system has 32 GB of DDR4-2600 memory and runs Linux 4.14.0.

Table 2. Applications Evaluated

Application	Version	Configuration/Inputs
FreeType [16]	2.5.3	Text file with 39,579 words (208,314 characters)
Hunspell [22]	9b	
Libjpeg [27]	1.6	Five JPEG images with 2500×1000 resolution
GUPS [20]		Global table length=2^32, Update sets=10^4, Chunk=65,536
NBench [42]	2.2.3	Consists of 10 sub-tests such as Sort, LU decomposition and Neural Net
OpenSSL [34]	1.1.1b	Custom testbench containing multiple cryptographic operations

Table 3. Empirical Security Analysis for SG^{XL}

			Number of page faults			Number of unique bigrams		
	Section	Size	Baseline	SG^{XL}	% change	Baseline	SG^{XL}	% change
FreeType [16]	Code	1.49 MB	224,251,608	1	99.99%	506	0	100%
Libjpeg [27]	Data	32.03 MB	24,360	49	99.80%	15,727	18	99.72%
Hunspell [22]	Data	4.6 MB	590,659	169,642	71.28%	22,869	8	99.97%
GUPS [20]	Heap	82 MB	662,272,033	594,000,830	10.30%	2,825,638	421	99.98%
NBench [42]	Data	16 MB	409,200	16	99.99%	182	5	97.25%
OpenSSL [34]	Code	1.75 MB	28,108,308	1	99.99%	1,203	0	100%

Porting applications to SGX are entire projects unto themselves. Applications typically need significant modifications to be able to make use of enclave execution in SGX. For example, applications cannot invoke system calls from within the enclave. We used Porpoise [41] to port the benchmarks (see Table 2) to use enclave execution, except for NBench, which had already been ported in prior work [42]. Porpoise [41] is a set of in-enclave libraries that shepherds the execution of certain instructions (e.g., syscall) that are forbidden in enclave-mode execution.

6.2 Empirical Security Analysis

While our defense is equally effective against all three page-address based attack vectors, we empirically evaluate SG^{XL} against the page-fault inducing attacks of Xu et al. [55]. In these attacks, the post-processing stage following the collection of page address sequences differs based on the victim program. For some, merely counting the number of page faults yields enough information about the secret; for others, post-processing relies on identifying a unique sequence of faulting addresses.

We measure two common metrics across all workloads for our security analysis. First, we count the number of page faults incurred, under the attack, with baseline SGX (current hardware), which uses 4-KB pages and SG^{XL}, which supports 2-MB pages. A large reduction in the number of faults would demonstrate the efficacy of SG^{XL} in thwarting attacks that rely on the number of page faults. However, the count may not be a good metric for attacks that exploit *unique sequences* of fault addresses. Therefore, we also collect the number of *unique bigrams* in fault address stream.[1] A larger number of unique bigrams indicates a larger number of unique sequences and thus, more information to infer the secrets.

Table 3 lists our findings. For each workload, the first column describes what part of the application was tracked for page faults. The next two (sub-)columns present the number of page faults incurred in baseline SGX and with SG^{XL}, respectively. The third sub-column shows the percentage

[1]Given a sentence in a particular alphabet, a bigram is a consecutive pair of alphabets in that sentence; here, each bigram is a pair of page fault addresses that appear in the page fault stream.

reduction in the page fault count. The next three sub-columns show the same for the number of unique bigrams.

We now discuss how each application is attacked and how the numbers in the Table 3 show the efficacy of SG^{XL}. For the first three applications, we replicated Xu et al.'s attacks [55].

FreeType: This application renders characters, and the attacker attempts to infer which character (secret) is being rendered by tracing code path execution by counting page faults on code pages. The execution path and thus the number of code pages accessed create unique signature for each character the application renders. From Table 3, we observe that SG^{XL} eliminates more than 99.9% of the page faults thus rendering the attack useless. This is possible, since a single large page (2 MB) can map the entire 1.5-MB code of `FreeType`.

Libjpeg: This application decompresses a picture, and the number of data pages accessed is different based on whether a pixel is on the edge of a figure. By counting the number of data page faults, attacker can trace the edges of the (secret) figure being decompressed. We observe more than 99.7% reduction in the page fault count that easily defeats the attack.

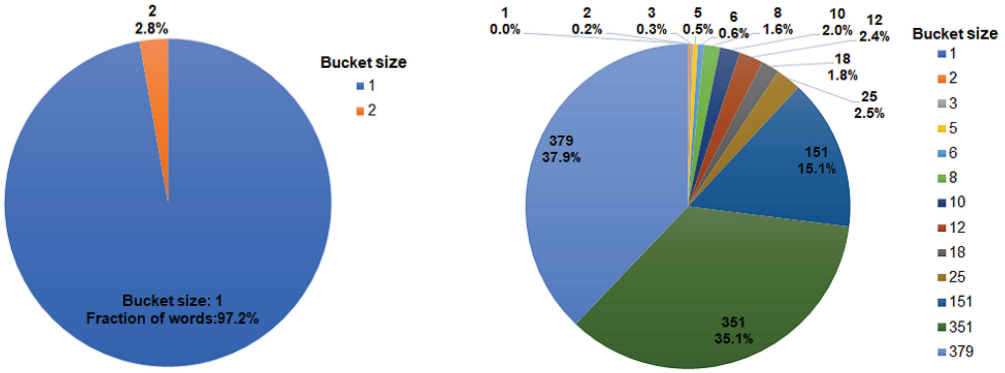
Note that the number of bigrams is irrelevant for attacking these two applications as they rely on page fault counts to infer secrets. However, SG^{XL} reduces even the number of unique bigrams by 100% and 99.7%, respectively.

