SGX-LAPD: Thwarting Controlled Side Channel Attacks via Enclave Verifiable Page Faults

Yangchun Fu[†] Erick Bauman[‡] Raul Quinonez[‡] Zhiqiang Lin[‡]

[†] Google Inc
[‡]The University of Texas at Dallas
firstname.lastname@utdallas.edu

Abstract. To make outsourcing computing more practical, Intel recently introduced SGX, a hardware extension that creates secure enclaves for the execution of client applications. With SGX, instruction execution and data access inside an enclave are invisible to the underlying OS, thereby achieving both confidentiality and integrity for outsourced computing. However, since SGX excludes the OS from its trusted computing base, now a malicious OS can attack SGX applications, particularly through controlled side channel attacks, which can extract application secrets through page fault patterns. This paper presents SGX-LAPD, a novel defense that uses compiler instrumentation and enclave verifiable page fault to thwart malicious OS from launching page fault attacks. We have implemented SGX-LAPD atop Linux kernel 4.2.0 and LLVM 3.6.2. Our experimental results show that it introduces reasonable overhead for SGX-nbench, a set of SGX benchmark programs that we developed.

Keywords: SGX · Trusted Execution · Controlled Channel Attack · Page Fault

1 Introduction

Trusted computing, or Trusted Execution Environment (TEE), is a foundational technology to ensure confidentiality and integrity of modern computing. Over the past few decades, a considerable amount of research has been carried out to search for practical ways for trusted computing, e.g., by using a formally verified operating system (OS) [16], or using a virtual machine monitor (VMM), hypervisor [25,9], system management mode (SMM) [30], and even BIOS [28] to monitor the kernel and application integrity, or with hardware support [17]. Increasingly, hardware based technologies for TEE (e.g., TPM [20], TrustZone [23]) have rapidly matured. The most recent advancement in this direction is the Intel Software Guard eXtensions (SGX) [18,13].

At a high level, SGX allows an application or part of an application to run inside a secure enclave, which is an isolated execution environment. SGX hardware, as a part of the CPU, prevents malicious software, including the OS, hypervisor, or even low-level firmware (e.g., SMM) from compromising its integrity and secrecy. SGX provides opportunities for securing many types of software such as system logs [15] and computer games [5]. The isolation enabled by SGX is particularly useful in cloud computing, where customers cannot control the infrastructure owned by cloud providers. Haven [6] pioneered the idea of enabling unmodified application binaries to run on SGX in a cloud

by utilizing a library OS [22]. VC3 [24] demonstrated privacy-aware data analytics in the cloud. Ohrimenko et al. [19] presented a number of privacy preserving multi-party machine learning algorithms running in SGX machines for cloud users, while Chandra et al. [7] provide a more scalable solution on larger models using randomization.

Unfortunately, since SGX excludes the OS kernel from its trusted computing base, SGX enclave programs can certainly be attacked by the underlying OS. A powerful demonstration of this is controlled channel attacks, which can extract application secrets using the page fault patterns of an enclave's execution [31]. In particular, by controlling the page table mappings of an enclave program, a malicious OS can observe a number of patterns regarding an application's page access footprint, such as the number of page faults, the base virtual address of the faulting pages, the sequence of page faults, and even the timing of page faults. If an attacker also has the binary code of the enclave program, he or she can recover a lot of secrets (e.g., text documents, outlines of JPEG images) based exclusively on the page access patterns.

Given such a significant threat from page-fault side channel attacks, it is imperative to design new defenses. Thus in this paper we present SGX-LAPD, a system built atop both OS kernel and compilers to ensure that the LArge Pages are verifieD by the enclave (LAPD) and attacker triggered page faults are detectable by the enclave itself. The key insight is that page-fault side channel attacks are very effective when the OS uses 4KB pages; if we can enlarge the page size, most programs will trigger few code page faults—and data page faults can also be significantly reduced (by three orders of magnitude if we use MB level pages). Thus, the challenge lies in how to make sure that the OS has cooperated and really provided large pages to the enclave programs.

Since the only trust for SGX programs is the underlying hardware and the enclave code itself, we have to rely on the enclave program itself to verify whether an OS indeed has provided large pages. As a page-fault attack often incurs significant delays during cross-page control flow transfers, an intuitive approach would be to detect the latency at each cross small-page control flow transfer point. However, there is no reliable way of retrieving the hardware timing information inside the enclave (e.g, `RDTSC` instruction is not supported in SGX v1 [14]), and meanwhile it can also be attacked by the OS. Note that `RDTSC` reads the Time-Stamp Counter from the TSC MSR which can be modified by `WRMSR` instruction [3]. Also, the API `sgx_get_trusted_time` provided by Intel SGX SDK is also only available in simulation mode.

Interestingly, we notice that each enclave contains a data structure, `EXINFO`, that tracks the page fault address if a page fault causes the enclave exit [14]. Therefore, we can detect whether an OS has indeed provided large pages by traversing this data structure when there is a page fault. However, when to incur a page fault is decided by the OS, and the enclave program has to deliberately trigger a page fault for such a verification. Therefore,, if we can instrument the enclave program to automatically inject a page access and then verify whether a page fault was triggered by checking the `EXINFO` data structure, we can then detect whether the underlying OS has cooperated. SGX-LAPD is designed exactly based on this idea.

We have implemented SGX-LAPD atop a recent Linux kernel 4.2.0 and LLVM 3.6.2. Specifically, we implemented an OS kernel module to enable the OS to support large page tables, and we implemented a compiler pass in LLVM to recognize the cross small-page control flow transfer points and insert the self-verification code. We

have evaluated our system using a number of benchmarks. In order to test SGX-LAPD on actual SGX hardware, we had to port a benchmark, since there are no existing SGX programs to test. We therefore manually created SGX-nbench, a modified version of nbench 2.2.3 running on real SGX hardware. Our experimental results show that SGX-LAPD introduces reasonable performance overhead for the tested benchmarks.

In short, we make the following contributions:

- We present SGX-LAPD, a system that uses large paging via kernel module and self-verifiable page faults through compiler instrumented code to defeat the controlled side channel attacks.
- We have also developed a new SGX benchmark suite SGX-nbench, for measuring the performance overhead of real SGX programs.
- We have evaluated SGX-LAPD with SGX-nbench and showed that it introduces reasonable overhead for detecting both non-present and non-executable page fault attacks.

2 Background and Related Work

In this section, we provide the background on the page fault side channel attacks using a running example in §2.1, and then discuss the possible defenses and related work in §2.2. Finally, we reveal how an enclave program handles exceptions in §2.3, which comprises the basic knowledge in order to understand our defense.

2.1 The Page-Fault Side Channel Attack

An SGX enclave program is executed in user mode (ring-3), and it has to ask the underlying OS to provide resources such as memory, CPU, and I/O. As such, this gives a hostile OS (ring-0) the opportunity to attack enclave programs from various vectors, such as manipulating system call execution (e.g., Iago [8] attacks) or controlling page fault access patterns to infer the secrets inside enclave programs [31].

The virtual memory pages of a process are managed by the underlying OS. Specifically, when launching a new process, the OS first creates the page tables and initializes the page table entries for virtual addresses specified in the application binary. When a process is executed, if the corresponding virtual page has not been mapped in the page table yet, a page fault exception will occur, and the CPU will report the faulting address as well as the type of page access (read or write) to the page fault handler, which will be responsible for mapping the missing pages. When a process terminates, the OS will delete the virtual to physical mappings and reclaims all the virtual pages.

Page faults for SGX processes are treated in the same way as regular processes, with the only difference that the page fault handler can observe just the base address of the faulting address. Therefore, by controlling the page table mappings, a hostile OS can observe all of the page access patterns of a victim SGX process. If the attacker also has the detailed virtual address mappings (e.g., when owning a copy of the SGX enclave binary), such a page fault attack is extremely powerful as demonstrated by Xu et al [31].

