Time and Order: Towards Automatically Identifying Side-Channel Vulnerabilities in Enclave Binaries

Wubing Wang Yinqian Zhang Zhiqiang Lin
Department of Computer Science and Engineering
The Ohio State University

Abstract

While Intel SGX provides confidentiality and integrity guarantees to programs running inside enclaves, side channels remain a primary concern of SGX security. Previous works have broadly considered the side-channel attacks against SGX enclaves at the levels of pages, caches, and branches, using a variety of attack vectors and techniques. Most of these studies have only exploited the "order" attribute of the memory access patterns (e.g., sequences of page accesses) as side channels. However, the other attribute of memory access patterns, "time", which characterizes the interval between two specific memory accesses, is mostly unexplored. In this paper, we present ANABLEPS, a tool to automate the detection of side-channel vulnerabilities in enclave binaries, considering both order and time. ANABLEPS leverages concolic execution and fuzzing techniques to generate input sets for an arbitrary enclave program, constructing extended dynamic control-flow graph representation of execution traces using Intel PT, and automatically analyzing and identifying side-channel vulnerabilities using graph analysis.

1 Introduction

Intel Software Guard eXtension (SGX) is a hardware addition that is available in recent Intel processors. It offers both integrity and confidentiality to application software running in a shielded execution environment—a secure enclave—even when the entire operating system is untrusted. Recent work has explored the use of Intel SGX for a variety of applications such as secure cloud data analytics [25], smart contracts [44], anonymity network [18], game hacking protection [7], and unmodified code execution [8, 31], which have outlined a promising future of SGX's broad adoption in both server-end and client-side computation.

Computer micro-architecture related side channels are not new. Side-channel attacks that exploit micro-architectural resources shared by mutually distrusting computing entities (*e.g.*, processes or threads) date back to the era of Pentium 4 [23, 24]. A malicious program or a virtual machine may manipulate the shared micro-architectural resources, such as CPU caches, branch prediction units, or function units, to learn the pattern with which these resources are used by the victim program and thereby infer secrets that dictate such a usage pattern. Over the past decades, computer micro-architecture has evolved drastically, but the issues of side channels remain. What differ in the SGX context are two fold: First, as SGX is designed to protect the confidentiality of applications that demand high levels of security, side channels become a major security threats. Second, because the adversary against SGX enclaves is assumed to have OS system-level privileges, a wider range of attacks are enabled. Particularly, over the past a few years, researchers have demonstrated that secrets can be leaked from a variety of attack vectors, such as branch prediction units [20], CPU caches [17], paging structures [35, 38, 43], and DRAM row buffers [38].

Completely eliminating side channels from CPU chips is unrealistic. Admitting this decades-old security concern, Intel recommends developers take special care to avoid side-channel vulnerabilities when writing enclave code [6]. However, developers are not experts of side channels and relying on regular program developers to solve side-channel issues is less promising. Moreover, there is no tool available that helps the developers automatically identify improper coding patterns in their enclave binaries.

In this paper, we aim to explore principles and techniques that automatically identify side-channel vulnerabilities in enclave binaries that allow a side-channel attacker who is able to observe execution traces of the *control flow* of an enclave program to infer sensitive information inside the enclave. The root cause of the vulnerability is the secret-dependent control flows that are inherent in the enclave code. More specifically, since side-channel attacks observe the runtime behavior of the enclave programs, an intuitive approach for the vulnerability identification would be to find a large set of secret values (*e.g.*, input of the enclave program), run the enclave program with these secret values, and collect the enclave's execution traces with respect to the control flow transfers (CFTs). The diversity

of the collected execution traces for different secret values is a viable indicator of the side-channel vulnerabilities—if all secret values correspond to the same execution trace, the enclave code is not vulnerable. With respect to the execution traces, there are both spatial (*i.e.*, order) and temporal (*i.e.* time) differences. A comprehensive solution should include both.

However, it is non-trivial to develop such a comprehensive approach for a number of reasons. First, how to generate the valid secret values (*e.g.*, program input) to expose the execution traces at different granularity (*e.g.*, branch, cache, or page). Second, how to collect the execution traces, especially the temporal information associated with the traces. We cannot use static analysis as it will not be able to resolve secret-dependent CFTs, and meanwhile cannot collect the precise time information. While we can use dynamic analysis, we still need to solve the coverage issues. Third, how to represent the execution traces and perform the cross-comparison, especially when there are multiple execution traces. Finally, how to quantitatively analyze the information leakage due to the detected vulnerabilities. Fortunately, we have addressed these challenges and built a tool dubbed ANABLEPS, by leveraging concolic execution and fuzzing techniques to generate input sets for an arbitrary enclave program, constructing extended dynamic control-flow graph representation of execution traces using Intel PT, and automatically analyzing and identifying side-channel vulnerabilities using graph analysis.

We have tested ANABLEPS with 8 programs and libraries, including text rendering, image processing, gnomic processing, and deep learning. Our tool has discovered numerous input leakage execution points for these programs. Our study also suggests automated tools can identify the side-channel vulnerabilities based on syntactic inputs and execution traces. However, the semantics (*i.e.*, the meaning) of the input is also of critical importance especially for the exploitation of the side-channel vulnerabilities.

Contributions. To summarize, the contributions of this paper are as follows:

- A novel and comprehensive approach to detecting both time-based and order-based control-flow side-channel vulnerabilities for enclave binaries.
- A practical implementation integrating fuzzing, symbolic execution, and hardware supported execution tracing.
- The first large-scale analysis of sensitive control-flow vulnerabilities for real world enclave binaries.

Roadmap. The rest of the paper is organized as follows. §2 presents necessary background knowledge including related works to facilitate our discussion of the problem and our motivation. In §3, we present the problem statement and a running example to highlight our key insights. We detail our design of ANABLEPS in §4. Then, we present how we implement ANABLEPS and evaluate its effectiveness in §5. We also made a number of case studies to understand the exploitability of the vulnerabilities in §6. §7 discusses the limitation of the approach and future research directions. Finally, §8 concludes the paper.

2 Background and Related Work

Intel SGX. At a high level, Intel SGX is a set of new instructions for the x86 architecture. These instructions allow application developers to protect sensitive code and data by utilizing a secure container called enclave [13]. The trusted hardware establishes an enclave by protecting isolated memory regions within the existing address space called Processor Reserved Memory (PRM) to assure confidentiality and integrity against other non-enclave memory accesses, including kernel, hypervisor, and other privileged code. The confidentiality of regions outside the PRM is protected by the memory encryption engine (MEE). Enclave programs with memory footprints larger than that is allowed by RPM can make use of memory regions outside the PRM via page swapping. Memory pages swapped out of the RPM need to be encrypted by MEE.

SGX Side-Channel Attacks. Side channels are the Achilles' Heel of Intel SGX's confidentiality guarantees. In the past few years, a variety of side-channel attacks have been demonstrated against SGX enclaves, particularly from the CPU's memory management perspective. For instance, it has been demonstrated that by controlling the *present* flag or the *reserved* flags of the page table entries (PTEs) [29, 43], the adversary could force the enclave program to trigger page faults when accessing a memory page, thus extracting sufficient amount of secrets (*e.g.*, image contours, user input, cryptographic keys). Most recently, it was shown that the page table access patterns can also leak the enclave secrets without actively triggering the page fault [35, 38], which can be achieved by monitoring the *accessed* flag of the PTEs.

Other micro-architectural side-channel attack vectors that have been studied on traditional hardware have also been found exploitable in SGX. It has been demonstrated that cache-based side-channel attacks can be migrated on SGX [9, 15, 17, 26], which can be more powerful than non-SGX settings. Branch prediction units have been demonstrated to leak the branch history inside the enclaves [20]. DRAM row buffer contention has been exploited to steal secrets from enclaves [38].

Most recently, Spectre [19], Meltdown [21], Foreshadow [32], and SGXPectre [10] attacks have been demonstrated to leverage speculative execution and out-of-order execution to read memory content protected by MMU isolation. These attacks are out of scope of this paper as they are micro-architecture vulnerabilities which cannot be solely addressed from software.

Existing Defenses. A number of enclave hardening techniques have been proposed to mitigate these side-channel attacks. To defeat page-level side-channel attacks, T-SGX [28] uses the Transactional Synchronization Extensions (TSX), DéJà Vu [12] relies on the execution time of the enclave program path, SGX-LAPD [14] explores the internal enclave data structures. To guard against cache side channels, Gruss *et al.* [16] encapsulates snippets of enclave code into hardware-supported memory, HyperRace [11] implements contrived data races. Varys [22] also proposes to reserve physical cores for secure enclave computation.

Closely related works to ours are Stacco [42], MicroWalk [40], and DATA [39], all of which detect side-channel vulnerabilities due to secret-dependent control flows. Particularly, Stacco [42] uses Intel Pin tools to detect vulnerabilities in SSL/TLS implementations, and it manually generates input to the SSL libraries, and MicroWalk [40] focuses on vulnerabilities in Intel IPP and Microsoft CNG. Similarly, DATA [39] only focuses on differential address trace analysis for cryptographic primitives. In contrast, as ANABLEPS works on arbitrary enclave binary, it must generate the large volume of input automatically and conduct vulnerability analysis without known semantics. These new design challenges differentiate our work and Stacco, DATA and MicroWalk. Outside the SGX context, CacheD [37] is also relevant to our work. However, in contrast to these works, ours considers more attack vectors.

3 Overview

3.1 Problem Statement and Definitions

The key objective of this work is to automatically identify the side-channel vulnerabilities caused by the secret-dependent control-flow transfers in the enclave programs. As enclave programs are typically shipped to the hosting services in the form of plaintext binary code, we anticipate the primary secret that the enclave developer would like to hide is the input to the enclave code. Therefore, the goal of the attacks is to learn, through a variety of side channels (*e.g.*, page accesses [29, 35, 38, 43], cache eviction [9, 15, 17, 26, 33], and branch prediction [20]), the input to the enclave programs.