Hunspell: This application searches a dictionary for input words to check their spellings. It does so by traversing linked lists whose head pointers are found by hashing the word. If the linked list nodes fall on different pages, then the sequence of page faults can reveal the word being looked up. The attacker traces data page faults and looks for unique sequences of fault address to infer a word. Consequently, the relevant metric here is the number of unique bigrams. Fewer unique bigrams suggests a lower likelihood of finding a sequence of faults that can uniquely identify a word. Table 3 shows that SG^{XL} reduces the number of unique bigrams by more than 99.7%.

To further understand the effectiveness of SG^{XL}, we analyzed the individual fault sequences for 1000 words in the dictionary. Fault sequences are not to be confused with bigrams (reported in Table 3). In particular, a sequence of faults could produce multiple bigrams. We grouped words in the input dictionary using the page fault sequence that they induce in Hunspell. If an input word induces a unique page fault sequence, then it can uniquely be identified. In contrast, if many words share the same page fault sequence, the attacker can at best identify the bucket of words that share that sequence, but cannot identify the input word uniquely. Figure 6 shows the distribution of page fault sequences for 1,000 words from the `en_US` Hunspell dictionary. Figure 6(a) shows that on Hunspell running within a traditional SGX enclave, 97% of the input words induced a unique page fault sequence (i.e., bucket size of 1). Fewer than 3% of the words shared their page fault sequence with another word. In those cases, the attacker can still guess a word with a probability of 50%.

In contrast, when we ran Hunspell atop SG^{XL}, no word induced a unique page fault sequence. Most words fell into buckets of size 151, 351, or 379 (a bucket size of 151 means that 151 words shared the same page fault sequence). On average, we found that the bucket size increases by 65× when SG^{XL} with 2-MB pages was employed.

GUPS, NBench and OpenSSL: We conducted similar experiments on these 3 micro-benchmarks. However, there were no secrets to infer here. The only purpose was to demonstrate the applicability of the defense across a wider set of workloads. `GUPS` is a micro-benchmark that allocates a large array. It then looks up one entry in random and updates it. Due to random accesses, even with the use of large pages in SG^{XL}, a relatively larger number of faults occur. However, we found page fault stream to be useless as the faults are across a few large pages that cover the heap (e.g., 10s). This is also captured by a large reduction in the number of unique bigrams with SG^{XL}.



(a) Baseline (b) SG^{XL}

Fig. 6. Distribution of words across page fault sequence buckets in Hunspell. Each bucket denotes a unique page fault sequence. The size of each bucket denotes the number of words sharing that page fault sequence. The smaller the bucket size, the easier it is for an attacker to identify an input word uniquely. The pie charts depicts the size of the bucket and the fraction of input words that fall into the corresponding bucket size. Without SG^{XL} (part (a)), over 97% of the input words can be uniquely identified with their page fault sequences (bucket size of 1). With SG^{XL} (part (b)), no word can be uniquely identified with its page fault sequence.

NBench consists of multiple micro-benchmarks such as sorting, neural nets, LU decomposition, Fourier transforms. SG^{XL} vastly reduces the number of faults and unique bigrams.

Lastly, we created a custom benchmark using the OpenSSL library. It includes widely used cryptographic functions such as DES, RSA, HMAC, and SHA. By monitoring accesses to code pages, an adversary can possibly infer the operation being performed. We observe that SG^{XL} can completely thwart information leakage through page faults, because the entire code could fit in a single 2-MB page.

6.3 Hardware Cost Evaluation

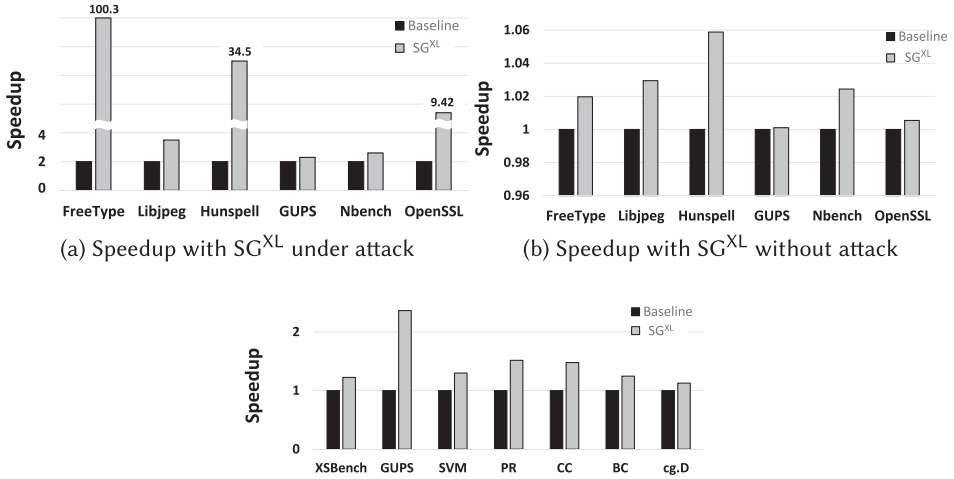
The only additional hardware state needed for SG^{XL} are ① 2-bit page size information in each EPCM entry and ② eight registers per core to keep virtual address ranges for segments of an SGX program that could be mapped with large pages (two registers each for the code, data, BSS, and heap segments).