A Running Example. To understand clearly the nature of the page fault side channel attack, we use example code from [31] as a running example to explain how SGX-LAPD

```
const char* WelcomeMessageForMale()
{
     char* mesg = "Hello sir!";
     return mesg;
}
const char* WelcomeMessageForFemale()
{
      char* mesg = "Hello madam!";
      return mesg;
}
const char* WelcomeMessage(GENDER s)
{
      const char* mesg;
      if (s == MALE) { //MALE
          mesg = WelcomeMessageForMale();
      } else { //FEMALE
          mesg = WelcomeMessageForFemale();
      }
      return mesg;
}
```

(a) Source Code

```
402000 <WelcomeMessageForMale>:
402000:  55                push   %rbp
402001:  48 89 e5          mov    %rsp,%rbp
...
                 402012:  c3               retq
...
403000 <WelcomeMessageForFemale>:
403000:  55                push   %rbp
403001:  48 89 e5          mov    %rsp,%rbp
...
403012:  c3                retq
...
404000 <WelcomeMessage>:
404000:  55                push   %rbp
404001:  48 89 e5          mov    %rsp,%rbp
404004:  48 83 ec 10       sub    $0x10,%rsp
404008:  89 7d fc          mov    %edi,-0x4(%rbp)
40400b:  85 ff             test   %edi,%edi
40400d:  74 07             je     404016 <WelcomeMessage+0x16>
40400f:  e8 ec ef ff ff    callq  403000 <WelcomeMessageForFemale>
404014:  eb 05             jmp    40401b <WelcomeMessage+0x1b>
404016:  e8 e5 df ff ff    callq  402000 <WelcomeMessageForMale>
40401b:  48 89 45 f0       mov    %rax,-0x10(%rbp)
40401f:  48 8b 45 f0       mov    -0x10(%rbp),%rax
404023:  48 83 c4 10       add    $0x10,%rsp
404027:  5d                pop    %rbp
404028:  c3                retq
```

(b) Disassembled Code

Fig. 1. Our Running Example.

works to defeat this attack. The source code of this example is shown in Figure 1(a). At a high level, this enclave program takes user input `GENDER` and returns a welcome string based on whether the `GENDER` is `MALE` or `FEMALE`. To show this program is vulnerable to the page fault attack, we compile its source code using LLVM deliberately with the option "`align-all-function=12`" that aligns each function at a 4KB boundary. The resulting disassembled code for this example is presented in Figure 1(b), where five control flow transfer instructions inside `WelcomeMessage` are highlighted.

We can notice that a hostile OS can infer whether a user enters `MALE` or `FEMALE` to the program by observing the page fault profiles. Specifically, when all other pages except 0x404000 are marked unmapped: if a subsequent page fault accesses page 0x403000 (for control flow transfer "`callq WelcomeMessageForFemale`"), then an attacker can infer `GENDER` is `FEMALE`; otherwise an attacker can conclude `GENDER` is `MALE` when page 0x402000 is accessed.

2.2 Possible Defenses and Related Work

In the following, we examine various possible defenses. At a high level, we categorize them into hardware assisted and software based defenses.

Hardware-Assisted Defenses. As the hardware of a system is usually in the TCB, it can be helpful to utilize the hardware to enforce security.

- **Enclave Managed Paging**. A very intuitive approach is to allow the enclave itself to manage the paging (i.e., self-paging [12]). Once the enclave has been granted this capability, it can disable paging out sensitive pages, or enforce large pages, etc.
- **Hardware Enforced Contractual Execution**. Recently, Shinde et al. [27] proposed having the hardware enforce a contract between the application and the OS. Such a contract states that the OS will leave a certain number of pages in memory; if a page fault that violates the contract does occur, the hardware reports the violation to the secure application.

While the hardware-assisted approaches sound appealing, they have to modify the hardware to add new mechanisms, such as securely delivering the page fault address to the application page fault handler without relying on the OS. In addition, hardware modifications require significant time before widespread adoption is possible.

Software-Based Defenses. Software defenses have significant advantages over hardware modifications, one of which is that they can work on existing platform. We focus more on software defenses due to this. Note that software approaches can have the freedom of rewriting the binary code or recompiling the program source to add new capabilities on the enclave program. A number of defenses can be designed:

- **ORAM**. ORAM [11,21] is a technique for hiding the memory contents and access patterns of a trusted component from an untrusted component. Initially it was a software obfuscation technique, but recently there has been increasing interest in applying ORAM to build practical cloud storage. Theoretically, ORAM can be applied to protect the page fault patterns, but ORAM has large space requirements and high overhead.
- **Normalization**. Another approach is to make sensitive portions of the code behave identically for all possible inputs. However, this is difficult because not only must all page accesses be identical, but also each execution branch should take the same time to execute. Meanwhile, as demonstrated by Shinde et al. [27] in their use of deterministic multiplexing to execute the sensitive code, such an approach runs the risk of imposing extremely high overhead (up to 4000X), as the execution of any path must also perform all the page faults that every other path might make.
- **Randomization and Noise Injection**. Alternatively, if the code is hard to normalize, then we can introduce randomization and noise to make attacks harder. For instance, we can apply the same principle as ASLR [29] by performing fine-grained randomization (e.g., [10,4]) of code and data locations to hide from an attacker what code or data is actually being accessed, or inject noise into normal program behavior to hide legitimate page accesses among random fake ones. However, the challenge lies in how to make the randomization or noise indistinguishable from the normal page fault patterns.
- **Detection**. If an application is able to detect a controlled page fault attack, then it has the ability to abort execution before an attacker can extract the secret. How-

ever, the challenge lies in extracting the unique signatures for this attack. Recently, T-SGX [26] leverages code instrumentation and Transactional Synchronization Extensions (TSX) mechanism to detect whether there is any exception occurs inside a transaction. Similar to T-SGX, we also take a detection approach and we both use compiler instrumentation to insert the detection logic. However, the difference is that T-SGX relies on TSX whereas SGX-LAPD does not depend on this hardware feature and instead it uses large pages.

2.3 Exception Handling Inside SGX

Since a page fault is an exception and SGX-LAPD needs to use some internal enclave data structures for the defense, we would like to examine in greater detail how SGX handles exceptions. The following study is based on the trace from a real SGX platform by executing our instrumented running example and confirmed with the description from the SGX programming reference [14].

By design, an exception will trigger an asynchronous enclave exit (AEX), and the CPU execution has to leave the enclave and come back through an `ENTER` or `ERESUME` instruction. In general, there are 10 exceptions [14] an SGX enclave can capture, and the type of the exception is stored in the `EXITINFO.VECTOR` field, which is at offset `0xA0` in the `GPRSGX` region, as illustrated in Figure 2. Note that the 4 bytes of `EXITINFO` contain the information that reports the exit reasons (i.e., which exception) to the software inside the enclave, and the first byte is the `VECTOR` field. The `GPRSGX` region holds the processor general purpose registers as well as the AEX information. Among the 10 exceptions, we are interested in GP, general protection fault, which is caused by illegal access, e.g., accessing thread control structure (TCS) inside an enclave and PF, the page fault exception. Exceptions such as DV (divide by zero), BP (int 3 for debugging), and UD (undefined instruction, e.g., executing `CPUID` inside enclave) etc., are out of our interest, though they are all handled similarly as GP by the CPU.

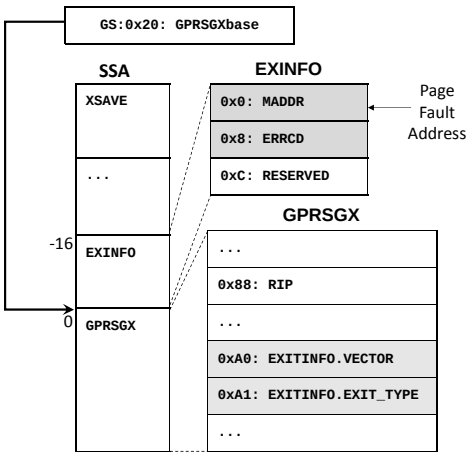

Fig. 2. The Layout of Involved Enclave Data Structures.

Page Fault Exceptions. An exception is handled by system software first, and then by the application defined code. A page fault exception can be entirely handled by the system software (only requires 3 steps of execution), but other exceptions such as GP, DV, or UD require 8 steps, as illustrated in Figure 3.