However, most of these prior studies on SGX side channels only consider the *order* attribute of memory access patterns, *i.e.*, which memory page (or cache set) has been accessed and in what order. Few has exploited the *time* of memory accesses as a side-channel vector. In fact, the first observation that *time* and *order* are the two key attributes of a side (and covert) channel can date back to the early 1990s [41]. As such, in our work, we consider both, and broadly define that an enclave program is vulnerable to side-channel attacks if different input can lead to different traces from either the executing *order* of each execution unit (*e.g.*, an instruction) or the *timing* at which each unit is visited.

Defining Side-Channel Vulnerabilities. More formally, given an enclave binary program p, a concrete input to p will lead to a concrete execution trace r, which is defined as $[(m_0,t_0),(m_1,t_1),(m_2,t_2),\cdots,(m_k,t_k)]$, where m_j is the address of the j^{th} execution unit and t_j is its timestamp relative to the beginning of the execution. When the memory addresses are normalized to be free of effects of randomization, for each input, there is a corresponding trace r.

***Definition* 1** *Given an enclave program p and an input I_i, the mapping function $\mathcal{E}(p,I_i)=r_i$, where r_i is the execution trace of p under the input I_i. Similarly, for a set of input I, we define the mapping function $\mathcal{E}(p,I)=\{r_i|r_i=\mathcal{E}(p,I_i),\forall I_i\in I)\}$. The entire input space is denoted $I_{\texttt{space}}$. Therefore, the entire space of execution traces $R=\mathcal{E}(p,I_{\texttt{space}})$.*

The mapping function $\mathcal{E}$ generates a program's execution trace under a specific input or a set of inputs, which allows us to define side-channel vulnerabilities as follows.

***Definition* 2** *Given an enclave program p and a set of input I, the program is considered to be vulnerable to side-channel attacks (under the input set I) if and only if $|\mathcal{E}(p,I)|>1$; the input set can be completely leaked through the side channels if and only if $|\mathcal{E}(p,I)|=|I|$.*

Informally, we define an enclave program p is vulnerable to side-channel attack if not all the input maps to the same trace. That is, the enclave program's execution is not input oblivious. However, even though the program is vulnerable to side-channel attack, the amount of leaked information can be different. The complete leakage captures the case that every input can be uniquely identified from the execution trace. It is worth noting that the set of input I is a subset of the entire input space $I_{\texttt{space}}$, *i.e.*, $I\in I_{\texttt{space}}$. In most practical scenarios, it is impossible to obtain $I_{\texttt{space}}$. Therefore, the definition of side-channel vulnerabilities is only meaningful when the program and its input set is fixed. In this paper, we consider two types of input set I: $I_{\texttt{syntactic}}$, the set of input generated automatically from program analysis, and $I_{\texttt{semantic}}$, the set of input provided by developers that are semantically meaningful.

Representing Execution Traces. To facilitate cross comparison of execution traces and directly pinpoint the secret-dependent control flow transfer (CFT) that leaks the information through side channels, execution traces need to be represented in proper data structures. String, in the form of linear trace $[(m_0,t_0),(m_1,t_1),(m_2,t_2),\cdots,(m_k,t_k)]$, however, is not an optimal choice as it will be quite challenging to identify the alignment (*i.e.*, anchor) point from the string. In our design, we choose to use a graph representation of the linear traces.

***Definition* 3** *An extended dynamic control-flow graph (ED-CFG) of a program p under input $I_i\in I$ is defined as a directed graph $G=<N,E>$, where $n_i\in N$ is a node of the graph that represents a basic block of the CFG; and $e_i\in E$ is a directed edge of the graph connecting two nodes that represents the dynamic CFT when p is executed with the input I_i.*

Also, each edge ($e_i \in E$) has a counter w_i (i.e., weight) to indicate how many times the edge is executed. The information of the program's execution order and time is embedded in each node $n_i \in N$. Each $n_i \in N$ has two ordered lists: `Order` $= [n_1^i, n_2^i, \cdots, n_k^i]$, *where n_j^i is the j^{th} successor of node n_i during the execution of p with input I_i;* `Time` $= [t_1^i, t_2^i, \cdots, t_k^i]$, *where t_j^i is the execution time to reach node n_j^i.*

An ED-CFG of an enclave program uniquely specify the execution trace of the program under a given input. More specifically, $\mathcal{G}^i$ represents the execution trace in a graph representation for the input I_i.

Execution Units in Side-Channel Attacks. An execution unit in the context of a side-channel attack is defined as the minimal single execution trace observable by attackers. For the enclave program execution, an attacker can mostly achieve the minimal execution unit at either cache level, or at page level. Typically, it is hard to observe the single instruction execution or basic block execution, but an attacker might be able to do so at certain scenario (*e.g.*, the branch shadowing attack [20] and the Nemesis attack [34]). Therefore, in our work we focus on the execution unit at page level (address aligned with 4K bytes), at cache level (address aligned with 64 bytes)[1], and at branch level.

***Definition* 4** *A page-level ED-CFG, $\mathcal{G}_p$, is a variant of $\mathcal{G}$, where each node of $\mathcal{G}_p$ contains the page execution unit (i.e., all the executed instructions that belong to a particular page, aligned with 2^{12} bytes), and each edge connects the CFTs between the pages. Similarly, we define the cache-level ED-CFG, $\mathcal{G}_c$, where each node contains the cache execution unit and edge captures the CFTs at cache level.*

Therefore, eventually for each input I_i, we will build $\mathcal{G}^i$ first, from which to derive $\mathcal{G}_p^i$ and $\mathcal{G}_c^i$. To detect the vulnerabilities, we will then cross compare $\mathcal{G}^i$, $\mathcal{G}_p^i$, or $\mathcal{G}_c^i$, respectively, for all input $I_i \in I$. If a trace is different (in terms of time or order of the specific execution units) among different user input, we conclude the enclave program is vulnerable to the corresponding side-channel attacks at different levels such as at branch, cache, or page. Further analysis can be performed on the graphs to quantify the vulnerability, or to identify the leaking code segments.

3.2 A Running Example

Next, we would like to use a simple running example to illustrate how to use $\mathcal{G}_p$ to detect the time and order side-channel vulnerabilities at the page granularity for the software running inside the SGX enclave. Detecting basic block-granularity and cacheline-granularity vulnerabilities is similar when given $\mathcal{G}_c$. In particular, we use the code snippet shown in Figure 1(d) as a running example. This code snippet is a simplified version of a barcode image processing function.

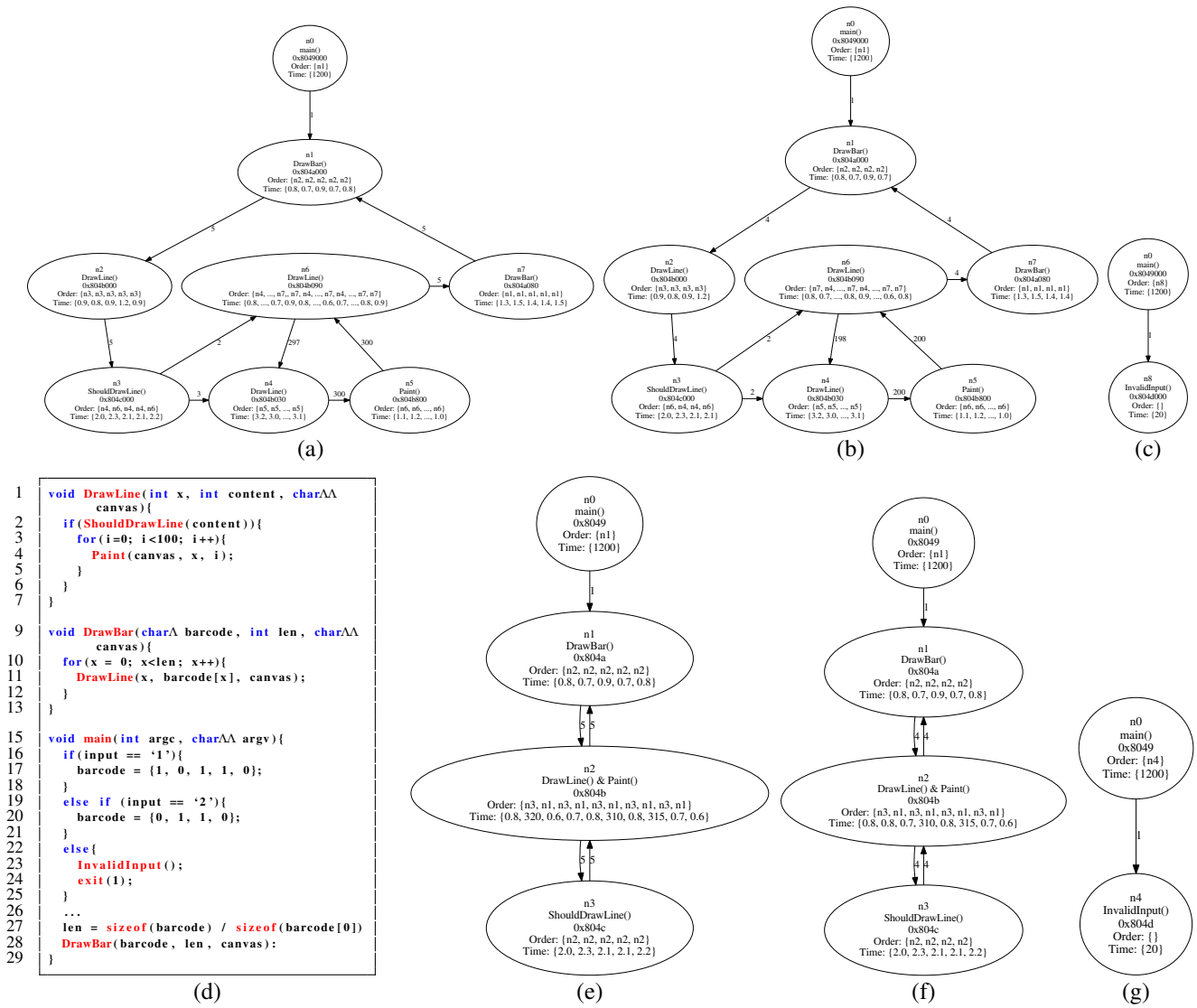
We notice in Figure 1(d) that this program takes three types of inputs: character '1', '2', or an illegal input. The program outputs two types of barcode, or an error message, accordingly. More specifically, function `main()` calls function `DrawBar()` if the input character is '1' or '2', otherwise returns an error (and `exit`). Function `DrawBar()` is used to draw a barcode on the canvas, and the weight of the canvas is decided by the length of the barcode. Then for each column of the barcode, it calls function `DrawLine()`, which calls the function `Paint()` in a loop if the given position is to draw a line.