While the exact size of EPCM entries is not publicly known, Intel SGX reference guide specifies that an EPCM entry would contain SECS identifier, enclave virtual address and several other metadata bits like valid, read/write [24]. Based on this, we calculate that a typical EPCM entry could be around 82 bits. For SG^{XL}, one need to add two more bits per entry for encoding page size. Given the EPC size of 128 MB and base page size of 4 KB, 32,768 EPCM entries would cover the entire EPC. Table 4 shows calculated EPCM area, access latency, and energy, without and without modifications for SG^{XL}. We used FabMem tool with HSPICE simulations to generate these numbers [10]. We find that SG^{XL} adds 2.37% area overhead for EPCM while there is barely any change in the EPCM access latency. Access energy increases by only 1.9% Beyond EPCM, SG^{XL} adds only eight registers per core and does not add any measurable area overhead.

As detailed in Section 5, the EPCM is looked up on a TLB fill (page table walk) to ensure system software has not modified the virtual to physical address mapping or page sizes. However, SG^{XL} does not add any additional latency in this path (Table 4). SG^{XL} looks up eight registers only on

Table 4. EPCM Area, Latency, and Energy Overhead

	SGX (Baseline)	SG^{XL}	% increase
Size (bytes)	335872	344064	2.439
Area (um^2)	582515.232	596334.024	2.372
Access latency (ns)	3.346	3.347	0.029
Access Energy (pJ)	200.865	204.74	1.929



(a) Speedup with SG^{XL} under attack

(b) Speedup with SG^{XL} without attack

(c) Projecting for EPC sizes with 1G pages

Fig. 7. Performance Evaluation.

page faults and thus cannot add any measurable additional latency as a page fault itself take several micro-seconds to get serviced.

In short, the hardware modifications required for SG^{XL} will not increase the latency in the critical path of operation. It adds minimal state overheads. We, therefore, find no reason to simulate SG^{XL} in processor simulators for evaluation.

6.4 Performance Implications

Enhanced security typically comes at a performance cost. However, no such tradeoff is necessary here. Figure 7(a) shows the speedup with SG^{XL} when applications are under attack. The speedup is calculated by dividing the execution time on the current SGX hardware by that with SG^{XL} mapping enclave memory with 2-MB pages. SG^{XL} significantly speeds up all applications under attack (up to 103.5×). This is expected as the use of large pages significantly reduces the number of faults (Table 3).

Figure 7(b) shows the speedup when the system is not under attack but SG^{XL} is employed. We observe that there is barely any change in performance. One could however, expect that performance could have still improved due to the likely reduction in the number of TLB misses due to use of large pages. However, the memory footprints (application *needs* to fit into EPC) of the workloads studied are too small to make any significant difference in performance.

Memory overheads. One of the implications of using large pages is the increase in memory footprint due to internal fragmentation. Table 5 shows the sizes of the application enclaves and

Table 5. Memory Bloat Caused by SG^{XL}

	Baseline	SG^{XL}	% increase
FreeType	5,236 KB	6,268 KB	1.19%
Libjpeg	35,884 KB	39,940 KB	1.11%
Hunspell	19,664 KB	23,516 KB	1.19%
GUPS	86,884 KB	90,964 KB	1.05%
NBench	2,652 KB	6,704 KB	252.79%
OpenSSL	9,936 KB	10,184 KB	1.02%

the bloat in memory caused by SG^{XL}. Only `NBench` showed a sgnificant memory bloat. `NBench` has a very small memory footprint with small segment sizes. Mapping these small segments with large pages results in a larger enclave size.

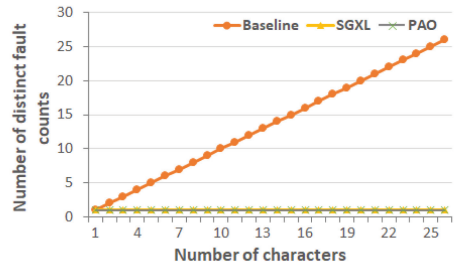
Projecting for larger EPC sizes. It has been speculated that future editions of the SGX will significantly increase the size of the EPC [25]. A larger EPC size is essential for commercially important applications with large memory footprints, e.g., databases, to benefit from SGX [38]. Applications with larger memory footprint are typically accompanied by larger address translation overheads [7]. We thus believe that SG^{XL} 's use of large pages to map enclave's address space could assume further importance, not only for enhanced security but also for performance, in future iterations of SGX.

To understand the importance of using large pages for applications with few GBs of memory footprint, we experimented with a couple of HPC programs (`XSBench` [54], `CG.D`), support vector machine [28], page rank [18], connected components [18], and between-ness centrality [18]. We also ran `GUPS` [20] with much larger memory footprint (32 GB). We studied the impact of 1-GB large pages on the performance of these applications but did not evaluate the security implications of using large pages with these applications. As stated earlier, 1-GB large pages reduce the resolution of page address stream by a large factor when compared to 4-KB pages. We believe using 1-GB pages could approximate PAO for applications with a much larger footprint, spanning beyond hundreds of MBs and few GBs. In Figure 7(c) we show the speedups achievable for these memory-intensive applications with 1-GB large pages (the largest page size available today in x86-64 processors) over the default 4-KB pages. All executions are outside the enclave. The impressive speedups (1.15× to 2.36×) reiterate the importance of mapping enclave memory with larger page sizes, as done in SG^{XL}, for larger EPC sizes.