Specifically, when an exception occurs, SGX hardware will automatically store the fault instruction address in the `GPRSGX.RIP` field and the exception vector in the `GPRS.EXITINFO.VECTOR` field (Step ❶), and meanwhile inform the CPU of the in-

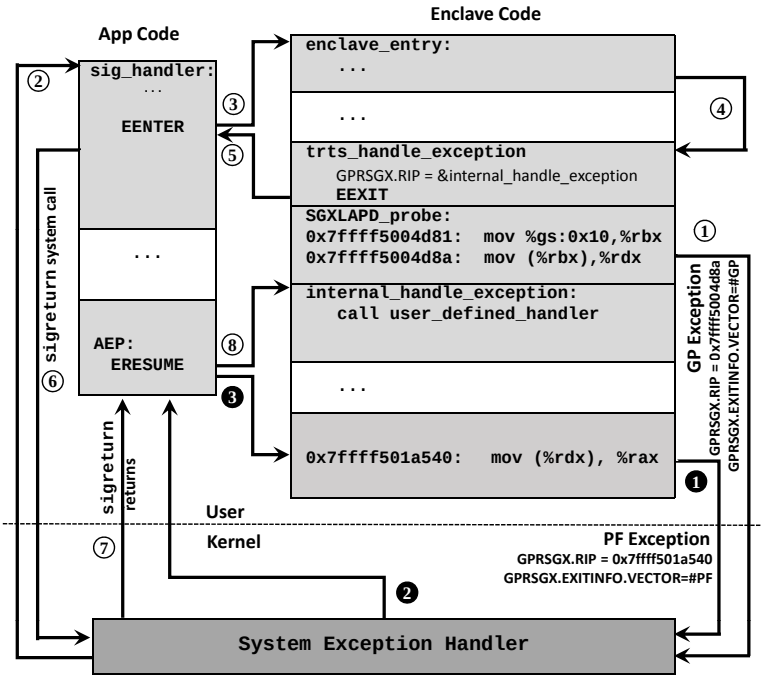

Fig. 3. Detailed CPU Control Flow Transfers in SGX Enclave Exception Handling.

struction to be executed next, which is defined as the Asynchronous Exit Pointer (AEP, which is normally just an ERESUME instruction). This is because an exception needs to be handled by system software, and the enclave internal address should not be exposed to the system software; the CPU will just execute AEP after handling the exception, so the address is not exposed. Also note that there are only two instructions that can enter an enclave: EENTER and ERESUME. EENTER always starts the execution execution at the enclave_entry address whereas ERESUME will use the internally maintained GPRSGX.RIP as the starting address.

For a page fault exception, SGX hardware will also store the fault address in MADDR as well as the corresponding error code (ERRCD) in the EXINFO structure, whose layout is presented in Figure 2. Note that EXINFO and EXITINFO are two different data structures, and EXINFO is only used for PF and GP exceptions, though both of them are stored in the State Save Area (SSA) page.

After the system software maps the missing page (Step ❷) for the page fault exception, the CPU will continue the execution in user space to execute the ERESUME instruction, which restores registers and returns control to where the exception occurred. Again, the ERESUME instruction is stored at address called AEP, which is defined by the EENTER instruction. After executing ERESUME (Step ❸), the CPU will continue the execution at the fault address that is captured by GPRSGX.RIP. For other exceptions such as GP, the CPU has to execute 8 steps to eventually resolve that exception.

Non-Page Fault Exceptions. Some exceptions cannot be completely resolved by the system software. In this case, the event will be triggered again if the enclave is re-entered using ERESUME to the same fault instruction (e.g., a divide by 0 instruction). Therefore, SGX supports resuming execution at a different location (to skip the fault instruction for instance). However, the fault instruction address is internally stored in GPRSGX.RIP field by the hardware inside enclave, and we must rely on the enclave code to update GPRSGX.RIP to a different instruction location, and then ERESUME to this new location. To tell the enclave and update GPRSGX.RIP, we have to use the EENTER instruction and then EEXIT.

Take a GP exception as an example, as illustrated in Figure 3: when enclave code accesses data in TCS (thread control structure, which is not supposed to be accessed by the enclave code), it triggers a GP exception (Step ①). The hardware stores the fault instruction address at GPRSGX.RIP and the exception number, namely #GP, in EXITINFO.VECTOR. Meanwhile, the hardware also passes the AEP address to the system software, which is the next instruction to be executed after handling the exception. The system exception handler processes this exception as SIGSEGV, which cannot be completely resolved without collaboration with the enclave code. Therefore, the control flow goes to the user space sig_handler (Step ②), which works together with the trts_handle_exception function inside the enclave to resolve the exception. More specifically, after learning more details about this exception, sig_handler executes EENTER at Step ③ and then the execution goes to the enclave_entry point.

Note that enclave_entry is defined in the enclave binary and initialized by EINIT, and EENTER will start to execute enclave code at enclave_entry, which normally contains a dispatch table. In our exception handling case, it will call trusted exception handling function trts_handle_exception (Step ④) to reset GPRSGX.RIP to the address of the internal_handle_exception function, and then it executes EEXIT at Step ⑤ to continue the execution of signal_handler, which further executes system call sigreturn (Step ⑥) to trap to the kernel. Then at Step ⑦, the sigreturn system call will return to AEP, which will execute ERESUME instruction (Step ⑧). Having set up the GPRSGX.RIP value with internal_handle_exception, enclave code will execute this function, call the corresponding user defined handler if there is one, and continue the execution.

To Capture Page Fault Exceptions. SGX hardware will not automatically report a page fault exception to EXINFO and EXITINFO unless the EXINFO-bit (namely SECS.MISCSELECT[0]) is set, and this bit can be controlled in SGX-v2, not in the current market available SGX v1. We have verified this observation in a real SGX-v2 machine with the help from the Intel SGX team. Note that SECS is the enclave control structure, which contains meta-data used by the hardware to protect the enclave and is created by the ECREATE instruction. Enclave developers can set the SECS.MISCSELECT field before invoking ECREATE to create the enclave. Once the EXINFO-bit is set, both GP and PF will be reported in the EXINFO structure. Therefore, an enclave can inject a GP exception to probe whether EXINFO-bit has been set, as we have demonstrated in the SGXLAPD_probe code in Figure 3.

3 System Overview

3.1 Scope and Assumptions

The focus of this paper is on defending the controlled channel attacks, which can be more specifically termed as page fault attacks. There are two types of page fault attacks: code page fault and data page fault. As a first step, we focus on the code page fault attacks and leave the protection of data page fault attacks to future work. Also, we focus on the Linux platform.

We assume the SGX hardware and the enclave program itself are trusted. While we wish for the OS to provide large pages, the OS may not cooperate and may cheat the enclave programs. Therefore, we will verify whether an OS indeed provides large pages from the application itself. Regarding the SGX hardware, the market available one is Skylake, and we focus on the x86-64 architecture. Typically, under this architecture, the CPU supports 4K and 2M page sizes [2]. We use 2M large pages. Also, we assume an attacker has a binary code copy of our enclave code, the same threat model as in [31].

3.2 Challenges and Approaches

Key Idea. The goal of SGX-LAPD is to minimize page fault occurrence by using large pages (i.e., 2MB). However, an OS may not provide large pages to the enclave program, and therefore the key idea of SGX-LAPD is to verify from the enclave itself whether an OS provides it 2MB or 4KB size pages. To perform the verification, fortunately we have another observation: if the OS is hostile and only provides 4KB size pages, but if there is no controlled page fault attack, the execution will still be normal; but if there is such an attack, then a cross 4KB page control flow transfer will trigger a page fault. If we have set the enclave to report page fault exceptions to `EXINFO`, we can detect this attack by checking the `MADDR` field in this data structure. Also, another reason to use 2MB pages is to minimize the page fault occurrences for enclave code, since most programs have less than 2MB code. If we do not use large pages, we cannot differentiate whether the page fault is malicious or benign when a real page fault occurs.

Challenges. However, there are still two major challenges we have to solve:

- **How to insert the verification code**. We certainly cannot manually insert the verification code into the enclave binary, as that would be error-prone and not scalable. Instead, we must resort to either binary code rewriting or compilers to automatically insert our code. Meanwhile, not all control flow transfers need the verification; we only need to check those that cross 4KB page boundaries and we must identify them to insert our code.
- **How to perform the verification**. At each cross 4KB page control flow transfer, we need to know how to traverse the `EXITINFO` and `EXINFO` structures inside the enclave in order to retrieve data such as the fault address. Meanwhile, we also have to decide whether the fault is legal or not since there could exist enclaves that have more than 2MB code.

Approaches. To address the first challenge, we decide to modify a mainstream compiler, LLVM, to automatically insert the large page verification code, which will be executed at run-time inside an enclave to make sure the OS really cooperates. The reason why we selected a compiler approach is because SGX essentially comes with a set

of new instructions, and it requires an ecosystem change for applications to really take advantage of its security features (unless one is directly running a legacy application inside the enclave using a library OS).

We use the insight we learned in §2.3 to address the second challenge. Specifically, we notice that inside the enclave, `%gs:0x20` always points to the `GPRSGX` region (as illustrated in Figure 2), from which we can easily reach `EXINFO`, which is at `(%gs:0x20)-16`, and `EXITINFO`, which is at `(%gs:0x20)+ 0xA0`. To allow legal control flow transfers across 2M page boundaries, our instrumented code will also collect the source address of the control flow transfer in addition to the target fault address. If this transfer crosses to another 2MB page, it will be considered legal. Next, we present our detailed verification algorithm using our running example.