Trace Construction. By providing input I_1 with '1', I_2 with '2', and an invalid input $I_{\texttt{invalid}}$, we get the corresponding execution traces $\mathcal{E}(p, I_1)$, $\mathcal{E}(p, I_2)$, and $\mathcal{E}(p, I_{\texttt{invalid}})$, from which to build $\mathcal{G}^1$, $\mathcal{G}^2$, and $\mathcal{G}^{\texttt{invalid}}$. As shown in Figure 1(a)(b)(c), each node represents the executed basic block, and each edge represents the CFT between the basic blocks. We also assigned an index for each node for easier locating them in the graph (*e.g.*, n_1 and n_2). Two ordered lists, `Order` and `Time`, associated with each node record the successor nodes (in execution order) and the execution time (in nanosecond *ns*) to reach them during execution. For instance, in Figure 1(a), the `Order` list of node n_6 is $[n_4, n_4, \cdots, n_7, \cdots, n_7, \cdots]$, which suggests that the execution of the program will first follow the edge from $n_6 \rightarrow n_4$ multiple times, then follow the edge from $n_6 \rightarrow n_7$. The first element of the `Time` list suggests the mean execution time to reach node n_4 for the first time is $0.8ns$.

The corresponding page-level ED-CFGs ($\mathcal{G}_p^1$, $\mathcal{G}_p^2$, and $\mathcal{G}_p^{\texttt{invalid}}$) are illustrated in Figure 1(e)(f)(g). For instance, the ED-CFG in Figure 1(a) can be converted to the page-level ED-CFG in Figure 1(e) in the following steps: First, node n_1 and n_7 of the original ED-CFG are both placed on page $0x804a$, they are merged to a single node n_1 in the page-level ED-CFG. Similarly, node n_2, n_4, n_5, and n_6 are merged into node n_2 in page-level ED-CFG. Edges between nodes of the same page are removed in the page-level ED-CFG; those crossing page boundaries are preserved or merged. For instance, the edge $n_2 \rightarrow n_3$ becomes the new edge $n_2 \rightarrow n_3$ in $\mathcal{G}_p^1$, and the edges $n_3 \rightarrow n_6$ and $n_3 \rightarrow n_4$ merges into the new edge $n_3 \rightarrow n_2$ in $\mathcal{G}_p^1$. We point out that it is not always straightforward to convert ED-CFG to page-level ED-CFG. Some basic blocks in ED-CFG may cross the page boundary. Dealing with these pages require additional efforts, which we will discuss in more details in §4.

Vulnerability Identification. By comparing the $\mathcal{G}_p$s ($\mathcal{G}$s or $\mathcal{G}_c$s), one can easily identify the side-channel vulnerabilities. For instance, by comparing Figure 1(e) and Figure 1(f), it can be seen that the two input values, '1' and '2', leads to different page-level execution orders: the sequence of $n_1 \rightarrow n_2 \rightarrow n_3 \rightarrow n_2 \rightarrow n_1$ is repeated one more time when the input is '1'. Figure 1(g) is very different from the other

[1] In this work, we simply model cache-based side-channel attacks on SGX assuming that the attacker is able to monitor the execution of the enclave program at the granularity of a 64-byte memory block. Interested readers can refer to Wang *et al.* [38] for more detailed discussion on attack techniques.




```
1  void DrawLine(int x, int content, charΛΛ
       canvas){
2    if(ShouldDrawLine(content)){
3      for(i=0; i<100; i++){
4        Paint(canvas, x, i);
5      }
6    }
7  }

9  void DrawBar(charΛ barcode, int len, charΛΛ
       canvas){
10   for(x = 0; x<len; x++){
11     DrawLine(x, barcode[x], canvas);
12   }
13 }

15 void main(int argc, charΛΛ argv){
16   if(input == '1'){
17     barcode = {1, 0, 1, 1, 0};
18   }
19   else if (input == '2'){
20     barcode = {0, 1, 1, 0};
21   }
22   else{
23     InvalidInput();
24     exit(1);
25   }
26   ...
27   len = sizeof(barcode) / sizeof(barcode[0])
28   DrawBar(barcode, len, canvas);
29 }
```

Figure 1: (a) $\mathcal{G}^1$, (b) $\mathcal{G}^2$, (c) $\mathcal{G}^{invalid}$, (d) Code snippet of our running example, (e) $\mathcal{G}_p^1$, (f) $\mathcal{G}_p^2$, and (g) $\mathcal{G}_p^{invalid}$

two $\mathcal{G}_p$s, easily differentiating $I_{\texttt{invalid}}$ from other input. We can validate this vulnerability by scrutinizing the code in Figure 1(d): Function `DrawLine()` is called five times when input value is '1', but four times when the value is '2'; the `main()` exits directly with invalid input.

Interestingly, an adversary can also infer more useful knowledge about whether a column in the barcode is a black line or white line. More specifically, according to the implementation of function `DrawLine()`, it will call function `Paint()` 100 times to draw a black line. Therefore, the execution time of function `DrawLine()` is much longer when it draws a black line in given position that of drawing a white line. With this information, the adversary can successfully recover the content of barcode by collecting the execution time of the page node on which function `DrawLine()` is placed. This vulnerability can be detected by scrutinizing the `Time` list of node n_2. Let $n_2.\texttt{Time}[k]$ denote the k^{th} element of the `Time` list of node n_2 (the index of an element starts with 1). The execution time to reach node n_1 from n_2, *i.e.*, $n_2.\texttt{Time}[2]$, $n_2.\texttt{Time}[6]$, $n_2.\texttt{Time}[8]$ of graph $\mathcal{G}_p^1$ are significantly larger (> 300 *ns*) than $n_2.\texttt{Time}[4]$ and $n_2.\texttt{Time}[10]$ ($<$ 1ns). Therefore, it can be inferred that the painted barcode is $[1,0,1,1,0]$, which correspond to input '1'.

3.3 Threat Model, Scope, and Assumptions

We assume the knowledge of at least the enclave binary code, especially the code layout and mapping. We assume there is no address space layout randomization (ASLR) with the enclave binaries (such as SGX-Shield [27]). We assume the adversary is capable of launching, resetting, and terminating the targeted enclaves, and is in control of the entire operating

system. This threat model is consistent to the controlled side-channel attacks [43] and also many other side-channel attacks against SGX enclaves [9, 15, 17, 26, 33].

Not all of the side channels are of our focus in this paper. In particular, we focus on identifying the side channels through branch, cache, page access behaviors, and timing information. Other side channels such as hardware architecture caused side channels (e.g., Meltdown [21] and Spectre [19]) are out of the scope. Also, we focus on the side channels from code access, and data access pattern caused side channel is out of scope.

4 Design

4.1 Input Generation

Since ANABLEPS uses dynamic analysis, it is important to generate the concrete input that covers as much as possible of the input space $I_{\texttt{space}}$. Fortunately, we are not the first to encounter such a problem, and many of the existing vulnerability identification tools all faces similar challenge. The state of the art is to combine both concolic execution (a.k.a, dynamic symbolic execution) and evolution fuzz testing (*e.g.*, AFL [1]) together to generate the best set of $I_{\texttt{syntactic}}$ (*e.g.*, Driller [30]). Therefore, when design ANABLEPS, we use the Driller approach and extend it for our purpose.

In particular, to start our analysis, we first use AFL [1] to execute the enclave program. The input generated by AFL is called $I_{\texttt{fuzz}}$. When fuzzing gets stuck and cannot explore the program path further, we use concolic execution to solve the path constraints and generate new input, which is called $I_{\texttt{concolic}}$. With the new input, we again let fuzzing execute first and only when fuzzing gets stuck, we invoke the concolic execution. When both fuzzing and concolic execution cannot explore the program path further, we terminate the input generation analysis.

During this stage execution, we have collected as many as possible of the program traces, which are the best effort to approximate R in the state of the art. and the minimal possible concrete input $I_{\texttt{syntactic}} = I_{\texttt{fuzz}} \cup I_{\texttt{concolic}}$ we use to expose these traces, denoted as $\mathcal{E}(p, I_{\texttt{syntactic}})$, where $\mathcal{E}(p, I_{\texttt{syntactic}}) \subseteq R$. At this stage, for each $I_i \in I_{\texttt{syntactic}}$, we have a corresponding $r_i \in \mathcal{E}(p, I_{\texttt{syntactic}})$. While we do know $|I_{\texttt{syntactic}}|$ (since each I_i is unique), we do not know $|\mathcal{E}(p, I_{\texttt{syntactic}})|$ yet, since we do not know whether each r_i is unique or not.