6.5 Approximating PAO with SG^{XL}

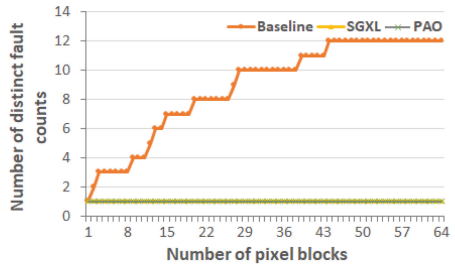
Figure 8 shows how closely SG^{XL} mimics PAO in practice. PAO theoretically guarantees absence of page address based attacks by requiring page access patterns to remain the same irrespective of the input to the enclave program.

As discussed in Section 6.2, attacks on `FreeType` and `Libjpeg` rely on page fault counts to infer secrets. In case of `FreeType`, if the rendering different characters generates a different number of faults then the attacker can easily infer the character (secret). However, if every character generates the same number of faults then the attack fails. Figure 8(a) shows how the number of unique page fault counts grows with the number of unique characters rendered under SGX, SG^{XL} , and PAO. We observe that the count of unique faults remains the same with both SG^{XL} , and PAO. Thus, it is not possible to infer secrets. Figure 8(b) shows similar measurements for `Libjpeg`. Here also, SG^{XL} and PAO performs the same way and leaves no unique page fault count information for attacker to identify the pixel block being compressed.



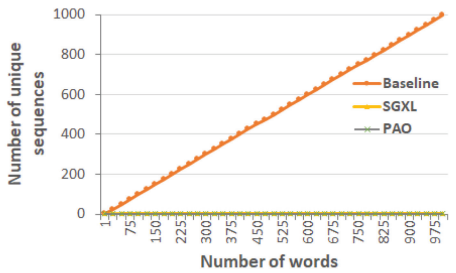
(a) FreeType

Number of characters	Number of distinct page fault counts		
	Baseline	SG^{XL}	PAO
0-6	6	1	1
0-12	12	1	1
0-18	18	1	1
0-26	26	1	1



(b) Libjpeg

Number of pixel blocks	Number of distinct page fault counts		
	Baseline	SG^{XL}	PAO
0-16	7	1	1
0-32	10	1	1
0-48	12	1	1
0-64	12	1	1



(c) Hunspell

Number of words	Number of unique sequences		
	Baseline	SG^{XL}	PAO
0-100	99	3	1
0-250	249	4	1
0-500	499	4	1
0-1000	996	4	1

Fig. 8. SG^{XL} 's approximation of PAO.

The attack on `Hunspell` relies on unique page fault address sequences to identify the words being looked up. We thus measure the number of unique fault address sequences generated as the number of unique words being looked up grows (Figure 8(c)). SG^{XL} closely approximates PAO—PAO generates one address pattern while SG^{XL} generates up to four while looking up one thousand unique words. In contrast, today's SGX hardware generates nearly a thousand unique patterns, leaving enough information for the attacker to exploit.

6.6 SG^{XL} and PAO-compiled Applications

Sinha et al.'s PAO-compiler introduces dummy accesses to obfuscate page access patterns, particularly at page boundaries [47]. To show how SG^{XL} can complement their PAO-compiler, we considered `FreeType`, `Hunspell`, and `Libjpeg`. These are the three applications against which Xu et al. [55] demonstrated controlled-channel attacks. We restricted ourselves to analyzing the specific functions that were targeted in the attacks demonstrated by Xu et al. We did not evaluate

Table 6. Dummy Accesses Needed to Ensure PAO

	Function/Data Structure	Baseline	SG^{XL}
FreeType	`load_truetype_glyph`	157	0
LibJpeg	`jpeg_idct_islow`	28	0
Hunspell	`tableptr[]` (dictionary)	852	3

how SG^{XL} could aid the PAO-compiler for the other applications, since the attack vectors for those applications were not defined.

We attempted to use Sinha et al.'s PAO-compiler to produce enclave applications. However, we were unable to produce PAO-compliant enclaves for any of the above applications. There were a number of reasons for this. First, their PAO-compiler prototype requires applications to be rewritten in a dialect of C, which we did not do, because that would require significant rewriting of the applications. Second, the PAO-compiler currently offers limited support for heap-allocated data structures, which the above applications use. In particular, the PAO-compiler requires the programmer to specify the size and type of object as an argument to `malloc()`.[2] Moreover, the prototype of the PAO-compiler that was shared with us did not support heap allocation at all. Finally, we determined (using microbenchmarks) that their PAO-compiler prototype does not currently scale to applications that consist of more than a few tens of lines of code. We thus used the `angr` binary analysis tool [45, 46, 48] to estimate the number of dummy accesses that the PAO-compiler would have inserted.

Table 6 presents the results. With SG^{XL}, we were able to fit the code/data to be protected in `FreeType` and `Libjpeg` within a 2-MB page and no dummy accesses were necessary. Contrast this with current SGX hardware that can use only 4-KB pages to map enclave memory. If an application's secret code or data exceeds a large page, then the PAO-compiler would have to instrument dummy accesses even with SG^{XL}. However, the number of dummy accesses that would be required with SG^{XL} would be significantly smaller (by around 512×). SG^{XL} can therefore be used to improve the efficiency of PAO-compiled applications. For instance, the secret data in `Hunspell` spans three 2-MB large pages resulting in a small number of dummy accesses when using SG^{XL}.

7 RELATED WORK

As detailed in Section 3, there are three attack vectors via which an adversarial system software can extract the page address stream of the enclave execution [51, 52, 55]. The research community has been investigating various lines of defense for these attacks (see Table 7).