3.3 The Verification Algorithm

The page fault exception attack can be triggered in two ways. The first and most straightforward way is to manipulate the page mapping (i.e., the `P`-bit in the page table) and make the target page unmapped. Then any code execution access will trigger a non-present (NP) page fault. This approach has been used by Xu et. al. [31]. However, we also determined that there is a second way to perform the attack by making the page non-executable when CPU paging mode is PAE or IA-32e to trigger a non-executable (NX) page fault. Therefore, we provide two strategies to detect these faults.

```
0000000000404000 <WelcomeMessage>:
 404000:   55                          push   %rbp
 404001:   48 89 e5                    mov    %rsp,%rbp
 404004:   48 83 ec 10                 sub    $0x10,%rsp
 404008:   89 7d fc                    mov    %edi,-0x4(%rbp)
 40400b:   85 ff                       test   %edi,%edi
 40400d:   eb 2f                       jmp    40403e <WelcomeMessage+0x3e>
           ...
 40403e:   74 69                       je     4040a9 <WelcomeMessage+0xa9>
 404040:   9c                          pushfq
 404041:   50                          push   %rax
 404042:   56                          push   %rsi
 404043:   52                          push   %rdx
 404044:   48 8d 35 b5 ef ff ff        lea    -0x104b(%rip),%rsi
 40404b:   65 48 8b 04 25 20 00        mov    %gs:0x20,%rax
 404052:   00 00
 404054:   c6 80 a0 00 00 00 00        movb   $0x0,0xa0(%rax)
 40405b:   8a 16                       mov    (%rsi),%dl
 40405d:   8a 90 a0 00 00 00           mov    0xa0(%rax),%dl
 404063:   80 fa 0e                    cmp    $0xe,%dl
 404066:   75 05                       jne    40406d <WelcomeMessage+0x6d>
 404068:   e8 a3 c4 ff ff              callq  400510 <abort@plt>
 40406d:   5a                          pop    %rdx
 40406e:   5e                          pop    %rsi
 40406f:   58                          pop    %rax
 404070:   9d                          popfq
 404071:   e8 8a ef ff ff              callq  403000 <WelcomeMessageForFemale>
 404076:   eb 2f                       jmp    4040a7 <WelcomeMessage+0xa7>
           ...
 4040a7:   eb 36                       jmp    4040df <WelcomeMessage+0xdf>
 4040a9:   9c                          pushfq
           ...
 4040da:   e8 21 df ff ff              callq  402000 <WelcomeMessageForMale>
 4040df:   48 89 45 f0                 mov    %rax,-0x10(%rbp)
 4040e3:   48 8b 45 f0                 mov    -0x10(%rbp),%rax
 4040e7:   48 83 c4 10                 add    $0x10,%rsp
 4040eb:   5d                          pop    %rbp
 4040ec:   9c                          pushfq
           ...
 40411b:   c3                          retq
```

Fig. 4. Final Disassembled Code For Function `WelcomeMessage` After SGX-LAPD Instrumentation for Non-Present Page Fault Detection.

We note that in terms of detection capability, the NX page fault approach can detect all attacks including both non-present and non-executable faults. However, the NP page fault approach cannot detect non-executable page faults. Therefore, in practice we recommend the use of the NX approach. Only when the CPU is set in non PAE nor IA-32e mode will the NP approach be useful. We provide both approaches just for the completeness of the defense.

(I). Detecting NP Page Faults. Since we have instrumented our verification code in the enclave binary at each cross 4KB-page control flow transfer point, we just need to invoke a target page read (basically inject an explicit page fault) and check whether

indeed there is a page fault. If so, a page fault attack is confirmed by checking field `EXITINFO.VECTOR`. To show how SGX-LAPD really performs this, we illustrate the final disassembly of function `WelcomeMessage` in our running example in Figure 4.

We can notice that for the four direct control flow transfers in `WelcomeMessage` (Figure 1), we each instrumented 49 bytes of code right before them. The last control flow transfer instruction `retq` has 47 bytes of instrumented code. More specifically, for the first ("`je 404049`") and third ("`jmp 4040df`") control flow transfers, our instrumented code directly performs a within-page jump (i.e., "`jmp 40403e`" and "`jmp 4040a7`") because there is no need for the verification, whereas for the second ("`callq 403000`"), and forth ("`callq 402000`") direct function call, and fifth ("`retq`") function return, our instrumented code first injects a target page read, and then traverses `EXITINFO` in `SSA` to detect whether there is a real page fault.

The full disassembly of our page fault verification code for the second control flow transfer "`callq 403000`" is presented in Figure 4 from `0x404040` to `0x404070`. In particular, our instrumented code will first save the flag register via `pushfq`, `rax`, `rsi`, and `rdx` in the stack, and then load the target address into `rsi`, i.e., "`lea -0x104b(%rip), %rsi`". After that, it loads the base address of `GPRSGX` into `%rax`, and assigns a zero to the field `EXITINFO.VECTOR` (to clear any prior exceptions recorded in the vector). Then *it performs a one-byte memory read access at the target address*, i.e., "`mov (%rsi), %dl`", to inject a page fault to test if there is any controlled side channel on the target page. After that, the enclave code checks the `EXITINFO.VECTOR` field. If it is set to be `0xe`, a page fault is detected (because there is a page fault for the three just executed instructions, and it must come from attack since enclave memory is not supposed to be swapped out) and we `abort` the execution; otherwise, we pop those saved registers and continue the execution.

Detecting NX Page Faults. Instead of using "`mov (%rsi), %dl`" to inject a read page fault, we need to really execute the target page in order to detect the NX page fault if there is any. The verification can be performed at either the destination page or the source page (if we inject a `callq *%rsi` and `retq` pair in the source and target).

If we perform the verification at the destination page, we need to track the source address (because we need to allow cross 2MB transfers) because a target page can be invoked by many different sources. On the other hand, since at each control flow transfer point we already know the source address, we decide to take the second approach, namely inject a `callq *%rsi` in the source page, and a `retq` in the target page to quickly return. To this end, we need to inject a `retq` in the beginning of each determined basic block, and probe the page fault by quickly returning from the target. We omit the details for brevity here since most of the code is similar to those in Figure 4.

4 Detailed Design

An overview of SGX-LAPD is illustrated in Figure 5. There are three key components: an SGX-LAPD-compiler and SGX-LAPD-linker that work together to produce the enclave code that contains large page verification code at any cross-small page control flow transfer points, and an SGX-LAPD kernel module that runs in kernel space to provide the 2MB pages for enclave code. In this section, we provide the detailed design for these three components.

```
Disassembly of section .text:
0000000000000000 <WelcomeMessageForMale>:
       0:   55                          push   %rbp
    ...
0000000000001000 <WelcomeMessageForFemale>:
    1000:   55                          push   %rbp
    ...
0000000000002000 <WelcomeMessage>:
    2000:   55                          push   %rbp
    2001:   48 89 e5                    mov    %rsp,%rbp
    2004:   48 83 ec 10                 sub    $0x10,%rsp
    2008:   89 7d fc                    mov    %edi,-0x4(%rbp)
    200b:   85 ff                       test   %edi,%edi
.LINST2_0_12:
    200d:   9c                          pushfq
    ...
.LINST2_0_11:
    203e:   74 69                       je     20a9 <WelcomeMessage+0xa9>
.LINST2_2_6:
    2040:   9c                          pushfq
    ...
    2044:   48 8d 35 b5 ef ff ff        lea    WelcomeMessageForFemale(%rip),%rsi
    ...
.LINST2_2_1:
    2071:   e8 8a ef ff ff              callq  1000 <WelcomeMessageForFemale>
.LINST2_2_25:
    2076:   9c                          pushfq
    ...
.LINST2_2_5:
    20a7:   eb 36                       jmp    20df <WelcomeMessage+0xdf>
.LINST2_1_8:
    20a9:   9c                          pushfq
    ...
.LINST2_1_1:
    20da:   e8 21 df ff ff              callq  0 <WelcomeMessageForMale>
.LBB2_3:
    20df:   48 89 45 f0                 mov    %rax,-0x10(%rbp)
    ...
    20ec:   9c                          pushfq
    ...
    211b:   c3                          retq
```

```
//In object file
Contents of section .SgxLapdCodeLabel :
OFFSET                          VALUE
0000000000000000                .LINST2_0_12:
0000000000000008                .LINST2_0_11
0000000000000010                .LINST2_1_8
0000000000000018                .LINST2_2_6
0000000000000020                .LINST2_2_1
0000000000000028                WelcomeMessageForFemale
0000000000000030                .LINST2_2_25
0000000000000038                .LINST2_2_5
0000000000000040                .LBB2_3
0000000000000048                .LINST2_1_8
0000000000000050                .LINST2_1_1
0000000000000058                WelcomeMessageForMale

RELOCATION RECORDS FOR [.SgxLapdCodeLabel]:
OFFSET           TYPE           VALUE
0000000000000000 R_X86_64_64    .text+0x000000000000200d
0000000000000008 R_X86_64_64    .text+0x000000000000203e
0000000000000010 R_X86_64_64    .text+0x00000000000020a9
0000000000000018 R_X86_64_64    .text+0x0000000000002040
0000000000000020 R_X86_64_64    .text+0x0000000000002071
0000000000000028 R_X86_64_64    .text+0x0000000000001000
0000000000000030 R_X86_64_64    .text+0x0000000000002076
0000000000000038 R_X86_64_64    .text+0x00000000000020a7
0000000000000040 R_X86_64_64    .text+0x00000000000020df
0000000000000048 R_X86_64_64    .text+0x00000000000020a9
0000000000000050 R_X86_64_64    .text+0x00000000000020da
0000000000000058 R_X86_64_64    .text+0x0000000000000000

//In final executable
Contents of section .SgxLapdCodeLabel :
OFFSET                          VALUE
0000000000000000                0x40400d
0000000000000008                0x40403e
0000000000000010                0x4040a9
0000000000000018                0x404040
0000000000000020                0x404071
0000000000000028                0x403000
...
```