4.2 Trace Construction

Next, we describe how the concrete execution traces $\mathcal{E}(p, I_{\texttt{syntactic}})$ were collected when running the enclave program with each given input $I_i \in I_{\texttt{syntactic}}$, and also describe how we construct various ED-CFG representations (*e.g.*, $\mathcal{G}^i$, $\mathcal{G}^i_p$, and $\mathcal{G}^i_c$) that are suitable for the vulnerability identification from $\mathcal{E}(p, I_{\texttt{syntactic}})$.

Trace Collection. ANABLEPS requires collecting information regarding both the execution order and time of an execution trace. There are a variety of approaches to collecting these traces, such as using Intel Processor Trace (PT) and Last Branch Records (LBR). The issue with LBR is that it only has limited number of entries and branch records can be lost if not collected in timely manners. Therefore, we use Intel PT to conduct dynamic analysis. Intel PT is a hardware feature available on recent Intel processors (*i.e.*, Broadwell or later families) to facilitate program debugging and performance profiling. It collects the information of the CFTs of a program with very small performance overhead. A useful feature of PT is that it also records timestamps together with the CFTs, and thus it perfectly fits the purpose of our design.

Although PT provides timestamps information of control flow transfers, it does not provide fine-grained time information of each execution unit, *e.g.*, an instruction. Moreover, because the Cycle Count (CYC) packets are generated right before the event packets such as Taken NotTaken (TNT) packets, which may include taken or not taken information of up to 6 consecutive conditional branches, precisely recording execution time is not even possible at the basic-block granularity. As such, ANABLEPS only approximates the execution time in its construction of ED-CFGs, which will be detailed later. Also, ANABLEPS sets the memory buffer large enough so that no packet is lost during the dynamic analysis. The recorded packets are then parsed and recorded in a log file, which will be used for ED-CFG construction.

ED-CFG Construction. We generate the ED-CFG $\mathcal{G}^i$, for a given input I_i and trace r_i, based on the execution order and time of each basic block tracked in the trace files by Intel PT. Eventually, a D-CFG will be firstly built according to the PT trace file, where each node represents a basic block, each edge represents the CFT between the blocks, and the weight of each edge represents how many times the corresponding CFT has been executed.

Next, we add the execution order and time information into D-CFG to make it become ED-CFG, namely $\mathcal{G}^i$, for input I_i. More specifically, in each node, we use two lists to record the execution order and time for every basic block. The order list records the next node to jump to, and time list records the execution time every time when current node gets executed. The execution order is acquired by traversing the PT trace file again. However, for the execution time of each basic block each time when it gets executed, we have to approximate it (get a lower bound and upper bound) since PT does not offer fine-grained time recording for each basic block.

Resolving the execution time for each basic block. To get the execution time for basic blocks, we have to rely on the CYC packet, which is generated before each Mini Timestamp Counter (MTC) packet, TNT and Target IP (TIP) packet. However, not all of CFTs between basic blocks will generate a CYC packet as one TNT packet can capture up to six ba-

sic block execution. Therefore, we have to approximate the execution time for each basic block.

We take the following strategies to estimate the upper bound and lower bound of the CPU cycles for a basic block. The upper bound is an over-estimated execution time for each basic block with the CPU cycles recorded in the CYC packet, and the lower bound is the shortest CPU cycles in theory.

- **Upper Bound**. The upper bound of a basic block execution time is the CPU cycles recorded in the CYC packet, regardless of the number of basic blocks the CYC packet has covered.

- **Lower Bound**. The lower bound of a basic block execution time is the sum of the latency of the instructions that belong to the basic block. The latency for each individual instruction is acquired from [4].

While we cannot provide precise estimate of the execution time for each basic block, fortunately, we will get the precise PT recorded information for many of the basic blocks when we merge them to generate $\mathcal{G}_p^i$ and $\mathcal{G}_c^i$, based on page or cache level execution unit if the basic blocks recorded by the TNT packets actually belong to these execution units.

$\mathcal{G}_p^i$ and $\mathcal{G}_c^i$ Generation. Once we have built $\mathcal{G}^i$ for each input I_i, next we would like to derive $\mathcal{G}_p^i$ and $\mathcal{G}_c^i$ such that our vulnerability identification can be performed. Since the difference between page level execution unit and cache level execution unit is only the address alignment is different (2^{12} vs. 2^8), in the following we just describe how we convert $\mathcal{G}^i$ to $\mathcal{G}_p^i$ ($\mathcal{G}^i$ to $\mathcal{G}_c^i$ is similarly converted).

The conversion is straightforward, we need to combine all the basic block nodes that belong to the same page into just a single page node, and add the corresponding edges when there is a CFT between the pages. Also, we have to split the basic block that crosses two pages. To make our algorithm simple, we just first get all of the page numbers for all of the executed basic blocks by traversing $\mathcal{G}^i$, and then we traverse $\mathcal{G}^i$ again to add the edges between the pages, and to add orders and timing on the page node. Especially, for timing information, we discard our lower and upper bound timing estimation for each basic block that was captured by the TNT packet if they all belong to the same page.

We take a two step approach to convert $\mathcal{G}^i$ to $\mathcal{G}_p^i$. The first step is to generate the corresponding node and edge for $\mathcal{G}_p^i$ by traversing $\mathcal{G}^i$, and the second step is to traverse $\mathcal{G}^i$ again to generate the execution order and timing information.

- **Generating nodes and edges**. We design an algorithm shown in algorithm 1 to illustrate this. At a high level, we need to combine all the basic block nodes that belong to the same page into just a single page node, and add the corresponding edges when there is a control flow transfer between the pages. Also, we have to split the basic block that crosses two pages. To make our algorithm simple, we just first get all of the page numbers for all of the executed basic blocks by traversing $\mathcal{G}^i$, and then we traverse $\mathcal{G}^i$ again to add the edges between the pages. The weights are updated accordingly when there is a cross-page control flow transfer.

Algorithm 1: Generating the nodes and edges for $\mathcal{G}_p^i$ from $\mathcal{G}^i$

```
begin
    G_p^i.N ← ∅
    G_p^i.E ← ∅
    foreach n ∈ G^i.node() do
        pg_num ← n.StartAddr() / 4096
        G_p^i.N ← G_p^i.N ∪ {pg_num}
        if n.StartAddr() / 4096 ≠ n.EndAddr() / 4096 then
            pg_num ← n.EndAddr() / 4096
            G_p^i.N ← G_p^i.N ∪ {pg_num}
            G_p^i.E ← G_p^i.E ∪ {< pg_num − 1, pg_num >}
            w(pg_num − 1, pg_num) ← w(pg_num − 1, pg_num) + 1
        end
    end
    N_tmp ← {G^i.Entry()}
    repeat
        n ← head(N_tmp)
        pg_num ← n.StartAddr() / 4096
        foreach n_s ∈ n.successor() do
            pg_next ← n_s.StartAddr() / 4096
            if pg_num ≠ pg_next then
                G_p^i.E ← G_p^i.E ∪ {< pg_num, pg_next >}
                w(pg_num, pg_next) ← w(pg_num, pg_next) + 1
            end
        end
        N_tmp ← N_tmp \ {n}
        N_tmp ← N_tmp ∪ {n.successor()}
    until N_tmp ≠ ∅;
    return G_p^i
end
```

- **Generating the order and timing**. Once we have generated the nodes and edges for $\mathcal{G}_p^i$, we then generate the order and timing information. The algorithm works similar to algorithm 1 with the differences that we need to record the new page order information, based on the original order recorded in $\mathcal{G}^i$ while traversing $\mathcal{G}^i$. Also, for timing information, we will accumulate the recorded timing information of the basic blocks that belong to the same page based on the execution order. We will discard our lower and upper bound timing estimation for each basic block that was captured by the TNT packet if they all belong to the same page.

4.3 Vulnerability Identification

ANABLEPS detects both order-based and time-based side-channel vulnerabilities by cross comparing the corresponding ED-CFGs. More specifically, comparing $\mathcal{G}_p$s reveals vulnerabilities at the page-level, which can be exploited by an adversary that monitors the enclave program's page accesses (through page faults or page table entry updates). Comparing $\mathcal{G}_c$s reveal vulnerabilities at the cache-level, which can be ex-

ploited by an adversary that monitors the enclave program's cache accesses. Directly comparing $\mathcal{G}$s reveal vulnerabilities at the basic-block level, which can be exploited by monitoring the branch prediction units [20]. In the following, we use $\mathcal{G}_p$ as examples to illustrate the process of vulnerability detection.

Order-based Vulnerability Detection. We compare every $\mathcal{G}_p^i$ with each other, the program is not vulnerable to page level attack if the order information of every edge been accessed in all $\mathcal{G}_p^i$s are the same. Otherwise, the attacker can infer the secret based on the differences. The algorithm for graph comparison is straightforward: $\mathcal{G}_p^i = \mathcal{G}_p^j$ if and only if the sets of node and edges are identical, including the `Order` list in each node, and the execution counts in the edges. In Figure 1(d)(e)(f), with different input, the execution order of the nodes are different. For instance, by comparing nodes n_1 in $\mathcal{G}_p^1$ and $\mathcal{G}_p^2$, the length of their `Order` lists is different, which can clearly differentiate the two graphs.

Time-based Vulnerability Detection. When any two graphs $\mathcal{G}_p^i$ and $\mathcal{G}_p^j$, $\forall I_i, I_j \in I$, are not vulnerable to order-based side channels. ANABLEPS needs to further investigate time-based vulnerabilities, by comparing the `Time` lists of the corresponding nodes. The comparison of the `Time` lists is as follows: The k^{th} element of n_l.`Time` in node n_l in graph $\mathcal{G}_p^i$ is compared with the k^{th} element of n_l.`Time` in graph $\mathcal{G}_p^j$. However, unlike comparison of the `Order` lists, where any difference can directly conclude the comparison, comparing the `Time` lists is more subtle. The execution time of a program can be influenced by many reasons, such as on-demand paging, caching, interrupts, *etc.*. In practice, each n_l.`Time`$[k]$ is a 2-tuple (t_{mean}, t_{std}), rather than a single value. The first element of the 2-tuple is the mean execution time to reach the successor node from multiple runs and the second element is the one standard deviation. With enough number of samples, the impact from side effects can be reduced.