T-SGX [42] and Deja Vu [9] use transactions, as supported by Intel TSX [12], to detect attacks. TSX allows page faults to be directly delivered to user transactions bypassing the adversarial system software. In T-SGX, the enclave application detects if it is under attack by setting a threshold for the number of observed page faults. Deja Vu uses TSX to set up a reference clock that enclave code can consult to determine if it has been interrupted too often. The main drawback of these defenses is that they are tailored to page-fault based attacks *only*. Consequently, these defenses are ineffective against other attack vectors that monitor the PTE without causing page faults [51, 52] and those that directly tap into the TLB to observe the page access stream [19] (Table 7).

SGX-Lapd [17] also proposes the use of large pages to defend against page fault based attacks. It instruments the enclave code to detect whether page faults were encountered at jumps that

[2]The SGX hardware that we use does not allow enclave code to dynamically allocate new memory on the heap. However, `malloc` requests by enclave code can be emulated by preallocating heap memory at enclave initialization, and satisfying `malloc` requests from this preallocated memory.

Table 7. Comparing SG^{XL} with Related Work

	Page faults		PTE monitoring	TLB tapping	Legacy support
	Code	Data			
T-SGX [42]	✔	✔	✖	✖	✖
DejaVu [9]	✔	✔	✖	✖	✖
SGX-LAPD [17]	✔	✖	✖	✖	✔
InvisiPage [2]	✔	✔	✔	✖	✔
PAO-compiler [47]	✔	✔	✔	✔	✖
SG^{XL}	✔	✔	✔	✔	✔

By "legacy support," we mean that the solution works without invasive source-level modifications.

cross 4-K page boundaries. The underlying idea is that if the code is indeed allocated on a large page (2 MB), then the enclave should not encounter page faults for control flows that cross 4-KB boundaries. At each indirect jump that crosses a 4-K page boundary, the inserted instrumentation code looks up the EXINFO structure in the enclave's thread-control structure populated by the hardware on a page fault. If the fault is unexpected, then the enclave detects an attack.

However, SGX-LAPD has at least three major deficiencies relative to SG^{XL}. ① It *only* defends against attacks on code pages, and *not* on data pages (Table 7). ② It defends against only *one of the three* known attack vectors (page faults). ③ Most importantly, the authors of T-SGX [42, Section III.D] illustrate that malicious system software can overwrite EXINFO and fool an enclave program into thinking that no page fault was raised. The authors of SGX-LAPD also acknowledge the possibility of this attack [17, Section 7], but contend that such attacks may be difficult to launch. Nevertheless, this is an attack that completely defeats the defense offered by SGX-LAPD. In fact, we argue that it is impossible to obtain strong guarantees about the page size allocated to the enclave without hardware support, e.g., the kind that SG^{XL} introduces.

Inspired by the Oblivious RAM (ORAM) literature, a number of projects have attempted to achieve memory access obliviousness by building oblivious primitives for specific application domains [3, 15, 31, 33, 39]. InvisiPage [2] is one such ORAM-inspired defense tailored for page address-based side-channels. InvisiPage uses enclave-managed paging [13, 14, 35] (not currently available on the SGX) that puts the enclave itself in charge of performing its own page management for EPC. This prevents the adversarial system software from directly manipulating PTEs of pages within the EPC. However, pages are routinely swapped from EPC to non-EPC main memory and then to secondary storage. Malicious system software can leverage this channel to identify the pages accessed by the enclave. InvisiPage defends against this channel by using ORAM to obfuscate page movements between the EPC and non-EPC memory. Although InvisiPage incurs performance overheads, it provides cryptographically strong security guarantees compared to other proposed solutions, in that it provides oblivious demand paging. Any access to a page de-allocated by the OS will leak the accessed address to the OS, and InvisiPage uses ORAM to cryptographically protect against such leaks. However, InvisiPage cannot defend against attacks that directly tap into the TLB [19]. On the contrary, SG^{XL} provides weaker guarantees but a practical defence against all the known attack vectors at no performance overheads. KLOTSKI [56] is another defense that uses an ORAM-based runtime to obfuscate the memory accesses by shuffling code and data in software caches. KLOTSKI uses a software memory management unit that coverts compile-time virtual addresses to runtime virtual addresses. This adds significant overhead and the article presents techniques to tradeoff security for performance.

Strackx and Piesens [49] develop an approach that pre-loads the TLB with page address translations that are needed by the enclave. By re-loading these translations each time the TLB is flushed

Table 8. Enclave Creation Times

	Enclave Creation Time (in ms)	
	Baseline	SG^{XL}
FreeType	0.177	0.192
Libjpeg	0.209	0.411
Hunspell	0.168	0.316
GUPS	0.588	0.862
Nbench	0.041	0.106
OpenSSL	0.197	0.216

(e.g., upon a page fault), this approach ensures that the adversarial OS cannot obtain page numbers by interrupting enclave execution. However, the hardware-implementation of this defense at the mercy of micro-architectural configurations (e.g., size/associativity of TLB), and the prototype is limited to enclaves of size <6 MB. Moreover, a TLB shootdown from a malicious OS can render this defense useless.

To our knowledge, the only work (other than SG^{XL}) that defends against all known attack vectors is the PAO-compiler of Sinha et al. [47]. It achieves PAO via a careful layout of enclave code and data and insertion of dummy accesses to obfuscate page accesses. Unfortunately, PAO-compiler imposes significant restrictions, e.g., the enclave program cannot have loops dependent on a secret. PAO-compiler also requires all code to be rewritten in a restricted dialect of C (Enclang) and thus prevents its applicability to legacy programs. Moreover, any workload of reasonable code and memory footprint would require significant number of dummy memory accesses. Due to these restrictions we hypothesize that their approach would not scale to any practical workloads. Nevertheless, as we illustrated in Table 6, SG^{XL} can complement and reduce the number of memory accesses in a PAO-compiler.