Fig. 6. Examples of SGX-LAPD Instrumented Code Label and the Corresponding Meta-Data.

4.1 SGX-LAPD-Compiler

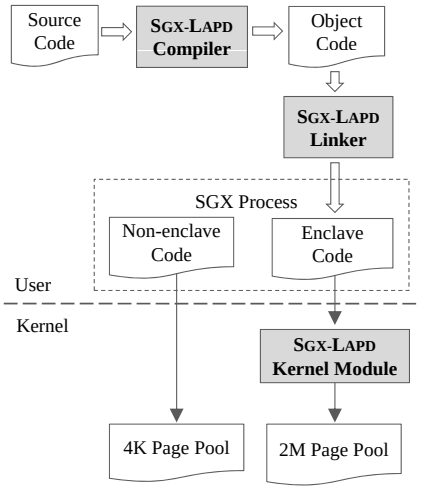

Fig. 5. SGX-LAPD Overview.

The goal of SGX-LAPD-compiler is to automatically insert the 4KB page fault detection code into each cross page control flow transfer (CFT) at various instructions such as `call/jmp/jcc/ret`. In particular, our compiler needs to track the source and target addresses for the CFTs, and also needs to keep the starting address of the inserted code such that we can later patch our instrumented code to NOP instructions (or other semantically equivalent ones) if the CFT is within a page. Note that only after the code is generated can this patching be performed (by our SGX-LAPD-linker) because we do not know the final concrete address before that.

The Meta-Data Used by Our Compiler. We define a data structure that tracks (1) the starting address of the inserted code, (2) the source, and (3) the target address, for each encountered CFT (except `retq` since we do not know its target address statically). We store this information in a special data section we created and we call it `.SgxLapdCodeLabel`. An example of these code labels is presented in Figure 6. In particular, for the first CFT "`je 20c4`", we store the starting address of the inserted

code at offset 0, which is the symbolic code label `.LINST2_0_12`. Then at offset 8 (recall that we are working on a 64-bit architecture), we store the source address of this CFT, which is `.LINST2_0_11`. Finally, at offset 0x10, we store the target address, which is `.LINST2_1_8`.

Meanwhile, during the compilation phase, we only know the symbol addresses for the code labels and the final concrete address is resolved during the linking phase. We have to thus create relocation entries to store these code label addresses and let the linker eventually resolve them. To this end, we also create relocation entries for each `.SgxLapdCodeLabel` item. After compilation, the value for these relocation entries will be the logic address within that particular object file. For instance, for the first entry `.LINST2_0_12`, whose value is `.text+0x0200d`, its final concrete address will be resolved by the linker (once the base address of `.text` is resolved). Also, mainstream compilers typically maintain the labels for each basic block starting address and CFT target address. We just need to parse the meta-data provided by compilers and use them for our purposes.

The Instrumentation Algorithm. At a high level, to perform the instrumentation, our compiler will iterate through each compiled function right after the code generation phase. For each basic block within a function, we will look for the CFT instructions (i.e., `call/jmp/ret` and conditional jumps `jcc`). For each CFT instruction, we get its source address and destination address and store them in the corresponding `.SgxLapdCodeLabel` section. Note for `retq`, since we do not know its target address statically, no target address meta data is needed for this instruction.

Since there are different types of CFTs, we have to instrument slightly different verification code. Note that the size difference is due to the different instructions used to fetch the target address for different CFT. Specifically, to detect *NP page faults*, we insert 49 bytes of assembly code if it is a direct CFT, and this assembly code is formed from a macro template with symbols as macro parameters. For indirect CFT, we insert 50 bytes of assembly if it is an indirect CFT through memory (e.g., "`call (%rax)`"), otherwise 45 bytes if it uses register (e.g., "`call %rax`"), right before the CFT instructions. For return CFT, we insert 47 bytes of assembly. For *NX page fault* detection, we insert 56 bytes, 57 bytes, 52 bytes, and 53 bytes respectively each for direct CFT, indirect CFT through memory, indirect CFT through register, and return CFT. We also store the starting address of the inserted code into `.SgxLapdCodeLabel` for direct CFTs. Note that the inserted assembly code will use the destination address symbol for the direct CFTs, and these symbol addresses will be automatically resolved during the linking phase. For all indirect CFTs (e.g., "`callq %rax`" and `retq`), we will directly use the correspondingly *run-time* value to access the target page in the inserted assembly code. In other words, we do not need to generate any meta-data for indirect CFTs as their target addresses are computed at runtime, and they also do not need patching.

4.2 SGX-LAPD-Linker

Symbol Address Resolution. The compiler generates the object file for each input source file using a logic address starting at offset 0. The function or global variable references are all through symbols. Their concrete addresses are not known until linking time, when the linker combines the same section (e.g., `.text`, `.data`) from each

object file. To assist the linker in calculating the address for each symbol, there is a relocation entry specifying the relative address to its section. SGX-LAPD leverages this mechanism, and generates symbols for each label that we want to know the final address of into the `.SgxLapdCodeLabel` section and the corresponding relocation record into the `.RELOCATION` section. Later, in the linking phase, the linker can resolve the concrete address for each label. For example, `.LINST2_0_12` is resolved as `0x40400d`, as shown in Figure 6.

Code Optimization. Our SGX-LAPD-compiler has instrumented each CFT due to the fact that we do not know whether any of these transfers will cross a 4KB page boundary in the final executable. Once the code is finally linked, we can scan the final executable to patch the overly instrumented code.

Thanks to our tracked meta-data, it becomes extremely simple to patch this code. Specifically, we know where to start the patching because our `.SgxNypdCodeLabel` section tracks the starting address of the instrumented code. We also know whether an instrumented CFT crosses a 4KB page boundary or not because we also know the source and destination addresses of this transfer from `.SgxNypdCodeLabel`. Note that `retq` is not included in this optimization since its destination address is unknown statically.

While we could patch all the inserted bytes to `NOP` instructions, we can just insert an unconditional jump to directly skip the unnecessary code instead. We also know how many bytes our instrumented code occupies (e.g., 49 bytes for direct CFT for non-present page fault detection). As such, we can directly rewrite the first two bytes in the beginning of the instrumented code to an unconditional jump instruction (e.g., "`eb 2f`" to skip the remaining 47 bytes of the 49 bytes of inserted code), as shown in the example code for the first and third CFT instructions in Figure 4.

4.3 SGX-LAPD-Kernel Module

The last component of SGX-LAPD is the kernel module that is responsible for providing 2MB pages for enclave code. While we can rewrite the OS kernel to provide 2MB pages for all processes, such a design would waste page resources for many other non-SGX processes. Therefore, we design a kernel module to exclusively manage the page tables for enclave code.