To generate (t_{mean}, t_{std}) for the list `Time` of each node, the program is executed with the same input $I_i \in I$ L times; so each n_l.`Time`$[k]$ (the k^{th} element of n_l) is also executed L times. The mean and standard deviation are calculated using these L execution time between node n_l and its k^{th} successor. In our implementation, $L = 10$.

Determining the Input Space for $\mathcal{G}^i$. Since the edge in $\mathcal{G}_p^i$ (and $\mathcal{G}_c^i$) can correspond to the jumps in different locations in the program, we can only use the one-to-one mapping relationship between $\mathcal{G}^i$ and I_i to determine the input space for $\mathcal{G}^i$. In particular, for each concrete input $I_{\texttt{syntactic}} \in \{I_{\texttt{fuzz}} \cup I_{\texttt{concolic}}\}$, we run the concolic execution with this seed input again, but we also track the corresponding path constraints for this seed input. Once we have collected the path constraints, we then use a constraint solver to solve the constraints. If no other input satisfies (or the execution time of the solver takes too much time to solve.[2]), it means the input is unique (I_i is completely leakable). Otherwise, we have to use application-specific knowledge to determine the leakage.

[2]We currently set up this time to be 90 minutes.

5 Evaluation

We have implemented ANABLEPS to detect the side-channel vulnerabilities for x86 and x86-64 ELF binaries by integrating and extending a number of open source tools. In particular, we extend Driller [30], which is built atop of AFL [1] and concolic execution, for *Input Generation*, and we use `perf` to configure Intel PT and dynamically collect the runtime information of each input. We built the PT packets decoder based on the open source library, `libipt` [3]. The ED-CFG construction and cross-comparison tool is built using python scripts by analyzing the PT packets, and matching the decoded address to the binary code with `pyelftools` library [5]. To quantify the input space for a given trace, we extended `angr` [36], an easily extensible python-based symbolic execution tool, to negate the constraints of the input we provide and calculate the input space. The prototype of ANABLEPS will be public available at `github.com/OSUSecLab/ANABLEPS`.

In this section, we present our evaluation results. We first describe how we set up the experiment in §5.1, and then describe the experimental results in §5.2. All of our evaluations are performed in Ubuntu Desktop 16.04LTS, running atop Intel i7-7700 CPU, with 32G physical memory.

5.1 Experiment Setup

Benchmark Selection. Ideally we would like to use the SGX programs for the test. However, there are not that many SGX programs available, and therefore we run the legacy applications with library OS (*e.g.*, Grephane-SGX [2]) support for the evaluation. In particular, we selected 8 programs from a variety of applications such as data analytics and machine learning, image processing, and text processing. The name of these programs is presented in the first column of Table 1.

Functionality Under Test. Each of the tested benchmark program contains quite sophisticated functionalities. Certainly, we cannot test all of their functionalities; we only tested the functionality of our interest (shown in the 2nd column of Table 1), based on our best understanding with the benchmarks. For instance, when testing `Genometools`, we know the genomic related program usually takes two types of input: `bed` format and `gff3` format. Converting between these two formats is a widely used operation in genomes. Therefore, we test the genome library `libgenometools.so` by converting `bed` format to `gff3` format.

Input Generation. To launch each of the testing program with Driller [30], we provide the seed inputs based on our best understanding of the program. Even with both AFL and concolic execution, we still cannot explore all the program paths. We therefore configure Driller [30] to run 48 hours for each

of the testing program. The number of syntactic inputs eventually generated are presented in the 3rd column of Table 1.

Trace Collection. With the input generated above, we run the tested program traced by Intel PT. The tested program is run outside of SGX in a debug mode. The execution time would be similar to that of executing inside enclaves, because instructions executed in the enclave-mode and non-enclave-mode have the same timing constraints (the main timing difference happens at ECalls/OCalls). Each input generated a separate trace file. The total size of the decoded PT trace file for each program is presented in the 4th column of Table 1. Depending on the size and input to the program, this size varies from a few Gigabytes to several hundreds of Gigabytes.

5.2 Experimental Results

Next, we present how ANABLEPS detects the branch level, page level, and cache level side-channel vulnerabilities based on each individual trace and their corresponding input. As we have described, from each input (and its corresponding execution trace), we first built their ED-CFGs, namely $\mathcal{G}^i$s, which are used to detect the branch level side channels. The total number of such ED-CFGs is presented in the 5th column of Table 1. Compared to the 3rd column of Table 1, we can notice that except for three benchmarks (namely Freetype, QRcodegen, and Genometools), the total number of unique $\mathcal{G}^i$s are all smaller than the total number of the syntactic inputs generated by ANABLEPS.

Detecting Order-based Side Channels. To detect order-based side channels, we first cross-compare all of the $\mathcal{G}^i$s ($\mathcal{G}^i_p$s or $\mathcal{G}^i_c$s) to detect whether there is any unique I_i that maps to a particular $\mathcal{G}^i$ ($\mathcal{G}^i_p$ or $\mathcal{G}^i_c$). As we are detecting order-based side channels, only `Order` of the $\mathcal{G}^i$s ($\mathcal{G}^i_p$s or $\mathcal{G}^i_c$s) are used in the comparison. Many inputs have such a one-to-one mapping $\mathcal{G}^i \leftrightarrow I_i$ ($\mathcal{G}^i_p \leftrightarrow I_i$ or $\mathcal{G}^i_c \leftrightarrow I_i$), which suggests that no other input I_j maps to the same $\mathcal{G}^i$. The branch-level, page-level and cache-level statistics for this mapping is reported in the 6th column of Table 1, the 3rd column of Table 2, and the 8th column of Table 2, respectively. From the table, we can notice that compared to the branch-level vulnerabilities, less one-to-one mappings are detected in page-level and cache-level. For instance, while all inputs of `dA` in deep learning can be recovered by branch-level side channel, they cannot be recovered by page-level side channels.

As the traces are dynamically collected, the node or edge which can differ any two $\mathcal{G}^i$s ($\mathcal{G}^i_p$s or $\mathcal{G}^i_c$s) must leak some secret of interest. It is possible that many nodes or edges only leak a partial secret. However, for some program, a set of vulnerable nodes can be used together to leak the entire secret (*e.g.*, the Deep Learning case uses two nodes to leak the entire secret). Moreover, it is also possible that part of leaked secret can be used to infer the entire secret (*e.g.*, the padding oracle attack for crypto algorithms only need to know if the padding is correct or not).

Detecting Time-based Side Channels. For those that have multiple inputs corresponding to the same trace, *i.e.*, one-to-N mappings ($\mathcal{G}^i \rightarrow I_i s$, $\mathcal{G}^i_p \rightarrow I_i s$ or $\mathcal{G}^i_c \rightarrow I_i s$), their statistics are reported in the 8th column of Table 1, the 4th column of Table 2, and the 9th column of Table 2, respectively. Next, we use the timing information to further differentiate $\mathcal{G}^i$ ($\mathcal{G}^i_p$ and $\mathcal{G}^i_c$) and see whether there is still one-to-one mapping (*i.e.*, $\mathcal{G}^i \leftrightarrow I_i$) after considering the timing differences. That is, we hope to determine whether there are time-based side-channel vulnerabilities when the program is not vulnerable to order-based side channels. In practice, only large enough time differences can be used to differentiate two traces. Therefore, thresholds are defined from empirical results. We report under three different threshold settings (*i.e.*, with $t_1 = 2ns, t_2 = 10ns$, and $t_3 = 20ns$), the number of such one-to-one mappings, and these results are reported in the last three columns of Table 1, the columns 5 to 7 and columns 10 to 12 of Table 2. We notice that it is relative hard to differentiate inputs based on timing information at branch level. However, many inputs can be further differentiated after applying time information at page or cache level.

Determining Input Spaces. Previous experiments are based on generated inputs I_is. However, not all inputs in the whole inputs set are generated. Therefore, we would like to know whether there is only one input I_i in the whole inputs set that can map to a particular $\mathcal{G}^i$, that is, if $|\{I_j | \mathcal{E}(p, I_j) = \mathcal{G}^i, \forall j \in I\}| = 1$, which can be determined by using concolic execution. If so, then the input I_i can be differentiated by order-based vulnerabilities. The total number of symbolic execution determined input $I_{\texttt{determinstic}}$ is reported in the 7th column of Table 1. We can see that for some applications, such as QRcodegen and Deep learning, $I_{\texttt{determinstic}}$ is non-zero, meaning at branch-level some inputs of these programs can be uniquely identified by execution traces. For some applications, $I_{\texttt{determinstic}}$ is zero, indicating by the constraint solver that there are other inputs that all have the same execution traces with generated inputs, *e.g.*, function `Sort` in `gsl`, although $|\mathcal{G}^i \leftrightarrow I_i|$ is non-zero (120 for `gsl`).

However, the concolic execution cannot finish for five programs (marked with $\times$ in the Table), including Hunspell, PNG, and Freetype, because of the limitation in either computation power or physical memory space. For these programs, ANABLEPS cannot answer if these execution traces will completely leak the information of the input.