8 DISCUSSION

SG^{XL} uses large pages to defend against page-address based side-channels. SG^{XL} reduces the resolution of the page access stream (obtained by observing page faults), thus leaving an attacker with very little information to infer secrets from within the enclave. We also observed empirically that SG^{XL} does not degrade the performance. In fact, most applications could benefit from SG^{XL}. However, we acknowledge that there are some constraints and tradeoffs associated with the SG^{XL} design described in this article, and we discuss them here.

The use of large pages may result in memory bloat and increase internal fragmentation. Given the limited size of the EPC in the current version of SGX, large pages could aggravate contention for EPC pages. Table 5 shows the increase in memory footprint for the applications that we considered. The use of large pages may also increase the burden on system software. In particular, the memory manager may be forced to move multiple 4-K pages to make room for a large page, resulting in increased page fault latencies. This factor could become more relevant with the upcoming version of SGX that supports dynamic memory allocation.

To understand the implications of using large pages on system software, we analyzed the time taken to setup enclaves using 4-KB pages and SG^{XL} . Table 8 shows the enclave creation times for different applications. Enclave creation mainly consists of adding pages to the EPC using the `eadd` instruction and calculating the hash of the loaded contents using `eextend`. All applications see a small increase in enclave creation times. However, we feel that this overhead is acceptable, since enclaves are not created and destroyed frequently. Also, the initialization time is in milliseconds, which is negligible when compared to the execution time of the application.

SGX's core design philosophy introduces rigidity in the page allocation strategy of SG^{XL}. In particular, SGX's threat model treats system software as untrusted, yet depends on it to manage the EPC. SGX requires the enclave configuration to be cryptographically measured (i.e., using hashes) before it is loaded into the SGX platform, and no further changes are allowed. In particular, this configuration includes the details of how pages are mapped. This requirement introduces rigidity in the page allocation strategy of SG^{XL}. Application developers are required to specify the page sizes to be used by the enclave during compile time. SG^{XL} hardware subsequently ensures that the system software has allocated the correct page sizes (during runtime). A more flexible adaptation would allow the host to automatically update 4-KB pages to large pages, similar to Linux's transparent huge pages support. Since SGX assumes that system software could be adversarial, SG^{XL} cannot adopt such a flexible strategy. However, if the secure runtime inside SGX is augmented to handle page management actions, this secure runtime could upgrade 4-KB pages to large pages transparently without an explicit request from the application creator.

Finally, SG^{XL} does require additional effort on the part of application developers to specify the secure page sizes to be used by their application. It may be possible to develop static binary analysis techniques or runtime profiling techniques to determine the "right" page size for different portions of an application to achieve security. However, we have not attempted to do so in this article. An application writer could also be conservative and request for large pages if he believes the application to be vulnerable to page-address based side-channel attacks.

9 CONCLUSION

SG^{XL} is a hardware-software co-designed approach that uses large pages to make page-address based side-channel attacks harder to mount against enclave applications. We believe that SG^{XL} represents a new point in the design space of defenses. While it does require minor changes to hardware, and does not guarantee defense against page-based side-channel attacks, it is unique in that it increases the difficulty of the attacks while generally *improving* the performance. SG^{XL} is *not* tied to a particular attack strategy, and is effective against all known variants of page-address based side-channel attacks.

ACKNOWLEDGMENTS

We thank Sirvisetti Venkat Sri Sai Ram for assisting us with Figure 7(c) and Kripa Shanker for help setting up Porpoise [41]. We also thank Michael M. Swift and Moinuddin Qureshi for comments on earlier drafts of this work.

REFERENCES

[1] S. Aga and S. Narayanasamy. 2017. InvisiMem: Smart memory defenses for memory bus side channel. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*.

[2] S. Aga and S. Narayanasamy. 2019. InvisiPage: Oblivious demand paging for secure enclaves. In *Proceedings of the 46th Annual International Symposium on Computer Architecture*.

[3] A. Ahmad, K. Kim, M. Sarfaraz, and B. Lee. 2018. Obliviate: A data oblivious file system for Intel SGX. In *Proceedings of the Networked and Distributed Systems Security Symposium*.

[4] I. Anati, S. Gueron, S. P. Johnson, and V. R. Scarlata. 2013. Innovative technology for CPU based attestation and sealing. In *Proceedings of the Workshop on Hardware and Architectural Support for Security and Privacy*.

[5] ARM. 2009. Security technology building a secure system using TrustZone technology (Whitepaper). *ARM Limited* (2009).

[6] S. Arnautov, B. Trach, F. Gregor, T. Knauth, A. Martin, C. Priebe, J. Lind, D. Muthukumaran, D. O'Keeffe, M. Stillwell, D. Goltzsche, D. Eyers, R. Kapitza, P. Pietzuch, and C. Fetzer. 2016. SCONE: Secure Linux containers with Intel SGX. In *Proceedings of the ACM/USENIX Symposium on Operating System Design and Implementation*.