Meanwhile, to really use SGX, Intel provides a number of hardware level data structures such as the Enclave Page Cache Map (EPCM) to manage the enclave page cache (EPC), a secure storage used by the CPU to store the enclave pages [14]. An enclave must run from the EPC, which is protected from non-enclave memory accesses. The EPC is initialized by the BIOS during boot time, and later each enclave process can use privileged instructions such as `ENCLS[EADD]` to add a page. In other words, we can directly instrument the corresponding SGX kernel code to manage the enclave process page tables.

In particular, the SGX kernel module is responsible for the management of enclave memory allocation and virtual-to-physical mapping. Each enclave page is allocated from a pool of EPC pages, and each EPC page has a size of 4KB. The process of adding an EPC page into an enclave is by first mapping a 4KB virtual page to a 4KB EPC page, then copying the corresponding contents to that EPC page via the `EADD` instruction.

While our SGX-LAPD-kernel module cannot directly add a 2MB EPC, it groups 512 small pages into a 2MB page. Note that those 512 smaller pages need to be contiguous in the physical address space, and the physical address of the first page is 2MB aligned. The SGX kernel module manages all the EPC pages and knows the physical address for each EPC page. We can control which physical pages are mapped to EPC pages.

5 Implementation

We have implemented SGX-LAPD for X86-64 Linux. We did not implement anything from scratch; instead we implemented SGX-LAPD-compiler atop LLVM 3.6.2, SGX-LAPD-linker atop `ld-2.24`, and SGX-LAPD-kernel module atop the Intel SGX kernel driver. Below we share some implementation details of interest.

Specifically, we modified the LLVM compilation framework to add a new `MachineFunction` pass into the LLVM backend. This new pass operates on LLVM's machine-dependent representation code. Note that our pass is running in the end of the compilation phase, so the code is ready to be emitted into assembly code. This also ensures that our inserted code is not optimized out by other passes. Inside this pass, we iterate each instruction within each basic block in order to identify all CFT instructions. For each CFT, the page fault detection code is inserted into the same basic block before the CFT instruction. We also add a new data section named `.SgxLapdCodeLabel` inside `MCObject FileInfo` class during the initialization phase. The `.SgxLapd CodeLabel` section is like the debug info section and can be removed by using the "`strip -R .SgxLapdCodeLabel`" command. Later in `AsmPrinter`, where the object file is created, we emit the meta data into the `.SgxLapdCodeLabel` section. In total, we added $1,500$ LOC to the LLVM framework.

To perform the linking of our compiled code, we modified the linker script to make sure the binary will be loaded into a 2MB-aligned starting address. Our linker also needs to use the meta-data inside the final ELF to optimize our instrumented code. We implemented our own optimization pass and integrated with linker `ld`. Basically, we parse the ELF header to locate the `.SgxLapd CodeLabel` section. Then the meta-data is used to decide whether each control flow transfer crosses a 4KB page boundary. Control flow transfers that happen inside the same page or cross a 2MB page boundary are considered valid (no verification check) and thus we insert unconditional jump to skip the verification code for those CFTs. In total, we added 150 LOC into `ld`.

Finally, we modified the Linux SGX kernel driver (initially provided by Intel) to support 2MB paging, which is only applied to the code pages of an enclave binary. Note that the data pages are still 4KB. We first instrumented `enclave_create` in the SGX kernel driver to record the base loading address and size of an enclave binary. We also make sure the EPC pages allocated to the enclave binary are contiguous and starting at a 2MB aligned physical address. Until an `EINIT` is executed, the enclave is not permitted to execute any enclave code, so before the execution of `EINIT`, all the enclave pages have been assigned and initialized. We can group each block of 512 small pages into a 2MB page by modifying the page table for the enclave process. In total, we added 200 LOC into the SGX kernel driver.

6 Evaluation

In this section, we present our evaluation result. We first describe how we create the benchmark programs and set up the experiment in §6.1, and then describe detailed result in §6.2.

6.1 The Benchmark and Experiment Setup

We have tested SGX-LAPD using two set of benchmarks: one is a manually ported nbench 2.2.3, which we call SGX-nbench, that runs atop a real SGX platform, and the other is the SPEC2006 benchmark that was not ported to SGX. It is important to note that no SGX applications currently exist that we can directly test, but we want to test the results of real SGX performance imposed by our solution. We therefore manually ported nbench into our SGX-nbench, which can be used to measure the true performance for any real SGX solutions. Meanwhile, since porting program to SGX platform requires non-trivial effort, SPEC2006 is in not running atop SGX enclave. We used SPEC2006 to exclusively measure how heavy of code instrumentation is for real programs.

SGX-nbench. We ported nbench 2.2.3, which contains 10 tests, to SGX-nbench. Specifically, we ported each benchmark to run inside an enclave in order to measure actual enclave performance. The difficulty of this task is that porting an application to run in SGX is nontrivial; libraries will not be available unless they are statically linked, and all system calls must be made outside the enclave. In addition, enclaves cannot execute certain instructions. Therefore, much of the code must be restructured in order to run inside an enclave. Porting a benchmark of the size and complexity of SPEC is a formidable task, so we focused on porting the more reasonably-sized nbench to measure real enclave performance.

In order to minimize the modifications to nbench, we moved only the minimal code required to run the timed portion of each benchmark into an enclave, and we left the rest of the benchmark code on the host application side. Specifically, we created an enclave application that we linked with modified nbench code; all the timing code stays outside the enclave, and the modified nbench code performs enclave calls to run the initialization code and timed code. The enclave contains the benchmark initialization functions (each benchmark needs to allocate one or more buffers and initialize them with starting data before the benchmark) and iteration functions (each benchmark performs n iterations until n is large enough that the elapsed time is greater than *min* seconds).

Our port added 5,279 LOC, modified 150 LOC, and removed 447 LOC from nbench 2.2.3. About half of the added LOC comprised enclave code or host application enclave initialization code, while the other added LOC were added to call the enclave functions for each of the benchmarks.

SPEC2006. We directly compiled SPEC2006 by using `clang` compiled from our modified LLVM framework. There are 31 benchmarks provided by SPEC2006, but only 21 are written in C/C++. We selected those 21 benchmarks to evaluate SGX-LAPD. In total, there are 12 integer benchmarks and 7 floating-point benchmarks. `998.specrand` and `999.specrand` are the common random number generator for integer suite and floating-point suite respectively.

Table 1. The building results for SPEC2006 and SGX-nbench

Benchmark	w/o Instrumentation			w/ NPD			w/ NXD		
	Size(KB)	#Direct CFT	#InDirect CFT	#Patch	Size(KB)	Increase(%)	#Patch	Size(KB)	Increase(%)
400.perlbench	1086	50152	1881	33375	5266	384.9	34651	5818	435.7
401.bzip	90	2029	120	1454	262	191.1	1572	286	217.8
403.gcc	3218	143634	5190	95564	15170	371.4	101562	16738	420.1
429.mcf	19	338	32	265	47	147.4	296	51	168.4
433.milc	132	3665	234	1497	440	233.3	2341	484	266.7
444.namd	327	6527	113	4653	863	163.9	5675	935	185.9
445.gobmk	3382	26701	2369	15185	5642	66.8	15838	5962	76.3
447.dealII	3240	101938	5722	70494	11596	257.9	80068	12856	296.8
450.soplex	375	13867	1467	7719	1551	313.6	9742	1723	359.5
453.povray	1027	32399	1747	16508	3739	264.1	21304	4107	299.9
456.hmmer	303	11108	478	6117	1227	305	8048	1355	347.2
458.sjeng	136	4541	189	2686	516	279.4	3118	564	314.7
462.libquantum	47	1113	104	592	139	195.7	727	155	229.8
464.h264ref	653	12533	875	8466	1721	163.6	9492	1869	186.2
470.lbm	19	140	20	71	31	63.2	110	31	63.2
471.omnetpp	655	25196	2503	6234	2819	330.4	11028	3175	384.7
473.astar	43	1062	90	647	135	214	888	147	241.9
482.sphinx3	186	6186	299	3315	702	277.4	3926	774	316.1
483.xalancbmk	4250	140253	9892	92143	16522	288.8	95686	18538	336.2
988.specrand	7	19	10	19	11	57.1	19	13	85.7
999.specrand	7	19	10	19	11	57.1	19	13	85.7
SGX-nbench	273	848	91	615	408	49.5	732	412	50.9
Average	885	26558	1520	16711	3128	212.5	18493	3455	244.1

Experiment Setup. All the benchmarks are compiled with Clang. Our tested platform is Ubuntu 14.04 with Linux Kernel 4.2.0, and our hardware is a 4-core Intel Core i5-6200U Skylake CPU running SGX-v1 at 2.3GhZ with 4G DDR3 RAM.