5.2.1 Performance Overhead

We also measured the performance of ANABLEPS, though it is an offline analysis tool. We report the execution time for each of the key component of ANABLEPS in Table 3. More specifically, during the Input Generation (IG) phase,

Benchmark Program	Functionality under Test	$\lvert I_{\texttt{syntactic}}\rvert$	Trace Size (GB)	$\lvert\mathcal{G}^i\rvert$	Detecting Branch Side Channel					
					$\lvert\mathcal{G}^i \leftrightarrow I_i\rvert$	$\lvert I_{\texttt{deteministic}}\rvert$	$\lvert\mathcal{G}^i \rightarrow I_i s\rvert$	$\lvert\mathcal{G}^i \leftrightarrow I_i\rvert$ t_1	t_2	t_3
Deep Learning	dA	214	76.8	214	214	214	0	-	-	-
	SdA	176	384.2	176	176	176	0	-	-	-
	DBN	152	139.0	152	152	152	0	-	-	-
	RBM	187	225.9	55	16	0	39	56	43	0
	LogisticRegression	198	25.1	41	18	0	23	31	5	0
gsl	Sort	220	2.8	154	120	0	34	0	0	0
	Permutation	200	3.0	135	100	×	35	0	0	0
Hunspell	Spell Checking	231	307.2	168	157	×	11	1	0	0
PNG	Image Render	294	82.3	135	120	×	15	0	0	0
Freetype	Character Render	206	352.6	206	206	×	0	-	-	-
Bio-rainbow	Bioinfo Clustering	128	51.3	119	118	0	1	0	0	0
QRcodegen	Generate QR Code	204	17.9	204	204	204	0	-	-	-
Genometools	bed to gff3 convertion	201	382.4	25	12	×	13	0	0	0

Table 1: The benchmark programs, their concrete input size, the corresponding PT trace size, and the result of branch level side channel detection

Benchmark Programs	Functionality Under Test	Detecting Page Side Channel					Detecting Cache Side Channel				
		$\lvert\mathcal{G}_p^i \leftrightarrow I_i\rvert$	$\lvert\mathcal{G}_p^i \rightarrow I_i s\rvert$	$\lvert\mathcal{G}_p^i \leftrightarrow I_i\rvert$ t_1	t_2	t_3	$\lvert\mathcal{G}_c^i \leftrightarrow I_i\rvert$	$\lvert\mathcal{G}_c^i \rightarrow I_i s\rvert$	$\lvert\mathcal{G}_c^i \leftrightarrow I_i\rvert$ t_1	t_2	t_3
Deep Learning	dA	127	12	65	52	9	214	0	-	-	-
	SdA	112	5	33	28	0	176	0	-	-	-
	DBN	128	9	15	11	0	152	0	-	-	-
	RBM	28	15	56	43	0	55	16	56	43	0
	LogisticRegression	6	9	82	24	0	18	23	82	24	0
gsl	Sort	17	12	0	0	0	33	16	0	0	0
	Permutation	100	2	0	0	0	100	2	0	0	0
Hunspell	Spell Checking	156	11	5	2	2	157	11	7	7	7
PNG	Image Render	103	25	1	1	1	111	22	10	6	0
Freetype	Character Render	206	0	-	-	-	206	0	-	-	-
Bio-rainbow	Bioinfo Clustering	39	9	1	0	0	118	1	0	0	0
QRcodegen	Generate QR Code	204	0	-	-	-	204	0	-	-	-
Genometools	bed to gff3 convertion	5	8	5	5	3	5	8	5	5	3

Table 2: The page level and cache level vulnerability detection results for the tested benchmark programs

Benchmark Programs	Functionality Under Test	IG (h)	TC (h)	VI (m)	CS (h)
Deep Learning	dA	48	26.1	7.9	31.3
	SdA	48	187.2	61.7	132.1
	DBN	48	63.3	43.8	82.3
	RBM	48	110.1	13.2	45.9
	LogisticRegression	48	7.2	1.8	8.4
gsl	Sort	48	0.62	0.2	2.5
	Permutation	48	0.57	0.2	-
Hunspell	Spell Checking	48	68.2	4.4	-
PNG	PNG Image Render	48	19.8	1.6	-
Freetype	Character Render	48	87.4	19.8	-
Bio-rainbow	Clustering bioinformatics	48	14.2	20.9	58
QRcodegen	Generate QR Code	48	8.89	22.4	126
Genometools	bed to gff3 convertion	48	192.6	15.2	-

Table 3: Performance overhead for running each component of ANABLEPS the tested programs. IG stands for Input Generation, TC stands for Trace Construction, VI stands for Vulnerability Identification, and CS stands for Constraint Solver

we configured ANABLEPS to run 48 hours for all of the benchmarks. Then, our Trace Construction (TC) component decodes the trace, builds each $\mathcal{G}^i$, $\mathcal{G}_p^i$, and $\mathcal{G}_c^i$. For the Vulnerability Identification (VI), ANABLEPS just performs the cross-comparison with the graphs we have built. Only when detecting the branch-level side channel, we invoke Constraint Solver (CS) to determine whether there is a unique input for a specific trace. This execution time is reported in the last column of Table 3. For certain programs that concolic execution cannot finish (marked with '-' in the Table), we cannot evaluate their performance overhead. We can notice that the bottleneck of the ANABLEPS is Trace Construction and Constraint Solver, which are affected by the size of execution trace files and computation power.

6 Exploitability of the Vulnerability

So far, we have discussed the design, implementation and evaluation of ANABLEPS in automatically detecting order and time based side-channel vulnerabilities. However, automated tools can only provide *syntactic*-level analysis. Oftentimes, such analysis cannot be directly translated into exploitability of the program, especially when the input space of interest (to the attackers) cannot be automatically determined. In this section, we discuss how ANABLEPS can be used by enclave program developers to analyze the exploitability of the vulnerabilities by providing the proper input, locating and exploiting the vulnerabilities.

6.1 Developer-assisted Vulnerability Analysis

Developer-supplied Input. While the automated syntactic analysis has provided a large number of inputs, not all of them are of interest to attackers. For instance, in the `PNG` example, not all input correspond to valid images; it is not very interesting to determine the errors in the PNG file formats. In practice, only software developers are able to identify the true secretive set of input that they would like to make indistinguishable. This is called *semantic*-level analysis. Developers may select the set of inputs I that she wishes to be indistinguishable from the execution traces and use ANABLEPS to analyze $\mathcal{E}(p, I)$.

The steps to perform such an analysis is similar to automated analysis described in §4. The only difference is that the *Input Generation* step and "determining input spaces for $\mathcal{G}^i$" of the *Vulnerability Identification* step can be skipped, as the set of input of interest is now provided by the developers. The output of the analysis would be $|\mathcal{G}^i \leftrightarrow I_i|$ that can be differentiated by order or time information of the execution traces.

Locating Vulnerabilities. Given a secretive set of input I, if $|\mathcal{E}(p, I)| > 1$, we would like to find the set of nodes in $\mathcal{G}$ that can be used to learn the inputs. That is, we would like to locate the vulnerabilities (*i.e.*, vulnerable node) in the graph and the program. With the method discussed in §4, we can differentiate *order-based vulnerable nodes* and *time-based vulnerable nodes*. The capability to easily locate vulnerabilities is one benefit of adopting ED-CFG to represent execution traces.

With the input set of $I_{\texttt{syntactic}}$, the statistics of the vulnerable nodes are shown in Table 4. The cache-level statistics are listed in the column 3 to 5, and the page-level statistics are reported in column 6 to 8. The total numbers of nodes in $\mathcal{G}_c^i$ and $\mathcal{G}_p^i$ are shown in column 3 and 6; the numbers of order-based vulnerable nodes are listed in column 4 and 7; and the numbers of time-based vulnerable nodes are listed in column 5 and 8, respectively. In the Table 4, the time-based vulnerable nodes are mutually exclusive with the order-based vulnerable nodes. According to the results presented in Table 4, ANABLEPS narrows down the number of nodes to be examined for side-channel vulnerabilities dramatically. On average, the number of order-based vulnerable nodes is only 18% of all nodes in $\mathcal{G}_c$, and 37% of all nodes in $\mathcal{G}_p$; the number of time-based vulnerable nodes is only 6% of all nodes in $\mathcal{G}_c$, and 13% of all nodes in $\mathcal{G}_p$. The fraction of vulnerable nodes can be further reduced with a developer-supplied input set that is of interest.

6.2 Case Studies of the Exploitability Analysis

In this section, we briefly summarize three interesting cases to show how ANABLEPS can help enclave developers identify side-channel vulnerabilities that can be exploited to extract sensitive information.

```
1  int binomial(int n, double p) {
2    ...
3    for(i=0; i<n; i++) {
4      r = rand() / (RAND_MAX + 1.0);
5      if (r < p) c++;
6    }
7    ...
8  }


11 void dA_get_corrupted_input(dA* this, int Ax, int Atilde_x, double p)
       {
12   int i;
13   for(i=0; i<this->n_visible; i++) {
14     if(x[i] == 0) {
15       tilde_x[i] = 0;
16     } else {
17       tilde_x[i] = binomial(x[i], p);
18     }
19   }
20 }
```

Figure 2: The deep learning vulnerable code

6.2.1 Deep Learning Algorithms

According to Table 2, there are 214 different inputs for algorithm `dA` that has unique $\mathcal{G}_c$, *i.e.*, $\mathcal{G}_c^i \leftrightarrow I_i$. Therefore, potentially the vulnerabilities in `dA` may lead to exploitable information leakage. In order to start analyzing the vulnerabilities in `dA` algorithm, we first manually selected inputs that might be of interest to attackers: a set of $|I|$ training data that differ only in values. Then, we feed these inputs to ANABLEPS. The output of ANABLEPS indicates that all selected inputs have unique cache-level execution traces, *i.e.*, $|\mathcal{E}(p, I)| = |I|$.

After locating the vulnerable nodes and some manual effort to examine the identified vulnerable nodes, we find the leakage primarily comes from function `dA_get_corrupted_input()`, which has a `for` loop that enumerates every element of array `x` and calls function `binomial()` if the element is not 0. The code snippet is shown in Figure 2.