[7] A. Basu, J. Gandhi, J. Chang, M. Hill, and M. Swift. 2013. Efficient virtual memory for big memory servers. In *Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA'13)*. ACM, New York, NY, 237–248. DOI: https://doi.org/10.1145/2485922.2485943

[8] A. Baumann, M. Peinado, and G. Hunt. 2015. Shielding applications from an untrusted cloud with Haven. *ACM Trans. Comput. Syst.* 33, 3 (September 2015).
[9] S. Chen, X. Zhang, M. Reiter, and Y. Zhang. 2017. Detecting privileged side-channel attacks in shielded execution with Deja Vu. In *Proceedings of the ACM Asia Conference on Computer and Communications Security*.
[10] N Choudhary, Salil Wadhavkar, Tanmay Shah, Sandeep Navada, H. Najaf-Abadi, and Eric Rotenberg. 2009. Fabscalar. In *Proceedings of the 4th Workshop on Architectural Research Prototyping*.
[11] J. Corbet. 2011. Transparent Huge Pages in 2.6.38. Retrieved from https://lwn.net/Articles/423584/.
[12] Intel Corporation. 2012. Intel transactional synchronization extensions. In *Intel Architecture Instruction Set Extensions Programming Reference (4th Generation Core Processors)*.
[13] V. Costan, I. Lebedev, and S. Devadas. 2018. Sanctum: Minimal hardware extensions for strong software isolation. In *Proceedings of the USENIX Security Symposium*.
[14] X. Dong, Z. Chen, J. Criswell, A. Cox, and S. Dwarakadas. 2018. Shielding software from privileged side-channel attacks. In *Proceedings of the USENIX Security Symposium*.
[15] B. Fisch, D. Vinayagamurthy, D. Boneh, and S. Gorbunov. 2017. IRON: Functional encryption using the SGX. In *Proceedings of the ACM Conference on Computer and Communications Security*.
[16] freetype [n.d.]. The FreeType Project: A Free, High-quality and Portable Font Engine. Retrieved from https://www.freetype.org.
[17] Y. Fu, E. Bauman, R. Quinonez, and Z. Lin. 2017. SGX-LAPD: Thwarting controlled side-channel attacks via enclave verifiable page faults. In *Proceedings of the 20th International Symposium on Research in Attacks, Intrusions and Defenses*.
[18] GAPBS [n.d.]. GAP Benchmark Suite Intended to Help Graph Processing Research. Retrieved from https://github.com/sbeamer/gapbs.
[19] B. Gras, K. Razavi, H. Bos, and C. Giuffrida. 2018. Translation leak-aside buffer: Defeating cache side-channel protections with TLB attacks. In *Proceedings of the USENIX Security Symposium*.
[20] gups [n.d.]. GUPS: HPCC RandomAccess Benchmark. Retrieved from https://github.com/alexandermerritt/gups.
[21] M. Hoekstra, R. Lal, P. Pappachan, V. Phegade, and J. del Cuvillo. 2013. Using innovative instructions to create trustworthy software solutions. In *Proceedings of the Workshop on Hardware and Architectural Support for Security and Privacy*.
[22] hunspell [n.d.]. Hunspell: Spell Checker of LibreOffice, OpenOffice.org, Mozilla Firefox 3, Thunderbird and Google Chrome. Retrieved from http://hunspell.github.io.
[23] T. Hunt, Z. Zhu, Y. Xu, S. Peter, and E. Witchel. 2016. Ryoan: A distributed sandbox for untrusted computation on secret data. In *Proceedings of the ACM/USENIX Symposium on Operating System Design and Implementation*.
[24] Intel. 2014. Software Guard Extensions Programming Reference, Revision 2. Retrieved from https://software.intel.com/sites/default/files/managed/48/88/329298-002.pdf.
[25] Intel-forum. 2018. SGX2.0 PRM Size. Retrieved from https://software.intel.com/en-us/forums/intel-software-guard-extensions-intel-sgx/topic/757950.
[26] Y. Kwon, H. Yu, S. Peter, C. Rossbach, and E. Witchel. 2016. Coordinated and efficient huge page management with Ingens. In *Proceedings of the ACM/USENIX Conference on Operating Systems Design and Implementation*.
[27] libjpeg [n.d.]. Libjpeg: A Widely Used C Library for Reading and Writing JPEG Image Files. Retrieved from http://libjpeg.sourceforge.net.
[28] liblinear [n.d.]. LIBLINEAR: A library for Large Linear Classification. Retrieved from https://www.csie.ntu.edu.tw/cjlin/liblinear/.
[29] J. Lind, C. Priebe, D. Muthukumaran, D. O'Keeffe, P. Aublin, F. Kelbert, T. Reiher, D. Golzsche, D. Eyers, R. Kapitza, C. Fetzer, and P. Pietzuch. 2017. Glamdring: Automatic application partitioning for Intel SGX. In *Proceedings of the USENIX Annual Technical Conference*.
[30] F. McKeen, I. Alexandrovich, A. Berenzon, C. V. Rozas, H. Shafi, and U. R. Savagaonkar V. Shanbhogue. 2013. Innovative instructions and software model for isolated execution. In *Proceedings of the Workshop on Hardware and Architectural Support for Security and Privacy*.
[31] P. Mishra, R. Poddar, J. Chen, A. Chiesa, and R. Popa. 2018. Oblix: An efficient oblivious search index. In *Proceedings of the IEEE Symposium on Security and Privacy*.
[32] J. Navarro, S. Iyer, P. Druschel, and A. Cox. 2002. Practical, transparent operating system support for superpages. In *Proceedings of the ACM/USENIX Conference on Operating System Design and Implementation*.
[33] O. Ohrimenko, F. Schuster, C. Fournet, A. Mehta, S. Nowozin, K. Vaswani, and M. Costa. 2016. Oblivious multi-party machine learning on trusted processors. In *Proceedings of the USENIX Security Symposium*.
[34] openssl [n.d.]. OpenSSL: Cryptography and SSL/TLS Toolkit. Retrieved from https://www.openssl.org/.
[35] M. Orenbach, P. Lifshits, M. Minkin, and M. Silberstein. 2017. Eleos: Exitless OS services for SGX enclaves. In *Proceedings of the European Conference on Computer Systems*.