6.2 Results

We complied the benchmarks with three settings: without Instrumentation, with Non-Present page fault Detection (NPD) and with Non-eXecutable page fault Detection (NXD). The evaluation tries to measure the overhead added to the compiler and programs caused by the instrumentation.

SGX-LAPD **Compiler.** Table 1 presents the building details for the SPEC2006 and SGX-nbench. To show how much code we needed to insert for each program, we reported the number of CFTs for each benchmark. We report the number of direct CFTs in the 3rd colume and the number of indirect CFTs in the 4th column. We also show the static binary size for each benchmark after compilation. The number of CFTs correlates with the size of the binary code; a larger code size will have more CFTs. Space overhead is due to the inserted code, so a program with more CFTs will have a higher space overhead. Table 1 shows that `400.perlbench` and `403.gcc` have the largest space overhead. Note that `445.gobmk` is as large as `403.gcc`, but only one-third is code. Hence, its space overhead is small. For SGX-nbench, we only report the size of code inside the enclave. On average, SGX-LAPD increases the static binary size by 213% with NPD and 244% with NXD.

In terms of compilation time, SGX-LAPD only introduces small overhead to the compiler. The building time for SPEC2006 is increased from 5672s to 5745s, with only additional 73 seconds more time. The building time for SGX-nbench is increased from 1.4s to 1.6s.

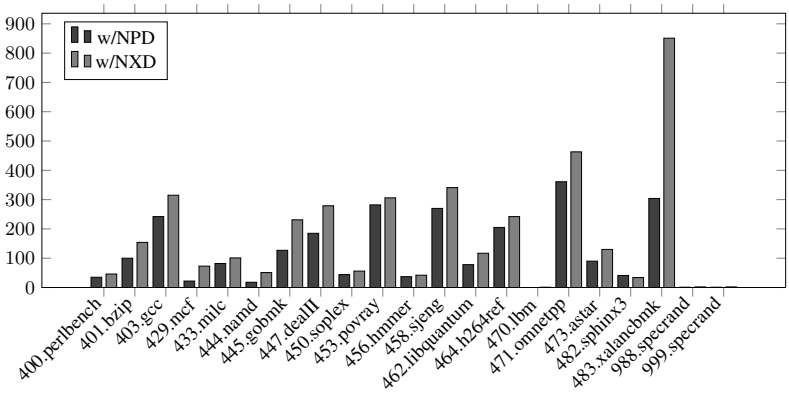

Fig. 7. Percent overhead for each of the SPEC2006 benchmarks.

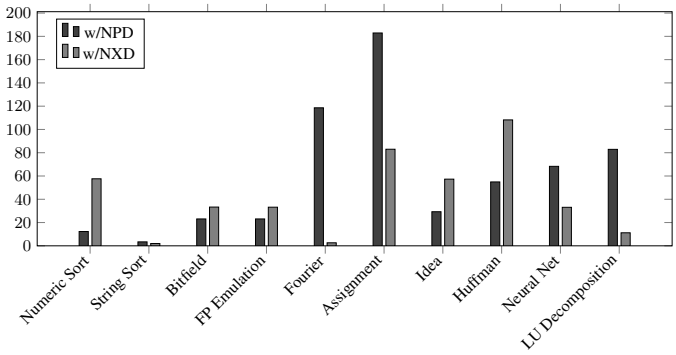

Fig. 8. Percent overhead for each of the SGX-nbench benchmarks.

SGX-LAPD **Linker.** In the linking phase, SGX-LAPD will optimize out the unnecessary instrumentation code. To show the efficiency of our optimization, we reported the number of patches for each benchmark in Table 1. As mentioned in §4.1, each direct CFT is associated with one piece of meta-data to record the instrumented code information. SGX-LAPD Linker scans that information to find all the direct CFTs for which the verification code does not need to be run and patches them with an unconditional branch. The patch number for NXD approach is larger. Currently, SGX-LAPD does not instrument the library code. We cannot use `call-ret` pair to check page fault, and thus verification code of CFT to library call need to be skipped. Note that this should not be a limitation of SGX-LAPD since in real application, the enclave code is built in static linked binary [1]. SGX-LAPD can instrument all the enclave code.

Runtime Performance. SGX-LAPD slows down the program execution time, which is caused by the additional page fault detection code inserted before each cross-page CFT. We evaluated the slowdown in both SPEC2006 and our own SGX-nbench. For SPEC2006, we measured the execution time overhead for each benchmark by running the instrumented benchmarks on their reference data sets 10 times, with a maximum

variance of 2%. In Figure 7, we present the execution time overhead for each SPEC2006 benchmark, shown as a percent increase over the normalized baseline performance of the non-instrumented version of each benchmark. Similar to space overhead, the NXD approach has a larger execution time overhead than NPD. In general, benchmarks with a larger number of CFTs will have higher overhead. As shown in Table 1, most of the verification code in `483.xalancbmk` cannot be skipped, which is why it has the largest overhead. On average, NPD introduces 120% overhead on SPEC2006, while NXD introduces 183% overhead.

For SGX-nbench, we used the performance result reported by the benchmark itself. In particular, SGX-nbench runs its benchmarks multiple times, taking the mean and standard deviation until it reaches a confidence threshold such that the results are 95% statistically certain. In Figure 8, we present the execution time overhead for each SGX-nbench benchmark, shown as a percent increase over the normalized baseline of the non-instrumented version. The average overhead of SGX-nbench is only 60% with NPD and 42% with NXD, smaller than SPEC2006. This is because SGX-LAPD only instruments the code inside the enclave. The host application code is not instrumented and has no overhead. This demonstrates the true performance of SGX-LAPD in real SGX programs.

7 Discussion

SGX-LAPD relies on the enclave code itself to detect page faults and verify whether an OS indeed provides large pages. All the code outside the enclave is not trusted, which means both the user level `sig_handler` and the kernel level system exception handler can be malicious. According to the detailed steps in exception handling described in Figure 3, we can notice that an attacker can execute the eight step exception handling instead of the three step page fault handling to reset the `GPRSGX.RIP` to some other instructions. But this relies on collaboration from the enclave code, which is trusted. Therefore, we have to note that such an attack is impossible unless the enclave code itself is compromised.

Meanwhile, we note that there might exist a race condition for a malicious OS to reset the `EXITINFO.VECTOR` right after entering the enclave as illustrated in Figure 4. More specifically, a malicious OS can first launch the page fault attack, causing `EXITINFO.VECTOR` to be set. When control returns to the enclave again but before our verification code, the malicious OS injects another interrupt (e.g., timer interrupt or other faults such as GP) and makes the enclave exit again (to reset `EXITINFO.VECTOR` and evade our detection). Fortunately, such an attack is challenging to launch. In particular, to launch this attack, attackers have to execute the enclave program using single step execution; otherwise it will be very challenging for them to control the timing. However, there is no way to execute enclave program using single step in the deployment mode (only debugging mode can), and attackers must rely on the extremely low probability to inject the interrupt or exception right after entering the enclave and before our checking code. But this time window is extremely short (just a few instructions).

In addition, there is a lot of room for further improvement of SGX-LAPD, particularly on where to instrument our detection code. For instance, our current design overly inserts a lot of intra-page control flow transfer page fault detection code in the enclave

binary, though we have patched the binary to skip executing that code. While our current design can be acceptable for small enclave binaries, especially considering the fact that we already ask the SGX to provide 2MB pages for the enclave code (such a design already wastes a large volume of space), we certainly would like to further eliminate this unnecessary code. We believe this would require iterative processing and instruction relocation. We leave this to one of our future works. On the other hand, if we were able to precisely identify the input-dependent CFTs, we would not have to insert excessive amounts of detection code. Therefore, the second avenue for future improvement is to identify the input-dependent CFTs. However, this is also a non-trivial task for a compiler since it would require a static input taint analysis. We leave this to another of our future works.

Finally, SGX-LAPD only stops code page fault attacks; attackers can still trigger data page faults. As mentioned in §3.1, we leave the defense for data page fault attacks to future work. We also would like to note that practical controlled channel attacks often require two kinds of page fault patterns, as demonstrated by Xu et. al. [31]. The first is the code page pattern which indicates the start or end of a specific function. The second can be either a code page fault pattern or a data page fault pattern, but it critically depends on the first code page fault pattern to be effective. By removing only code page fault patterns, SGX-LAPD can still make data page fault attacks much harder.