The execution of `dA_get_corrupted_input()` and `binomial()` may be exploited to leak training data information. Whether or not function `binomial()` is called by `dA_get_corrupted_input()` reveals the value of array `x`. The function call sequence can be learned through cache-level side channels. The two functions are located in the same page but different cachelines. After compilation, the `for` loop in `dA_get_corrupted_input()` is compiled into two cachelines, denoted m_1 and m_2, function `binomial()` is compiled into two consecutive cachelines. We denote the first cacheline as m_3. Therefore, if the i^{th} element of array `x` is 0, the order of the executed cachelines is $[m_1, m_2]$; otherwise, the execution order becomes $[m_1, m_2, m_3, m_2]$. This order-based side-channel vulnerability on the cache-level can completely leak the training data of the deep learning algorithm.

6.2.2 Freetype Font Engine

According to Table 2, there are 206 inputs that have unique $\mathcal{G}_p^i$. To validate the page-level vulnerability, we generated some printable characters as input and fed them to ANABLEPS. The result indicates that every input corresponds to a unique $\mathcal{G}_p^i$.

Programs	Functionalities Under Test	Cache Level			Page Level		
		#Nodes	#Order-Based Vulnerable Nodes	#Time-Based Vulnerable Nodes	#Nodes	#Order-Based Vulnerable Nodes	#Time-Based Vulnerable Nodes
Deep Learning	dA	69	9	4	13	2	3
	SdA	109	12	21	22	3	3
	DBN	126	17	81	14	3	10
	RBM	68	8	27	13	2	7
	LogisticRegression	48	2	16	11	0	7
gsl	Sort	31	12	0	11	5	0
	Permutation	99	30	0	29	15	0
Hunspell	Spell checking	302	48	9	47	27	10
PNG	PNG Image Render	640	170	90	53	39	2
Freetype	Character Render	1054	263	18	82	20	13
Bio-rainbow	Bioinfo Clustering	214	16	0	24	2	1
QRcodegen	Generatee QR	176	32	18	15	6	3
Genometools	bed to gff3 convertion	1901	231	9	147	53	5

Table 4: Locating vulnerable nodes in $\mathcal{G}_c$ and $\mathcal{G}_p$

```
1  static void psh_glyph_interpolate_strong_points(...){
2    ...
3      for (; count>0; count--, point++){
4        ...
5          if (psh_point_is_edge_min(point))
6            point->cur_u = ...;
7          else if (psh_point_is_edge_max(point))
8            point->cur_u = ...;
9          else{
10           data = ...;
11             if (delta<=0)
12               point->cur_u = FT_MulFix(...) + ...;
13             else if (delta >= hint->org_len)
14               point->cur_u = FT_MulFix(...) + ...;
15             else
16               point->cur_u = FT_MulDiv(...) + ...;
17         }
18         psh_point_set_fitted(point);
19     }
20 }
```

Figure 3: The freetype vulnerable functions

ANABLEPS has helped us identify the vulnerable nodes. In fact, there are more than one vulnerable nodes. To illustrate these vulnerabilities, we explain the leakage through function `psh_glyph_find_strong_points()` at the page level. The code snippet is shown in Figure 3. `psh_glyph_interpolate_strong_points()` includes a loop to interpolate every strong point into the glyph. Adversaries can recover the strong points position according to the page sequence. More specifically, function `psh_point_is_edge_min()` is placed in page m_1. Functions `FT_MulFix()` and `FT_MulDiv()` are placed in another page, denoted m_2. The page of function `psh_glyph_interpolate_strong_points()` is denoted m_3. The access order of these pages leaks information of the interpolated point: When a point is not marked as a strong point, the order of page access is $[m_3]$; when the strong point is located in the edge, the order of page access is $[m_3, m_1, m_3, m_1, m_3]$; otherwise, the sequence would be $[m_3, m_1, m_3, m_2, m_3, m_1, m_3]$. Given the sequence of this function, the attacker can learn whether each point is strong or not. Though the example does not completely leak the content of the data, it illustrates how leakage can be identified.

```
1  8050920 <get_parser>:
2  ...
3  8050b05:    89 1c 24          mov    %ebx,(%esp)
4  8050b08:    83 c3 02          add    $0x2,%ebx
5  8050b0b:    e8 e0 63 00 00    call   8056ef0 <
       _unguarded_linear_insert>
6  8050b10:    39 de             cmp    %ebx,%esi
7  8050b12:    75 f1             jne    8050b05 <get_parser+0x1e5>
8  ...

11 8056ef0 <_unguarded_linear_insert>:
12 ...
13 8056f28:    89 c2             mov    %eax,%edx
14 8056f2a:    89 d8             mov    %ebx,%eax
15 8056f2c:    0f b7 18          movzwl (%eax),%ebx
16 8056f2f:    66 89 1a          mov    %bx,(%edx)
17 8056f32:    0f b6 50 ff       movzbl -0x1(%eax),%edx
18 8056f36:    8d 58 fe          lea    -0x2(%eax),%ebx
19 8056f39:    0f b6 70 fe       movzbl -0x2(%eax),%esi
20 8056f3d:    c1 e2 08          shl    $0x8,%edx
21 8056f40:    01 f2             add    %esi,%edx
22 8056f42:    66 39 ca          cmp    %cx,%dx
23 8056f45:    77 e1             ja     8056f28 <
       _unguarded_linear_insert+0x38>
24 ...
```

Figure 4: The assembly code of `std::sort`

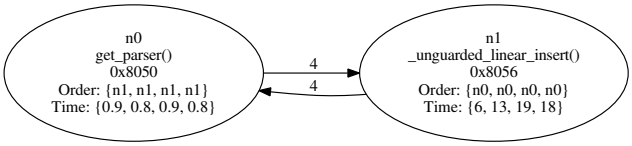


Figure 5: A subgraph of $\mathcal{G}_p$ for function `std::sort`

6.2.3 Hunspell

Hunspell is a popular spell checker. Xu *et al.* identified that Hunspell is vulnerable to page-level controlled channel attacks due to its input-dependent access pattern to data pages [43]. But its control flow was considered immune to side-channel attacks. However, as shown in Table 2, ANABLEPS identifies various control-flow side-channel vulnerabilities that may be exploited by attackers.

With the help of ANABLEPS, we narrow down our attention to the `get_parser()` function of Hunspell, in which the function `std::sort(vector.begin(), vector.end())` is called to sort the data in the vector. We found this function both have cache-level and branch-level order-based vulnerability and page-level time-based vulnerability. This is a function implemented in C++ standard library. After compilation,

the linear insertion algorithm is used in this sort function with the snippet of assembly code in Figure 4. According to the code snippet of function `get_parser()`, the function `_unguarded_linear_insert()` is called when an element in the unsorted vector is to be inserted into the sorted vector. As such, by monitoring the execution sequence that involves this function, the attacker is able to learn the number of elements to be sorted in page-level, cache-level and branch-level. Moreover, function `_unguarded_linear_insert()` contains a loop to compare the element to be inserted with elements already in the sorted vector. According to the insertion sort algorithm, the number of loops in function `_unguarded_linear_insert()` reflects the number of comparisons during the insertion, which can be used to infer the location of an element after the insertion.

Such leakage can be easily identified in $\mathcal{G}_p$s with time-based vulnerability. A subgraph of $\mathcal{G}_p^i$ of a particular input I_i is shown in Figure 5. The edge $n_0 \rightarrow n_1$ is executed 4 times, which reflects that four elements are being sorted. The elements of `Time` list in node n_1 reveals the number of comparisons in function `_unguarded_linear_insert()`: the first element corresponds to no comparison, the second element corresponds to 1 comparison, the third and fourth elements correspond to 2 comparisons. Therefore, the page level order-based vulnerability in Hunspell, or more precisely the `sort` algorithm implemented in the standard C++ library, can only leak the number of elements to be sorted; however, the time-based vulnerability can be exploited to leak the list to be sorted if sorting result is known. We specially tested the `sort` algorithm by providing a set of $|I|$ unsorted lists that correspond to the same sorted list after sorting. As expected, ANABLEPS reports $|\mathcal{E}(p, I)| = |I|$ for this set of inputs.

7 Limitations and Future Work

Although we have demonstrated that ANABLEPS is capable of identifying side-channel vulnerabilities in enclave binaries, we only made a first step and there are a number of avenues for future works. First, the currently design only considers side-channel vulnerabilities due to secret-dependent control flows. Leakages due to secret-dependent data accesses are out of scope currently. Interestingly, the differences in the data access pattern caused by divergence in the control flow can actually be identified by ANABLEPS's control-flow based vulnerability analysis. What is missed by ANABLEPS is memory pointers or array indexes that are determined by the secret values. One of the future works is to extend ANABLEPS in handling of these vulnerabilities.

Second, while ANABLEPS has integrated the state-of-the-art input generation tools such as fuzzing and concolic execution, it still cannot generate the complete set of input. Currently, we rely on developers' knowledge to remediate this limitation since developers have the best understanding of the semantic of the enclave program and its input space. Certainly, any advances in the research of test case generation itself will improve ANABLEPS.

Third, the capability of the constraint solver is limited. Given an input to a program, ANABLEPS relies on symbolic execution to collect constraints. These constraints are solved by a constraint solver to determine the size of $\mathcal{G}^i$'s input space. However, not all the constraints can be solved (e.g., hash functions). Also, a solver may take too much time to solve a constraint. Currently, ANABLEPS requires the solver to return the result in 90 minutes. Otherwise, it considers unsolvable. Any advancement of constraint solver will make ANABLEPS more efficient.