[36] A. Panwar, S. Bansal, and K. Gopinath. 2019. HawkEye: Efficient fine-grained OS support for huge pages. In *Proceedings of the ACM Conference on Architectural Support for Programming Languages and Operating Systems*.

[37] A. Panwar, A. Prasad, and K. Gopinath. 2018. Making huge pages actually useful. In *Proceedings of the ACM Conference on Architectural Support for Programming Languages and Operating Systems*.

[38] C. Priebe, K. Vaswani, and M. Costa. 2018. EnclaveDB: A secure database using SGX. In *Proceedings of the IEEE Symposium on Security and Privacy*.

[39] S. Sasy, S. Gorbunov, and C. W. Fletcher. 2018. ZeroTrace: Oblivious memory primitives from Intel SGX. In *Proceedings of the Networked and Distributed Systems Security Symposium*.

[40] F. Schuster, M. Costa, C. Fournet, C. Gkantsidis, M. Peinado, G. Mainar-Ruis, and M. Russinovich. 2015. VC3: Trustworthy data analytics in the cloud using SGX. In *Proceedings of the IEEE Symposium on Security and Privacy*.

[41] K. Shanker, A. Joseph, and V. Ganapathy. 2020. An evaulation of methods to port legacy code to SGX enclaves. In *Proceedings of the ACM SIGSOFT Joint European Software Engineering Conference and Symposium on Foundations of Software Engineering*.

[42] M.-W. Shih, S. Lee, T. Kim, and M. Peinado. 2017. T-SGX: Eradicating controlled-channel attacks against enclave programs. In *Proceedings of the Networked and Distributed Systems Security Symposium*.

[43] S. Shinde, D. Le Tien, S. Tople, and P. Saxena. 2017. Panoply: Low-TCB Linux applications with SGX enclaves. In *Proceedings of the Networked and Distributed Systems Security Symposium*.

[44] S. Shinde, S. Tople, D. Kathayat, and P. Saxena. 2015. Protecting Legacy Applications with a Purely Hardware TCB. Technical Report No. NUS-SL-TR-15-01. National University of Singapore.

[45] Yan Shoshitaishvili, Ruoyu Wang, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. 2015. Firmalice - Automatic detection of authentication bypass vulnerabilities in binary firmware. In *Proceedings of the ISOC Network and Distributed Systems Security Symposium*.

[46] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Audrey Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. 2016. SoK: (State of) the art of war: Offensive techniques in binary analysis. In *Proceedings of the IEEE Symposium on Security and Privacy*.

[47] R. Sinha, S. Seshia, and S. Rajamani. 2017. A compiler and verifier for page access oblivious computation. In *Proceedings of the ACM SIGSOFT Symposium on Foundations of Software Engineering*.

[48] Nick Stephens, John Grosen, Christopher Salls, Audrey Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. 2016. Driller: Augmenting fuzzing through selective symbolic execution. In *Proceedings of the ISOC Network and Distributed Systems Security Symposium*.

[49] R. Strackx and F. Piessens. 2017. The Heisenberg Defense: Proactively Defending SGX Enclaves against Page-Table-Based Side-Channel Attacks. arxiv:1712.08519v1. Retrieved from https://arxix.org/abs/1712.08519v1.

[50] C. Tsai, D. E. Porter, and M. Vij. 2017. Graphene-SGX: A practical library OS for unmodified applications on SGX. In *Proceedings of the USENIX Annual Technical Conference*.

[51] J. Van Bulck, N. Weichbrodt, R. Kapitza, F. Piessens, and R. Strackx. 2017. Telling your secrets without page faults: Stealthy page table-based attacks on enclaved execution. In *Proceedings of the USENIX Security Symposium*.

[52] W. Wang, G. Chen, X. Pan, Y. Zhang, X. Wang, V. Bindschaedler, H. Tang, and C. A. Gunter. 2017. Leaky cauldron on the dark land: Understanding memory side-channel hazards in SGX. In *Proceedings of the ACM Conference on Computer and Communications Security*.

[53] B. Xing, M. Shanahan, and R. Leslie-Hurd. 2016. Intel® software guard extensions (Intel® SGX) software support for dynamic memory allocation inside an enclave. In *Proceedings of the Workshop on Hardware and Architectural Support for Security and Privacy*.

[54] xsbench [n.d.]. XSBench: A Monte Carlo Macroscopic Cross Section Lookup Benchmark. Retrieved from https://github.com/ANL-CESAR/XSBench.

[55] Y. Xu, W. Cui, and M. Peinado. 2015. Controlled channel attacks: Deterministic side-channels for untrusted operating systems. In *Proceedings of the IEEE Symposium on Security and Privacy*.

[56] Pan Zhang, Chengyu Song, Heng Yin, Deqing Zou, Elaine Shi, and Hai Jin. 2020. Klotski: Efficient obfuscated execution against controlled-channel attacks. In *Proceedings of the 25th International Conference on Architectural Support for Programming Languages and Operating Systems*.

Received March 2020; revised September 2020; accepted November 2020