8 Conclusion

We have presented SGX-LAPD, a system that leverages enclave verifiable large paging to defeat controlled page fault side channel attacks based on the insight that large pages can significantly reduce benign page fault occurrence. A key contribution of SGX-LAPD is a technique that explicitly verifies whether an OS has provided large pages by intentionally triggering a page fault at each cross small page control flow transfer instruction and validating with the internal SGX data structure updated by the hardware. We have implemented SGX-LAPD with a modified LLVM compiler and an SGX Linux kernel module. Our evaluation with a ported real SGX benchmark SGX-nbench shows that, while space and runtime overhead can be somewhat high, as a first step solution SGX-LAPD can still be acceptable especially considering the difficulties in fighting for the controlled side channel attacks. Finally, the source code of SGX-LAPD is available at https://github.com/utds3lab/sgx-lapd, and the source code of SGX-nbench is available at https://github.com/utds3lab/sgx-nbench.

Acknowledgement

We thank Mona Vij from Intel for the assistance of the test with SGX-v2. We are also grateful to the anonymous reviewers for their insightful comments. This research was partially supported by AFOSR under grant FA9550-14-1-0119, and NSF awards CNS-1453011, CNS-1564112, and CNS-1629951. Any opinions, findings, conclusions, or recommendations expressed are those of the authors and not necessarily of the AFOSR and NSF.

References

1. Intel software guard extensions (intel sgx) sdk. https://software.intel.com/en-us/sgx-sdk.
2. The linux kernel archives. https://www.kernel.org/doc/Documentation/vm/hugetlbpage.txt.
3. Intel 64 and ia-32 architectures software developer'smanual. http://www.intel.com/content/dam/www/public/us/en/documents /manuals/64-ia-32-architectures-software-developer-manual-325462.pdf, Dec. 2015.
4. BACKES, M., AND NÜRNBERGER, S. Oxymoron: Making fine-grained memory randomization practical by allowing code sharing. In *USENIX Security Symposium* (2014).
5. BAUMAN, E., AND LIN, Z. A case for protecting computer games with sgx. In *Proceedings of the 1st Workshop on System Software for Trusted Execution* (2016), ACM, p. 4.
6. BAUMANN, A., PEINADO, M., AND HUNT, G. Shielding applications from an untrusted cloud with haven. In *Proceedings of the 11th Symposium on Operating Systems Design and Implementation (OSDI)* (Broomfield, Colorado, Oct. 2014), pp. 267–283.
7. CHANDRA, S., KARANDE, V., LIN, Z., KHAN, L., KANTARCIOGLU, M., AND THURAISINGHAM, B. Securing data analytics on sgx with randomization. In *Proceedings of the 22nd European Symposium on Research in Computer Security* (Oslo, Norway, September 2017).
8. CHECKOWAY, S., AND SHACHAM, H. Iago Attacks: Why the System Call API is a Bad Untrusted RPC Interface. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (Houston, TX, Mar. 2013), pp. 253–264.
9. CHEN, X., GARFINKEL, T., LEWIS, E. C., SUBRAHMANYAM, P., WALDSPURGER, C. A., BONEH, D., DWOSKIN, J., AND PORTS, D. R. Overshadow: a virtualization-based approach to retrofitting protection in commodity operating systems. In *Proceedings of the 13th international conference on Architectural support for programming languages and operating systems* (Seattle, WA, USA, 2008), ASPLOS XIII, ACM, pp. 2–13.
10. GIUFFRIDA, C., KUIJSTEN, A., AND TANENBAUM, A. S. Enhanced operating system security through efficient and fine-grained address space randomization. In *USENIX Security Symposium* (2012), pp. 475–490.
11. GOLDREICH, O. Towards a theory of software protection and simulation by oblivious rams. In *Proceedings of the nineteenth annual ACM symposium on Theory of computing* (1987), ACM, pp. 182–194.
12. HAND, S. M. Self-paging in the nemesis operating system. In *OSDI* (1999), vol. 99, pp. 73–86.
13. HOEKSTRA, M., LAL, R., PAPPACHAN, P., PHEGADE, V., AND DEL CUVILLO, J. Using innovative instructions to create trustworthy software solutions. In *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy (HASP)* (Tel-Aviv, Israel, 2013), pp. 1–8.
14. INTEL. Intel Software Guard Extensions Programming Reference (rev2), Oct. 2014. 329298-002US.
15. KARANDE, V., BAUMAN, E., LIN, Z., AND KHAN, L. Securing system logs with sgx. In *Proceedings of the 12th ACM Symposium on Information, Computer and Communications Security* (Abu Dhabi, UAE, April 2017).
16. KLEIN, G., ELPHINSTONE, K., HEISER, G., ANDRONICK, J., COCK, D., DERRIN, P., ELKADUWE, D., ENGELHARDT, K., KOLANSKI, R., NORRISH, M., SEWELL, T., TUCH, H., AND WINWOOD, S. sel4: Formal verification of an os kernel. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles* (2009), SOSP '09, pp. 207–220.
17. MCCUNE, J. M., PARNO, B. J., PERRIG, A., REITER, M. K., AND ISOZAKI, H. Flicker: An Execution Infrastructure for TCB Minimization. In *Proceedings of the ACM EuroSys Conference* (Glasgow, Scotland, Mar. 2008), pp. 315–328.

18. MCKEEN, F., ALEXANDROVICH, I., BERENZON, A., ROZAS, C. V., SHAFI, H., SHANBHOGUE, V., AND SAVAGAONKAR, U. R. Innovative instructions and software model for isolated execution. In *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy (HASP)* (Tel-Aviv, Israel, 2013), pp. 1–8.
19. OHRIMENKO, O., SCHUSTER, F., FOURNET, C., MEHTA, A., NOWOZIN, S., VASWANI, K., AND COSTA, M. Oblivious multi-party machine learning on trusted processors. In *25th USENIX Security Symposium (USENIX Security 16)* (Austin, TX, Aug. 2016), USENIX Association, pp. 619–636.
20. PEREZ, R., SAILER, R., VAN DOORN, L., ET AL. vTPM: virtualizing the trusted platform module. In *Proceedings of the 15th Usenix Security Symposium (Security)* (Vancouver, Canada, July 2006), pp. 305–320.
21. PINKAS, B., AND REINMAN, T. Oblivious ram revisited. In *Advances in Cryptology–CRYPTO 2010*. Springer, 2010, pp. 502–519.
22. PORTER, D. E., BOYD-WICKIZER, S., HOWELL, J., OLINSKY, R., AND HUNT, G. C. Rethinking the library os from the top down. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (Newport Beach, CA, Mar. 2011), pp. 291–304.
23. SANTOS, N., RAJ, H., SAROIU, S., AND WOLMAN, A. Using arm trustzone to build a trusted language runtime for mobile applications. In *ACM SIGARCH Computer Architecture News* (2014), vol. 42, ACM, pp. 67–80.
24. SCHUSTER, F., COSTA, M., FOURNET, C., GKANTSIDIS, C., PEINADO, M., MAINAR-RUIZ, G., AND RUSSINOVICH, M. VC3: Trustworthy Data Analytics in the Cloud using SGX. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy* (2015).
25. SESHADRI, A., LUK, M., QU, N., AND PERRIG, A. Secvisor: a tiny hypervisor to provide lifetime kernel code integrity for commodity oses. In *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles* (Stevenson, Washington, USA, 2007), SOSP '07, pp. 335–350.
26. SHIH, M.-W., LEE, S., KIM, T., AND PEINADO, M. T-sgx: Eradicating controlled-channel attacks against enclave programs. In *Proceedings of the 2017 Annual Network and Distributed System Security Symposium (NDSS), San Diego, CA* (2017).
27. SHINDE, S., CHUA, Z. L., NARAYANAN, V., AND SAXENA, P. Preventing your faults from telling your secrets: Defenses against pigeonhole attacks. *arXiv preprint arXiv:1506.04832* (2015).
28. SUN, K., WANG, J., ZHANG, F., AND STAVROU, A. SecureSwitch: BIOS-assisted isolation and switch between trusted and untrusted commodity oses. In *Proceedings of the 19th Annual Network and Distributed System Security Symposium (NDSS)* (San Diego, CA, Feb. 2012).
29. TEAM, P. Pax address space layout randomization (aslr). *http://pax.grsecurity.net/docs/aslr.txt*.
30. WANG, J., STAVROU, A., AND GHOSH, A. K. Hypercheck: A hardware-assisted integrity monitor. In *Recent Advances in Intrusion Detection, 13th International Symposium, RAID 2010, Ottawa, Ontario, Canada, September 15-17, 2010. Proceedings* (2010), pp. 158–177.
31. XU, Y., CUI, W., AND PEINADO, M. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy* (2015).