8 Conclusion

In conclusion, we designed and implemented ANABLEPS, a software tool for automatically vetting side-channel vulnerabilities in SGX enclave programs. ANABLEPS is the first side-channel vulnerability analysis tool that considers both time and order of a program's memory access patterns. It leverages concolic execution and fuzzing techniques to generate input sets for an arbitrary enclave program, constructs extended dynamic control-flow graph representation of execution traces using Intel PT, and automatically analyzes and identifies side-channel vulnerabilities using graph analysis. With ANABLEPS, we have uncovered a large number of side channel leaks in enclave binaries we tested. Our experimental results also demonstrate ANABLEPS can be used by both security analysts and software developers to identify the side-channel vulnerabilities for enclave programs.

Acknowledgments

We would like to thank the anonymous reviewers for their very helpful comments. This work was supported in part by the NSF grants 1750809, 1718084, 1834213, 1834215, and 1834216 as well as a research gift from Intel.

References

[1] american fuzzy lop. `http://lcamtuf.coredump.cx/afl/`. (Accessed on 04/28/2018).

[2] Graphene / graphene-SGX library os - a library os for linux multi-process applications, with intel SGX support. `https://github.com/oscarlab/graphene`.

[3] libipt - an intel(r) processor trace decoder library. `https://github.com/01org/processor-trace`.

[4] Lists of instruction latencies, throughputs and micro-operation breakdowns for intel, amd and via cpus. `https://www.agner.org/optimize/instruction_tables.pdf`.

[5] pyelftools - parsing elf and dwarf in python. `https://github.com/eliben/pyelftools`.

[6] Intel® software guard extensions enclave writer's guide. https://software.intel.com/sites/default/files/managed/ae/48/Software-Guard-Extensions-Enclave-Writers-Guide.pdf, 2017. Revision 1.02, Accessed May, 2017.

[7] E. Bauman and Z. Lin. A case for protecting computer games with SGX. In Proceedings of the 1st Workshop on System Software for Trusted Execution (SysTEX'16), December 2016.

[8] A. Baumann, M. Peinado, and G. Hunt. Shielding applications from an untrusted cloud with haven. ACM Transactions on Computer Systems (TOCS'15), 2015.

[9] F. Brasser, U. Müller, A. Dmitrienko, K. Kostiainen, S. Capkun, and A.-R. Sadeghi. Software grand exposure: SGX cache attacks are practical. In USENIX Workshop on Offensive Technologies, 2017.

[10] G. Chen, S. Chen, Y. Xiao, Y. Zhang, Z. Lin, and T. H. Lai. Stealing intel secrets from SGX enclaves via speculative execution. In Proceedings of the 2019 IEEE European Symposium on Security and Privacy, June 2019.

[11] G. Chen, W. Wang, T. Chen, S. Chen, Y. Zhang, X. Wang, T.-H. Lai, and D. Lin. Racing in hyperspace: Closing hyperthreading side channels on SGX with contrived data races. In 2018 IEEE Symposium on Security and Privacy (SP'18). IEEE, 2018.

[12] S. Chen, X. Zhang, M. K. Reiter, and Y. Zhang. Detecting privileged side-channel attacks in shielded execution with déjà vu. In 12th ACM on Asia Conference on Computer and Communications Security (ASIA CCS '17). ACM.

[13] V. Costan and S. Devadas. Intel SGX explained. Technical report, Cryptology ePrint Archive, Report 2016/086, 20 16. http://eprint. iacr. org.

[14] Y. Fu, E. Bauman, R. Quinonez, and Z. Lin. 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 (RAID'17), September 2017.

[15] J. Götzfried, M. Eckert, S. Schinzel, and T. Müller. Cache attacks on intel SGX. In 10th European Workshop on Systems Security (EuroSec'17). ACM, 2017.

[16] D. Gruss, J. Lettner, F. Schuster, O. Ohrimenko, I. Haller, and M. Costa. Strong and efficient cache side-channel protection using hardware transactional memory. In USENIX Security Symposium (USENIX Security'17). USENIX Association, 2017.

[17] M. Hähnel, W. Cui, and M. Peinado. High-resolution side channels for untrusted operating systems. In 2017 USENIX Annual Technical Conference (USENIX ATC 17). USENIX Association,, 2017.

[18] S. M. Kim, J. Han, J. Ha, T. Kim, and D. Han. Enhancing security and privacy of tor's ecosystem by using trusted execution environments. In (NSDI'17), 2017.

[19] P. Kocher, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom. Spectre attacks: Exploiting speculative execution. ArXiv e-prints, Jan. 2018.

[20] S. Lee, M.-W. Shih, P. Gera, T. Kim, H. Kim, and M. Peinado. Inferring fine-grained control flow inside SGX enclaves with branch shadowing. In 26th USENIX Security Symposium (USENIX Security'17). USENIX Association, 2017.

[21] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg. Meltdown. ArXiv e-prints, Jan. 2018.

[22] O. Oleksenko, B. Trach, R. Krahn, M. Silberstein, and C. Fetzer. Varys: Protecting SGX enclaves from practical side-channel attacks. In 2018 USENIX Annual Technical Conference (USENIX ATC 18). USENIX Association, 2018.

[23] D. A. Osvik, A. Shamir, and E. Tromer. Cache attacks and countermeasures: the case of AES. In 6th Cryptographers' track at the RSA conference on Topics in Cryptology, 2006.

[24] C. Percival. Cache missing for fun and profit. In 2005 BSDCan, 2005.

[25] F. Schuster, M. Costa, C. Fournet, C. Gkantsidis, M. Peinado, G. Mainar-Ruiz, and M. Russinovich. Vc3: Trustworthy data analytics in the cloud using SGX. In 2015 IEEE Symposium on Security and Privacy (SP'15). IEEE, 2015.

[26] M. Schwarz, S. Weiser, D. Gruss, C. Maurice, and S. Mangard. Malware guard extension: Using SGX to conceal cache attacks. Springer International Publishing, 2017.

[27] J. Seo, B. Lee, S. Kim, M.-W. Shih, I. Shin, D. Han, and T. Kim. SGX-shield: Enabling address space layout randomization for SGX programs. In In Proceedings of the 2017 Annual Network and Distributed System Security Symposium (NDSS), 2017.

[28] M.-W. Shih, S. Lee, T. Kim, and M. Peinado. T-SGX: Eradicating controlled-channel attacks against enclave programs. In Proceedings of the 2017 Annual Network and Distributed System Security Symposium (NDSS'17), 2017.

[29] S. Shinde, Z. L. Chua, V. Narayanan, and P. Saxena. Preventing page faults from telling your secrets. In 11th ACM on Asia Conference on Computer and Communications Security (ASIA CCS'16). ACM, 2016.

[30] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, and G. Vigna. Driller: Augmenting fuzzing through selective symbolic execution. In (NDSS'16), 2016.

[31] C.-C. Tsai, D. E. Porter, and M. Vij. Graphene-SGX: A practical library os for unmodified applications on SGX. In Proceedings of the USENIX Annual Technical Conference (ATC'17), 2017.

[32] J. Van Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, and R. Strackx. Foreshadow: Extracting the keys to the intel SGX kingdom with transient out-of-order execution. In 27th USENIX Security Symposium (USENIX Security'18). USENIX Association, 2018.

[33] J. Van Bulck, F. Piessens, and R. Strackx. SGX-step: A practical attack framework for precise enclave execution control. In Proceedings of the 2Nd Workshop on System Software for Trusted Execution, (SysTEX'17), 2017.

[34] J. Van Bulck, F. Piessens, and R. Strackx. Nemesis: Studying microarchitectural timing leaks in rudimentary cpu interrupt logic. In Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS'18). ACM, 2018.

[35] J. Van Bulck, N. Weichbrodt, R. Kapitza, F. Piessens, and R. Strackx. Telling your secrets without page faults: Stealthy page table-based attacks on enclaved execution. In Proceedings of the 26th USENIX Security Symposium (USENIX Security'17). USENIX Association, 2017.

[36] R. Wang, Y. Shoshitaishvili, A. Bianchi, A. Machiry, J. Grosen, P. Grosen, C. Kruegel, and G. Vigna. Ramblr: Making reassembly great again. In Proceedings of the 24th Annual Symposium on Network and Distributed System Security (NDSS'17), 2017.

[37] S. Wang, P. Wang, X. Liu, D. Zhang, and D. Wu. Cached: Identifying cache-based timing channels in production software. In 26th USENIX Security Symposium (USENIX Security'17). USENIX Association, 2017.

[38] W. Wang, G. Chen, X. Pan, Y. Zhang, X. Wang, V. Bindschaedler, H. Tang, and C. A. Gunter. Leaky cauldron on the dark land: Understanding memory side-channel hazards in SGX. In Proceedings of the ACM SIGSAC Conference on Computer and Communications Security, (CCS'17). ACM, 2017.

[39] S. Weiser, A. Zankl, R. Spreitzer, K. Miller, S. Mangard, and G. Sigl. DATA – differential address trace analysis: Finding address-based side-channels in binaries. In 27th USENIX Security Symposium (USENIX Security'18). USENIX Association, 2018.

[40] J. Wichelmann, A. Moghimi, T. Eisenbarth, and B. Sunar. Microwalk: A framework for finding side channels in binaries. In Proceedings of the 34th Annual Computer Security Applications Conference. ACM, 2018.

[41] J. C. Wray. An analysis of covert timing channels. J. Comput. Secur., 1992.

[42] Y. Xiao, M. Li, S. Chen, and Y. Zhang. Stacco: Differentially analyzing side-channel traces for detecting SSL/TLS vulnerabilities in secure enclaves. In Proceedings of the ACM SIGSAC Conference on Computer and Communications Security, (CCS'17). ACM, 2017.

[43] Y. Xu, W. Cui, and M. Peinado. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In Proceedings of the 2015 IEEE Symposium on Security and Privacy (SP'15). IEEE, 2015.

[44] F. Zhang, E. Cecchetti, K. Croman, A. Juels, and E. Shi. Town crier: An authenticated data feed for smart contracts. In Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS'16). ACM, 2016.